1

(last page backward)

OFB – Object oriented Function Block Graphic

–

using LibreOffice draw

–

Handling

Dr. Hartmut Schorrig
www.vishia.org 2024-10-09

Table of Contents

1 Handling with OFB diagrams and LibreOffice draw

1.1 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

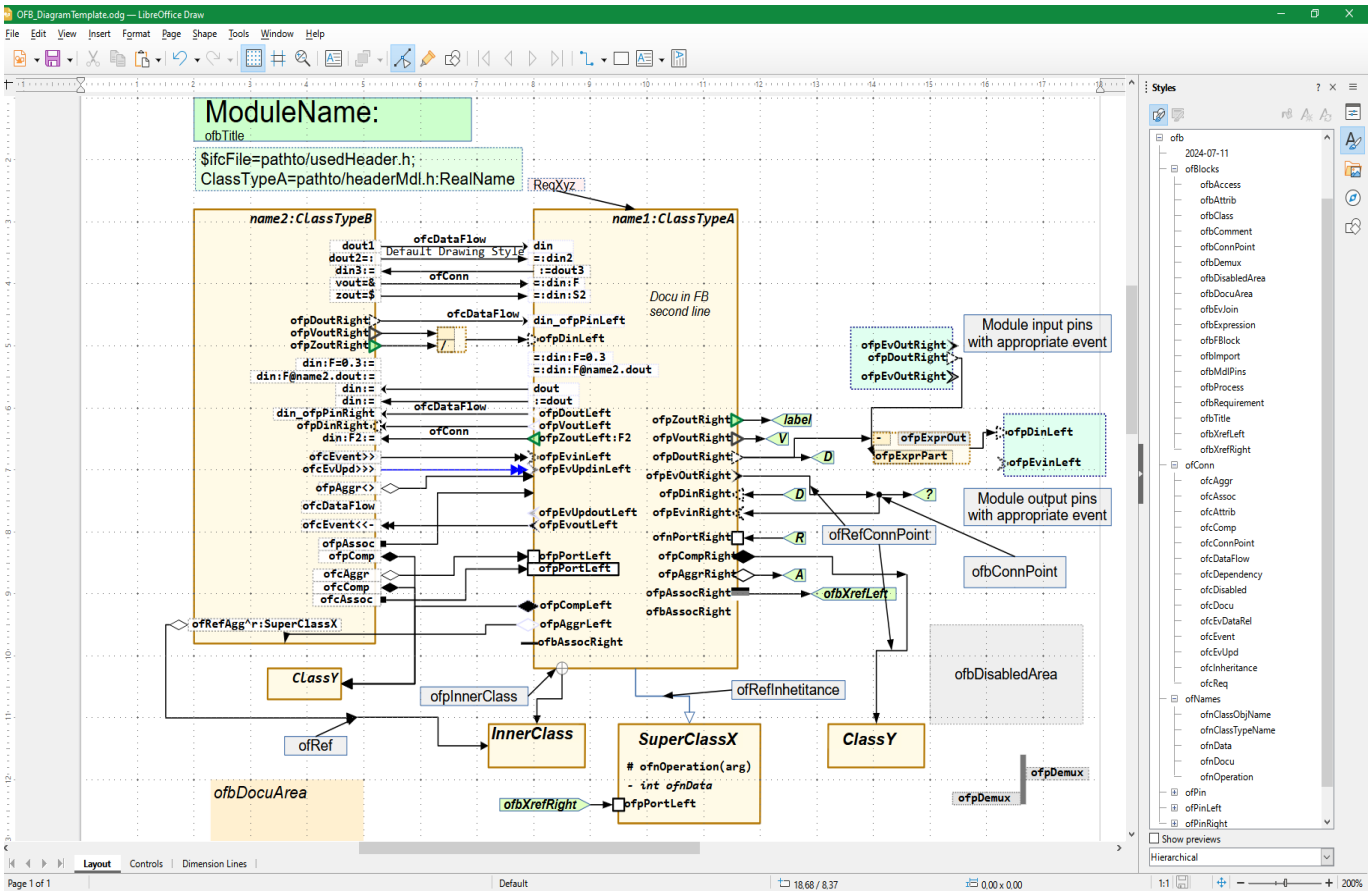


Figure 1: odg/OFB_DiagramTemplate.png

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 html / Basics-OFB_VishiaDiagrams.pdf: ***3.2 Using figures with styles (indirect formatted) for element*** page ***8***. The identifier of the style sheet is here used also as name, only for documentation.

The class left `ClassType name` contains simple connection elements of the base style `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 ***1.2 All styles*** page ***8***

1.2 All styles

1.2.1 GBlock styles, ofb

GBlock (*Graphic Block*) styles should be assigned to shapes that represent blocks that can contain 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 ***1.5.1 Module in file organized in pages*** page ***20***

- `ofbImport`: This is a shape which contains the association between aliases and the real used type and the interface files (header files in C/++) for used modules, which are given in a target language. See ***1.5.1 Module in file organized in pages*** page ***20***

- `ofbMdlPins`: This is a shape which contains the pins of the module, see ***1.5.1 Module in file organized in pages*** page ***20***

- `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 ***1.6 Possibilities of Graphic Blocks (GBlock)*** page ***24***

- `ofbExpression`: This is an expression FBlock or also named *“FBexpr”*, see ***1.7 Expressions inside the data flow*** page ***38***

- `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 ***1.9 Execution order, Event and Data flow*** page ***58***

- `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 ***1.8 Connection possibilities*** page ***52***

- `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 `ofConn` 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.

- `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 `ofConn`. 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 ***1.10 Showing processes*** page ***61***

- `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.

1.2.2 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 ***1.6.1 Difference between class, type and instance ("Object")*** page ***24***

- `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).

1.2.3 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 ***1.8 Connection possibilities*** page ***52***.

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

- `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 diamon, only a reference.

- `ofpComp`: `<*>name<*>` It is an 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. If a type, not an FBlock instance, marked only with `:type` for the connected (referenced) FBlock is given then the code generation produce a memory allocation on construction. If a named FBlock is given, this FBlock is part of the modules objects, then it is in fact an aggregation, but thought as composition. ***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` Data input, without leading or trailing marker. But it may have operators as described in ***1.7.1 Expression parts as input*** page ***38***. ***Mnemonic hint:*** That’s why additional pin kind markers are too mucj.

.- `ofpDout`: `:=name=:` Data output, the data are locally defined. ***Mnemonic hint:*** `=` is often used for assignment (to the output). `:=` or also `=:` is known for assignment in IEC61499 textual language and also other automation device languages, originally from Algol or Pascal.

.- `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 output, the `$` should associate to a “S” for state variable.

- `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.

- `ofpExprPart`: It is an input of an expression. It has no specific designation for the pin kind, it should be used only in expressions. Instead a simple `ofPin` cannot be used there. See ***1.7 Expressions inside the data flow*** page ***38***

- `ofpExprOut`: It is an output of an expression. It has no specific designation for the pin kind, it should be used only in expressions. Instead a simple `ofPin` cannot be used there. See ***1.7 Expressions inside the data flow*** page ***38***

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

1.2.4 Connector styles, ofc

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

- `ofcAggr`: It shows a non filled diamond on the start of the connector as in UML.

- `ofcAssoc`: It shows a very small filled rectangle (0.6 mm) on the start of the connector, to distinguish from the standard connector

- `ofcComp`: It shows a filled diamond on the start of the connector as in UML.

- `ofcConnPoint`: This style is attended to use as connection to a connection point or to connect two connectors. It has a very small arrow on end (0.6 mm).

- `ofcDataFlow`: attended to use but not necessary for data flow (can be removed in future, do not use it).

- `ofcDataFlow`: attended to use but not necessary for data flow (can be removed in future, do not use it).

- `ofcEvent`: attended to use but not necessary for event flow (can be removed in future, do not use it).

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

- `ofcInheritance`: Inheritance between types also able to apply from a FBlock to a class GBlock (without name). If the referenced GBlock is a FBlock with name, the instance is not used. As familiar in UML the end is a non filled symmetric triangle arrow.

- `ofcDependency`: Dependency between types (the source type uses the destination type). As In UML a long dashed line with an open arrow on end.

- `ofcDocu`: From a `ofbComment` GBlock to the appropriate destination, a gray dotted line with a small filled arrow on end.

- `ofcReq`: From a `ofbRequirement` GBlock to the appropriate destination, a gray dashed line with a small filled arrow on end.

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

- `ofcEvDataRel`: For connectors between pins to associate event and data. Todo: If this connector style is applied at least between two pins of a FBlock or FBtype, then an automatically association between all shown pins in the GBlock is not done. See ***1.6.2 GBlocks for each one function, data – event association*** page ***26***

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.

1.3 Texts in graphic blocks and pins

The text entries in all graphic boxes and pins are built with the same syntax. See also html / Impl-OFB_VishiaDiagrams.pdf: ***1.3.3.4 Evaluating Pin texts*** page ***24***

The simplest form, used for FBlock is:

```
name : type
```

or exact in ZBNF syntax

```
nameType::= [ <$?descr> ]  : <$?type>
          |   <$?descr>  [ :  <$?type> ].
```

It means either the `descr` or the `type` is optional, But the given type should always written with a colon before. The type is always after the name (or description). This is used 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 a `?` (see following). All other character. Especially operators are part of the description. See ***1.7.2 More possibilities of DinExpr*** page ***40***.

