(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 2025-07-23

Table of Contents

OFB – Object oriented Function Block Graphic – using LibreOffice draw – Basics and Handling..... 1
1 Open/Libre Office for Graphical programming..... 2
2 Join FBlock Diagrams and UML-Class Diagrams..... 3
3 Approaches for the graphic, basic consideration..... 4
3.1 Question of sizes and grid snapping in diagram..... 4
3.2 Using figures with styles (indirect formatted) for element..... 7
3.3 Pins..... 8
3.4 Connectors of LibreOffice for References between classe..... 9
3.5 Connect Points for more complex reference..... 10
3.6 Diagrams with cross reference Xref..... 11
3.7 Outfit of the GUI in LibreOffice draw..... 12
4 Capabilities and concepts of OFB diagrams..... 14
4.1 Graphic blocks, pins and text fields inside a GBlock..... 14
4.2 Connections..... 14
4.3 Show same FBlocks multiple times in different perspective..... 14
4.4 More as one page for the FBlock or class diagram..... 15
4.5 Translating the content of a graphical diagram..... 16
4.6 Function Block and class diagram thinking in one diagram..... 16
4.7 Using events instead sample times in FBlock diagrams..... 18
4.8 Storing the textual representation of OFB in IEC61499..... 20
4.9 Source code generation from the graphic..... 21
4.10 Run and Test and Versioning..... 22
5 Handling with OFB diagrams and LibreOffice draw..... 24
5.1 All Kind of Elements with there style..... 28
5.2 All styles..... 30
5.2.1 GBlock styles, ofb..... 30
5.2.2 Name styles, ofn..... 31
5.2.3 Connector styles, ofc..... 32
5.2.4 Pin styles, ofp..... 34
5.3 Texts in graphic blocks and pins..... 36
5.3.1 Syntax in colored ZBNF..... 36
5.3.2 The complete Syntax of texts for pins and FBlocks..... 37
5.3.3 Syntax of input to a pin..... 38
5.3.4 Examples for description and type..... 39
5.3.5 What contains descr, for expressions and pin designation for FBlocks. .39
5.3.6 type and sizeArrayType..... 40
5.3.7 nrGpos, order of pins after grave..... 41
5.4 Data types..... 42
5.4.1 One letter for the base type..... 42
5.4.2 Unspecified types..... 44
5.4.3 Array data type specification..... 44
5.4.4 Container type specification..... 45
5.4.5 Structured type on data flow..... 46
5.4.6 Data type forward and backward test and propagation..... 47
5.4.7 Using a module with non deterministic data types..... 48

5.4.8 Integer Data types and their scaling and decimal point.....51
5.5 One Module, Inputs and Outputs, file and page layout.....52
5.5.1 Module in odg file(s) organized in pages.....52
5.5.2 Alias control and import.....52
5.5.3 Module pins.....53
5.5.4 Order of pins.....54
5.5.5 The module's input.....56
5.5.6 The module's output.....58
5.6 Possibilities of Graphic Blocks (GBlock).....64
5.6.1 Difference between class, type and instance ("Object").....64
5.6.2 GBlocks for each one function, data – event association.....66
5.6.3 Aggregations are corresponding to ctor or init events.....67
5.6.4 Predefined FBlocks or definition on demand, relation with source code 68
5.6.5 Possibility of inputs of FBlocks.....70
5.6.6 Possibilities of outputs of FBlocks.....72
5.6.7 Expression GBlocks.....74
5.6.8 GBlocks for operation access in line in an expression - FBoper.....74
5.6.9 Conditional execution with boolean FBexpr.....76
5.6.10 Data flow event related – or persistent data.....78
5.6.11 Sliced or Array FBlocks, Demux and array data.....80
5.7 Connection possibilities.....82
5.7.1 Pins.....82
5.7.2 name : Type on pins.....86
5.7.3 Connectors.....86
5.7.4 Connection points.....87
5.7.5 Xref.....87
5.7.6 Using GBmux and GBdemux for connections.....88
5.7.7 Connections from instance variables and twice shown FBlocks.....88
5.7.8 Textual given connections.....88
5.7.9 Admissibility check of connections.....89
5.7.10 Data type test and conversion on inputs.....89
5.7.11 The direction of references and the data flow.....90
5.7.12 More outputs to one input.....90
5.8 Expressions inside the data flow (FBexpr).....92
5.8.1 Expression as rectangle and input pins as rectangle ofpExprPart.....92
5.8.2 More possibilities of DinExpr.....94
5.8.3 Data Type specification and value casting in expressions.....100
5.8.4 Data types with fractional bits in expressions , using saturation.....102
5.8.5 Any expression in FBexpr.....107
5.8.6 Output possibilities, variable after expression.....108
5.8.7 Set elements to a array of structure variable.....109
5.8.8 Output with ofpExprOut.....110
5.8.9 FBexpr as data set.....110
5.8.10 FBoper, operation for a FBlock.....111
5.8.11 How are expressions presented in IEC61499?.....112
5.8.12 FBexpr capabilities compared to other FBlock graphic tools.....114
5.9 Operations to FBlocks inside the data flow (FBoperation).....116
5.9.1 void Operation with input(s) and reference output.....116

5.9.2 What is stored in the IEC61499 FBcl.fbd file:.....117
5.9.3 Operation with return value and reference outputs.....118
5.9.4 Join_OFB for inputs for calculation order.....119
5.9.5 A FBoperation as simple getter.....119
5.10 FBlocks in slices, access to slices.....120
5.10.1 Vectors in expression.....120
5.10.2 Vectors and scalar FBlocks.....121
5.10.3 Slices of named FBlocks.....122
5.10.4 Mux and Demux, build vectors with Mux.....123
5.10.5 Build vectors with elements, access to vector elements.....123
5.11 Execution order, Event and Data flow, Event chains and states.....124
5.11.1 Event and Data flow.....124
5.11.2 Event chains for each one operation, state variables.....127
5.12 Drawing and Source code generation rules.....128
5.12.1 Writing rules in target language used from generated code from OFB 128
5.12.2 Life cycle of programs in embedded control: ctor, init, step and update 129
5.12.3 Using events in the module pins and FBlocks, meaning in C/++.....130
5.12.4 More possibilities, definition of special events.....132
5.13 Showing processes.....134
5.14 Converting the graphic – source code generation.....136
5.14.1 Calling conversion with code generation.....137
5.14.2 Handling of include in C/++ or import and real used type names.....140
5.14.3 Error messages while translating.....140
5.14.4 Templates for code generation.....141
5.15 Presentation of the graphic and results in files.....142
5.15.1 The original odg format (Overview).....142
5.15.2 Graphic saved with the option The original odg format (Overview)….142
5.15.3 The FBcl format or IEC61499, file.fbd.....144
5.15.4 The original odg format (Overview).....146
6 Overview show styles of this document.....148

(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 parameterization (with given *Operation and Monitoring* tools). For updatíng 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 Join FBlock Diagrams and UML-Class Diagrams

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).

3 Approaches for the graphic, basic consideration

Table of Contents

3 Approaches for the graphic, basic consideration....4
3.1 Question of sizes and grid snapping in diagram....4
3.2 Using figures with styles (indirect formatted) for element....7
3.3 Pins.... 8
3.4 Connectors of LibreOffice for References between classe....9
3.5 Connect Points for more complex reference....10
3.6 Diagrams with cross reference Xref....11
3.7 Outfit of the GUI in LibreOffice draw....12

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

3.1 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.

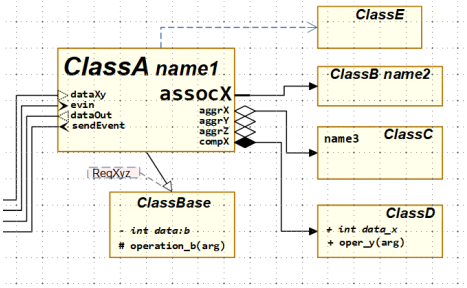


Figure 1: View A4-width as Part (280 DPI)

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.

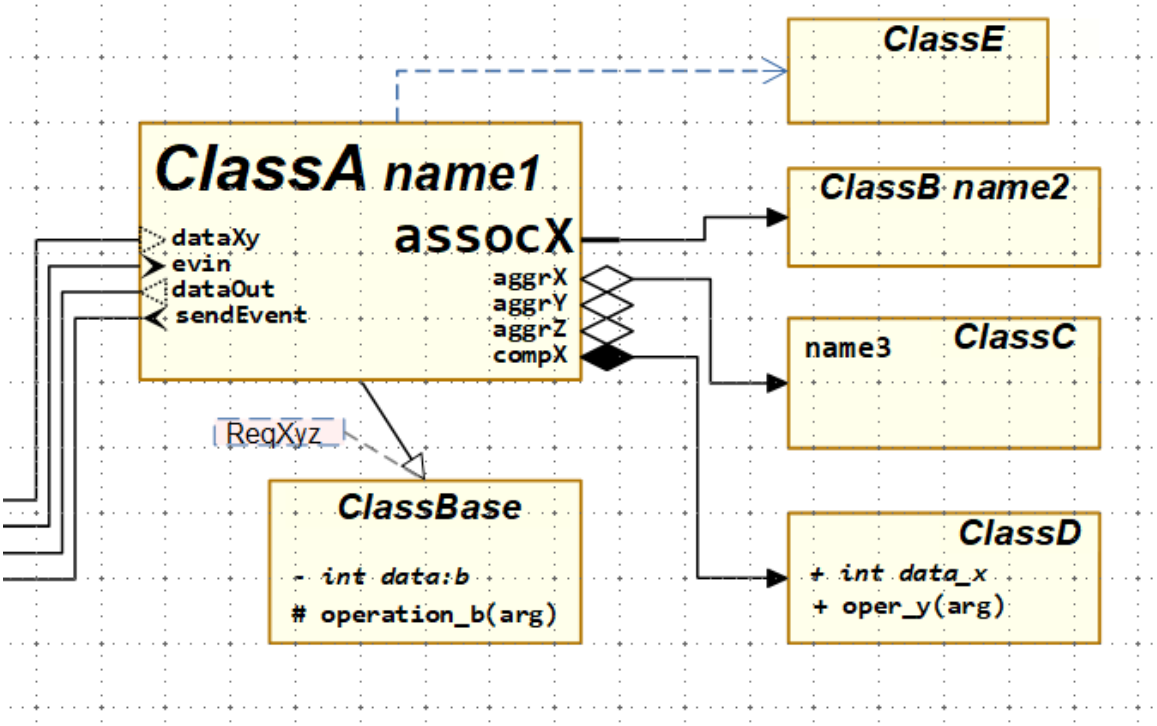


Figure 2: View in original size if this document is displayed with 2 pages on screen (112 DPI)

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:

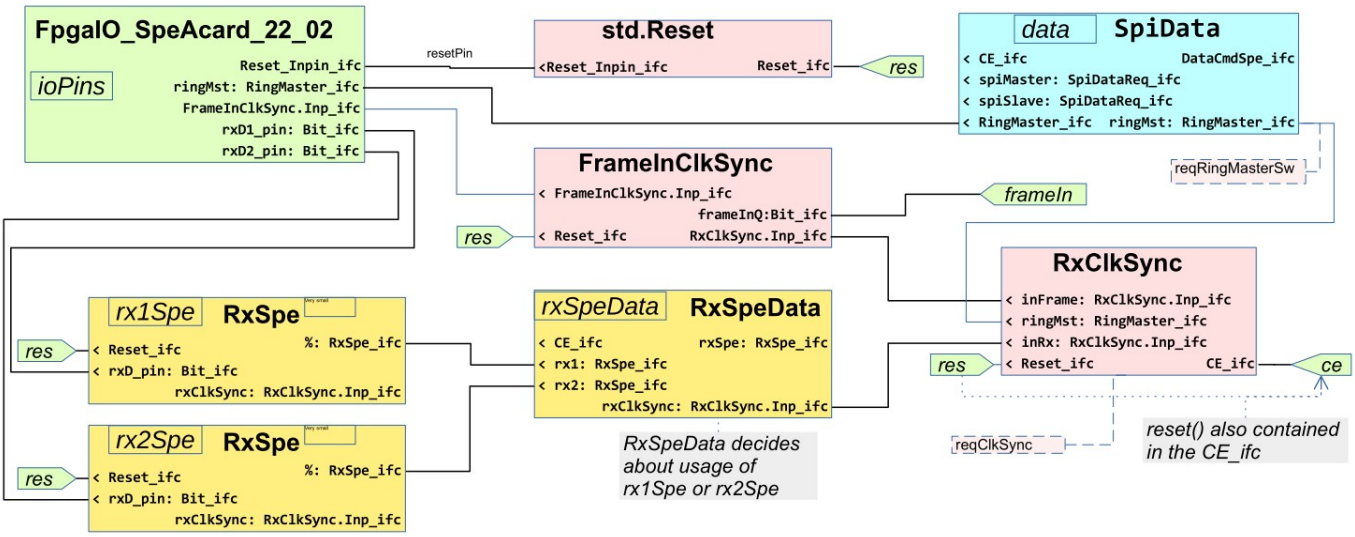


Figure 3: Example for a Module Diagram

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.

3.2 Using figures with styles (indirect formatted) for element

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 *Error: Reference source not found* on page *Error: Reference source not found*

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.

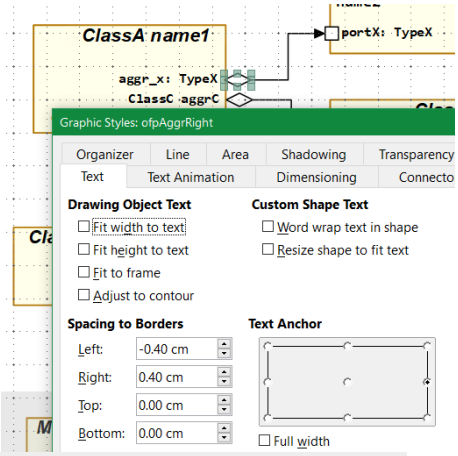


Figure 4: Style_ofpAggrRight_TextProp.png

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.

3.3 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.

3.4 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.

3.5 Connect Points for more complex reference

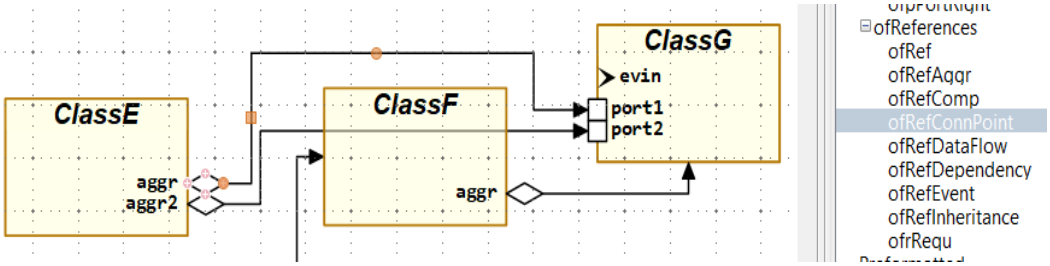


Figure 5: Figure 5:

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.

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.

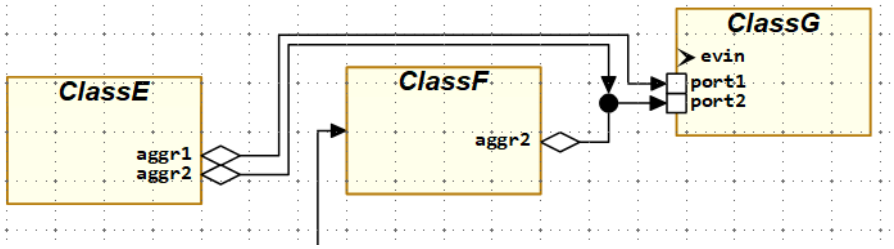


Figure 6: OFB/ConnPoint.png

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`.

3.6 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.

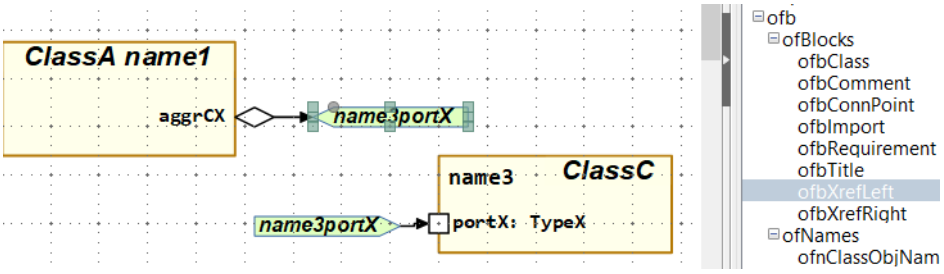


Figure 7: UMLdiagramXrefExample.png Cross Reference usage

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.

3.7 Outfit 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.

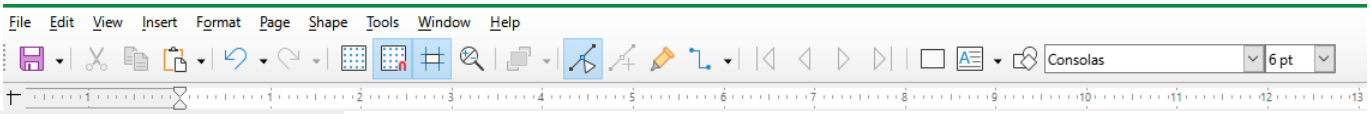


Figure 8: LOffc/IconsTop1.png

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

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.

empty page

4 Capabilities and concepts of OFB diagrams

Table of Contents

4 Capabilities and concepts of OFB diagrams....14
4.1 Graphic blocks, pins and text fields inside a GBlock....14
4.2 Connections....14
4.3 Show same FBlocks multiple times in different perspective....14
4.4 More as one page for the FBlock or class diagram....15
4.5 Translating the content of a graphical diagram....16
4.6 Function Block and class diagram thinking in one diagram....16
4.7 Using events instead sample times in FBlock diagrams....18
4.8 Storing the textual representation of OFB in IEC61499....20
4.9 Source code generation from the graphic....21
4.10 Run and Test and Versioning....22

4.1 Graphic blocks, pins and text fields inside a GBlock

The diagram contains primary Graphic Blocks (GBlock) which are associated to one of the style `ofb…`. This GBlocks should not overlap, should have a well distance each other.

Secondly the graphic consists of ***pins***, which are part of a GBlock. Pins are associated with a style `ofp…` or only `ofPin`. The pins should be associated to a GBlock. This is done via its positions. At least a pin should have one coordinate (left, right, top, bottom) inside the GBlock area, then it is associated to the GBlock. The pins can jut out a little from the GBlock so that the connection points are properly visible.

Third, the GBlock can contain text fields, also possible a little bit jut out, but usual inside the GBlock, with a style `ofn…`. It is for the name and type of a `ofbFBlock` or also for some attributes and operations as known in UML.

See *5.2.1 GBlock styles, ofb* page *30* and *5.6 Possibilities of Graphic Blocks (GBlock)* page *64*

4.2 Connections

The connections between GBlocks and usual between its pins are done with the “connector” in LibreOffice. The connector is glued to pins (or GBlocks) and follow graphical movements. The connected pins and GBlocks are associated with this glue points.

Connectors should have a style `ofc...`. If a pin is not full determined for its kind, the style of the connector determines it. Vice versa a common style for a connector is possible if the pin kind is well defined. Faulty pairing are reported as warning or error.

4.3 Show same FBlocks multiple times in different perspective

There is an interesting and important principle using in UML class diagrams. A class can be presented in more as one perspective in several diagrams, and also more as one time in one diagram. The class is presented by its name, it is also able to find it in the repository of the UML data. The diagrams plays only the role of presentation of the class with its properties just in several perspective.

In opposite, traditional Function Block Diagrams shows one FBlock as one instance. Often the FBlock does not need a specific name, then it is automatically named

The ***OFB*** approach uses the principle, showing also a FBlock in several perspectives, in opposite to traditional FBlock diagrams, but similar as UML. It means, a FBlock as one instance can be shown more as one time in the same diagram or in several pages of the same module also in several files. The FBlock is dedicated by its instance name with a type or by its type name. Drawing a second FBlock with the same name is the same instance. All FBlocks with the same type describes this type in sum.

This principle enables showing complex large FBlocks in several perspectives. Different connections are shown on different places, also the same connection can be shown more as one. For example inputs of one functionality of a FBlock are shown on one page with focus of that input signals, other input signals are shown on a second page, and the output connections and processing are shown on a third one. Also the connections are unique dedicated by its pin name on the named FBlock with the named type. This offers more overview.

The dispersion of one FBlock connectivity in several views may be seen as disadvantage, it becomes confusing. But notice, there are search operations and evaluations of the graphic which gives an overview to find all locations of the same FBlock instance. The idea is newly for FBlock diagrams, look for its advantage.

Now this idea is also usable for the class description idea: Any FBlock instance is dedicated by its type. The type is the class type.

All occurrences of the same type of Flocks are properties of its class. Also FBlock with only the type name, without instance name presents the class properties. The sum of all is the property. This is true for the type of a c FBlock which is a class as also for the connectivity of an instance of a FBlock in several graphic presentations.

Look for example to . The FBlock with name `h3p` is assigned to the type `BpParam`, left bottom. But this block is drawn twice, the second is magenta, has not the type identification because the name is unique, and shows the instance with another event input `ctorObj` and some other data. This is another functionality associated to this same instance, and also to the same class.

