Capabilities and Concepts OFB

OFB (Object-Oriented Function Block) is a graphical modelling approach and description language for the development of embedded control software. It combines object-oriented structures with a function-block-based graphical representation to define application behaviour at a higher level of abstraction.

Models are created using LibreOffice Draw, which serves as the graphical editor. A Java-based translator extracts the model information from the LibreOffice documents, processes and organises it, and generates source code.

The primary target language is C; however, the translation architecture is designed to be extensible to additional output languages. Support for e.g. VHDL is considered as viable option for FPGA - oriented development workflows.

This chapter provides a technical overview of the OFB concept and its capabilities. It corresponds to Chapter 4 of the complete documentation; all cross-references relate to the full document set.

Written by Dr. Hartmut Schorrig & Bernhard Horn, 2026-03-22

Table of Contents

4 Capabilities and concepts of OFB diagrams

Table of Contents

4.1 Modules and Files

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

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

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

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

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

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

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

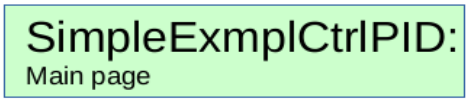


Figure 0: og/ofbTitle-1.png

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

Modules can also be organised into reusable libraries.

4.1.1 Module as Library – Definition of FBtypes

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

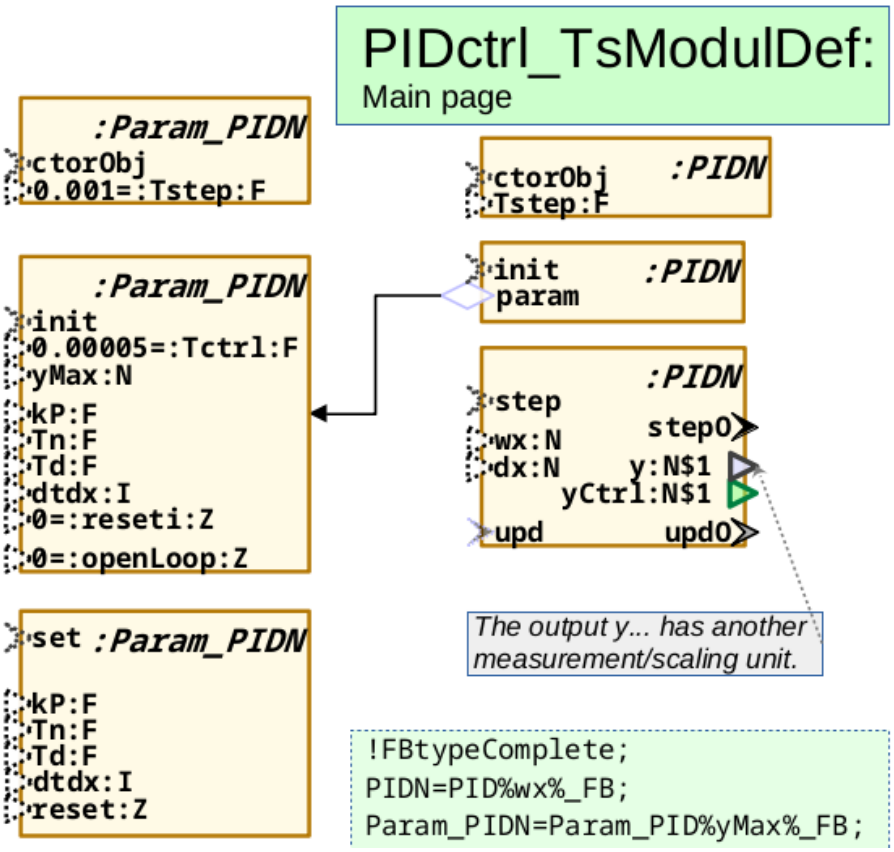


Figure 1: ExmplCtrl/LibCtrl_PID-FBtype.png

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

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

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

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

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

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

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

4.1.2 Module as Functionality - with Interface and Content

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

In particular, the handling of event inputs/outputs and event flow is described in *4.4.3 Event Determination from Data Flow (OFB Approach)* page *11* which even use this example and explains data flow, event synchronisation, and step timing, including the mechanisms required to ensure data consistency.