For pins some more possibilities are given:

```
@fbSrc@pin.elemAccess:cast=:descr:type
```

or also more simple:

```
.elemAccess:cast=:descr:type
```

or only

```
:cast=:descr:type
.elemAccess=:descr:type
```

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

```
+
```

There is one more syntay possibility: The text can end with

```
…?specificDesignation
```

This can have a special meaning.

General it is:

```
name[ix]:type[size]@connection=value
```

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]`"

1.4 Data types

Table of Contents

In the ***Error: Reference source not found Error: Reference source not found*** the input `x:F` is designated as float input with the letter `F`. This is very space-saving but still obvious. Other tools sometimes have only a *"Pin dialog"* where the type can be selected and can optional show the type in the graphic, but then all types destroying the overview. The idea only using one character should be seen as proper, the number of types used are not too much. This is for the standard usual numeric types. The type of aggregations are determined by the destination class. A type name can be given additionally if necessary.

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

1.4.1 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 A
+-ANY_DERIVED                         L
+-ANY_ELEMENTARY                      E
  +-ANY_MAGNITUDE                     M
  | +-ANY_NUM                         N
  | | +-ANY_REAL                      G
  | | | LREAL                           D
  | | | REAL                            F
  | | +-ANY_INT                       K
  | | LINT, DINT, INT, SINT       J I S B
  | | ULINT, UDINT, UINT, USINT   Q U W V
  | +-TIME                             T
  +-ANY_BIT                           b
  | +-LWORD, DWORD, WORD, BYTE    q u w v
  +-BOOL                              Z
  +-CHAR                              C
  +-ANY_STRING
  | STRING                            c
  | WSTRING     (not specified)
  +-ANY_DATE                          H
  | 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
```

Complex types, not defined in IEC61499

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

The shown character for this types (green) are used for OFB, based on this basic types:

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

1.4.2 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.

1.4.3 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.

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`.

For expressions there is a simple way to build vectorized values and access to elements:

TODO

1.4.4 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.

1.4.5 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 defragmentation 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:

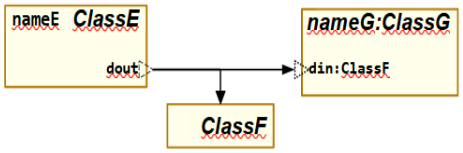


Figure 2: OFB/DflowStructData1.png

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:

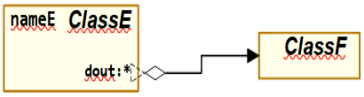


Figure 3: OFB/DflowStructData1.png

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`.

1.4.6 Data type forward and backward propagation

The input variables of the PID controller do not need this type declaration here, because the type is forwarded. But it is shown nevertheless, gets more clarity for usage. The type of the output variable `y:F` do also not need to be shown if or because the module is well defined in its interface for explicitly types or for type forwarding.

More step times or calculation events: In this example automatically an event chain is generated from `stslow` (means a slower step time) to the expression block with the `w1`variable, and forward to the event output `stslowO` (not shown here). Because `w1` of of style `ofpZout…` it needs updated with the correspond `updslow` event on the module's input block. If the value of the `ofpZout` variable is connected to outputs of the module with also the `updslow` event, the appropriate data flow will be assigned to this event chain till `updslowO`.

Data consistence: If the value of the `ofpZout` variable is used in another event chain, as shown here for built `dwx`, the stored value of the last calculation (after update) is used. In this case the value comes from another step time or calculation event, just the `stslow`, and hence consistently data all from this update event can be used. The consistence of the data should be guaranteed by a proper implementation. For example a slower step time can prepare values in with higher calculation effort, but the update of this values is done in a high priority interrupt which cannot be interrupted by another. The update needs only copying of values, or as better solution switch only a pointer to a double buffer system, if the update event is registered for the interrupt. Then the values are always consistent.

old:

You can show data and event pins on classes, but the connections are only sensible between the instances. This is familiar for FBlock diagrams. The t**ype of data pins** can be given immediately on the pin (after colon), but can be also forward propagated by a data flow. Simple arithmetic operations do not change the type of source pins and forward the type to the destination pins. Specific operations (for example access to the real and imaginary part of a complex value or to an array element) does not change the numeric type but influences the real/complex or array property of the type. Specific FBlocks can forward the type of inputs to the type of outputs. A backward propagation (as in Simulink) is not designed, because sometimes a mix of forward and backward propagation is more confused by the user. An important property of FBlock diagrams is, that the numeric type of pins in library FBlocks are not determined, instead a type dedication as `ANY_NUMBER` (in IEC61499) or such can be used. In Simulink it is determined as "***inherit***" type. It means that the types in the usage of the FBlocks depends from its using environment. For code generation either any template should be used (C++) or the FBlock should be existing as variant with all necessary types, or the FBlock implementation is a macro (C language) where the compiler associates the type.

1.5 One Module, Inputs and Outputs, file and page layout

Table of Contents

1.5.1 Module in file organized in pages

On 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`:

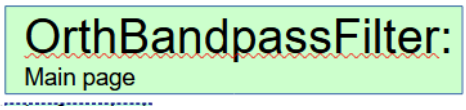


Figure 4: og/ofbTitle-1.png

The first word separated with colon is the name of the module, should be an identifier. The following text 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.

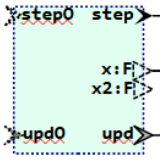
If the page contains an area with style `ofbDisableArea` then all shapes which are inside or only touches this area are not evaluated. This is a simple and proper obvious possibility to deactivate parts of the graphic without removing in the graphic, similar as commented parts in textual sources.

How does it works, see also 8.3.3 todo label

1.5.2 Module pins

Module pins should be contained in a shape or graphical block (***GBlock***) with the style `ofbMdlPins`

Figure 5: og/ofb-1.png



This ***GBlock*** should contain data input and output pins, whereby for practical reason the output pins (usual right on side, left in the block frame) are separated from the input pins (usual left on side, right in the ***GBlock*** as shown here).

But also associated events should be given. The events are important for association to the data.

The module's data input pins are of style `ofpDout…`, usual `ofpDoutRight`. Why dout: because they are data outputs to the inner connection of the module, they are data inputs seen from outside, from usage of the module. shows module's data inputs. Adequate, the module's data outputs are of style `ofpDin…`, usual `ofpDinLeft`.

The module's event input are `ofpEvout…` and `ofpUpdEvout…`. Both are shown in right side.

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 (`ofpEvout… `)

The given update event is associated for the update operation proper to the prepare operation.

It means for this , the module has one operation