4.4 More as one page for the FBlock or class diagram

The chapter above *4.3 Show same FBlocks multiple times in different perspective* allows simple to disperse a diagram over a lot of pages (as necessary) because the same FBlock instance can be shown for example with its input signal wiring, and on another page with its output signals, or group of signals. This allows formally descriptions more near to explanations. One Image (one side) should present one aspect. Which – this is document- or explanation oriented. Data flow connections can also be joined by Xref blocks.

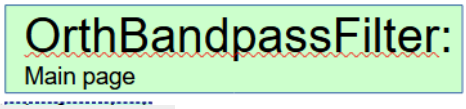


Figure 9: ofbTitle-1.png

Any page need have a title block, of style `ofbTitle`. It contains the name of the module and a short text what it contains.

The pages can contain several modules. The association of module diagrams to files.odg is an important topic. If you have related modules, you can store all it in one file. On the other hand it is possible to have more as one file for one module. This should only be regarded while translation the module.

4.5 Translating the content of a graphical diagram

The graphic is read from the inner data (`content.xml` inside a `LibreOffice.odg` file) and first mapped to inner Java data which are stored in `>>OdgPage`, `>>OdgShape` and then in `>>OdgGBlock` and `>>OdgGpin`. Note: The links refer to the inner software documentation (Javadoc). This is for further explanation if you are interest on how does it work.

In the next step this graphic oriented data are converted to `>>FBlock_FBcl`, `>>FBtype_FBcl`, `>>Pin_FBcl` and `>>PinType_FBcl` and some derived instances. This level of mapping is shown in a textual presentation of the graphic, which is named FBcl (Function Block connection language), see TODO

4.6 Function Block and class diagram thinking in one diagram

One of the basic ideas of the OFB approach is just, join UML thinking and FBlock thinking. UML presents in class diagrams relations between classes. A class is an abstraction of implementation. The implementation uses instances (of classes).

In opposite, ordinary Function Block Diagrams only work with instances. A "class" is an unused word in this way of thinking. But in fact, using a Function Block type from a Library is *“instantiation of a class”*, the library block type is the class.

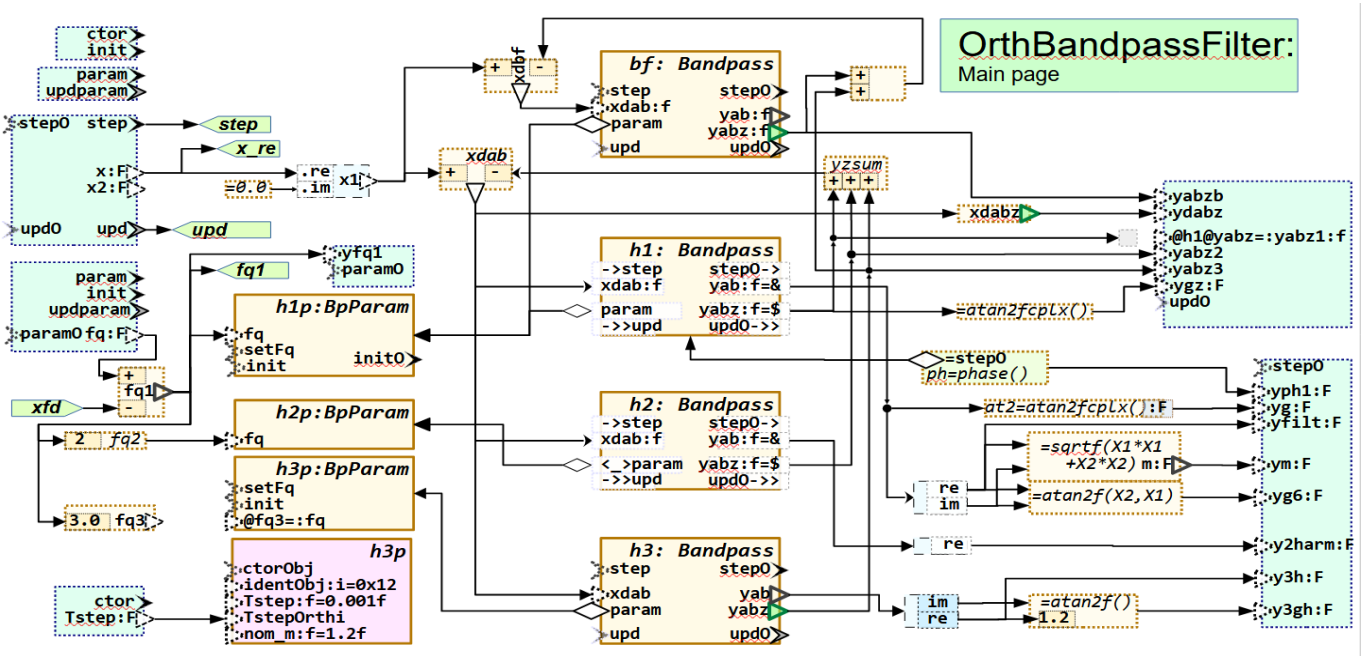


Figure 10: OrthBandpassFilter.odg.png

This image shows primary a Function Block Diagram (FBlock diagram). The green parts are the input and output pins of the module. Some FBlocks presents expressions, these are with dashed lines. The other FBlocks presents instances (each three from the same type) which are connected with data flow.

But from the `Bandpass` FBlocks to the `BpParam` FBlocks there are aggregations. That shows two things:

a) There is an aggregation from the type (class) `Bandpass` to the class `BpParam`. This is a relation of a class diagram.

b) The aggregation from `bf` and `h1` is initialized to refer `h1p`, as also `h2` refers `h2p` and `h3` refers `h3p`. This is a property of the FBlock instances.

The relation shown with the aggregation can be seen also as data flow, but in the opposite direction. Initially the address of the `h1p` FBlock is provided to the `bf` and `h1` FBlock, to refer it, adequate for `h2` and `h3`. Hence, the diagram contains information about class (or type) relations as class diagram and information about instance relations as Function Block Diagram with data flow.

The combination in thinking of FBlock instances, its type (the class) and several operations, here presented by the several events is a kind of ObjectOriented thinking. The “Object” is the instance of a well defined type, the type (class) has some properties valid for all Objects of this type, and it has operations.

The last one aspect, given operations, is also shown in the green block right mid with `phase():F`. This is a shape of style `ofbExpression` but with an aggregation. It means the expression aggregates a FBlock instance, which are the data for the given operation in the expression, and hence the operation is associated to the data type, it is an Object Orientated operation (or method as often named). The second specifity is, this operation should not have side effects, it does not change data in the aggregated object, because it is designated as expression term. This is an important feature of ***Functional Programming***, and unfortunately not so much considered in Object Orientation, but important. In C++ implementation this is an operation ending with `const` after the closing parenthesis if the function definition line:

```
float Bandpass::phase() const {...}
```

but for example in Java it has not a proper counterpart, Java does not know a designation for const operations, unfortunately. (It is not the final keyword!).

In opposite, operations which change data should be present as FBlock with the adequate event. The event characters the operation, as shown on all FBlocks, especially the three different operations shown in two FBlocks `h3p` left bottom. Note that `setFq(float fq)` and `init(float fq)` are defined in the same FBlock, only possible in combination with init.

4.7 Using events instead sample times in FBlock diagrams

Usual for FBlock diagrams sample times are familiar. It follows from the basic approach that the FBlock connections are executed cyclically. That is so in some applications, for example industrial automation control. But sometimes events also play a role. In ordinary automation control often this is regarded by polling (quest of input signals) in a cyclically kind, because their basic operation system supports firstly cycles. The importance of events was often not the focus when such systems were created, although events were common and well-known in other areas of software technology. For example Simulink works basically with *"sample times"* but has specific opportunities (*"triggered subsystem"*) to deal with events.

Well, the assignment of signals and FBlocks to events ***includes working with sampling times***, but also triggered operations. More as that, the ***event flow presents*** better as a data flow the ***execution order of FBlocks***. Only using the data flow sometimes it is not well as necessary predicted. If the execution order is internal information (the user does not see it unless you study the generated source code), then uncertainties remain.

The UFBgl tool allows the automatic derivation of the event flow from the data connections (data flow). The event flow is shown in the textual representation of the graphic and can be viewed or analyzed. It is also possible to determine a specific event connection in the graphic by the user.

...

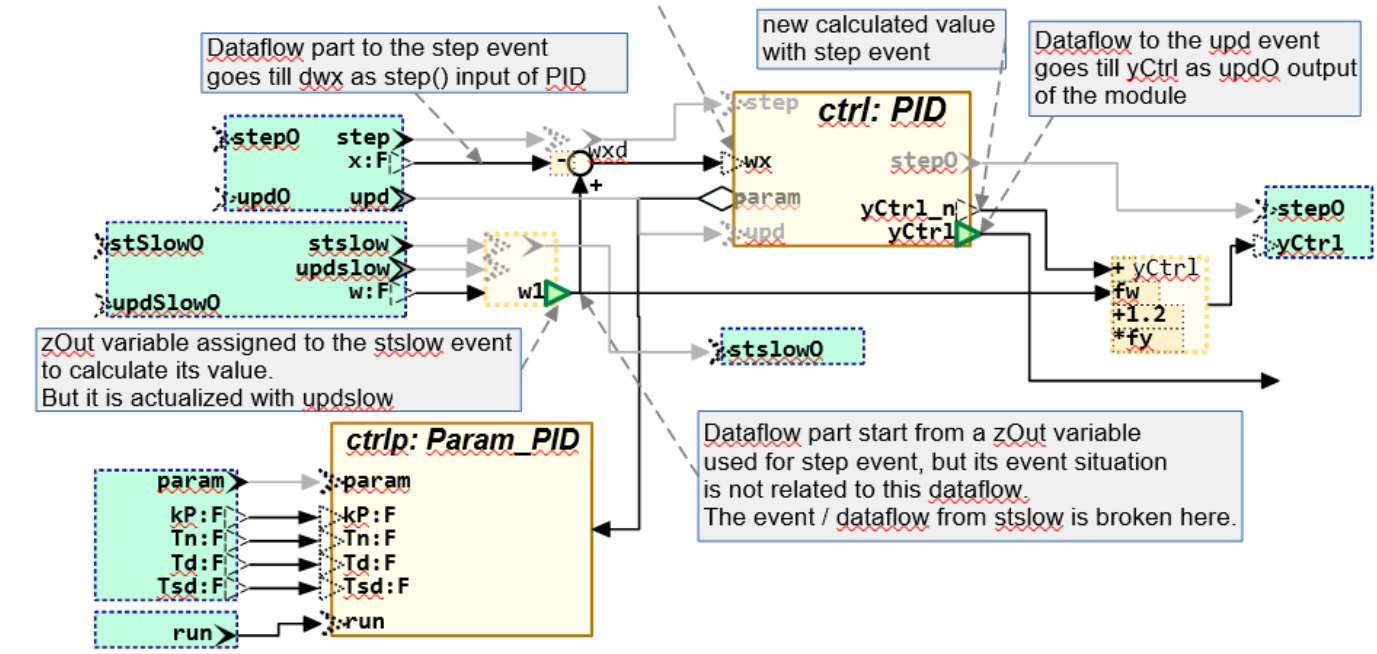


Figure 11: OFB/DataFlowPID4.png

The *Error: Reference source not found* is an example primary as Function Block diagram with a ***data flow***. The ***event flow*** shown in gray is not necessary to be drawn. Here it is only shown in gray what is automatically generated. But the ***event pins*** should be determined as shown (***drawn black***). With the given event pins the data are related to the events, instead to *"sample times"*. Here the `x` ist related to `step`, and the `w` to `stepslow`. The *reference value* `w` comes from another sample time or just with another event. The data flow from `x` to the output `yCtrl` is given, hence `yCtrl` is related to the step ***event chain*** and it is delivered with the `stepO` output event. The value stored in the `w1` variable is a *"state value"* set with the `stepSlow`event and only used, similar as after a *"Rate Transition"* in Simulink.

But this image has also an ***Aggregation*** from the `PID` controller FBlock to its Parameter FBlock. This is ***UML***. In Runtime, the address of the parameter instance is delivered to the `ctrl: PID` one time on initializing the system. It means that is a ***data flow*** from `ctrlp_`

`Param_PID` to `ctrl: PID` revers to the aggregation line.

The green blocks of style `ofbMdlPins` are responsible to determine the module pins from/to outer or just the type of the module. Each `ofbMdlPins` block is responsible to associate event-data relations (as also familiar in IEC61499 diagrams), but additionally the update pin is also associated here:

It means that the input variable `x` is bind to the input event `step`. It presents the `step()` operation (should be called cyclically in the step or sample time). Because the `x` is forwarded by data flow to the `ctrl: PID`, also the event `step` is forwarded. Due to the interface definition of the `PID` type the input `dwx` is associated to the `PID` event input `step`. Hence the data flow `x` → `ctrl.dwx` determines also an event flow from `step` → `ctrl.step`.

The role of "*update*" comes from the mealy and moore automate thinking for logic and it is also familiar in numeric solutions for control: All values are first prepared. Preparation uses always the values from the step time before (or in binary logic preparation of D inputs of Flipflops uses only values of the Q outputs of the clock cycle before). That is the ordinary role of the step event. The update event now realizes the switch of all state values (or clock for Q in Flipflop logic) from the old to the current step to use for the next step. In a sample or step time of a controlling logic first all modules executes the prepare event which is here named `step`. If all parts have been prepared, then the update comes. This assures exactly working for solutions of differential equations and typically for controller theory, it is the Euler principle for numerical integration.

A FBlock can also propagate output values with the prepare event, it depends from the functionality. In Simulink as similar solution an input of an S-Function can be designated as `ssSetInputPortDirectFeedThrough(port,1)` if it influences an output or not (set to 0, default).

In this example shown the output `y.ctrl` is set newly with the `ctrl.upd` event. Hence an event connection between `ctrl.upd` and `upd` of the module accompanies the data flow from `ctrl.y` to the modules `yCtrl` output. The relation between `step`, `stepO`, `upd`, `updO`in the PID FBlock type is clarified by the class definition of `PID`.

Next you see a code snippet of the textual representation of this module in IEC61499, see next chapter:

```
FUNCTION_BLOCK CtrlExample
EVENT_INPUT
  param WITH Td, Tn, Tsd, kP;
  run;
  stslow WITH w;
  ...
END_EVENT
EVENT_OUTPUT
  stepO WITH yCtrl;
  ...
VAR_INPUT
  Td : REAL;
  Tn : REAL;
  ...
VAR_OUTPUT
  yCtrl : REAL;
END_VAR
FBS
  ctrl : PIDf_Ctrl_emC;
  ctrlp : Param_PID;
  w1 : Expr_FBUMLgl( expr:='+;;' );
  wxd : Expr_FBUMLgl( expr:='-+;;' );
  yCtrl : Expr_FBUMLgl( expr:='+; ...
END_FBS
EVENT_CONNECTIONS
  run TO ctrlp.run;
  stslow TO w1.prep;
  updslow TO w1.upd;
  step TO wxd.prep;
END_CONNECTIONS
DATA_CONNECTIONS
  Td TO ctrlp.Td; (*dtype: F *)
  Tn TO ctrlp.Tn; (*dtype: F *)
```

4.8 Storing the textual representation of OFB in IEC61499

It is interesting and promising that the widely proven FBlock programming in the IEC61131 standard for industrial automation control (tools such as Siemens Simatic *FBD in TIA-Portal* or Beckhoff *Codesys*) has been further developed to the IEC61499 standard. This development was started in ~2006, Also Siemens was one of the driver in that time. The IEC61131 is used since many years for automation programming. The IEC61499 is standardized and used, but not from the global meaningful players, they only monitors this development. The reason (in my mind and experience) is not disadvantages of IEC61499, it is more a too widely usage, supporting and maintenance of the long term existing IEC61131.

The IEC61499 has introduced an event coupling between function blocks. This determines the stepping order better than the ordinary net lists in IEC61131, but it allows also to distribute the implementation of one Function Block Diagram over several automation stations. Event connections between distant stations forces automatically network communication implementation and assures the correct order of execution in the dispersed station, without additional effort. That's the advantage for automation programming. But the more universal character of event coupling inclusively state machine thinking can also basically used for embedded control programming.

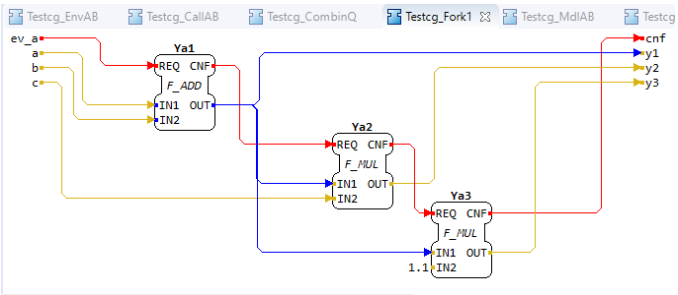


Figure 12: 4diac/Testcg_Fork1.png

A chain of events in the same implementation platform (same thread in a CPU) defines a statement order. Different event chains are related to operations, which can be called either cyclically (for step time driven thinks) of also from the state behavior or independent for example on user accesses.

But the drawing of the event connections in a IEC61499 diagram is an additional effort. The image shows an example with event coupling for simple data relations with the graphical edition tool 4diac. In most cases an event flow (chain) is also determined by the data flow. Evaluation of the data flow results in an event connection, which should not be drawn manually. It is automatically detected during the evaluation of the graphic, and stored in the data model. Only if dedicated event relations are necessary, the events should be drawn in graphic.

The IEC61499 standard is used to store the content of UFBgl diagrams in textual form. This allows also a proper comparability if details in the diagrams are changed. That is a high importance to use this tooling in the development of software, a proper traceability of changes is necessary. With pure graphics, this is often not properly supported, one of the reasons for the still widespread use of textual programming.

It is also possible to read this stored IEC61499 textual files for processing for sub modules, and for code generations, as well as reading IEC61499 fbd files from other tools to merge here.

4.9 Source code generation from the graphic

As is usual with some FBlock graphics, code generation from the graphic is a prerequisite for being able to work productively with it. This chapter should only give an overview. Refer for more opportunities in chapter ToDO

The evaluation of the graphic is done with a Java command line process as (shortened)

```
java -cp tools/vishiaBase.jar;
 … tools/vishiaFBcL.jar
 … org.vishia.fbcl.Ufbconv
 … -dirGenSrc:src/UFBglExmpl/cpp/genSrc
 … src/UFBglExmpl/odg/OrthBandpassFilter.odg
```

This reads the graphic, writes anyway a IEC61499 fbd file, and writes here C-language header and implementing code.

The graphic is shown (as part, one page) in *Error: Reference source not found*. The generated code looks like (shortened)

```
/**Generated by org.vishia.fbcl.
made by ...
#ifndef HGUARD_OrthBandpassFilter
#define HGUARD_OrthBandpassFilter
#include <emC/Ctrl/OrthBandpass_Ctrl_emC.h>

typedef struct OrthBandpassFilter_T {
  struct { // Locale struct for all din
    float x; // OrthBandpassFilter.x
    float x2;
    float fq;
  } din;

  struct { // Locale struct for all dout
    bool initOk;
    ...
  } dout;

  float_complex xdab; // Expression xdab

  OrthBandpassF_Ctrl_emC_s h1; // h1
  Param_OrthBandpassF_Ctrl_emC_s h1p; // h1p
  OrthBandpassF_Ctrl_emC_s h2; // h2
  ...

} OrthBandpassFilter_s;


void step_OrthBandpassFilter ( );

void upd_OrthBandpassFilter ( );
...
#endif
```

The implementation file is generated as:

```
/**Operation step(...)
 */
```

```
void step_OrthBandpassFilter
( OrthBandpassFilter_s* thiz
                              , float x, float
x2 ) {
  // --> x1.prep otx:evChainExprSetvar
  float_complex x1;
  x1.re = x; // Y D otx:evChainExprSetvar
  x1.im = 0; // Y D otx:evChainExprSetvar
  ...
  thiz->xdab.re = ( x1.re - ( thiz->h1.ya ...
  thiz->xdab.im = ( x1.im - ( thiz-
>h1.yabz.im
   + thiz->h3.yabz.im));
  step_OrthBandpassF_Ctrl_emC(&thiz->h1,
                              thiz->xdab);
  ...
```

There are some stuff which is regarded beside the event flow and hence the execution order. The types of all elements are forward and backward propagated. For the here used complex data types the operations are duplicated respectively specific functions are created, and so on.

The code generation is controlled by textual template files using the java class OutTextPreparer, see

Any user can proved its own templates for code generation, can copy the originals and modify, or can write its own template for other languages or only specific style guides. For pure C language an object oriented style is used of course to represent the instances of classes. classes are presented by struct { } with its associated operations with a thiz reference to the own struct. This can be encapsulated also by C++.

4.10 Run and Test and Versioning

Only yet minutes:

- Compilation in a PC platform (Visual Studio, Eclipse CDT, ...
- Environment for running in C/++ as given (familiar for C development)
- Physical simulations cannot be done, maybe as future development.
- But coupling with another Simulation tool for physics is very recommended, use your own tool. Can bei Simulink, Modelica, or what ever.
- The coupling should be always possible with shared memory on the same PC. For Simulink such an SharedMem Sfunction block, configurable due to a header file on the counterpart, is existing since ~2021, aks me. Should be documented also here.

Versioning:

- Store the odg graphic
- Store the IEC61499 textual representation for compare which changes.
- Store the generated sources in the target language “Secondary Sources”.