```
step_OrthBandpassfilt(…, float x, flost x2);
```

and one operation

```
upd_OrthBandpassfilter(…);
```

without data arguments. For prepare and update see chapter ***Error: Reference source not found Error: Reference source not found*** 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 presentation of the module's event out pins for prepare and update, style `ofpEvin…`or `ofbEvUpdin…` (optional) means, that the module's input event are not ending in a state machine, which has specific output events, instead this are operations with immediately output data and a created output event if they are calculated. From outside, without knowledge of the inner module functionality, this module can be seen as a black box Standard FBlock with a simple regular state machine. It means, each event reacts with an output event, and does not really change the state, or it has defacto no states.

A module with FBlocks with state machines are not realized in the current version (2024-08) of OFB, but it is planned in the near future. But then the module's output events are not given.

To complete this description, have a look to page one of this module as a whole. The same image is used more times in this documentation, because it shows some important concepts on one example:

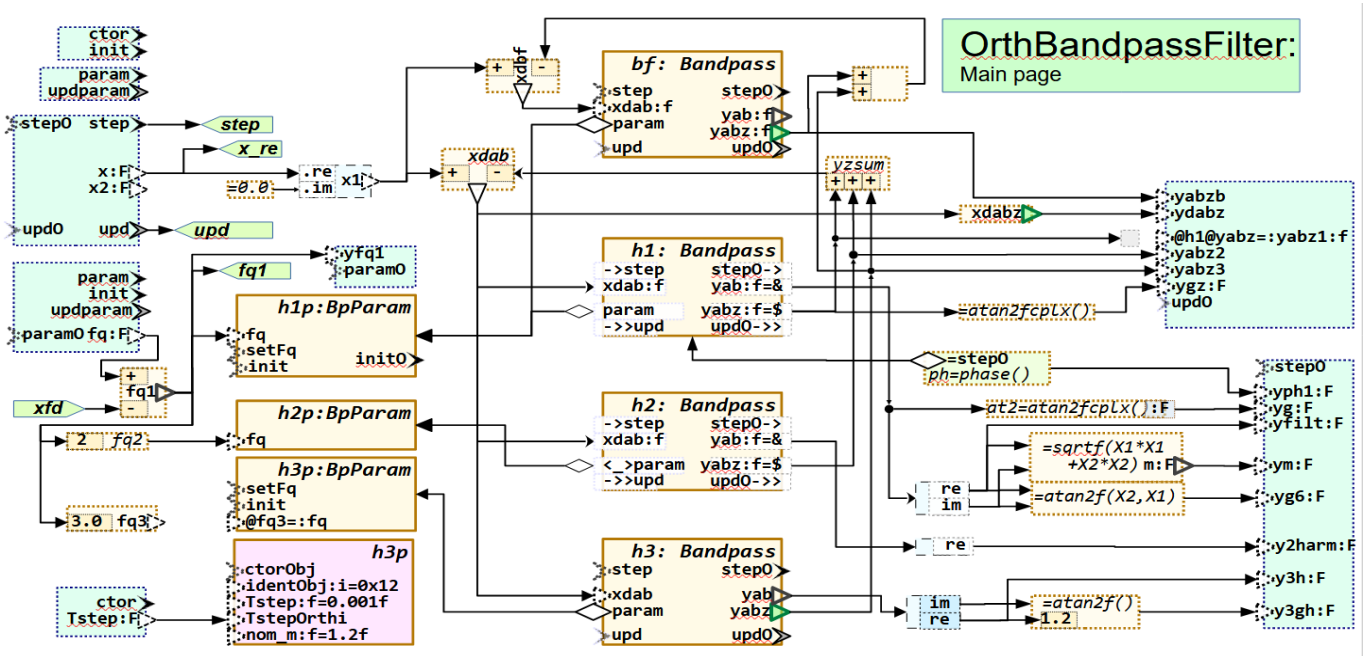


Figure 6: OrthBandpassFilter.odg.png

In left top are some input events and data, and proper output pins for the prepare event are right side for `stepO`, and also right side above with `updO`.

The code generation can offer an operation prototype with these output values which should be implemented outside, but it is called inside the module if the module's output event is activated. That is the pure event driven

implementation. This output event operation can be implemented either to send the events via communication in an event queue, or via inter-process-communication to any other device, or it can be implemented to organize the call of an operation of another module.

The other more simple more manual programming approach is, only offer the calculated values in data struct.

1.5.3 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, and as order in the generated code. If you think on reproducible build, then it is important that a repeated generation of code should create the same source code 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. But any where the order of the pins should be determined. It may be sensible to sort the pins by its name (alphabetically), but it is better to sort the pins by its graphic position of first usage. If the pins are used furthermore, in other pages or in the same page twice, it is not essential. The first detection in graphic determines.

To have an overview this part of is repeated here: in a part as For the approach of using the graphic position, the graphic here contains left top first the both events for `ctor` and `init`. It means the first event (left, top) is `ctor`. Then `init` comes. This is the order of the event in the IEC61499 fbd file and also in generated code. First the `ctor_…()` operation comes in the implementation source, then the `init_…()`. But the data for `ctor` and `init` are not designated here, it is in another `ofbMdlPins` block.

The order is first the order of the `ofbMdlPins` ***GBlocks***, and then the order of the pins inside each ***GBlock***.

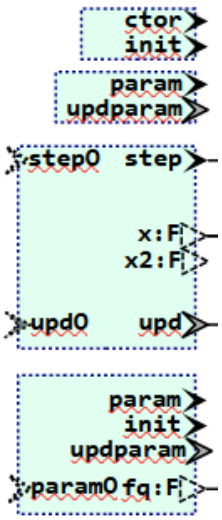


Figure 7: odg/ofbMdlPins-2.png

For the 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. Look on . You see that 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.

For this example it is very easy. First comes `ctor` and `init` from the first GBlock, then `param`and `updparam` from the second GBlock, then `stepO`, `updO`, `step`, `x`, `x2` and `upd` in this order from the third GBlock, and last `paramO` and the rest from the forth GBlock.

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.

1.5.4 The module's output

It may be possible to adapt the code generation that instead access to output variable any time an operation call for a *“getter”* is generated in the code, and hence executed with the core sources. This is if for example in C++ all instance variables are encapsulated as private. But often especially for generated code which follows stronger rules as manual written one, the immediate access to the variable in a data struct is desired. Then the special solution to call a function, not only a getter, really to execute a functionality may be desired. Such a function may have also input arguments and may have output values called by reference if more as one output is necessary. One output value is usual returned by value.

1.6 Possibilities of Graphic Blocks (GBlock)

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

Table of Contents

1.6.1 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, 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 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 with several graphic shapes (GBlocks). It means a class or a FBlock can be shown in different contexts, see also html / Basics-OFB_VishiaDiagrams.pdf: ***4.2 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 But sometimes this works not properly, then either “*Format – Clear direct Formatting*” on the shape helps, or Menu “*Format – Text Attributes*” and adjust it.

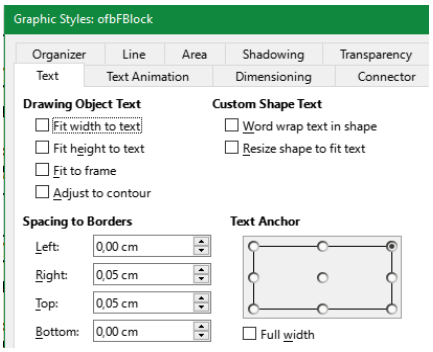


Figure 8: odg/ofbFBlock-TextStyle.png

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, 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.

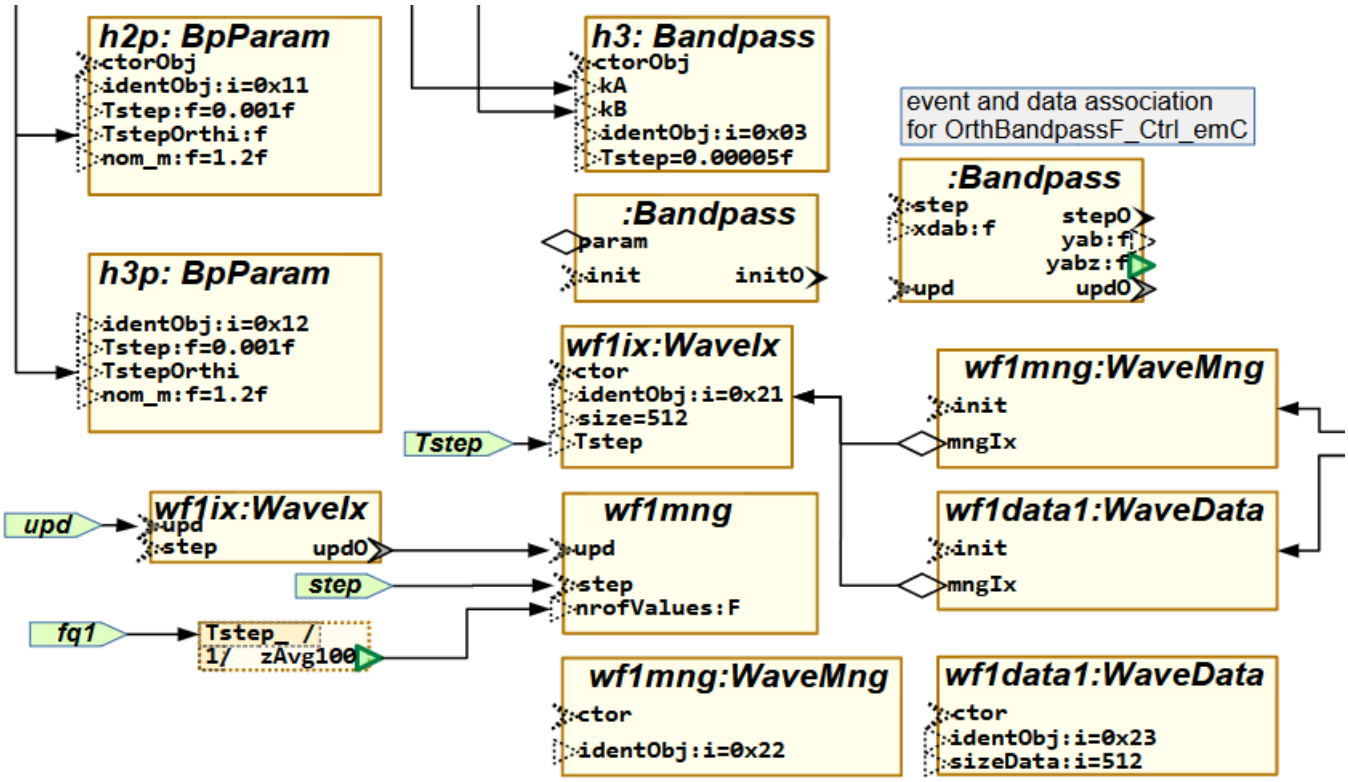


Figure 9: odg(ExmplFBlocksTypes.png

1.6.2 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 ***Error: Reference source not found Error: Reference source not found***

- 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,

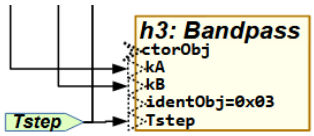


Figure 10: odg/FBlock_ctorObj.png

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 11: 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 etc.
- 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.

1.6.3 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 12: 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 ***Error: Reference source not found Error: Reference source not found***.

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.

1.6.4 Expression GBlocks

Expressions are elaborately described in the next chapter ***1.7 Expressions inside the data flow***. The difference between expressions and also the following described FBoper and FBaccess and ordinary FBlocks on the other side in the data flow 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 and FBaccess 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 UFGgl is, have only a small set of different block kinds, and use familiar textual notation of the pins to dedicated the operation.

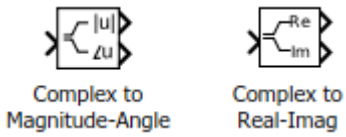


Figure 13: Simulink standard library blocks SmlikLibCplxMagnAngle_CplxReIm.png

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. The *Complex to Real.Imag* is a `ofbAccess` GBlock, anyway cheap in implementation, and the *Complex to Magnitude-Angle* should be offered by a specific FBlock in the users responsibility with a proper appropriate implementation.

1.6.5 How are expressions presented in IEC61499?

The IEC614499 does only know FBlocks and their types. Expressions are built from standard FBlocks. As presented in the chapter before, that is not the approach in OFB, instead describing expressions 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. For a common expression the expression type is the

```
FUNCTION_BLOCK Expr_OFB
EVENT_INPUT
  prep WITH expr, expr, X1999, K1999
END_EVENT
EVENT_OUTPUT
  prepO WITH y;
END_EVENT
VAR_INPUT
  expr : STRING;
  X1999 : ANY_NUMERIC;
  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. Separated with comma `,` it is first the kind of operation and the operators for the X… signals. After semicolon `;` the second section holds operators of the `K…` pins. The next section after ; can contain a mathematics or elsewhere given function to execute in the expression.

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 / Impl-OFB_VishiaDiagrams.pdf: ***1.1.5 FBexpr_FBcl: FBlock for expressions, presentation in FBlock_FBcl*** on page ***12***.

1.6.6 GBlocks for operation access in line in an expression - FBoper

See also html / Approaches-OFB_VishiaDiagrams.pdf: ***1.1 GBlocks, FBlocks and FBoper - what is a FBlock***

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 ***1.6.2 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):

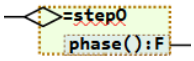


Figure 14: odg/FBoperGetter.png

The right figure shows a simple getter possible as part of an expression. The aggregation refers the proper FBlock, see also . The `=step0` means, that the operation (getter) can be called only after the `step0` output event of the referenced FBlock. It means the data to get are prepared after finishing the correspond step event. In ordinary textual languages such things are given by the line sequence (calling order). For graphical programming the events determines the order.

This getter ***FBoper*** can be used more as one time in the graphic. It is not an only repeated graphic presentation (due to html / Basics-OFB_VishiaDiagrams.pdf: ***4.2 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:

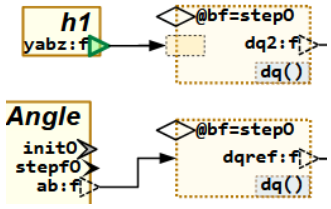


Figure 15: odg/FBoperInOut

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 ***1.8 Connection possibilities*** page ***52***).

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;
```

1.6.7 Data Access Blocks

Data can be arrays or also structured types. To set and access parts of it, without specific operations in FBlocks, a GBlock of style `ofbAccess` can be used.

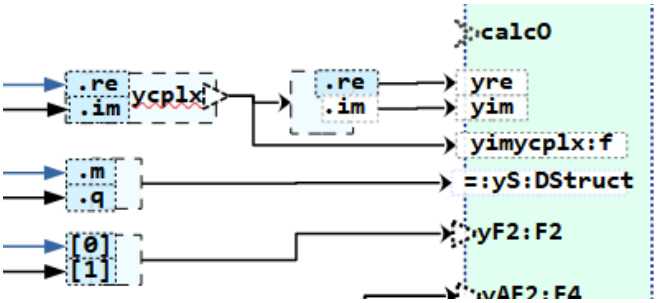


Figure 16: ofbAccessExmpl1.png

The image shows some variants of the data access. General the GBlock is of style `ofbAccess`. The pins of the FBlock are either `ofPin`, the common style for pins, or `ofpAccessElem`. The common `ofPin` can be used anywhere because the elements are detected as such as located in the `ofbAccess` GBlock. But the `ofpAccessElem` can have a more proper style. In the image above only the right side `.im` is styled with `ofPin`.

The access can be a get or set, depending of the outgoing or incoming data flow. The arrangement of the pins, left or right, or also on top or bottom in one line, does not play any role.

The text in the pins inside the `ofbAccess` describes either the array element which is/are accessed, or an element in a structured data type, then with starting dot.

Access to structure elements can also be written in a deeper struct with for example `.c1.re` as access to a complex variable `c1` to its real part named `re`. The type of structure elements are not forward propagated from the structure. It means they should be determined by its environment using the data type propagation (see chapter ***1.4.6 Data type forward and backward propagation*** or it can be declared also in the pin using the writing style `.access:Dtype`. If the type does not match you get problems in the generated code for compilation.

Access to array elements can be written with more dimensions with the pattern `[1,2]` for a two-dimensional array with access to element `[1][2]` in C/++ writing style. You can also access a whole sub array by writing lesser indices. Means `[1]` is the first row (whole second dimension) of a two dimensional array.

Also variable can be used for the indices, write `[ix,0]` to access the first element from 0 in the second dimension, and the ix-given part in the first dimension. Whereby the variable should be able to find inside the Variables of the Module as output of a FBexpr.

The type of array elements is automatically detected from the access.

1.6.8 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. 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:

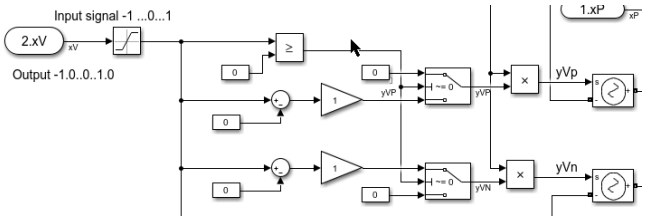


Figure 17: smlk/Exmp_if_switch.png

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;
}
```

The enabled and triggered subsystem are other specific blocks in Simulink for conditional operations: The internal function is only executed with a condition outside.

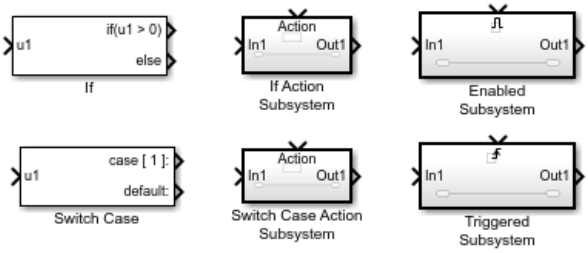


Figure 18: smlk/SmlkLibCondFBlocks.png

The image above shows some specific *'Subsystems'* for conditional operations.

In the OFB graphic with its event orientation the conditional execution (if-else-construct) is simple:

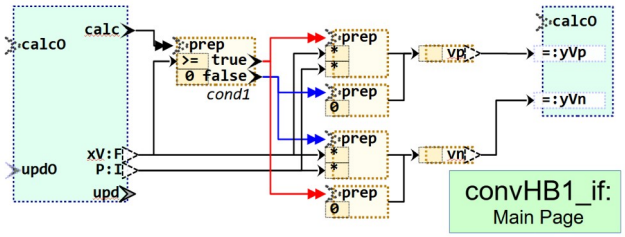


Figure 19: OFB/exmpTrueFalse.png

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.

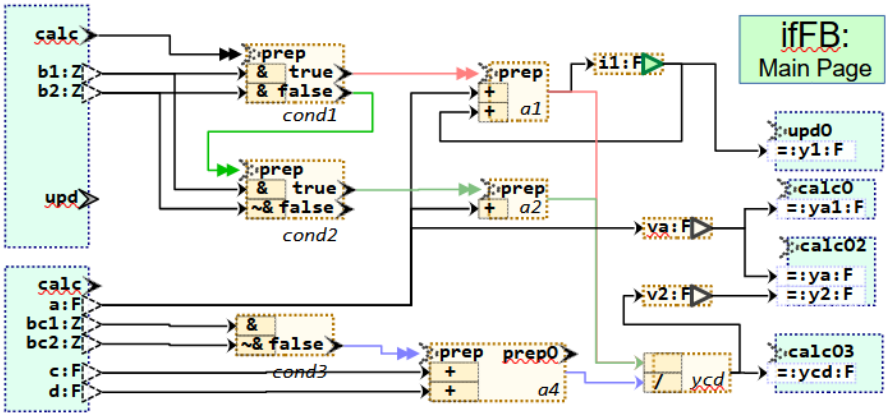
The event flow is evaluated as following:

```
EVENT_CONNECTIONS
calc TO cond1.prep
  cond1.true TO d_4.prep
    d_4.prepO TO vp_X.prep
      vp_X.prepO TO vp.prep
        vp.prepO TO JOIN_calcO.J1
  cond1.true TO d_6.prep
    d_6.prepO TO vn_X.prep
      vn_X.prepO TO vn.prep
        vn.prepO TO JOIN_calcO.J2
  cond1.false TO d_3.prep
    d_3.prepO TO vn_X.prep
      vn_X.prepO TO vn.prep
        vn.prepO TO JOIN_calcO.J2
  cond1.false TO d_5.prep
    d_5.prepO TO vp_X.prep
      vp_X.prepO TO vp.prep
        vp.prepO TO JOIN_calcO.J1
JOIN_calcO.J TO calcO
upd TO updO
END_CONNECTIONS
```

This event connections shown in that kind in the fbd file documents also the code generation order. The generated code is similar as shown above.