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. The advantage of that is: The content of the graphic is comparable between progress of development (versions). Whereby not the graphic appearance is in focus (better seen in original graphic), but the content for functionality and code generation.

To have an overview look on the following image:

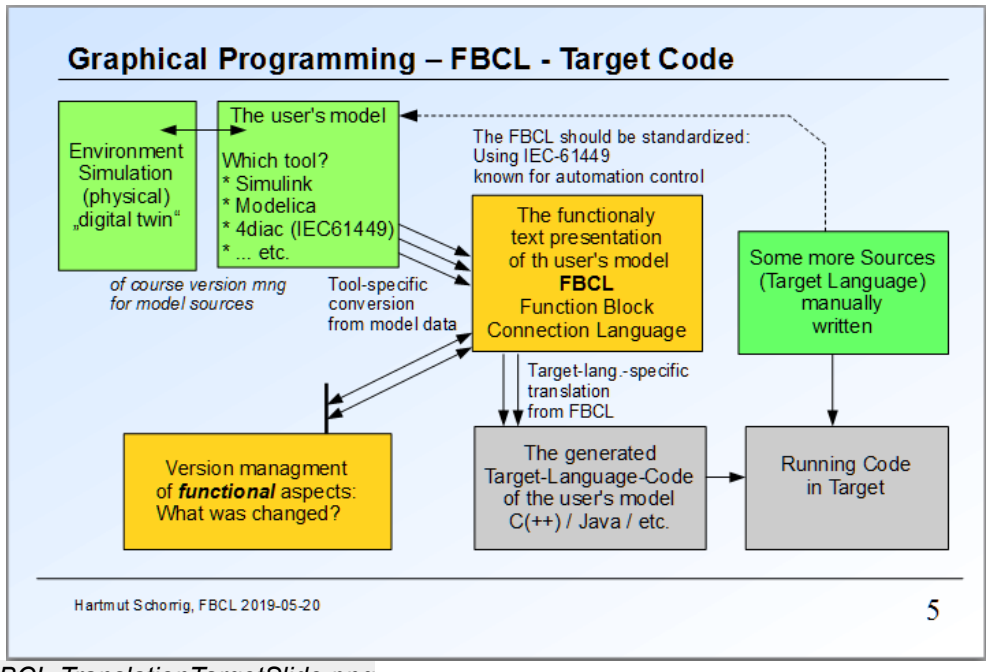


Figure 13: Fbcl/FBCL-TranslationTargetSlide.png

This is an older image from 2019, but it shows the whole truth. The so named FBCL (Function Block connection language) is here shown as textual representation of the graphic, whereby here the usage of Open/LibreOffice for the graphic was not yet present. But the using of IEC61499 was already found as coding standard for the textual graphic representation.

This figure shows also the topics of simulation of the functionality shown in the graphic, also

including usage of manual written (core) sources in the target language.

5 Handling with OFB diagrams and LibreOffice draw

Table of Contents

5 Handling with OFB diagrams and LibreOffice draw....24
5.1 All Kind of Elements with there style....28
5.2 All styles....30
5.2.1 GBlock styles, ofb....30
5.2.2 Name styles, ofn....31
5.2.3 Connector styles, ofc....32
5.2.4 Pin styles, ofp....34
5.3 Texts in graphic blocks and pins....36
5.3.1 Syntax in colored ZBNF....36
5.3.2 The complete Syntax of texts for pins and FBlocks....37
5.3.3 Syntax of input to a pin....38
5.3.4 Examples for description and type....39
5.3.5 What contains descr, for expressions and pin designation for FBlocks....39
5.3.6 type and sizeArrayType....40
5.3.7 nrGpos, order of pins after grave....41
5.4 Data types....42
5.4.1 One letter for the base type....42
5.4.2 Unspecified types....44
5.4.3 Array data type specification....44
5.4.4 Container type specification....45
5.4.5 Structured type on data flow....46
5.4.6 Data type forward and backward test and propagation....47
5.4.7 Using a module with non deterministic data types....48
5.4.8 Integer Data types and their scaling and decimal point....51
5.5 One Module, Inputs and Outputs, file and page layout....52
5.5.1 Module in odg file(s) organized in pages....52
5.5.2 Alias control and import....52
5.5.3 Module pins....53
5.5.4 Order of pins....54
5.5.5 The module's input....56
5.5.5.1 call by value....56
5.5.5.2 call by reference....56
5.5.5.3 set input variables....57
5.5.6 The module's output....58
5.5.6.1 Using public variable for the output....58
5.5.6.2 Access inner variable of the module for output....58
5.5.6.3 Operation for outputs access 'getter'....60
5.5.6.4 Event operations with return value and / or output variable by reference....62
5.5.6.5 Return a reference or variable by double reference....63
5.6 Possibilities of Graphic Blocks (GBlock)....64
5.6.1 Difference between class, type and instance ("Object")....64
5.6.2 GBlocks for each one function, data – event association....66
5.6.3 Aggregations are corresponding to ctor or init events....67
5.6.4 Predefined FBlocks or definition on demand, relation with source code....68
5.6.5 Possibility of inputs of FBlocks....70

5.6.5.1 Inputs as local arguments of the event operation ofpDin....70
*5.6.5.2 Call by value or call by reference ofpDin& *....70*
5.6.5.3 Instance variable for inputs ofpVin....70
*5.6.5.4 Instance variables as reference ofpVin& *....71*
5.6.6 Possibilities of outputs of FBlocks....72
*5.6.6.1 Reference and return output ofpDout() & *....72*
5.6.6.2 Instance variable with public access ofpVout....72
5.6.6.3 Output access via operation ofpDout()....73
5.6.6.4 Operation access returns the value or the reference ofpDout()....73*
5.6.6.5 Access Zout values ofpZout....73
5.6.7 Expression GBlocks....74
5.6.8 GBlocks for operation access in line in an expression - FBoper....74
5.6.9 Conditional execution with boolean FBexpr....76
5.6.10 Data flow event related – or persistent data....78
5.6.11 Sliced or Array FBlocks, Demux and array data....80
5.7 Connection possibilities....82
5.7.1 Pins....82
5.7.2 name : Type on pins....86
5.7.3 Connectors....86
5.7.4 Connection points....87
5.7.5 Xref....87
5.7.6 Using GBmux and GBdemux for connections....88
5.7.7 Connections from instance variables and twice shown FBlocks....88
5.7.8 Textual given connections....88
5.7.9 Admissibility check of connections....89
5.7.10 Data type test and conversion on inputs....89
5.7.11 The direction of references and the data flow....90
5.7.12 More outputs to one input....90
5.8 Expressions inside the data flow (FBexpr)....92
5.8.1 Expression as rectangle and input pins as rectangle ofpExprPart....92
5.8.2 More possibilities of DinExpr....94
5.8.2.1 Operation on expression input: factors in Add expression, variables....94
5.8.2.2 Access to elements of the input connection to use....95
5.8.2.3 Description of all possibility, syntax/semantic of DinExpr....95
5.8.2.4 Some examples for DinExpr....99
5.8.3 Data Type specification and value casting in expressions....100
5.8.4 Data types with fractional bits in expressions , using saturation....102
5.8.4.1 Example - How is it done in pure C programming....102
5.8.4.2 Same Example graphical....103
5.8.4.3 Why saturation or limitation is neccessary....104
5.8.4.4 Limit or saturation input(s)....105
5.8.4.5 Condition on overflow....106
5.8.5 Any expression in FBexpr....107
5.8.6 Output possibilities, variable after expression....108
5.8.7 Set elements to a array of structure variable....109
5.8.8 Output with ofpExprOut....110
5.8.9 FBexpr as data set....110
5.8.10 FBoper, operation for a FBlock....111
5.8.11 How are expressions presented in IEC61499?....112
5.8.12 FBexpr capabilities compared to other FBlock graphic tools....114

5.9 Operations to FBlocks inside the data flow (FBoperation)....116
5.9.1 void Operation with input(s) and reference output....116
5.9.2 What is stored in the IEC61499 FBcl.fbd file:....117
5.9.3 Operation with return value and reference outputs....118
5.9.4 Join_OFB for inputs for calculation order....119
5.9.5 A FBoperation as simple getter....119
5.10 FBlocks in slices, access to slices....120
5.10.1 Vectors in expression....120
5.10.2 Vectors and scalar FBlocks....121
5.10.3 Slices of named FBlocks....122
5.10.4 Mux and Demux, build vectors with Mux....123
5.10.5 Build vectors with elements, access to vector elements....123
5.11 Execution order, Event and Data flow, Event chains and states....124
5.11.1 Event and Data flow....124
5.11.2 Event chains for each one operation, state variables....127
5.12 Drawing and Source code generation rules....128
5.12.1 Writing rules in target language used from generated code from OFB....128
5.12.2 Life cycle of programs in embedded control: ctor, init, step and update....129
5.12.3 Using events in the module pins and FBlocks, meaning in C/++....130
5.12.4 More possibilities, definition of special events....132
5.13 Showing processes....134
5.14 Converting the graphic – source code generation....136
5.14.1 Calling conversion with code generation....137
5.14.2 Handling of include in C/++ or import and real used type names....140
5.14.3 Error messages while translating....140
5.14.4 Templates for code generation....141
5.15 Presentation of the graphic and results in files....142
5.15.1 The original odg format (Overview)....142
5.15.2 Graphic saved with the option The original odg format (Overview)....142
5.15.3 The FBcl format or IEC61499, file.fbd....144
5.15.4 The original odg format (Overview)....146

empty

5.1 All Kind of Elements with there style

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

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

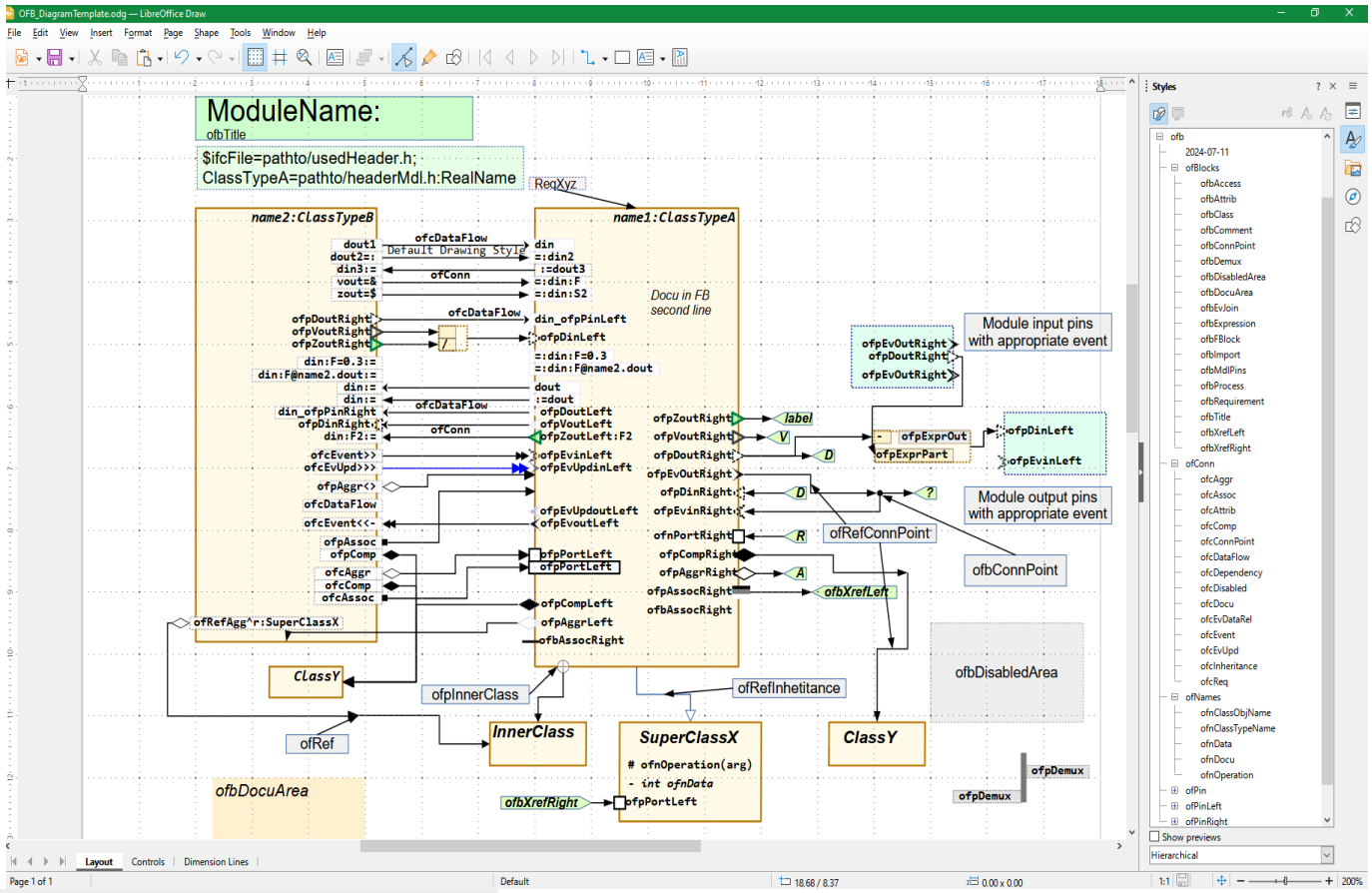


Figure 14: odg/OFB_DiagramTemplate.png

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.

If you use first time this OFB concept, copy the template file in your working space saved as `yourName.odg`. Then delete all content. The styles remain, they are important. Alternatively use an example from the download, it should contain the same styles.

Then open the `OFB_DiagramsTemplate.odg` as second LibreOffice draw Window

You can use this drawing content in `OFB_DiagramsTemplate.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.

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

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.

The class in the mid with `name: ClassTypeA` contains all connection elements for the concept described in *3.2 Using figures with styles (indirect formatted) for element* page *7*. 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 `ofPinRight` and `ofPinLeft`, 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 styles* page *30*

5.2 All styles

5.2.1 GBlock styles, ofb

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 *5.5.1 Module in odg file(s) organized in pages* page *52*

- `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 *5.5.1 Module in odg file(s) organized in pages* page *52*

- `ofbMdlPins`: This is a shape which contains the pins of the module, see *5.5.1 Module in odg file(s) organized in pages* page *52*

- `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 *64*

- `ofbExpression`: This is an expression FBlock or also named *“FBexpr”*, see *5.8 Expressions inside the data flow (FBexpr)* page *92*

- `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 *5.11 Execution order, Event and Data flow, Event chains and states* page *124*

- `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 *82*

- `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.

- `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 *5.13 Showing processes* page *134*

- `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.

5.2.2 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 *64*

- **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).

5.2.3 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 *66*

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

5.2.4 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.

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 *82*.

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 `ofpAggr…` Aggregation

`<_>name<_>` same as `ofpAggr…` 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 *5.8.1 Expression as rectangle and input pins as rectangle ofpExprPart* page *92*.

- `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 *92*

.- `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 Texts 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 *5.2.4 Pin styles, ofp* page *34*.

See also html (www) / Impl-OFB_VishiaDiagrams.pdf (www): *7.3.3.4 Evaluating Pin texts* page *25*

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 texts 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.

`*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 More possibilities of DinExpr* page *94*. 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@?stepO`2` This is a typical example for the aggregation pin of an operation FBlock, see also *5.8.10 FBoper, operation for a FBlock* page *111*. `fb2@` is the aggregated FBlock, `stepO` is a ***special*** designation, used for the event. After them also the ***nrGpos*** is given.

`fb2@=:?stepO`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 **ofbImport** shape, see 5.5.2 Import and alias control page 52. This enable the opportunity to use an 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 Examples for description and type

The chapter *5.3.1 Syntax in colored ZBNF* page *36* 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 More possibilities of DinExpr* page *94*.

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 More possibilities of DinExpr* page *94*.

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

5.3.5 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 *36* 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 More possibilities of DinExpr* page *94*.

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 The module's output* page *58* and *5.6.6 Possibilities of outputs of FBlocks* page *72* 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 *5.6.6.1 Reference and return output ofpDout() & ** page *72*
- `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(`.

5.3.6 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 *42* 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 *44*. 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

```
+
```

5.3.7 *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

"`: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

5.4 Data types....42
5.4.1 One letter for the base type....42
5.4.2 Unspecified types....44
5.4.3 Array data type specification....44
5.4.4 Container type specification....45
5.4.5 Structured type on data flow....46
5.4.6 Data type forward and backward test and propagation....47
5.4.7 Using a module with non deterministic data types....48
5.4.8 Integer Data types and their scaling and decimal point....51

In the *Figure 10: OrthBandpassFilter.odg.png* 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 (hour) or the question which hour. 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 *47*) 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:

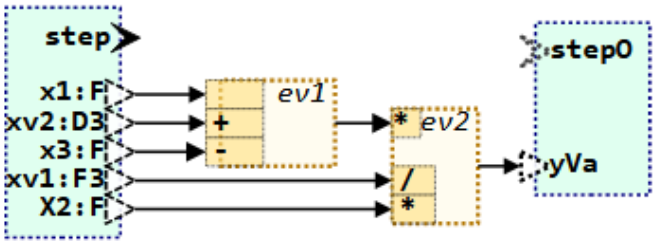


Figure 15: OFB/ExprVect-A.png

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:

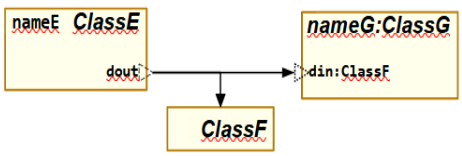


Figure 16: OFB/DflowStructData1.png

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:

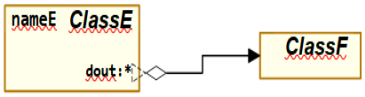


Figure 17: OFB/DflowStructData1.png

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

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.

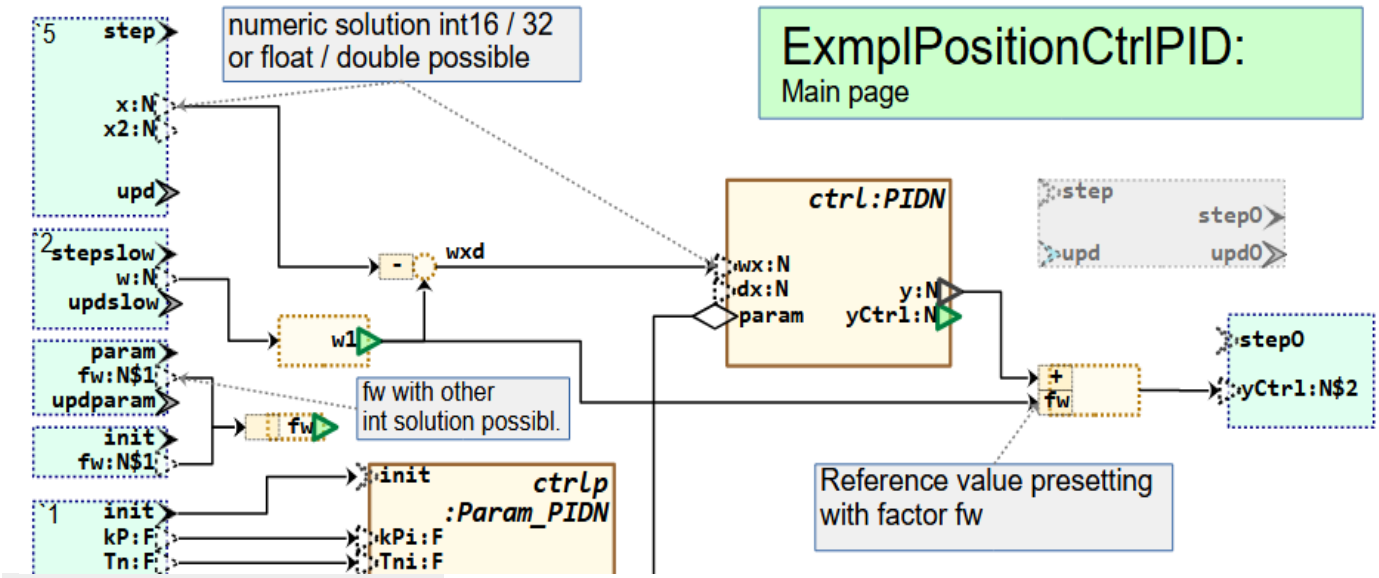


Figure 18: odg/PositionCtrlPID_1.png

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 19: 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:

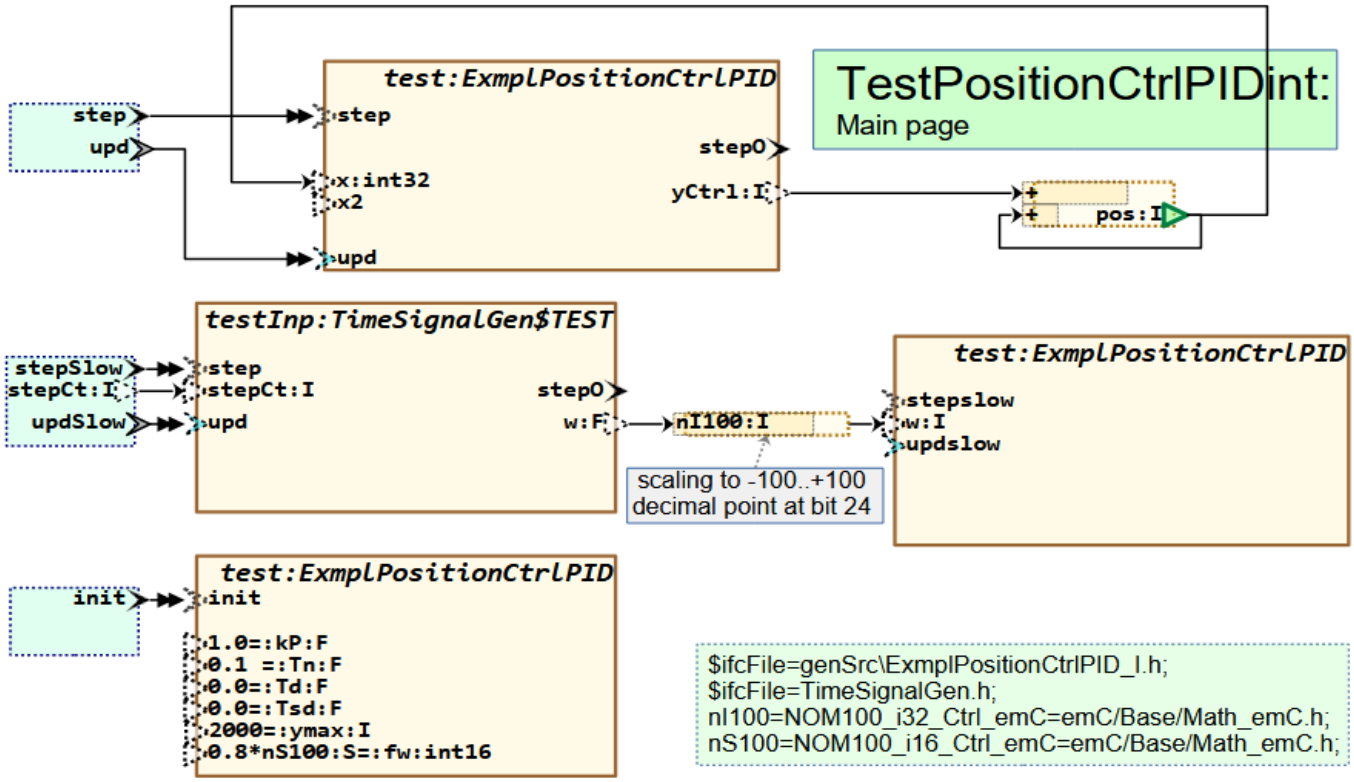


Figure 20: odg/PositionCtrlPID_Test_int32

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 `$mdlType=ExmplPositionCtrlPID_%x$step%` 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:

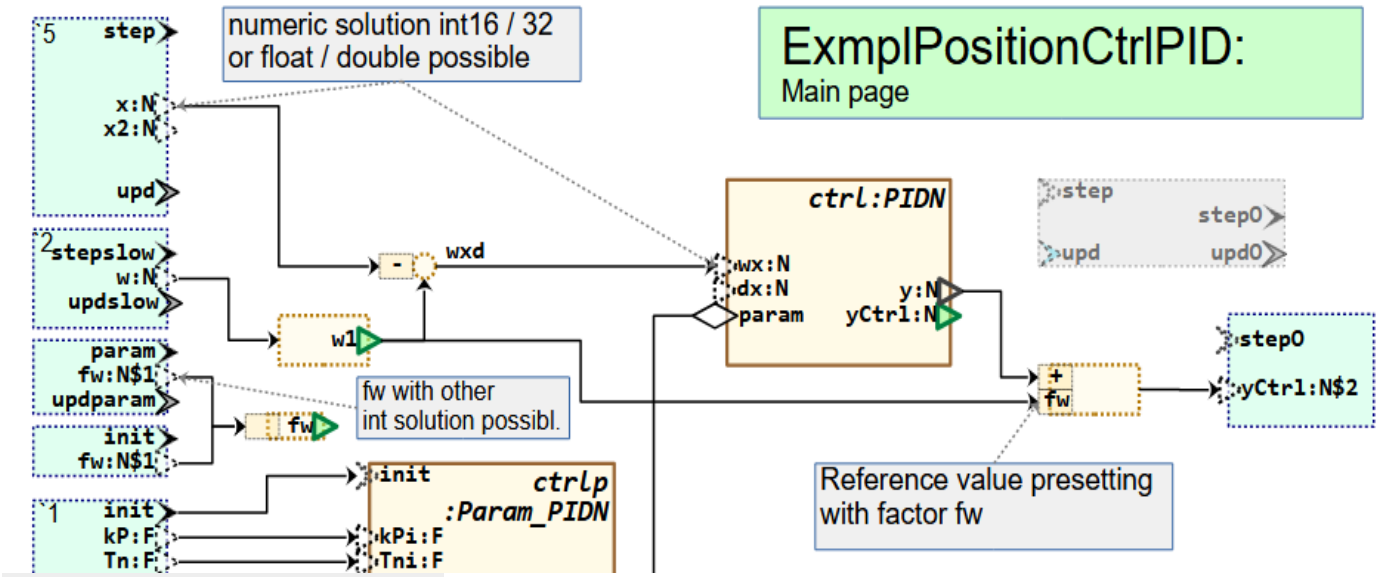


Figure 21: odg/PositionCtrlPID_1.png

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 *47* 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 *5.3.6 type and sizeArrayType* page *40*. 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 One Module, Inputs and Outputs, file and page layout

Table of Contents

5.5 One Module, Inputs and Outputs, file and page layout....52
5.5.1 Module in odg file(s) organized in pages....52
5.5.2 Alias control and import....52
5.5.3 Module pins....53
5.5.4 Order of pins....54
5.5.5 The module's input....56
5.5.6 The module's output....58

5.5.1 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`:

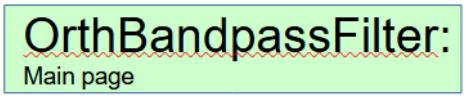


Figure 22: og/ofbTitle-1.png

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.

5.5.2 Alias control and import

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

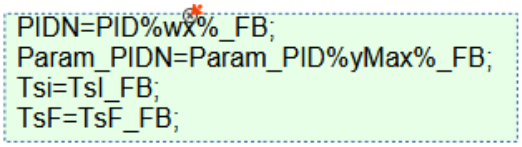


Figure 23: odg/ofbAlias.png

This example is related to the description of 5.4.7 Using a module with non deterministic data types page 46 and has here the second and important meaning to assign the simple name of a FBtype to a name build with one of its data type char.

But the simple meaning is: Using an short identification, an alias for a FBtype and association the full qualified name. Here the alias `Tsi` is `TsI_FB`. But note, this may not be the used name in the generated code (it would be `T1i_Ctrl_emC` for C language)

This shape of `ofbAlias` box can contain:

```
$mdlType=...;
```

an divergent name of the module for code generation.

```
$cfgAliasImport=...;
```

Path to a file which is used local only for this module.

describes files as interface, to import, for C/++ code generation header files to include.

```
alias = used_name;
```

associates an alias name, used in the graphic, to the used name for code generation.

```
alias = constant_value;
```

Also usable for constant values.

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

For the import and determining the real used name of types there is a second mechanism. Either module specific after the `$cfgAliasImport=` or also as argument `-cfg` for the translator (see *5.14.1 Calling conversion with code generation* page *137*):

A textual file controls the real used names in code generation for the FBtype names after this alias translation, and also the associated header files for C/++ target code, or other import files for other languages. The detail description for that is contained in *5.14.2 Handling of include in C/++ or import and real used type names* page *140*.

5.5.3 Module pins

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

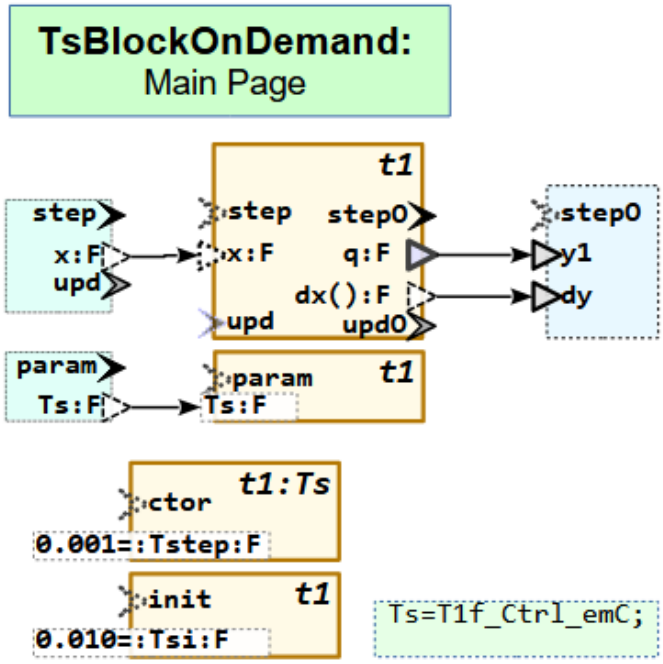


Figure 24: OFB/TsBlockOnDemand.png

The image shows a complete simple example module. Left their are two `ofbMdlInp` ***GBlocks*** for the `step` and `upd` event (operation) with `x` as data, and for the `param` event or operation with `Ts` as input. Right side there is only one output GBlock with `stepO` as output event and the both shown data.

Each GBlock for `ofbMdlInp` and `ofbMdlOut` should contain only one event or the prep or step event and the associated update event, style `ofpEvin...` and `ofpEvUpdin...` for the `ofbMdlInp` and `ofpEvout` and the related `ofpEvUpdout` for the `ofbMdlOut`.

If `ofp...in` pins are used in an `ofbMdlOut` GBlock, they are accepted also, but as `ofp...out` pins, and vice versa. The reason for that is, intrinsically a module input is an output for the inner data flow, and vice versa. But using the `ofp...out` pins for `ofbMdlOut` is at least necessary for `ofpZout`.

With the association of data to events the data are associated to this event, or in other words, it builds the arguments to the event operation in the order given from top to down. Whereby, data to update events does not exists, the data are associated to the prepare event (`ofpEvin… )`

The given `ofpEvupdin...` event is associated for the update operation proper to the prepare operation with the `ofpEvin...`. It means, the module has one operation

```
step_TsBlockOnDemand(…, float x);
```

and one operation

```
upd_TsBlockOnDemand(…);
```

without data arguments. For the concept of prepare and update see chapter *5.12.2 Life cycle of programs in embedded control: ctor, init, step and update* page . The association of the prepare event (here `step`) with 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 operation.

The relation between the prepare (step) `ofpEvin` and the related `ofpEvout` in the `ofbMdlOut` GBlock is clarified by the data flow, or by given event connections in the graphic. In the shown simple example without state machine, only with the data flow it is clarified that the `stepO` follows the `step` after calculation. That presumes, that also for the one FBlock the data flow is associated to `step ->> stepO`. Because this module has no `ofpZout` pins on its `ofbMdlOut`, it has also no `ofpEvUpdout` pin. This may be necessary if `ofpZout` data should be

calculate by combinatorics in the update phase, but such calculations should be only simple.

Also the `param` event has no counterpart as `ofpEvout`. If parameterize is done, it does not need any more action.

But sometimes a `updateParam` event may be interesting, because with the update the a new parameter set can be get valid, and this should be done in timing coordination with a maybe fast step operations (in its update operation). - But this are implementing or architecture details of usage.

Also, this module has no GBlock for the ctor which may also an extra ofpEvin. But because the module has FBlocks with ctor, as shown, the ctor evin (means a ctor operation) is created without necessary GBlock for that. The same is with init. An ofpEvin for init is then necessary, if init has associated data.

See also the following chapter *Error: Reference source not found* page *Error: Reference source not found* for further possibilities.

5.5.4 Order of 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.

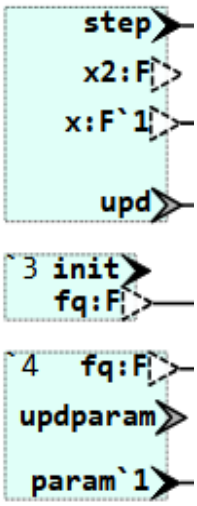
b) Determine the pin order in the first used GBlock by a sort number, named `nrGpos`. This number is written as last information in the pins text as `` `123 `` The character before the number is the *'grave'* (hexa 0x60), usual able to found on a US keyboard left top, on a German keyboard right top.

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 25: 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.

5.5.5 The module's input

5.5.5.1 call by value

The simplest form for inputs are using the symbol `ofpDin…` in a module's input GBlock with style `ofbMdlInp`. as shown in Figure 25: odg/ofbMdlPins-2.png on the page before, or some other images. This pins are translated to formal arguments for the operation and 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 also a complex numeric type is a structured type. See following image:

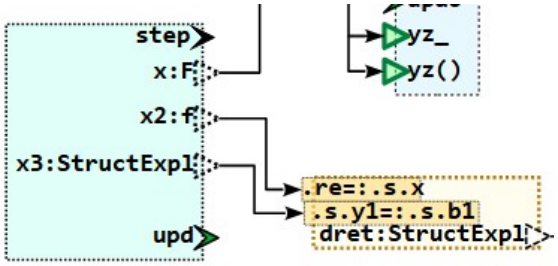


Figure 26: BasicTest/ModuleInoutDef_Inp.png

The generated code for the forward declaration of the step operation of bf 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 call on scalar inputs an expression can be connected, but the structured data type inputs needs a variable, can be a variable after expression. This original example delivers:

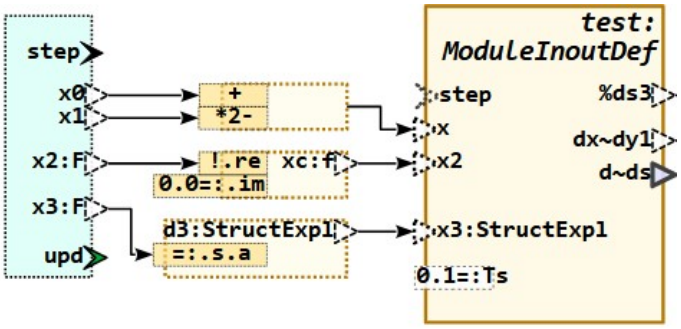


Figure 27: BasicTest/ModuleInoutUse_Inp.png

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. The here stack local variables are delivered as value, which forces an internal `memcpy` as usual for call by value.

5.5.5.2 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 slang. 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).

5.5.5.3 set input variables

A simple set of global variables in the data of the called FBlock is 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.

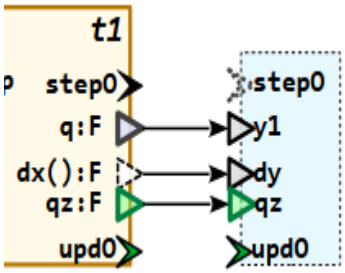
This feature, using `ofpVin` pins, is just not implemented (2025-07), but possible to implement if necessary.

5.5.6 The module's output

5.5.6.1 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 28: 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:

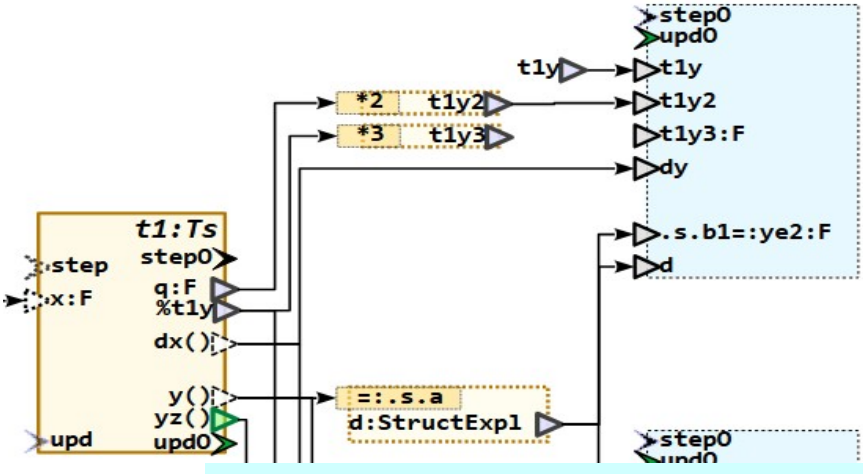
```
  .... + test->y1 + ...
```

5.5.6.2 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 *5.8.6 Output possibilities, variable after expression* page *108*) or also as variable as call by reference or return destination of FBlocks (see *5.6.6.1 Reference and return output ofpDout() & ** page *72*). Also, inner variables can be better designed as (see *5.4.5 Structured type on data flow* page *46*).

Figure 29: 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 *5.6.6.1 Reference and return output ofpDout() & ** page *72* 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.

5.5.6.3 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.

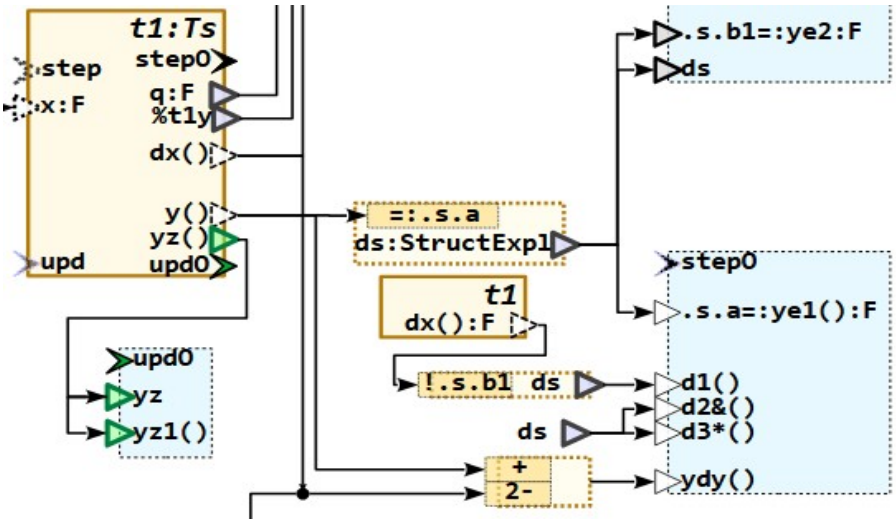
Another important advantage is: An operation may contain not only a simple access to elsewhere public data, it can be also the execution of a more complex expression, maybe also executing a longer expression in hard coded target language. For example a `atan2()` operation is a little bit longer, important 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 30: 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 `stepO` 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 slang. 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)

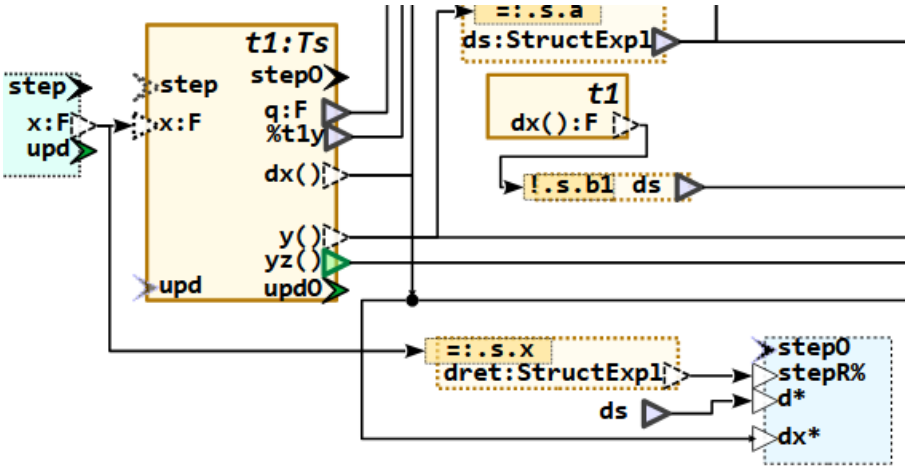
```
void upd_ModuleInoutDef ( ModuleIno ...
  upd_T1f_Ctrl_emC(&thiz->t1);
  //
  // Module outputs due to the event updO:
  thiz->mEvout_upd |= MASK_upd_updO;
  thiz->yz_ = yz_T1f_Ctrl_emC(&thiz->t1);
```

But follow also the next chapter

5.5.6.4 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 31: 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:

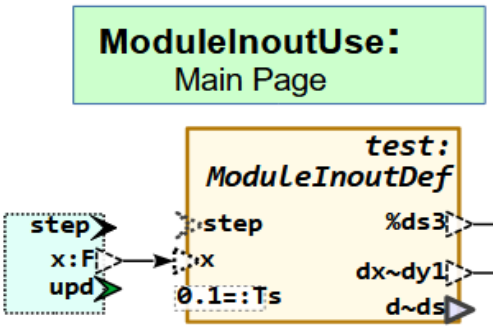
```
StructExpl_BasicTest_s step_ModuleInoutDef ( ModuleInoutDef_s* thiz
  , float x
  , StructExpl_BasicTest_s* d  // output per reference in otx: EventOperBody
  , float* dx  // output per reference in otx: EventOperBody
  ) { // ##oper_step

  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 32: 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 *5.6.6 Possibilities of outputs of FBlocks* page *72*. 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 *68*.



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.

5.5.6.5 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.

5.6 Possibilities of Graphic Blocks (GBlock)

This chapter should show all possibilities for Function block shapes (FBlocks).

Table of Contents

5.6 Possibilities of Graphic Blocks (GBlock)....64
5.6.1 Difference between class, type and instance (“Object”)....64
5.6.2 GBlocks for each one function, data – event association....66
5.6.3 Aggregations are corresponding to ctor or init events....67
5.6.4 Predefined FBlocks or definition on demand, relation with source code....68
5.6.5 Possibility of inputs of FBlocks....70
5.6.6 Possibilities of outputs of FBlocks....72
5.6.7 Expression GBlocks....74
5.6.8 GBlocks for operation access in line in an expression - FBoper....74
5.6.9 Conditional execution with boolean FBexpr....76
5.6.10 Data flow event related – or persistent data....78
5.6.11 Sliced or Array FBlocks, Demux and array data....80

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 *4.3 Show same FBlocks multiple times in different perspective* page *14*

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.

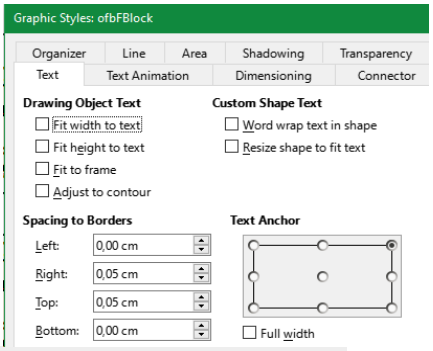


Figure 33: odg/ofbFBlock-TextStyle.png

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.

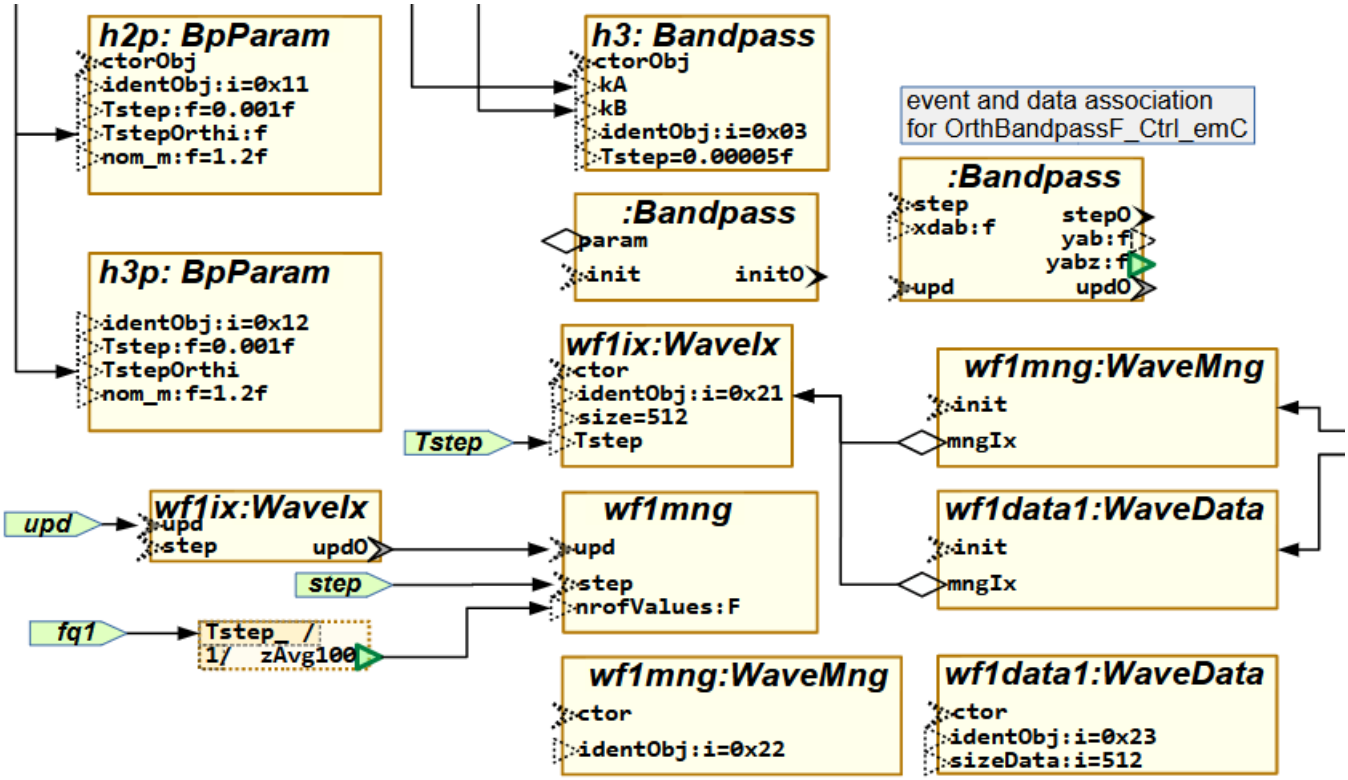


Figure 34: odg(ExmplFBlocksTypes.png

--

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 *5.12.2 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,

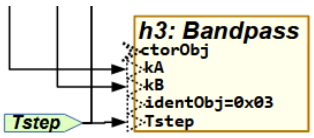


Figure 35: odg/FBlock_ctorObj.png

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 36: 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 `stepO`, `upd` and `updO` 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 corresponding 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 37: 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 *5.12.2 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:

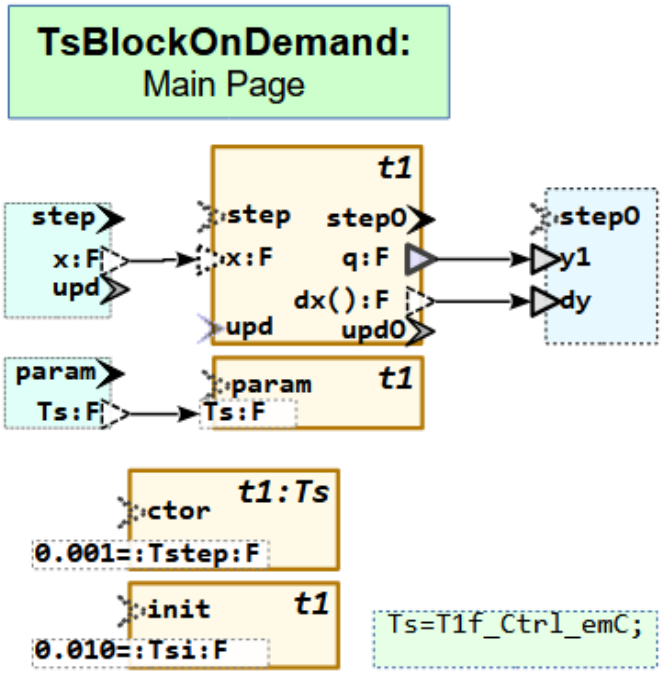


Figure 38: odg/TsBlockOnDemand.png

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 *5.5.2 Alias control and import* page *52*.

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.

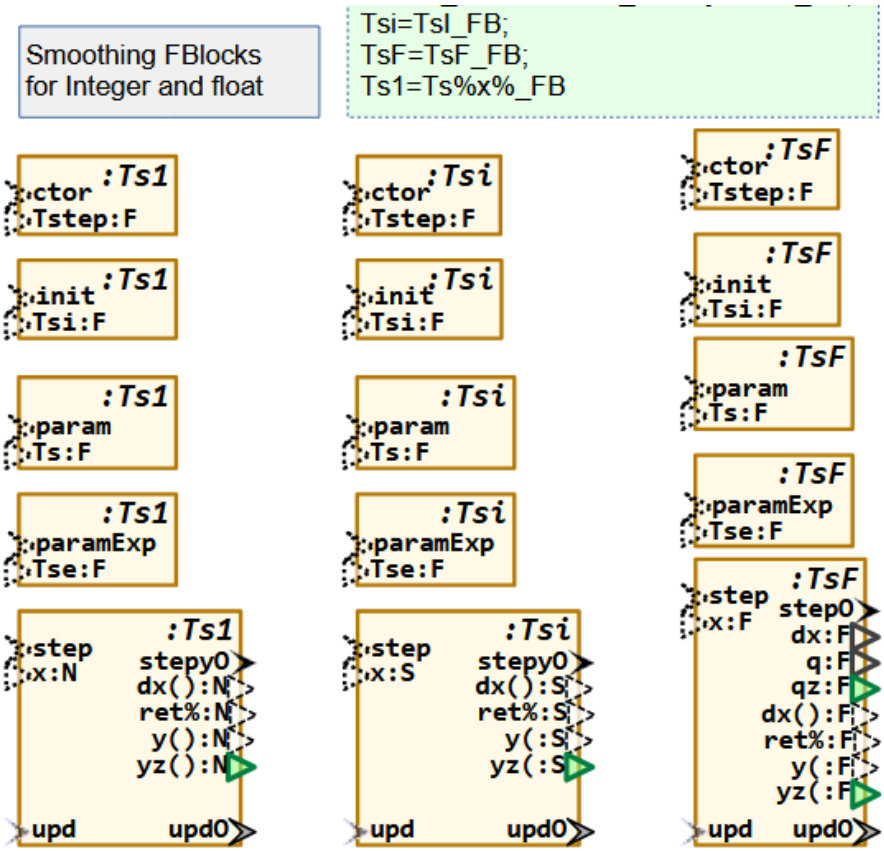
It is better to have prototype definitions or just predefined FBlocks. A module diagram with this predefined FBlocks describes only and exact the interface to the given legacy code, or just the interface to another translated module.

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 39: 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 40: 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 Possibility of inputs of FBlocks

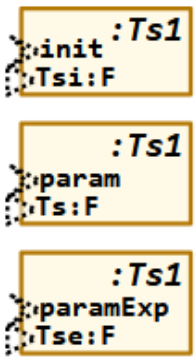
5.6.5.1 Inputs as local arguments of the event operation ofpDin

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 41: 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.

*5.6.5.2 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

5.6.5.3 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.

*5.6.5.4 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

5.6.6 Possibilities of outputs of FBlocks

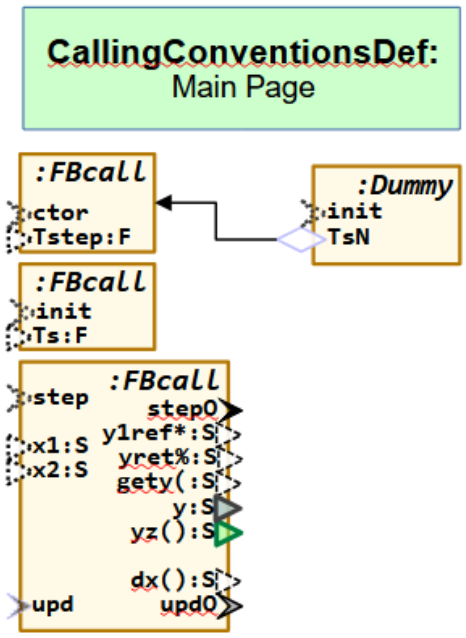


Figure 42: BasicTest/CallingconventionsDef.png

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.

*5.6.6.1 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:

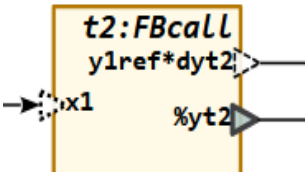
```
int16 step_FBcall_... (FBcall_..._s* thiz
, int16 x1, int16 x2, int16* dx);
```

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 43: BasicTest/Calling ConventionsUse _DoutRefRet.png



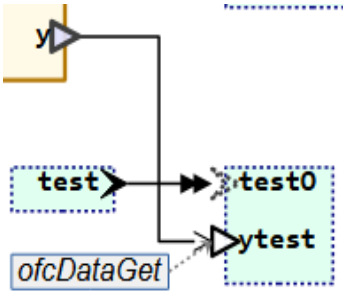
The kind of the Dout in the using GBlock determines the kind of the variable. Here `ofpDoutRight` is used for `y1ref`. it builds a local (stack-) variable, and for the return variable a `ofpVoutRight` is used which builds an instance variable.

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`. jdoc:org.vishia.fbcl.readOdg.OdgNameTypeArray#OdgNameTypeArray(,,,)

5.6.6.2 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 44: 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.

5.6.6.3 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).

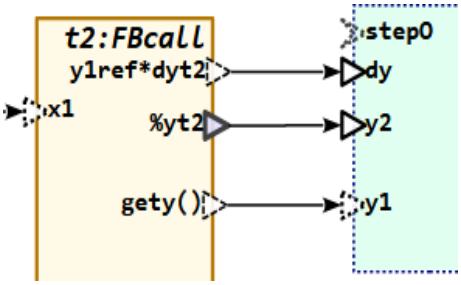


Figure 45: BasicTest/CallingConventionsUse_DoutOper.png

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);
}
```

5.6.6.4 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

5.6.6.5 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:

5.6.7 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.

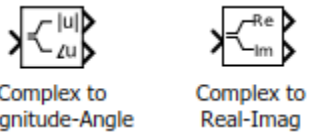


Figure 46: Simulink standard library blocks SmlikLibCplxMagnAngle_CplxReIm.png

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.

5.6.8 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):

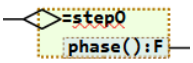


Figure 47: odg/FBoperGetter.png

The right figure shows a simple getter possible as part of an expression. The aggregation refers the proper FBlock, see also . The `=stepO` means, that the operation (getter) can be called only after the `stepO` 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 *4.3 Show same FBlocks multiple times in different perspective*), it is really each an operation call for each graphic presentation.

That fact is more able to explain with the following example:

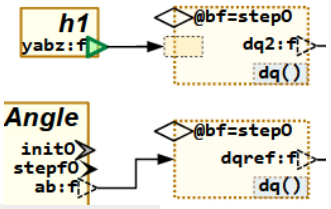


Figure 48: odg/FBoperInOut

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 *82*).

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, `=stepO` on the aggregation means, that the FBoper can be executed only after valid `stepO`, 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.stepO` 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.stepO` is connected to the FBoper. This can be seen in the produced fbd file, for this example:

```
EVENT_CONNECTIONS
bf.stepO TO dq2_X.prep;
bf.stepO TO JOIN_dqref_X_prep.J1;
gref.stepfO TO JOIN_dqref_X_prep.J2;
JOIN_dqref_X_prep.J TO dqref_X.prep;
```

5.6.9 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.

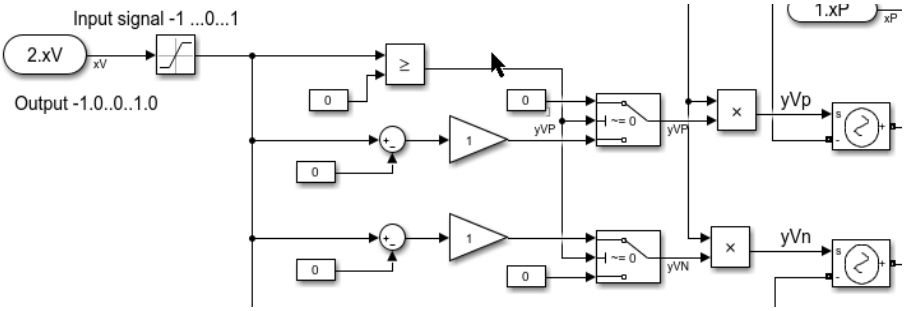


Figure 49: smlk/Exmp_if_switch.png

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;
}
```

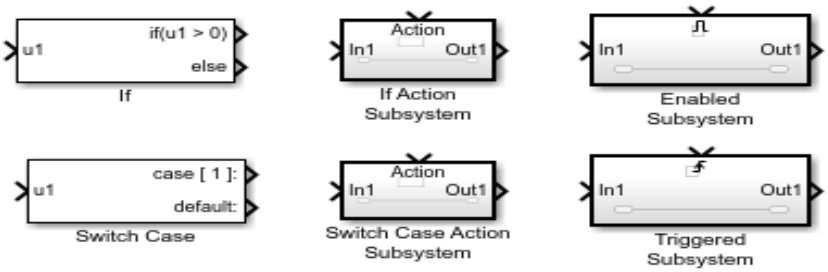


Figure 50: smlk/SmlkLibCondFBlocks.png

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.

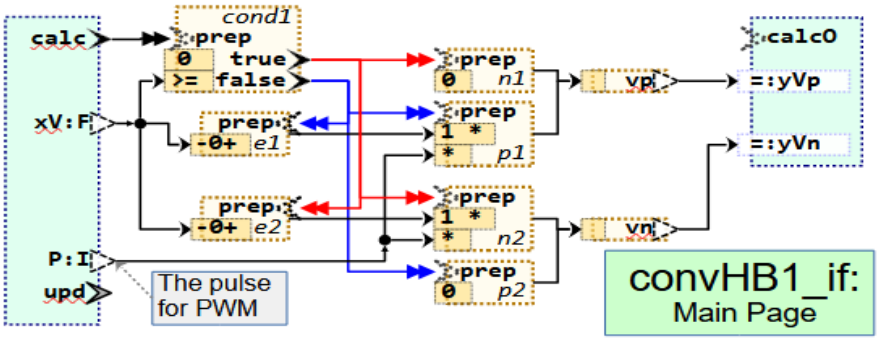


Figure 51: OFB/exmpTrueFalse.png

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:

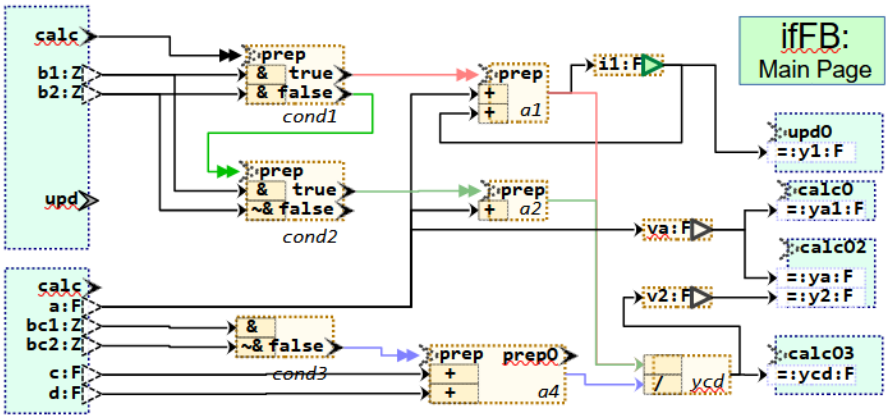
```
EVENT_CONNECTIONS
  calc TO cond1.prep;
  calc TO e1.prep;
  JOIN_calcO.J TO calcO;
  cond1.true TO e2.prep;
  cond1.true TO n1.prep;
  cond1.false TO e1.prep;
  cond1.false TO p2.prep;
  e1.prepO TO p1.prep;
  e2.prepO TO n2.prep;
  n1.prepO TO vp_X.prep;
  n2.prepO TO vn_X.prep;
  p1.prepO TO vp_X.prep;
  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 52: 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
.....
```

5.6.10 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

5.6.11 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:

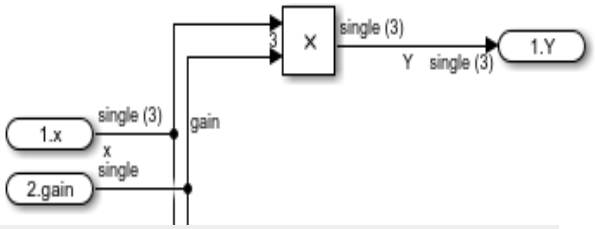


Figure 53: smlk/Exmp_Multiply_Vector_Scalar.png

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:

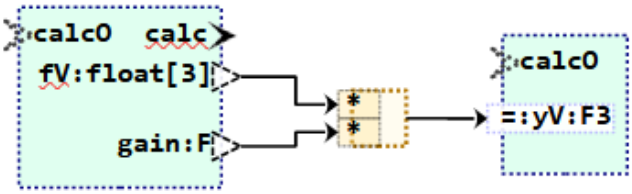


Figure 54: OFB/Exmp_Multiply_Vector_Scalar.png

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:

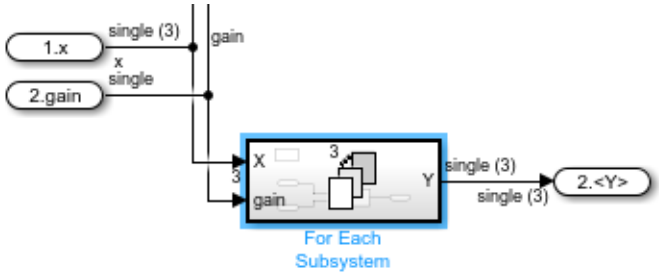


Figure 55: smlk/Exmp_Multiply_Vector_Scalar.png

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:

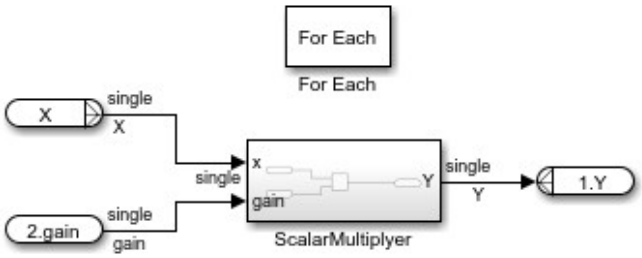


Figure 56: smlk/Exmp_ForEachSub_InnerScalar Mult.png

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:

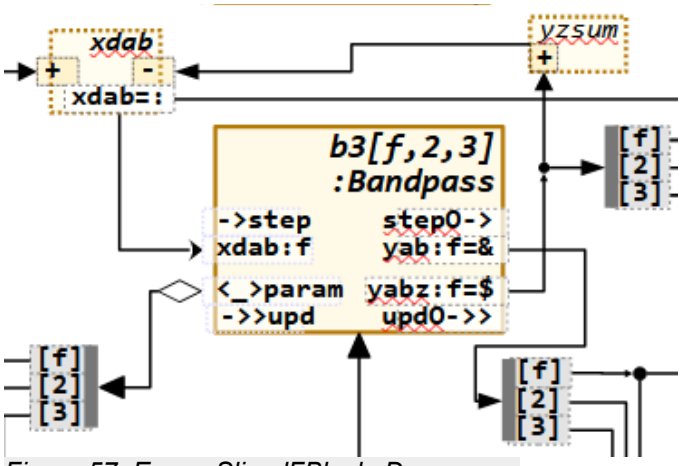


Figure 57: Exmp_SlicedFBlock_Demux.png

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 56: 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 *90*), 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 Connection possibilities.....82
5.7.1 Pins.....82
5.7.2 name : Type on pins.....86
5.7.3 Connectors.....86
5.7.4 Connection points.....87
5.7.5 Xref.....87
5.7.6 Using GBmux and GBdemux for connections.....88
5.7.7 Connections from instance variables and twice shown FBlocks.....88
5.7.8 Textual given connections.....88
5.7.9 Admissibility check of connections.....89
5.7.10 Data type test and conversion on inputs.....89
5.7.11 The direction of references and the data flow.....90
5.7.12 More outputs to one input.....90

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 58: 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.:

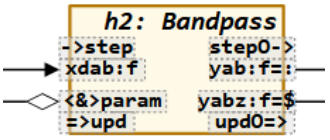


Figure 58: odg/FBpin_ofPinOnly.png

The pin kind designations are described in *5.2.3 Connector styles, ofc* page *32*. 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:

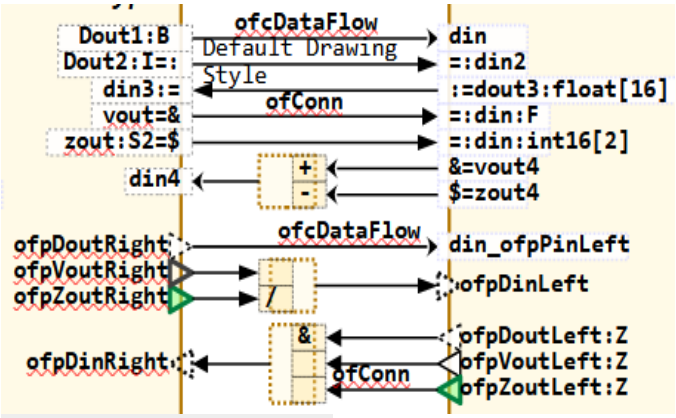


Figure 59: PindefDinout.png

The image above shows a detail of the https://vishia.org/fbg/deploy/OFB_DiagramTemplate.odg (*5.1 All Kind of Elements with there style* page *28*) 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 *3.7 Outfit 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 Texts in graphic blocks and pins* page *36*.

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 60: odg/FBpin_ofp.png

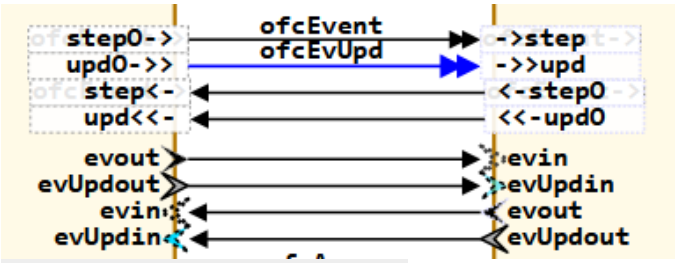


Figure 61: OFB/PindefEvinput.png

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 *5.11 Execution order, Event and Data flow, Event chains and states* page *124*

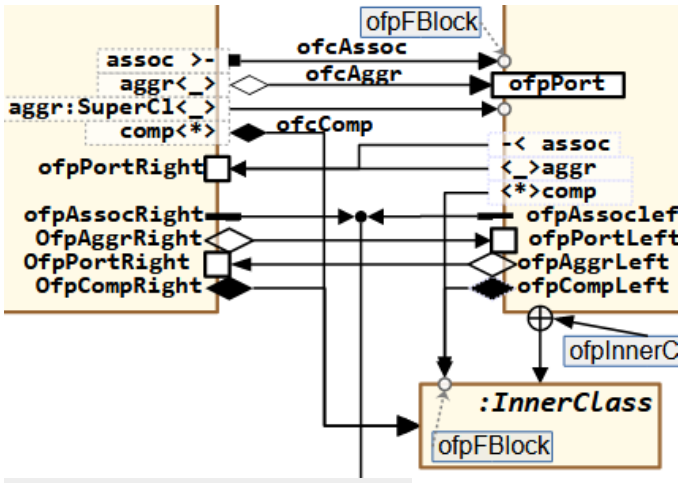


Figure 62: OFB/PindefRefPort.png

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.

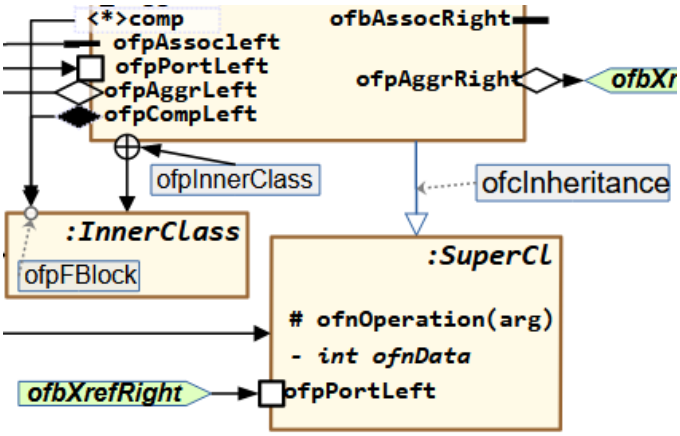


Figure 63: OFB/PindefRefPort.png

This image above shows the right continuing of the https://vishia.org/fbg/deploy/OFB_DiagramTemplate.odg (*5.1 All Kind of Elements with there style* page *28*)

`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>:

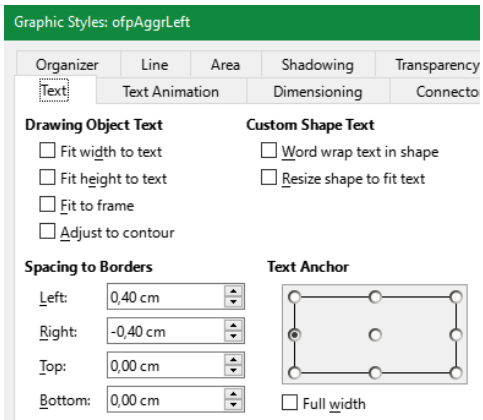


Figure 64: odg/Fbpin_ofpStyleText.png

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 *47*, 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

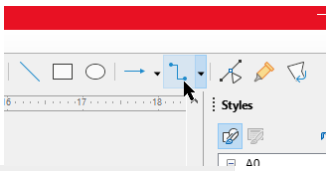


Figure 65: odg/Connector-Icon.pdf

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 *5.2.3 Connector styles, ofc* page *32*.

It is also interesting to have a line connector:

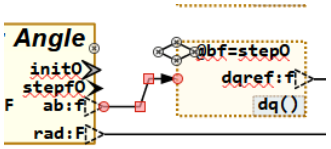


Figure 66: odg/LineConnectorExmpl1.png

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:

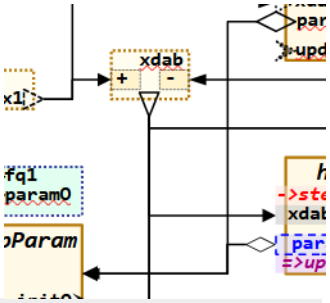


Figure 67: odg/LineConnectorExmpl1.png

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:

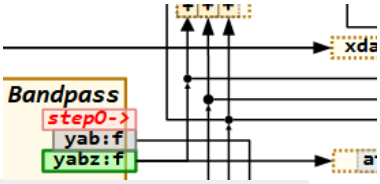


Figure 68: odg/ConnectionPoints1.png

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 *3.4 Connectors of LibreOffice for References between classe* page *9*

5.7.5 Xref

This is already described in *3.6 Diagrams with cross reference Xref* page *11*. 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 *5.6.11 Sliced or Array FBlocks, Demux and array data* page *80* 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:

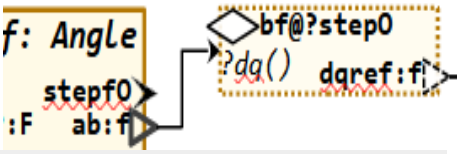


Figure 69: odg/ConnectionFromFBlockOut.png

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:

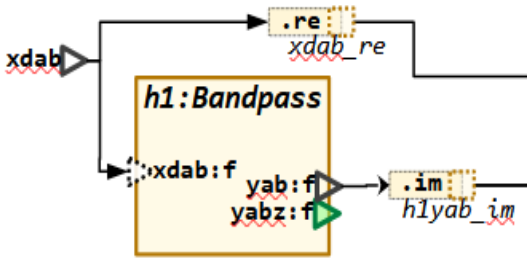


Figure 70: odg/ConnectionFromVariable.png

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 *80*

5.7.8 Textual given connections

It is also possible to write the connections simple as text:

Figure 71: 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.

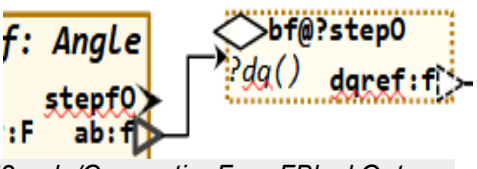


Figure 72: odg/ConnectionFromFBlockOut.png

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 *5.6.8 GBlocks for operation access in line in an expression - FBoper* page *74*

The graphical connected variant for an adequate approach is shown in:

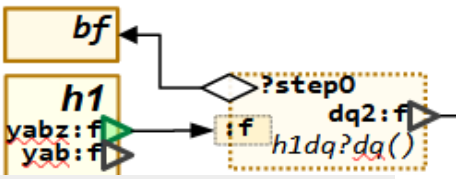


Figure 73: odg/FBoperGetterAggrConn.png

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`.

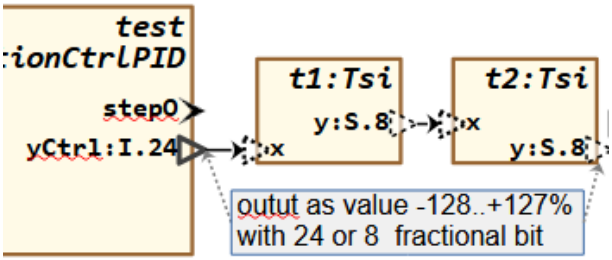


Figure 74: odg/ExmplAutoCastData.png

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

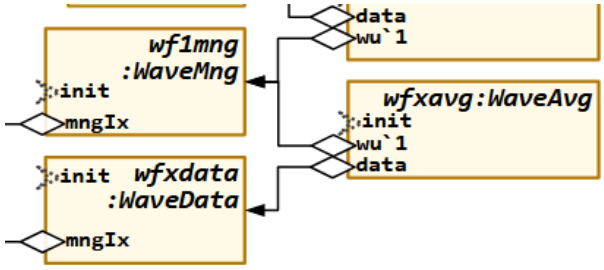


Figure 75: odg/ConnectionAggregation.png

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 *5.6.9 Conditional execution with boolean FBexpr* page *76*. One data input can have more driving sources, each for each condition.

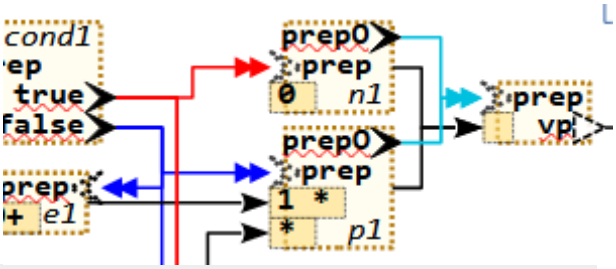


Figure 76: OFB/ExmpTrueFalseConnConditional.png

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 *5.6.9 Conditional execution with boolean FBexpr* page *76*.

But as in the next image shown, the variable vp can also drawn twice instead have two connections to the variable:

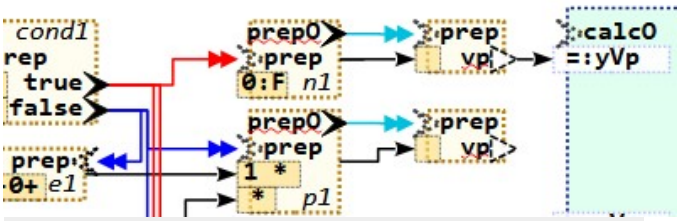


Figure 77: OFB/ExmpTrueFalseConnConditional1.png

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:

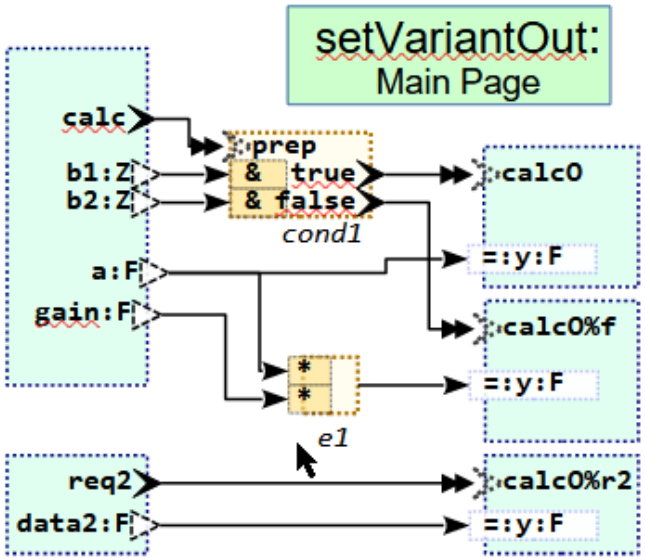


Figure 78: OFB/setVariantOut.png

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/cmpGen/genSrcCmp/setVariantOut.c::2'bool cond1'*2!1'}'::43::-//::*

```
  bool cond1 = false;
  thiz->mEvout_calc = 0;
  cond1 = (b1 && b2);
  if( cond1 ) {
    y1 = (a);
    thiz->mEvout_calc |= MASK_calc_calcO;
    thiz->y = y1;
  }
  if(!cond1) {
    y1 = ((a * gain));
    thiz->mEvout_calc |= MASK_calc_calcO;
    thiz->y = y1;
  }
```

include:../../BasicTest/cmpGen/genSrcCmp/setVariantOut.c::'Operation req2(...)'!0'}'::43::-//