But this is not the only one possibility of condition. It may be more complex:

Figure 20: 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

  thiz->va = a;  //calc --> va_X.prep  genEx
  //
  //Module outputs due to the event calcO:
  thiz->mEvout_calc |= MASK_calc_calcO;  //
  thiz->ya1 = thiz->va;    // otx: setMdlOut
  //
  if(cond2 && !cond3) {        // otx: exprC
    //Module outputs due to the event calcO2
    thiz->mEvout_calc |= MASK_calc_calcO2;
    thiz->ya = thiz->va;    // otx: setMdlOu
    thiz->y2 = thiz->v2;    // otx: setMdlOu
  }  //Condition Bits
}
```

1.6.9 Sliced and Array FBlocks

In FBlock graphics usual one GBlock (graphic Block) is one FBlock. But also Simulink knows a "slicing". It means there, a submodule of type

For simple Expressions, Slicing is not a specific effort, both in Simulink as also in OFB draw graphic. Look on the two examples:

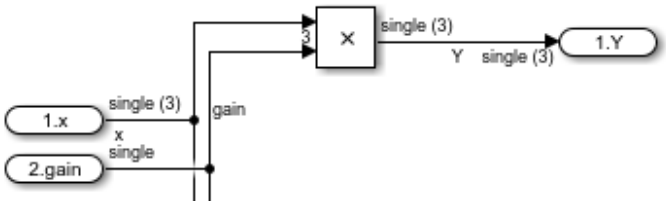


Figure 21: smlk/Exmp_Multiply_Vector_Scalar.png

This is Simulink. The Multiplier above calculates a float[3] vector with a scalar gain, resulting again a float[3]. The graphic detects automatic the scalar of one of the inputs.

The same is done in OFB graphic:

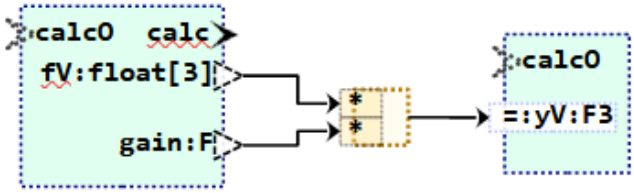


Figure 22: OFB/Exmp_Multiply_Vector_Scalar.png

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 usage of a specific FBlock type which does not deal with vectors (arrays). Simulink has the solution of a *"For Each Subsystem"*, looks like:

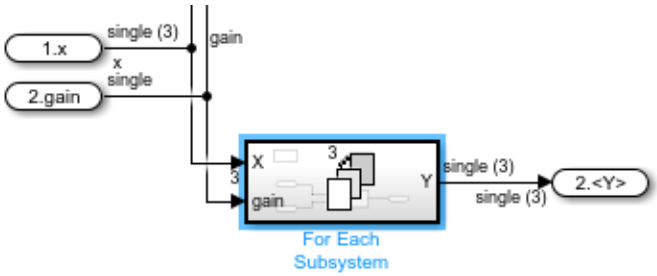


Figure 23: smlk/Exmp_Multiply_Vector_Scalar.png

Intern a FBlock which can only deal with scalars is used:

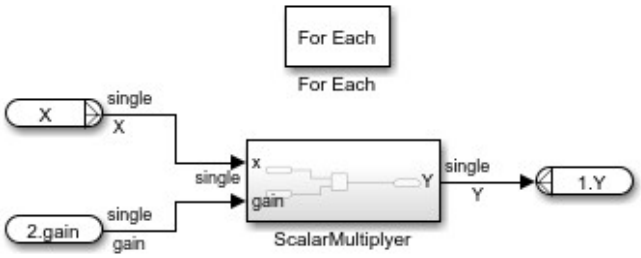


Figure 24: smlk/Exmp_ForEachSub_InnerScalarMult.png

This specific "Subsystem" has for-each pins, which are outside vectors as shown in

(empty page)

1.7 Expressions inside the data flow

Table of Contents

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 `ofcConn` for input and output, whereby the input connector can have a text for the expression.

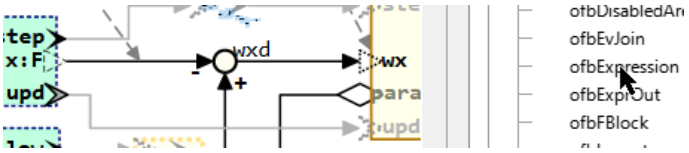


Figure 25: odg/ExpressionExmp.png

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.

1.7.1 Expression parts as input

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 `ofbExprPart` 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.

They can contain operators and also a factor as constant or as variable. The basic form to add and sub is:

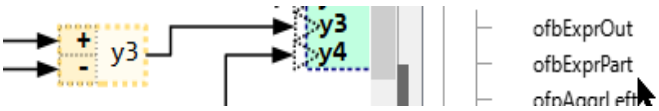


Figure 26: odg/ExpressionExmp.png

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:

- `+ -` numeric ADD FBexpr with unary operator `-` possible.
- `* / %` numeric MULT (DIV, Modulo) FBexpr with unary operator – possible. The `%` is the modulo operator. A FBexpr with only a `/` operator builds the reciprocal from the input

For both operators, the inputs can be modified by an additional operation written textual in this `ofbExprPart` box, see following ***1.7.2 More possibilities of DinExpr***.

- `&` boolean or bit wise AND, with unary operator `~` possible 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 boolean and bits for an exclusively OR.
- `<< >>` Bit shift operators.
- `== != <> < <= > >=` For numeric, boolean or bit wise comparison, with unary operator `~` or `-` possible for bit wise negate or numeric negate. More as one inputs can be used. `<>` 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:

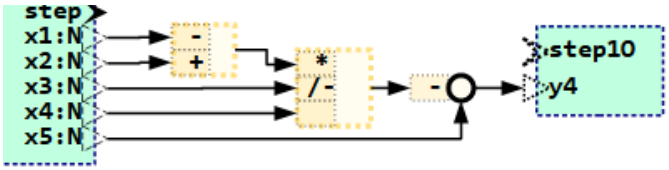


Figure 27: any image

The 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 one operand to divide. The `-` after `/-` is the unary `-` for the `x2` input. All of this should be intuitive understandable.

But to reinforce it look on a boolean example:

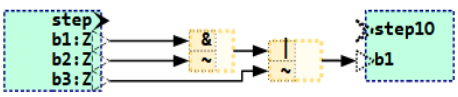


Figure 28: any image

This is

```
yb1 = (b1 & !b2) | !b3;
```

In C/++ Syntax. 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.

1.7.2 More possibilities of DinExpr

But there are more possibilities using `ofpExprPart`:

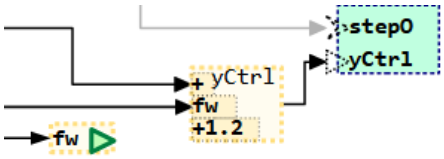


Figure 29: odg/ExpressionExmpK2const.png

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 model. It is wired to the `K2` input in the IEC61499 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.

1.7.2.1 Example with division, factors in Add expression and variables

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

Figure 30: 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.

1.7.2.2 Access to elements of the input connection to use

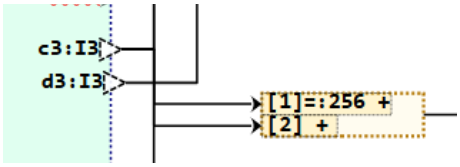


Figure 31: ExprInpArrayAccessMult256.png

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.

1.7.2.3 Description of all possibility, syntax/semantic of DinExpr

See also chapter html / Impl-OFB_VishiaDiagrams.pdf: ***1.3.7 Preparation of Expressions from odg*** page ***30***

Any part is optional.

The first input access possibilities are also possible on each Din on a FBlock, not only on an expression.

- `@fbSrc@pinSrc`: This is a textual connection. If the String starts with a `@` then it should not have a connection. Instead the connection is given textual. A module pin is named with `@pin`. A module variable is named as `@varName`. If the pin has also a connection, it's the same as twice connections and causes an error message on twice driven inputs. But you can use the pin connection for an outgoing connection to another input.

- `@fbSrc[1]@pinSrc`: The index is regarded to a sliced FBlock, See TODO.

- `.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]`.

- `:valueCast` this is a type casting, as last operation of the access description before the `=:`. The given data type can be one of the standard types see chapter ***1.4 Data types*** page ***14***, 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. If a `:valueCast` is used, the input type on the connection is free, determined by the input, not tested.

- `=:` 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.

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).

- 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 ***1.7.5 Set components to a 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 ***1.7.1 Expression parts as input*** 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()`)

1.7.2.4 Some examples for DinExpr

TODO

1.7.3 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:

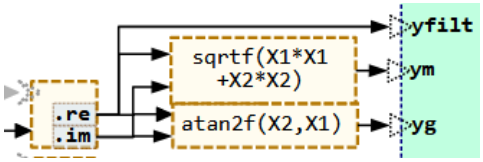


Figure 32: odg/ExprAnyX1X2.png

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 ***1.7.9 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.

1.7.4 Output possibilities

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.

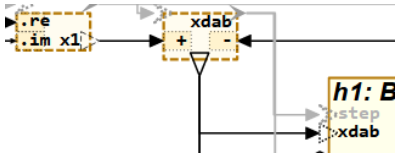


Figure 33: odg/ExprOutpin.png

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 html / Basics-OFB_VishiaDiagrams.pdf: ***4.6 Storing the textual representation of UFBgl 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:

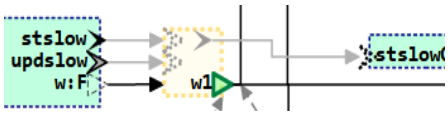


Figure 34: odg/ExprOutStateUpd.png

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 ***1.7.1 Expression parts as input***. 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.

1.7.5 Set components to a variable

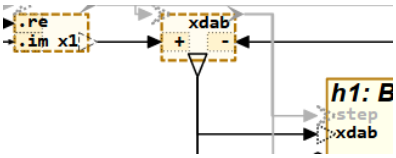


Figure 35: odg/ExprOutpin.png

Input is .re or such or also [1] or [index].

The output must be a variable. The type should proper to the input descriptions. Simplest case: complex, type given with `:f`, `:d` or also an array given with `:F3` as float array or also `:f3` as complex array. More possibility use a structured type whereby the structure should be defined in the target language (in C in header file). `:structType` see ***1.7.8 Type specification in expressions***

Generally variables as expression output can be drawn more as time. If the expression has no input, then this variable can be accessed, not set. If the expression is this kind of set a component, different components 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 equal.

1.7.6 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 ***1.7.3 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 36: 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.

1.7.7 FBexpr as data access

If you look at the you see on input `.re` and `.im`. This expression needs an output variable, which collects the real and imagine part and delivers a complex value.

The opposite expression is

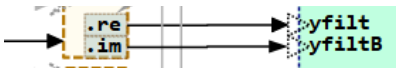


Figure 37: odg/ExprOutReIm.png

Here the outputs are drawn in graphic style `ofpExprOut` with internal text starting with the dot. On access (without output variable) from the input the adequate part, here from the complex value, is accessed.

The same as for `.re` and `.im` can be done for elements of an array. The collect (on the `ofpExprPart`) and the access (on the `exprOut`) 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. The size of the array variable on a collect expression should be dedicated, given with the type specifier, see next chapter.

1.7.8 Type specification in expressions

In the texts of the expression inputs and outputs (`ofpExprPart`, `exprOut` and also the pins on output `ofpDout`..., `ofpVout`... `ofpZout…` the text on the pin can contain a `:<:n:….>:Type` as suffix. This can be written after a variable name (for the out pins) as also for all other possibilities for the expression part and output. The type designation follows chapter ***1.4 Data types***. The types should be semantically sensible. In this kind the size of an array can be defined, see example:

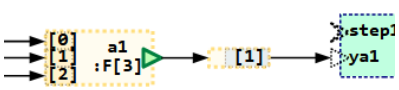


Figure 38: odg/ExprArray.png

Here the text to the output is wrapped, this is not important. But it ends with `:F[3]`, means it is a `float[3]` array in C/++ or also Java language. The right expression then accesses the element 1.

1.7.9 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 ***1.6.6 GBlocks for operation access in line in an expression - FBoper*** on page ***30***

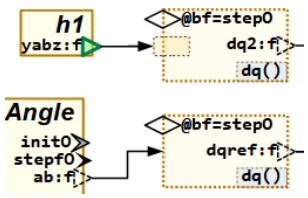


Figure 39: odg/FboperInOut.png

1.7.10 FBexpr fblock types

An `ofbExpression` is also a FBlock, with a specific FBlock type. The difference to a non `ofbFBlock` shape is, there is no library FBlock behind. The target code is built only with the given graphic information of the `ofbExpression` GBlock.

But there are fundamental types of FBexpr which are present by internal standard FBtype instances. The types are able to see in the

```
FBS
  ...
  y3_X : Expr_FBUMLgl( expr:='.+,+,+;,,;;' )
  ...
```

inside the written fbd file (or also possible input as IEC61499 syntax file). The types are automatically set depending on input or output designation.

- `Expr_FBUMLgl`: This is a simple expression, which has the same types on all input and output pins. The expression has not an output variable, hence it is evaluated in line in the target code.

- `ExprCplx2ReIm_OFB`: This is an expression which needs a variable on its only one input of a complex type. The output(s) should be an `ofbExprOut` with access string `.re` or `.im` to access the real and imagine part. The numeric data types are the same, but the input is complex, the outputs are real.

- `ExprReIm2Cplx_UFB`: This is an expression which needs a variable on its only one output of a complex type. The input(s) should be an `ofbExprPart` with access string `.re` or `.im` to set the real and imagine part. The numeric data types are the same, but the output is complex, the inputs are real.

- `ExprArrayAccess_UFB`: This is an expression which needs a variable on its only one input of an array type. The output(s) should be an `ofbExprOut` with access string `[1]` to access an element of the array. The basic numeric data types are the same, but the sizeArray designation is different. The outputs can be also arrays, if a whole segment of a multi dimensional array is selected.

- `ExprSetArray_UFB`: This is an expression which needs a variable on its only one output of an array type. The input(s) should be an `ofbExprPart` with access string `[1]` to access an element of the array. The basic numeric data types are the same, but the array size designation is different. The inputs can be also arrays, if a whole segment of a multi dimensional array is set.

- `ExprStructAccess_UFB`: This is an expression which needs a variable on its only one input of a user defined type, which should be a `struct` or similar in the target language. The output(s) should be an `ofbExprOut` with access string `.name` to access an element of the `struct`, or also invoke a getter operation with the given name, depending on the code generation. The data types should be determine independently for all inputs and outputs, depending on the given `struct`.

- `ExprSetStruct_UFB`: This is an expression which needs a variable on its only one output of a user defined type, which should be a `struct` or similar in the target language. The input(s) should be an `ofbExprPart` with `.name` to set the named element of the `struct`, or also invoke a setter operation with the given name, depending on the code generation. The data types should be determine independently for all inputs and outputs, depending on the given `struct`.

1.7.11 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)

1.8 Connection possibilities

Table of Contents

1.8.1 Pins

The pin appearance does not play any role for the interpretation and converting of the graphic, but it is essential for manual view. For interpretation the associated style is essential.

The first idea for OFB was, using one pin style which is proper for appearance, and defining several styles for the connection kinds between pins (aggregation, composition, data or event flow etc). Then the connector style determines 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 or just a type, class, FBtype.

Hence, the sometimes existing `ofRef…` or `ofc…` styles aren't used for content semantic, only for appearance. All styles for connectors between pins are the same for functionality, only different in appearance. But styles of connectors between the whole Graphic blocks are used, see ***1.2.4 Connector styles, ofc*** page ***11***

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:

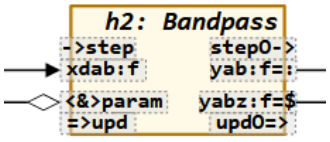


Figure 40: odg/FBpin_ofPinOnly.png

The pin kind designations are described in ***1.2.4 Connector styles, ofc*** page ***11***. 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:

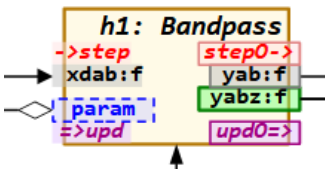


Figure 41: odg/FBpin_of.png

Here all pins have its proper specific style `ofpEvin` etc. but not the `ofp...Left` and .`ofp...Right` style. It is applied also to text fields. The text, background and frame is colored. Red is for events (following IEC61499 for diac). Outputs have borders. Inputs have no frames. The aggregation is blue, with dashed frame. Here the diamond symbol on the connector type `ofcAggr` is helpful for viewing, but not necessary for graphic evaluation.

The pin kind designations are not necessary, but here given for the event pins. If they are given and non proper to the pin kind, an ERROR is shown on evaluation of the graphic. It means it can be written also with the proper pin style.

If you do not like colors for the styles, because colors may be used for other things (mark functionality), the appearance of the styles can be changed to gray and black. If the meaning of the pin is still understandable, by naming, positioning etc, then it is ok. You can additionally use the pin kind designations. Then for example the `param` pin is gray with maybe dashed line, determined as aggregation by the `ofpAggr` style, and additionally for the user view it is obviously that it is an aggregation because of using the diamond in `ofcAggr` style for the connection.