```
/**Operation req2(...)
 */
void req2_setVariantOut ( setVariantOut_...
  , float data2
) {
  float y1;
  thiz->mEvout_req2 = 0;
  y1 = (data2);
  thiz->mEvout_req2 |= MASK_req2_calcO;
  thiz->y = y1;
}
```

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

5.8 Expressions inside the data flow (FBexpr)....92
5.8.1 Expression as rectangle and input pins as rectangle ofpExprPart....92
5.8.2 More possibilities of DinExpr....94
5.8.3 Data Type specification and value casting in expressions....100
5.8.4 Data types with fractional bits in expressions , using saturation....102
5.8.5 Any expression in FBexpr....107
5.8.6 Output possibilities, variable after expression....108
5.8.7 Set elements to a array of structure variable....109
5.8.8 Output with ofpExprOut....110
5.8.9 FBexpr as data set....110
5.8.10 FBoper, operation for a FBlock....111
5.8.11 How are expressions presented in IEC61499?....112
5.8.12 FBexpr capabilities compared to other FBlock graphic tools....114

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.

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 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.

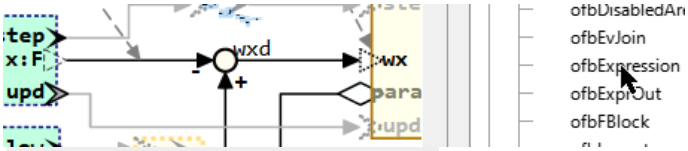


Figure 79: odg/ExpressionExmp.png

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.

5.8.1 Expression as rectangle and input pins as rectangle `ofpExprPart`

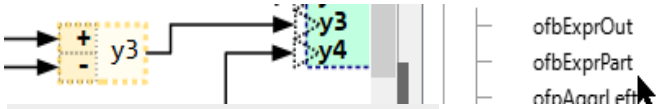


Figure 80: odg/ExpressionExmp.png

The other possibility is using a rectangle box with the style `ofbExpression`, in the following text referred to as ***FBexpr***: (*"Function Block as expression"*). 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 internal type of this elements is `DinExpr_FBcl` and hence ***DinExpr*** is written for that in the following text.

The DinExpr can contain the operator for this input, but also a factor as constant or as variable and also a type casting, see *5.8.2.3 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:

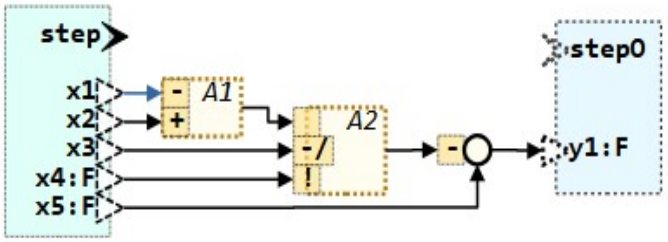


Figure 81: OFB/ExprExmpCombi.png

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:

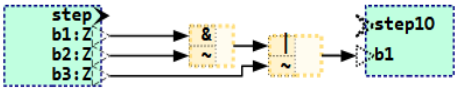


Figure 82: any image

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 More possibilities of DinExpr

But there are more possibilities using `ofpExprPart`:

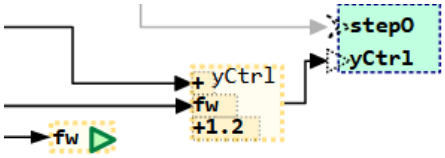


Figure 83: odg/ExpressionExmpK2const.png

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.

5.8.2.1 Operation on expression input: factors in Add expression, variables

There is also a possibility to write two variables in the expression input, but only if the input is not connected:

Figure 84: 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.

5.8.2.2 Access to elements of the input connection to use

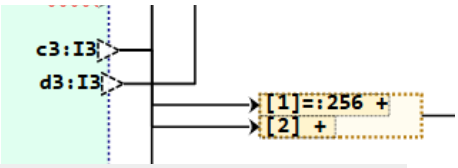


Figure 85: ExprInpArrayAccessMult256.png

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.

5.8.2.3 Description of all possibility, syntax/semantic of DinExpr

See also chapter html (www) / Impl-OFB_VishiaDiagrams.pdf (www): *7.3.7 Preparation of Expressions from odg* page *33*

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 *5.6.11 Sliced or Array FBlocks, Demux and array data* page *80* 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 *42*. 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 *5.8.7 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 *5.8.1 Expression as rectangle and input pins as rectangle 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()`)

5.8.2.4 Some examples for DinExpr

TODO

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`.

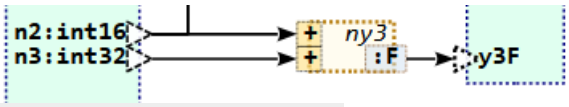


Figure 86: odg/ExmplAddFSI.png

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:

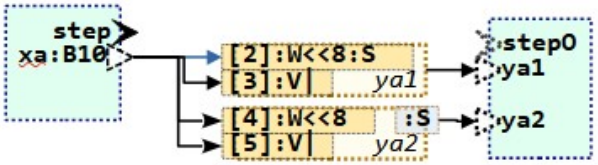


Figure 87: odg/ExmplShiftOr16.png

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:

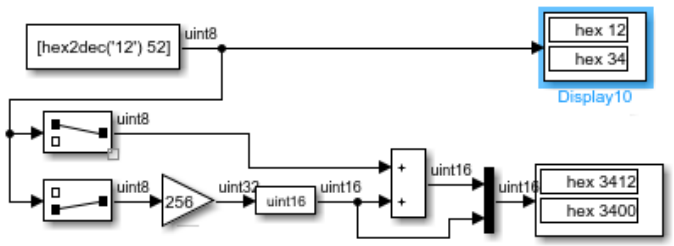


Figure 88: smlk/Exmp_shiftAdd8bit.png

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

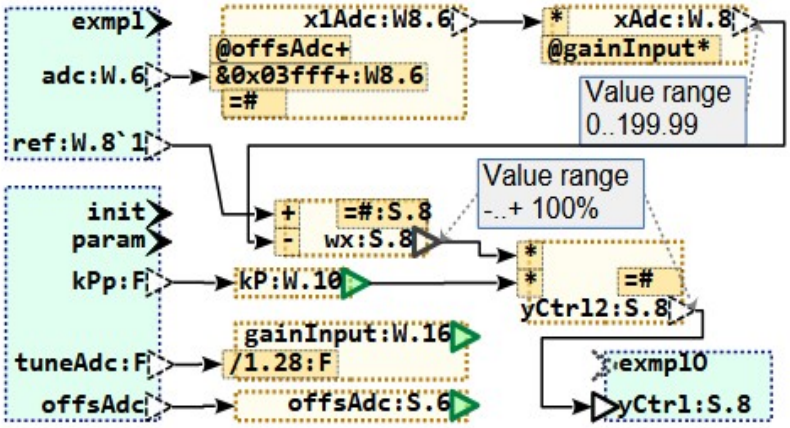
uses 6 bit before decimal point. The given float number is proper converted on compile time, no floating point arithmetic on a cheap controller. On multiplying here, the value range of the difference `wx` is multiplied with the `kP` gain. It means on gain=10.0 means 10% difference between reference and 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 89: 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 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:

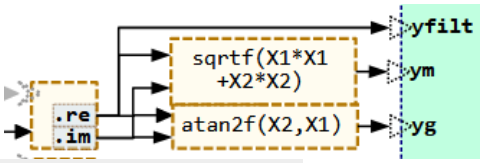


Figure 90: odg/ExprAnyX1X2.png

The `ofbExpression` shape or block 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 on the inputs using the .code template scripts, see chapter *5.8.10 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.

5.8.6 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.

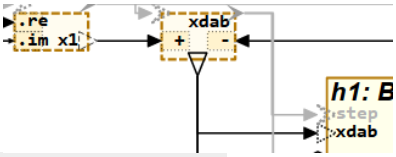


Figure 91: odg/ExprOutpin.png

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 *4.8 Storing the textual representation of OFB in IEC61499*, page *20*) 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:

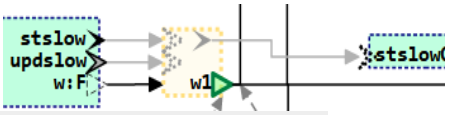


Figure 92: odg/ExprOutStateUpd.png

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 *5.8.1 Expression as rectangle and input pins as rectangle 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.

5.8.7 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.

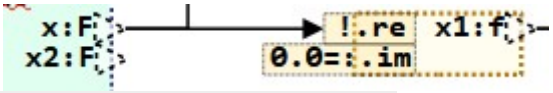


Figure 93: odg/Exmpl1SetElemvar.png

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 texts for pins and FBlocks* page *37* 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.

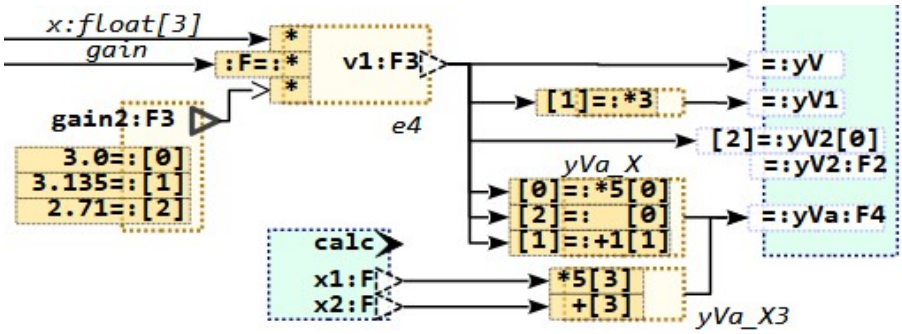


Figure 94: odg/ArraySlideDemux_VectorAccExpr.png

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`.

5.8.8 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 *5.8.5 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

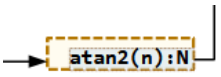


Figure 95: odg/ExprAtan2.png

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.

5.8.9 FBexpr as data set

This is a snapshot from the BandpassFilter example which have on input but needs internally complex values.

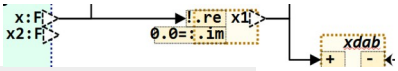


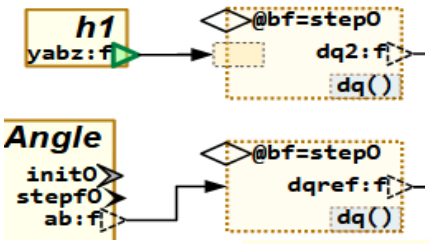
Figure 96: odg/ExprOutReIm.png

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.

5.8.10 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 *5.6.8 GBlocks for operation access in line in an expression - FBoper* on page *74*

empty

5.8.11 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 *5.6.7 Expression GBlocks* page *74*. 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.

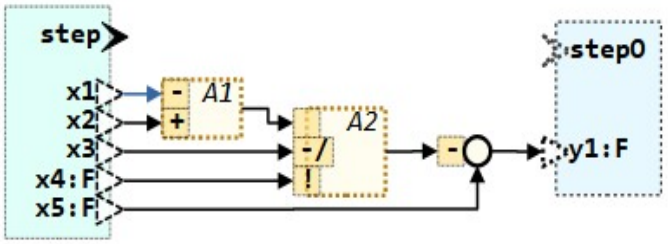


Figure 98: OFB/ExprExmpCombi.png

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:

- 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 html (www) / Impl-OFB_VishiaDiagrams.pdf (www): *7.1.6 FBexpr_FBcl: FBlock for expressions, presentation in FBlock_FBcl* on page *12*.5.8.10 FBexpr, FBoper and

5.8.12 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 Operations to FBlocks inside the data flow (FBoperation)....116
5.9.1 void Operation with input(s) and reference output....116
5.9.2 What is stored in the IEC61499 FBcl.fbd file:....117
5.9.3 Operation with return value and reference outputs....118
5.9.4 Join_OFB for inputs for calculation order....119
5.9.5 A FBoperation as simple getter....119

5.9.1 void Operation with input(s) and reference output

As shown in the overview chapter *5.6.8 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:

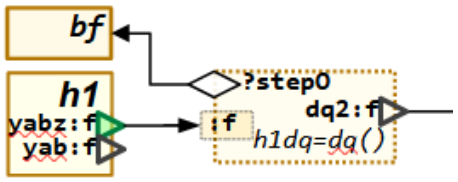


Figure 99: OFB/Fboper_h1dq.png

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 *5.6.8 GBlocks for operation access in line in an expression - FBoper* page *74*.

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 More possibilities of DinExpr* page *94*. 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:

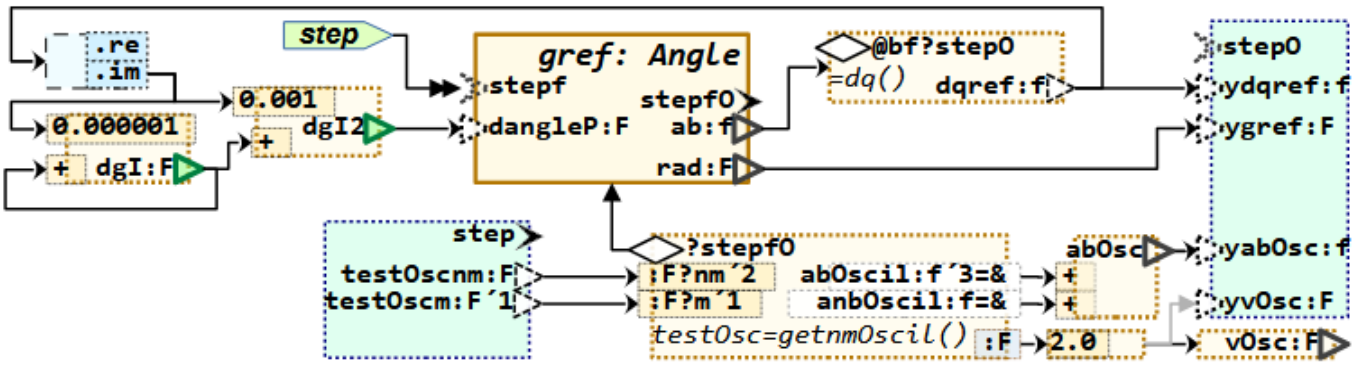


Figure 100: OFB/FBoper_getnmOscil.png

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 *5.5.4 Order of pins* page *54*), 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 (`stepfO`) 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 100: 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 *5.11.1 Event and Data flow* page *124*.

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 *5.6.11 Sliced or Array FBlocks, Demux and array data* page *80*. That chapter shows also a small comparison with Simulink © Mathworks.

5.10.1 Vectors in expression

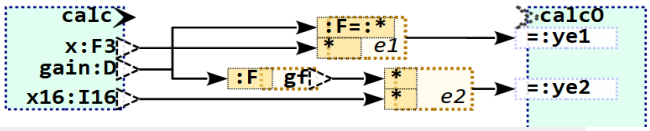


Figure 101: odg/ArraySlideDemux_VectorExpr.png

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/cmpGen/genSrcCmp/ArraySlideDemux.c::'gf = '#13::45::2:-::6:+

```
  gf = ((float)(gain));  // #genExprOut_gf...
  step_Ts1FiltSimple(&thiz->ts1[0], x[0]);...
    ......
  thiz->mEvout_calc |= MASK_calc_calcO;  /...

  thiz->ye1[0] = ((float)(gain) * x[0]); /...
  thiz->ye1[1] = ((float)(gain) * x[1]); /...
  thiz->ye1[2] = ((float)(gain) * x[2]); /...

  thiz->ye2[0] = (gf * x16[0]); // #genExp...
  thiz->ye2[1] = (gf * x16[1]); // #genExp...
  thiz->ye2[2] = (gf * x16[2]); // #genExp...
  thiz->ye2[3] = (gf * x16[3]); // #genExp...
```

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/cmpGen/genSrcCmp/FBcl/ArraySlideDemux.fbd::'FBS'#4::45::2:-::3:+

```
FBS
  JOIN_calcO : Join_OFB                    ...
    ......
  e2 : ARRAY [0..15] 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/cmpGen/genSrcCmp/FBcl/ArraySlideDemux.fbd::'TO e1.X2'#1::45

```
  x$calc TO e1.X2;  (*:F3  m_evinMdl=0x1  ...
```

include:../../BasicTest/cmpGen/genSrcCmp/FBcl/ArraySlideDemux.fbd::'TO e1.X1'#1::45

```
  gain$calc TO e1.X1;  (*  m_evinMdl=0x1  ...
  gain$calc TO e1.X1;  (*  m_evinMdl=0x1  ...
```

5.10.2 Vectors and scalar FBlocks

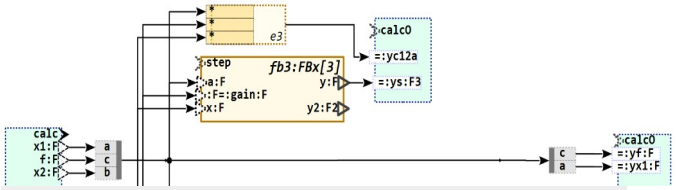


Figure 102: odg/ArraySlideDemux_VectorFBlock.png

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/cmpGen/genSrcCmp/ 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/cmpGen/genSrcCmp/FBcl/ ArraySlideDemux.fbd::'Ts1FiltSimple'#1::45

```
  ts1 : ARRAY [0..2] OF Ts1FiltSimple( fs:...
```

(empty)

In the generated C-source there is:

include:../../BasicTest/cmpGen/genSrcCmp/ 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/cmpGen/genSrcCmp/ 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/cmpGen/genSrcCmp/ ArraySlideDemux.c::'upd_Ts1FiltSimple'#9::45

```
  upd_Ts1FiltSimple(&thiz->ts1[0]);  // #F...
  upd_Ts1FiltSimple(&thiz->ts1[1]);  // #F...
  upd_Ts1FiltSimple(&thiz->ts1[2]);  // #F...
  //
  // Module outputs due to the event updO:...
  thiz->mEvout_upd |= MASK_upd_updO;  // #...
  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

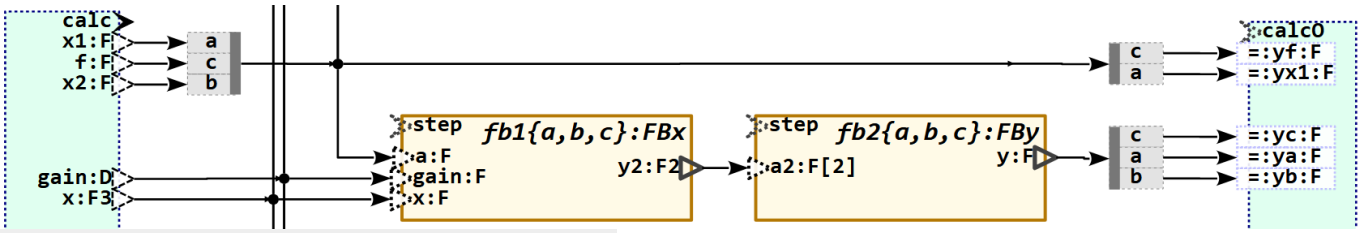


Figure 103: odg/ArraySlideDemux_DemuxFBlock.png

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/cmpGen/genSrcCmp/FBcl/ArraySlideDemux.fbd::'fb1a : FBx'#6::45

```
  fb1a : FBx_FB                          (*...
  fb1b : FBx_FB                          (*...
  fb1c : FBx_FB                          (*...
  fb2a : FBy_FB                          (*...
  fb2b : FBy_FB                          (*...
  fb2c : FBy_FB                          (*...
```

And the header file also:

include:../../BasicTest/cmpGen/genSrcCmp/ArraySlideDemux.h::'fb1a'#6::45

```
    FBx_s fb1a;  // fb1a:FBx_FB@5'60(59..8...
    FBx_s fb1b;  // fb1b:FBx_FB@5'60(59..8...
    FBx_s fb1c;  // fb1c:FBx_FB@5'60(59..8...
    FBy_s fb2a;  // fb2a:FBy_FB@5'100(92.....
    FBy_s fb2b;  // fb2b:FBy_FB@5'100(92.....
    FBy_s fb2c;  // fb2c:FBy_FB@5'100(92.....
```

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/cmpGen/genSrcCmp/FBcl/ArraySlideDemux.fbd::'TO fb1a.a;'#1::45

```
  x1$calc TO fb1a.a;  (*:F  m_evinMdl=0x1 ...
```

and in C/++ it appears as

include:../../BasicTest/cmpGen/genSrcCmp/ArraySlideDemux.c::'step_FBx(&thiz->fb1a'#6::45

```
  step_FBx(&thiz->fb1a, x1, (float)(gain),...
  step_FBy(&thiz->fb2a, thiz->fb1a.y2);  /...
  step_FBx(&thiz->fb1b, x2, (float)(gain),...
  step_FBy(&thiz->fb2b, thiz->fb1b.y2);  /...
  step_FBx(&thiz->fb1c, f, (float)(gain), ...
  step_FBy(&thiz->fb2c, thiz->fb1c.y2);  /...
```

include:../../BasicTest/cmpGen/genSrcCmp/ArraySlideDemux.c::'thiz->yc = '#3::45

```
  thiz->yc = thiz->fb2c.y;    // #genDinAc...
  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

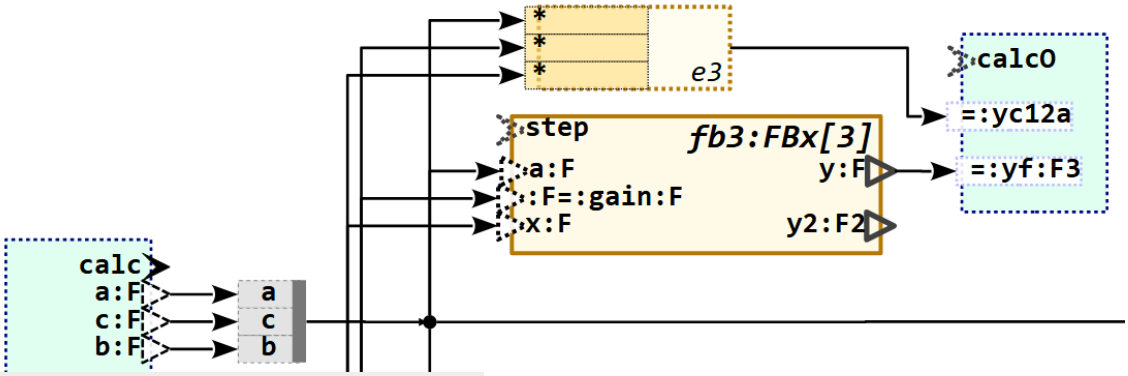


Figure 104: odg/ArraySlideDemux_VectorFBlock.png

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 103: 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 Execution order, Event and Data flow, Event chains and states

5.11.1 Event and Data flow

As also explained in chapter *5.6.2 GBlocks for each one function, data – event association*page *66*, events are associated to the data. In chapter *4.7 Using events instead sample times in FBlock diagrams* on page *18*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:

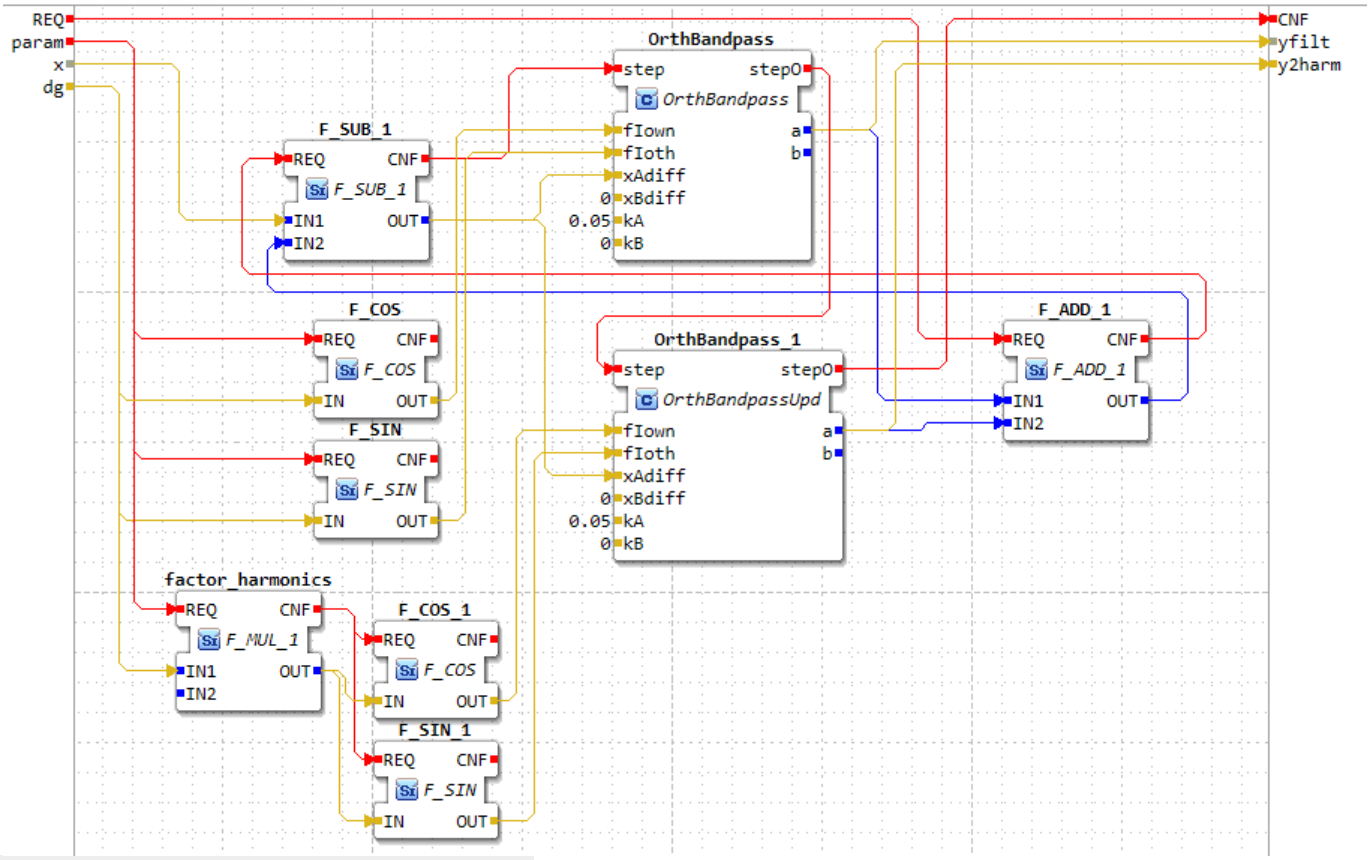


Figure 105: 4diac/OrthBandpassFilterAppl.png

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.

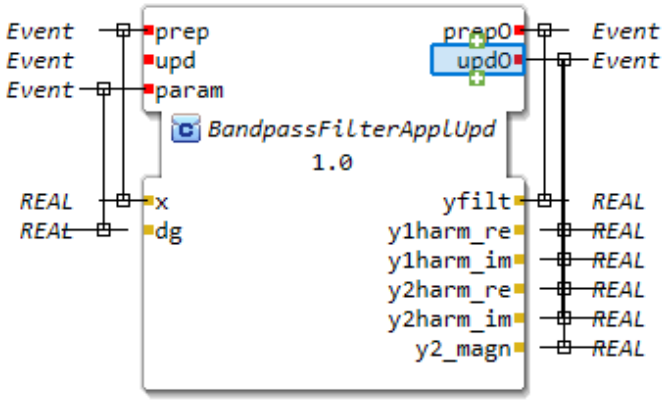


Figure 106: 4diac/OrthBandpassFilterApplUpd_ifc.png

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.