The third variant for the pins are small figure as shown in the next image:

Figure 42: odg/FBpin_ofp.png

This may be the best viewable form. The aggregation have a figure as a diamond, as known from UML. Events are similar an arrow, because determining the execution flow. Data pins are triangles in arrow from determining the direction.

To get the figures you can pick up them from the template or other existing odg modules. Of course you can drawing also your own forms. The style assignment is only essential.

The texts are written outside of the figure, left side for right side pins and right side for left side pins. You can also rotate the shape, adequate. That is the reason to have styles for `odp...Left` and `odp...Right`. Sometimes it is necessary to insert a leading or trailing space to have a distance, her for `param`. This is possible and does not influence the graphic evaluation, spaces are trimmed.

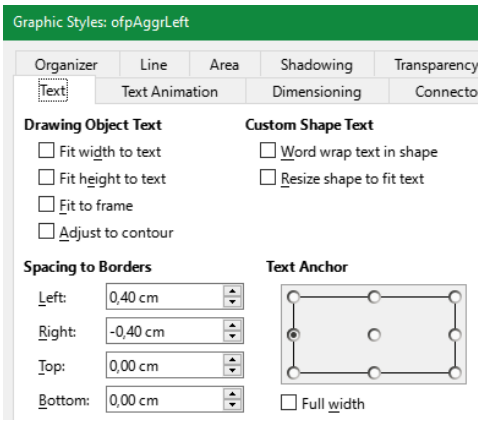


Figure 43: odg/Fbpin_ofpStyleText.png

The figure above shows the necessary settings to place the text right side to the shape of length 0.4 cm.

1.8.2 Connectors

It is very simple to draw a connector from an output to an input using the

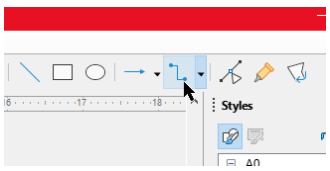


Figure 44: odg/Connector-Icon.pdf

The used `Default Drawing Style` is sufficient for the pin connections. For connections between FBlocks and FBtype blocks (without instance name) the proper ofc… styles should be used, see ***1.2.4 Connector styles, ofc*** page ***11***.

It is also interesting to have a line connector:

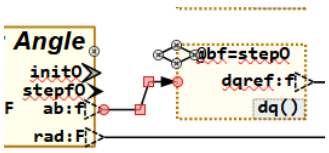


Figure 45: odg/LineConnectorExmpl1.png

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.

1.8.3 Connection points

One fast usable possibility is to organize the connectors from the source with proper positioning:

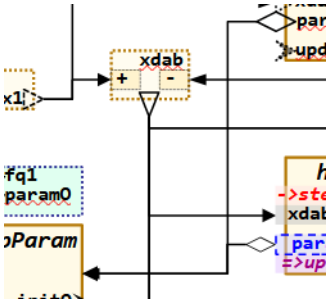


Figure 46: odg/LineConnectorExmpl1.png

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:

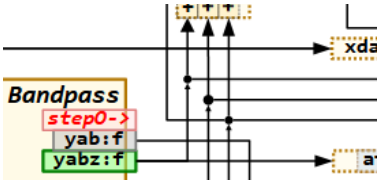


Figure 47: odg/ConnectionPoints1.png

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 html / Basics-OFB_VishiaDiagrams.pdf: ***3.4 Connectors of LibreOffice for References between classe*** page ***11***

1.8.4 Xref

This is already described in html / Basics-OFB_VishiaDiagrams.pdf: ***3.6 Diagrams with cross reference Xref*** page ***13***. 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.

1.8.5 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:

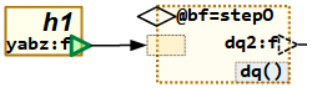


Figure 48: odg/ConnectionFromFBlockOut.png

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:

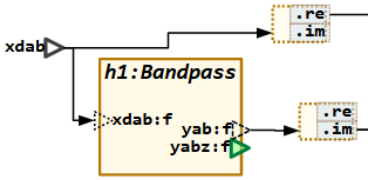


Figure 49: odg/ConnectionFromVariable.png

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 ***13***

1.8.6 Textual given connections

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

Figure 50: 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.

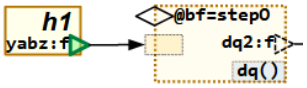


Figure 51: odg/ConnectionFromFBlockOut.png

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 `=step0` is the here necessary designation of an event order, see ***1.6.6 GBlocks for operation access in line in an expression - FBoper*** page ***30***

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

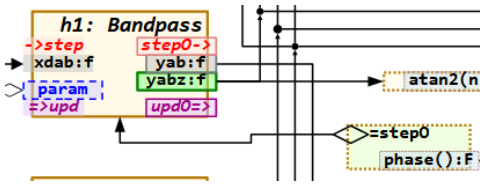


Figure 52: odg/FBoperGetterAggrConn.png

Here the `h1` FBlock is aggregated and shown immediately in the graphical context.

1.9 Execution order, Event and Data flow

As also explained in chapter ***1.6.2 GBlocks for each one function, data – event association*** page ***26***, events are associated to the data. In chapter html / Basics-OFB_VishiaDiagrams.pdf: ***4.5 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:

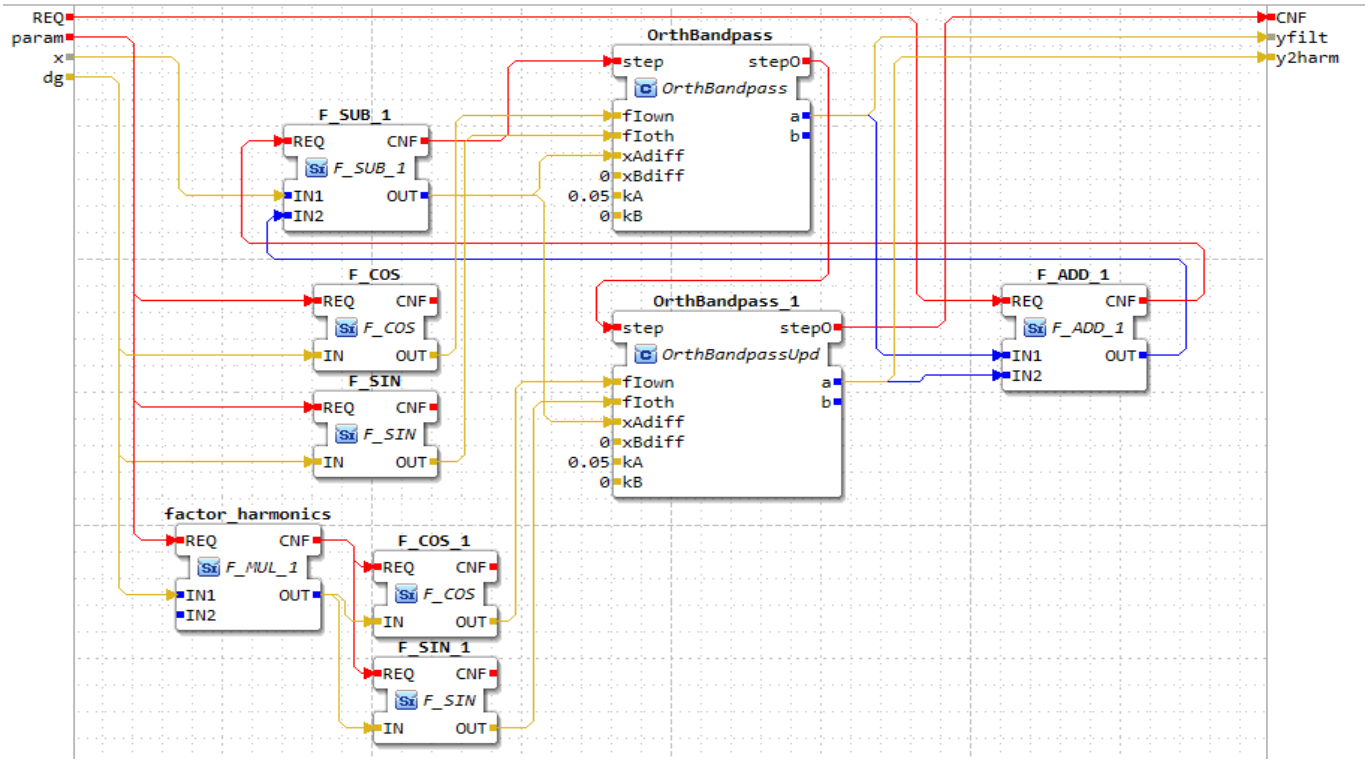


Figure 53: 4diac/OrthBandpassFilterAppl.png

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.

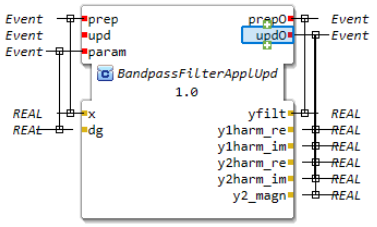


Figure 54: 4diac/OrthBandpassFilterApplUpd_ifc.png

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.

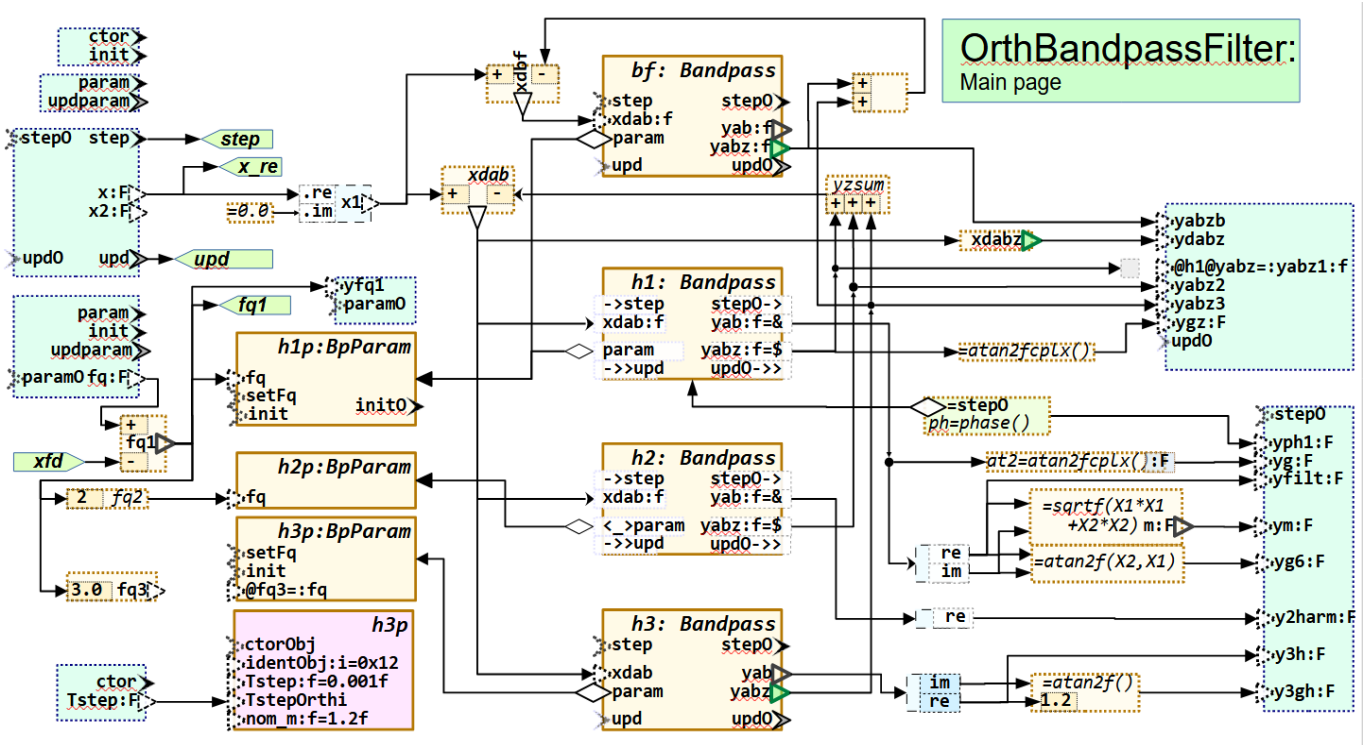


Figure 55: odg/

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 ***1.7 Expressions inside the data flow*** page ***38*** 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 ***Error: Reference source not found Error: Reference source not found*** page ***Error: Reference source not found***. 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”*.

1.10 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 page)

1.11 Drawing and Source code generation rules

Table of Contents

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

1.11.1 Writing rules in the 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 ***Error: Reference source not found Error: Reference source not found*** 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 leading `_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 spedific names determined by the evin of the FBlock.

1.11.2 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);

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 (...) {
prep_FB1(&dataFB1, ... &FB1, &FB2)
prep_FB2(&dataFB1, ... &FB1, &FB2)
upd_FB1(&dataFB1, ...)
upd_FB2(&dataFB2, ...)
```

As you see, first all ***preparations*** are done for new states, using the current ones. Then ***update*** the new states to the current ones comes for the next step. This is similar also of D and Q on Flipflops in digital logic.

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.

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

See chapter html / Basics-OFB_VishiaDiagrams.pdf: ***4.5 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.

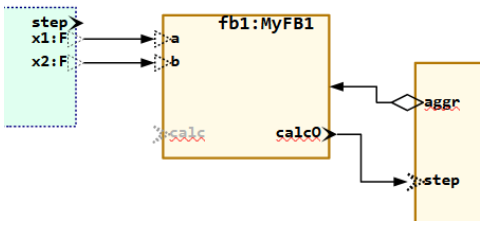


Figure 56: ExmplEvDeflt_calcOstep.png

Look for a not simple but should be obvious example in

- The both input values `x1` and `x2` are associated to a module input event `step`, usual the module gets a `step_..(..., float x1, float x2)` operation.

- The `fb1` has a named output event `calc0`. Hence for the input variables the input event, here drawn in gray as not active, is `calc`. The called operation is `calc_MyFB1(…)`. If the FBlock would not have any event designation, a `prep` event will be created as default.

- But notice, that an event – data association can also be drawn on another position of the graphic, proper to the rule "*Any element of the functionality can be shown more as one time in different contexts*" described in chapter html / Basics-OFB_VishiaDiagrams.pdf: ***4.2 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 `calc0` does not influence the input data association between `calc0` is an output event.

- For this example it is shown in the graphic that a called `calc_...(fb1...)`operation is followed by a `step_..(fb2...)` operation of the next FBlock because this is dedicated by the here shown event connection. In this special case the `fb1` has no data output which should elsewhere determine the calculation order (or just event connection). Hence it should be dedicated by the drawn event connection.

- The aggregation from the second `fb2` to the `fb1` needs an initialization. For that both FBlocks gets an `init` → `init0` event pair per default (as nowhere other it is dedicated in another way, just as default). The own address of the `fb1` as "port" output is related to the `init0`event, and the aggregation is related to the `init` event of the right FBlock.

- And also for construction a `ctor` and a `ctor0` event is associated to all FBlocks which are not expressions.

With this simple rules the code generation from OFB to C language in the default version (can be adapted, see TODO) is compatible with your basic function blocks in C language.

Then you don't need specific extra definitions outside of the Libre/Open Office graphic.

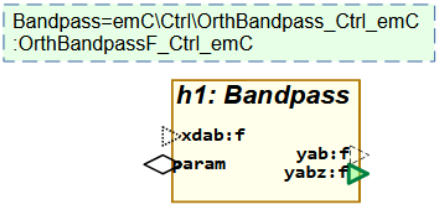


Figure 57: FBlockSimpleUsage.png

This is the only necessity in the graphic to use it together with the existing code in C/++ language:

- The green box is of style `ofbImport` and declares the alias `Bandpass` in the graphic as full Module type `OrthBandpass_Ctrl_emC` which is the module's name in C language (see ./../../../vishia/emc/html/Ctrl/OrthBandpass.html (http://www.vishia.org/emc/html/Ctrl/OrthBandpass.html).

- The input events `step`, `init`, `ctor` and the output events `step0` and `init0`, are automatically created because here events are not defined.

- Because at least one output with the graphic style `ofpZout...` is given, also the input event `upd` and the output event `upd0` is automatically defined.

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

associated to `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.

1.11.4 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)

1.12 Converting the graphic – source code generation

As fast mentioned also in chapter html / Basics-OFB_VishiaDiagrams.pdf: ***4.7 Source code generation from the graphic***page , 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.

FBcl/UFBglConvAndTestSlide.png :: id=__Img_FBcl_UFBglConvAndTestSlide ::title=Figure 49: Fbcl/UFBglConvAndTestSlide.png. :: style=ImageCenter :: size=18.0cm*10.82cm.>

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)

1.12.1 Calling convension with code generation

The code generation from Open/LibreOffice odg files can be performed with:

```
@REM This file is the batch file to call java and similar the argument file.
cls
if not exist ..\cpp\genSrc mkdir ..\cpp\genSrc
if not exist ..\fbcl mkdir ..\fbcl
@REM use --@file:label, the file is this file itself as %0
java -cp ../../../tools/vishiaBase.jar;../../../tools/vishiaUFBgl.jar org.vishia.fbcl.UFBglConv
--@%0:args
@echo off
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 of the
argument,
REM one space and ## after the args label defines remove trailing spaces and remove comments
after this ##
REM --- is a commented argument for the java main routine
::args ##
::---dirStdFB:src/libModules_fbd/fbd
::---codeTpl:d:\vishia\fbg\source.wrk\src\srcJava_vishiaFBcl\java\org\vishia\fbcl\translate\
cHeader.txt
::-dirGenSrc:../cpp/genSrc
::---dirCmpGenSrc:src/ExmplGenSrc/cmpGen
::-dirFBcl:../fbcl
::---dirCmpFBcl:src/ExmplGenSrc/cmpGen
::-dirDbg:../fbcl/dbg ## output directory for some log files for data debugging
::---ifbd:path/to/file.fbd ## for a inner module
::---ifbd:path/to/othermodule.fbd ## can be given more as one
::-odg ## writes an file.odg as inner data presentation
::-oxmltest ## possibility to write back the read content.xml
::---oxmldatahtml
::---datahtml ## possibility to write the internal data in html
::-i:../odg/MyExample.odg ## The input odg file to translate
pause
```

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

The input file is the last argument after `-i:`. 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. Also other modules can be read.

For used modules the rule is: First name the used module, then the using module in the `-i:` argument. Then the using module can participate on the existing definition of the used module. Elsewhere some default mechanism are effective, if the used module is not full specified while using, and this will be seen in a not proper code generation.

Especially files 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 can/should be used, see TODO later.

The extension of the `-i:` file determines how to read it. `.odg` is OpenLibreOffice, `.fbd` is a IEC61499 file. `.slx` should be for Simulink (yet TODO), all other graphic sources should/can be translated adequate.

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.

The three `-dirGenSrc:` `-dirFBcl:` `-dirDbg:` 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.

The three directories `-dirCmpGenSrc:` `-dirCmpFBcl:` `-dirCmpDbg:` are only for internal test to compare results with given files after code changes (test evaluation).

The `-codeTpl:` option (possible more as one) describes paths to otx files (OutTextpreparer) for code generation. If this argument is not given, internally files for C code generation are used. See next chapter.

- 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 `-dirDbg:` 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.
- 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 ***Error: Reference source not found Error: Reference source not found*** page (Java internals) in a readable html file.

1.12.2 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.

2 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;
}
```

```
##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

more

also C or C++ language `cppOperation()` given

and

a part of a otx: `<:otx: VarV_UFB:`

A nomination of a style, this is `cOtx` or just `CodeOtx`.

A `Marker` with style `cM` should be demonstrative

wait `what is` `cV`?

and `VARIABLES` in a IEC614499 source

Docu file: Approaches-OFB_VishiaDiagrams

Docu file: Basics-OFB_VishiaDiagrams

Docu file: Impl-OFB_VishiaDiagrams