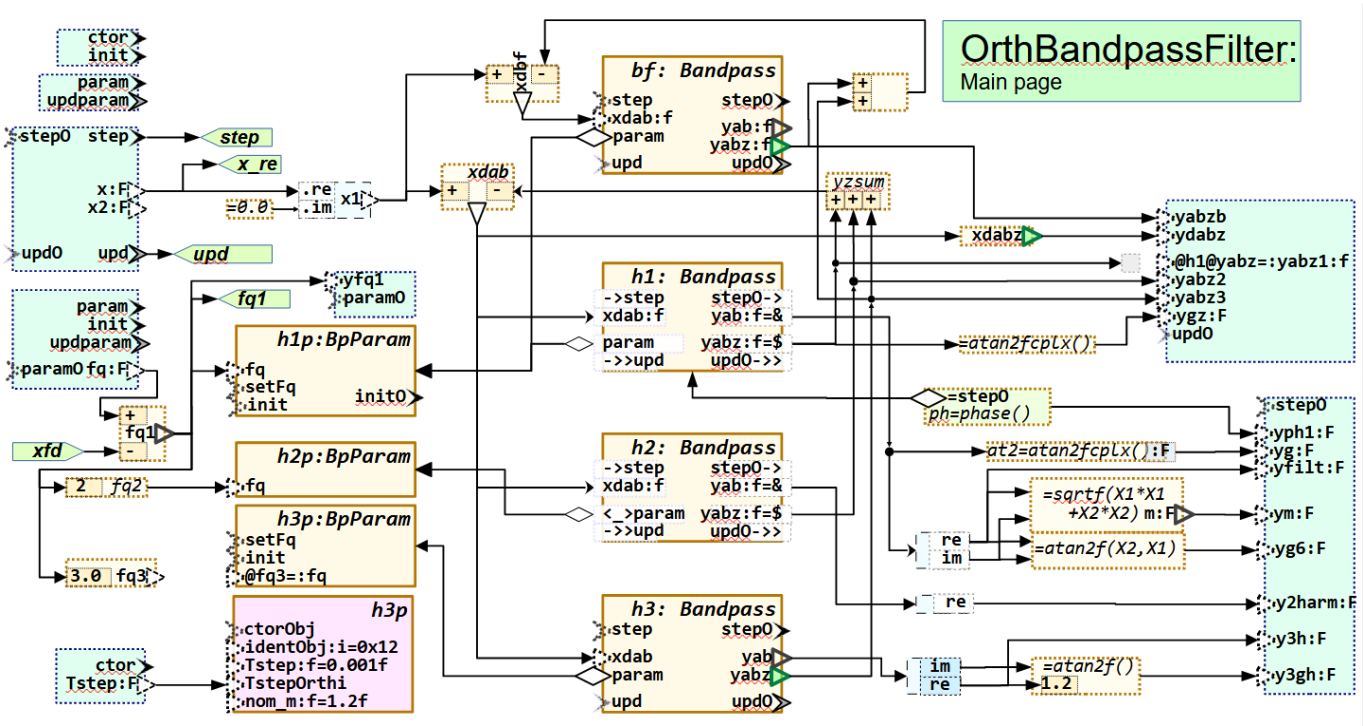


Figure 107: odg/OrthBandpassFilter.odg.png

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 *92* 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 *5.11 Execution order, Event and Data flow, Event chains and states* page *124*. 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”*.

5.11.2 Event chains for each one operation, state variables

The example before in *Figure 107: 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

5.12 Drawing and Source code generation rules

Table of Contents

5.12 Drawing and Source code generation rules.....128
5.12.1 Writing rules in target language used from generated code from OFB.....128
5.12.2 Life cycle of programs in embedded control: ctor, init, step and update.....129
5.12.3 Using events in the module pins and FBlocks, meaning in C/++.....130
5.12.4 More possibilities, definition of special events.....132

C/++ is only one example for a target language but it is the most familiar, hence it is used her for description.

5.12.1 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 *5.14.4 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.

5.12.2 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.

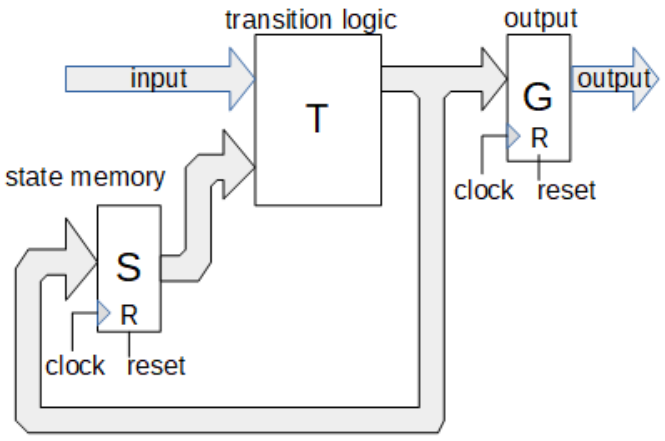


Figure 108: PrepUpd/q-input-trans-qout.png

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.

5.12.3 Using events in the module pins and FBlocks, meaning in C/++

See chapter *4.7 Using events instead sample times in FBlock diagrams* 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.

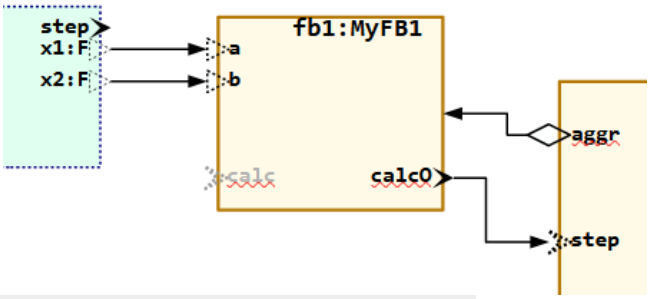


Figure 109: ExmplEvDeflt_calcOstep.png

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 `calcO`. 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 *4.3 Show same FBlocks multiple times in different perspective* page . If the data inputs are associated to another event there, this is valid. Then the here shown `calcO` does not influence the input data association between `calcO` 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` → `initO` 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 `initO`event, and the aggregation is related to the `init` event of the right FBlock.

- And also for construction a `ctor` and a `ctorO` 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.

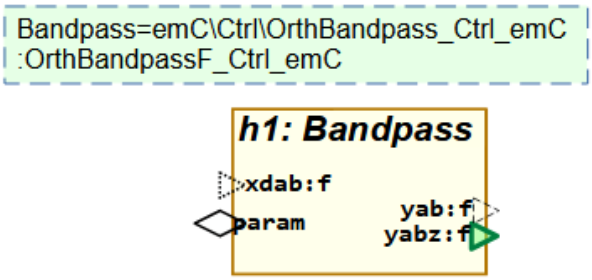


Figure 110: FBlockSimpleUsage.png

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 `stepO` and `initO`, 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 `updO` is automatically defined.

- All data inputs are associated to the `step`, all data outputs which are not `ofpZout` are associated to `stepO`. All `ofpZout` outputs are associated to `updO`.

- 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.

5.12.4 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)

5.13 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)

5.14 Converting the graphic – source code generation

As fast mentioned also in chapter *4.9 Source code generation from the graphic*page *21* , 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.

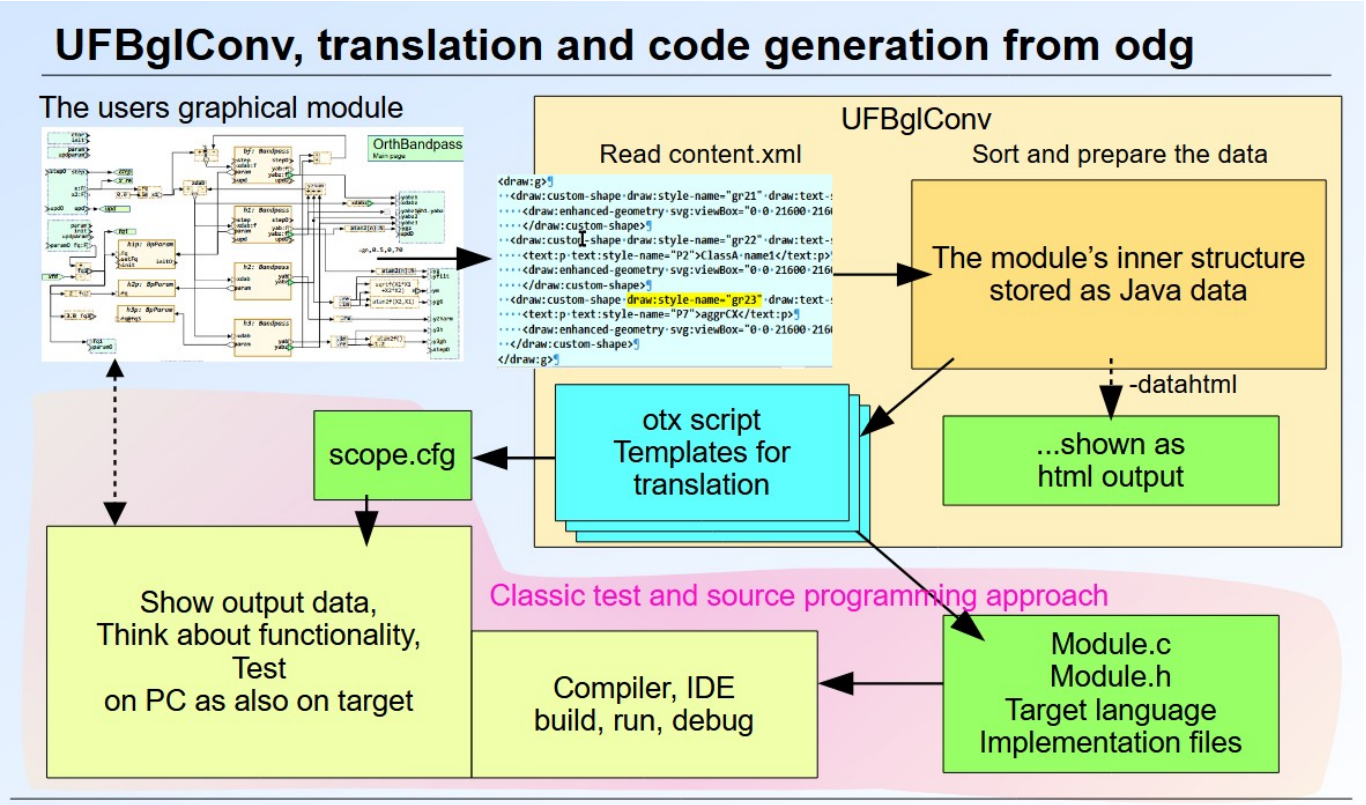


Figure 111: FBcl/OFBConvAndTestSlide.png

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.

This otx scripts have a syntax described in:

./../../Java/pdf/OutTextPreparer.pdf (www)

(empty)

5.14.1 Calling conversion with code generation

The code generation from Open/LibreOffice odg files can be performed with the following batch script::

include:../../BasicTest/makeScripts/genSrc_odg.bat::$::92::1:dir::

```
@echo off
echo called: %0 %1dir
REM %1 may be "NODIFF"
set RETDIR=%CD%
cd /d %~d0%~p0\..
REM This file is the batch file to call java and also the argument file.
REM clean build only if it is an empty directory, usual a dangling link (JUNCTION) on non...
if not exist ..\..\build\*.txt call ..\..\+createClean_mklink_build.bat
REM clean cpp\genSrc only if it is an empty directory, usual a dangling link (JUNCTION) o...
if not exist cpp\genSrc\*.c call +createClean_cppGenSrc.bat
REM set the java class path in JCP as central batch:
call ..\Organize_OFB\SetJCP.bat
REM works in a loop for simple repe...loop
:loop
@echo off
echo CD=%CD%
REM use --@file:label, the file is this file itself as %~d0%~p0%~n0%~x0 as absolute p...java
echo on
java -cp %JCPVISHIA% org.vishia.fbcl.OFBconv --@%~d0%~p0%~n0%~x0:args
@echo off
if ERRORLEVEL 1 (
  if "%1"=="NOPAUSE" (
    echo ERROR java OFBconv exit with %ERRORLEVEL% >>cpp/genSrc/report/log.txt
  ) else (
    echo ERROR: %ERRORLEVEL%
    pause
  )
)
echo =============== finished OFBconv Java==============================
REM the arguments are written in lines which are comments for the batch processing ::
REM characters before the label args are identification for the arg lines, but not part o...
REM one space and ## after the args label defines remove trailing spaces and remove comme...
REM --- is a commented argument for the java main routine
REM -tplCode:@org.vishia.fbcl references to inside the jar file given otx templates
REM -tplCode:makeScripts/scope.otx is an additional otx, for that the other should be given.
::args ##
::---ifbd:path/to/Module.fbd          ## select translation only for this modu...ifbd
::-i:../Templates_OFB/odg/LibCtrl_emC.odg  ## The declaration (lib-) input odg file to tr...
::-im:PIDctrl_TsModulDef              ## Use a specific module for this input file
::---xm:BandpassFilterModulDef          ## Exclude a specific module for this input file
::---cfg:../Templates_OFB/makeScripts/LibCtrl_emC.alias.cfg     ## cfg for code genera...cfg
::-i:odg/BasicTest.odg              ## The input odg file to transla...odg
::---im:ModuleSpec                    ## select translation only for this modu...im
::-cfg:makeScripts/local.aliasHeader.cfg
::-dirCmpn=.                        ## directory name builds $srcCmpnDir, as part of ...cmpn
::---dirStdFB:src/libModules_fbd/fbd
::-tplCode:@org.vishia.fbcl.writeFBcl.WriterCodegen:cHeader.otx     ## possible use other...
::-tplCode:@org.vishia.fbcl.writeFBcl.WriterCodegen:cImpl.otx
::-tplCode:../Templates_OFB/makeScripts/scopeEthernetComm.otx   ## code generation tem...otx
::-dirGenSrc:cpp/genSrc               ## The output directory for generated souce co...out
::-dirCmpGenSrc:cmpGen/genSrcCmp      ## directory for compare with source code gener...cmpr
::-fbg:cpp/genSrc/FBcl/               ## write raw content of each module to this dire...fbg
::-dirFBcl:cpp/genSrc/FBcl            ## directory for generated FBcl fil...out
::---dirFBcl:fbcl                       ## better use cpp/genSrc for more simple comparison
::-dirCmpFBcl:cmpGen/genSrcCmp/FBcl   ## dir to compare generated FBcl fil...cmpr
::---dirReport:cpp/genSrc/report        ## output directory for some log files for dat...rep
::-dirReport:../../build/$srcCmpnDir/report ## output directory for some log files for...rep
```

```
::-log:../../build/$srcCmpnDir/$srcModuleName.OFBconv.log      ## output file for l...log
::---dirReport:report
::-dirDbg:../../build/$srcCmpnDir/report/dbg       ## output directory for some log fi...dbg
::---odg                                  ## writes an file.odg as inner data presentati...odg
::-oxmltest                             ## possibility to write back the read content.x...xml
::---oxmldatahtml                         ## possibility to write internal data as html
::---datahtml                             ## possibility to write the internal data in html
REM show file differences:
if not "%1"=="NODIFF" call ..\Organize_OFB\fdiff.bat cpp\genSrc cmpGen\genSrcCmp
REM runs in loop only if not called with LOOP, else superior batch possib...loop
if "%2"=="LOOP" (
  echo ---
  echo repeat generation?
  pause
  cls
  goto :loop
)
REM exit with directory on ca...dir
cd /D %RETDIR%
exit /B 0
```

This is the whole content of the batch file `src/BasicTest/makeScripts/genSrc_odg.bat` in the example download, inclusively some explanations.

The approach here is: The batch file prepares some directories, and the calls the OFB converter as Java command line invocation. The same file is also used as command argument file.

`dir` this two lines on begin and end of the script assures that it can be called from another directory, for translate more as one project.

`JCP` This environment variable is set to the jar files as *Java Class Path*. Here also a possibility for debug is intended, should be commented.

`loop` For simple usage this batch runs in a loop, but only, see on end `loop`, if it is not called with `NOPAUSE`

`java` Java should be available on your system. All Java versions from Java-8 (Oracle), also OpenJDK.

The argument `--@%~d0%~p0%~n0%~x0:args` means, that this batch file itself is used as argument file. All arguments are written after the line starting with `::args ##`. and then each line beginning with `::` is used for one argument. Arguments starting with `---` are commented out - for flexibility.

`odg` `-i:…` defines an input.odg file. More as one such argument, hence more input files are possible. A Module can have some pages in more input files, all they are summarized 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.

`ifbd` `-ifbd:…` 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.

`im` `-im:…` If this option is used, only the named Modules (more as one possible) are translated from the odg file. This is usal for specific tests.

`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 current directory.

The `-dirStdFB:` is used to look for files, which are used as modules but not given as `-i:` argument. In this (may be more as one) directories proper module files are searched.

`otx` `-tplCode:…` This is the path to 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 commented argument.

`cfg` `---cfg:$!` This is an enhancement used for more configuration possibilities, yet commented.

`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).

`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:

- 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 *5.14.1 Calling conversion with code generation* page (Java internals) in a readable html file.

5.14.2 Handling of include in C/++ or import and real used type names

TODO

5.14.3 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`

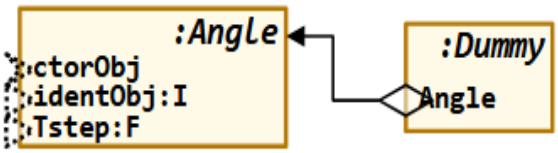
This is an error which occurs only if a generation otx script was changed. The reported script exists twice, maybe the first occurrence is in another script. The names of the sub scripts are unique over all scripts. The error is immediately output independent of usage of this sub script. Please fix it.

`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 112: odg/HowtoCreateTHIS.png



To access a FBlock as reference, for example via FBoper (see *5.8.10 FBoper, operation for a FBlock* page *111*) 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.

5.14.4 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.

5.15 Presentation of the graphic and results in files

Table of Contents

5.15 Presentation of the graphic and results in files.....142
5.15.1 The original odg format (Overview).....142
5.15.2 Graphic saved with the option The original odg format (Overview).....142
5.15.3 The FBcl format or IEC61499, file.fbd.....144
5.15.4 The original odg format (Overview).....146

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.

5.15.1 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.

5.15.2 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
```

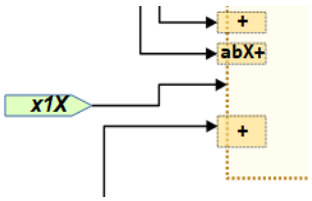
The output is cut here on right side. But the principle should be able to recognize. It contains the graphic position with page, coordinates in mm. The mm-value after the page is the vertical row, where some FBlocks are centralized in vertical order. This arrangement is important for the order of FBlocks 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 113: 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 a 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.

5.15.3 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/cmpGen/genSrcCmp/FBcl/ArrayBandpassFilter.fbd::'F"VAR_INPUT'::45::5:-:: 'END_EVENT':+::+4:-::'END_EVENT':+

```
FUNCTION_BLOCK ArrayBandpassFilter
EVENT_INPUT
  ctor WITH Tstep$ctor, q1$ctor, qh$ctor, ...
  init WITH fq$init; (* kind=Evin@1 ~evdat...
  param WITH fq$param; (* kind=Evin@2 ~evd...
    ......
END_EVENT
EVENT_OUTPUT
  stepO WITH yph1, Yg, Yfilt, ym, yg6, Y2h...
  updO WITH yabzb, ydabz, yabz, yabz2, yab...
    ......
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/cmpGen/genSrcCmp/FBcl/ArrayBandpassFilter.fbd::'VAR_INPUT'#30::45::3:-:: 'END_VAR':+::+4:-::'END_VAR':+::+1:-

```
VAR_INPUT
  fq$init : REAL; (* kind=Din@0 ~evdata=0x...
  fq$param : REAL; (* kind=Din@1 ~evdata=0...
    ......
END_VAR
VAR_OUTPUT
  yfq1 : REAL; (* kind=Dout@0 ~evdata=0x4 ...
  yabzb : CREAL; (* kind=Dout@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 *42*.

Further follows:

include:../../ExmplBandpassFilter/cmpGen/genSrcCmp/FBcl/ArrayBandpassFilter.fbd::'FBS'#30::45::3:-::'b3f':+::+6:-

```
FBS
  JOIN_dqref_X_dq : Join_OFB                ...
```

```
  JOIN_e_3_9_40_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 *5.8.11 How are expressions presented in IEC61499?* page *112*.

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/cmpGen/genSrcCmp/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.

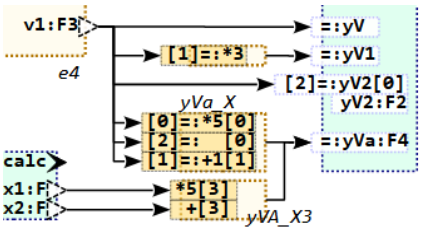
include:../../BasicTest/cmpGen/genSrcCmp/FBcl/ArraySlideDemux.fbd::'END_FBS'#30::45::4=-::'v1.prepO'=+::+6=-::'END_CONN'=+::+2=-

```
END_FBS
EVENT_CONNECTIONS
  calc TO e1.prep;  (* mEvMdl=1/1 cond= sr...
  calc TO e3.prep;  (* mEvMdl=1/1 cond= sr...
    ......
  v1.prepO TO e_5_13_96.prep;  (* mEvMdl=1...
  v1.prepO TO yVa_X.prep;  (* mEvMdl=1/1 c...
  yVa_X.prepO TO JOIN_calcO.J11;  (* mEvMd...
  yVa_X3.prepO TO JOIN_calcO.J11;  (* mEvM...
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 114: ArraySlideDemux_VectorAccExpr-2.png



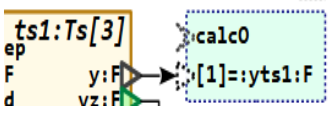
Array access to variables is part of the connection

```
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 115: 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;
```

5.15.4 The original odg format (Overview)

One odg file

empty page

6 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