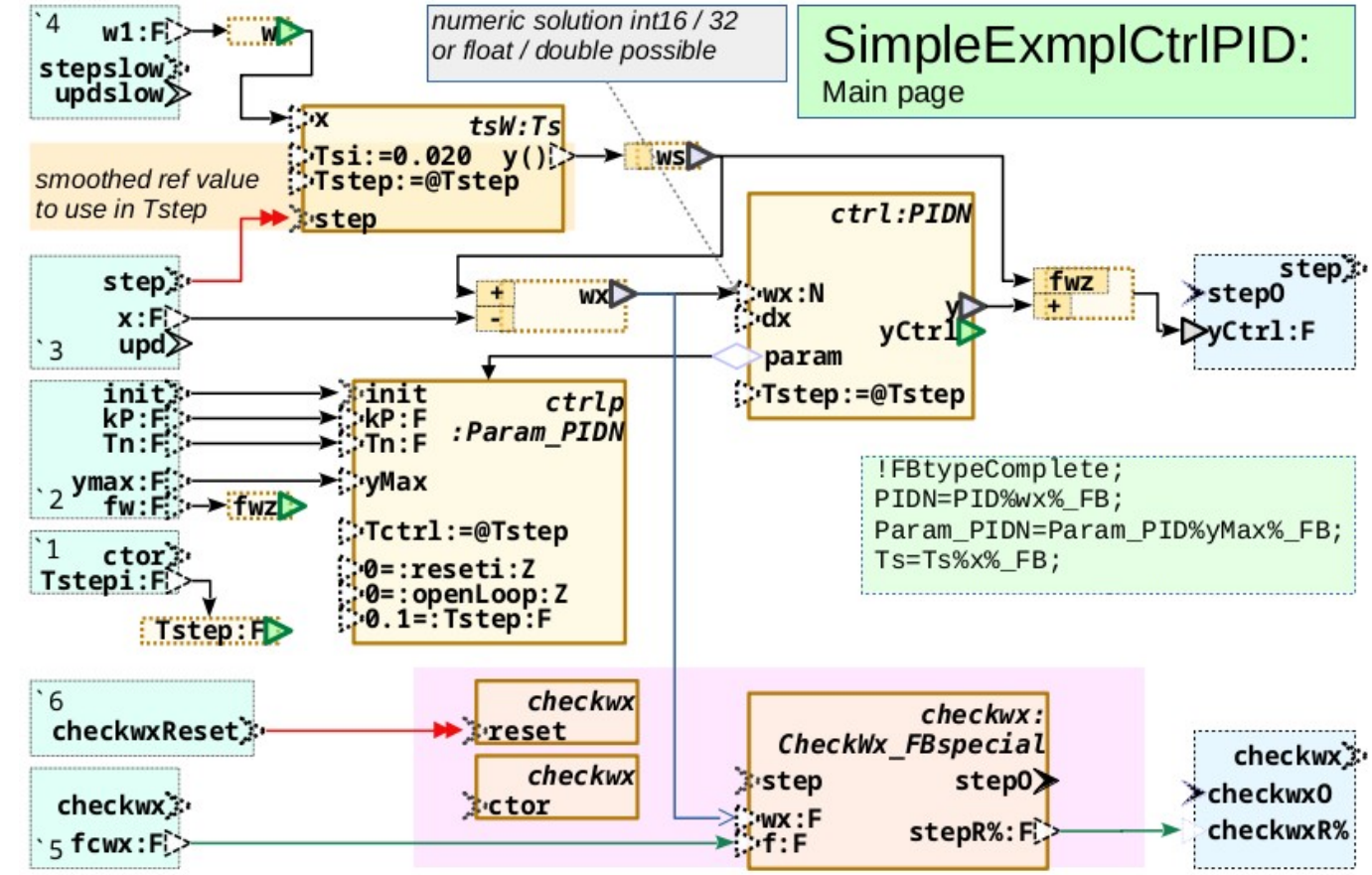


Figure 2: ExmplCtrl/SimpleExmplCtrlPID.png

Key structural aspects of a module:

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

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

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

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

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

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

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

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

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

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

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

4.1.3 Graphic Blocks, Pins, Texts and Connections

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

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

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

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

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

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

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

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

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

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

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

4.2 Additional FBlock capabilities

4.2.1 Multiple View of the same FBlock in Graphic

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

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

Key rules:

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

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

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

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

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

4.2.2 Aggregations and Associations Between FBlocks

OFB supports UML like structural relations:

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

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

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

4.2.3 Vectorisation and Sliced FBlocks

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

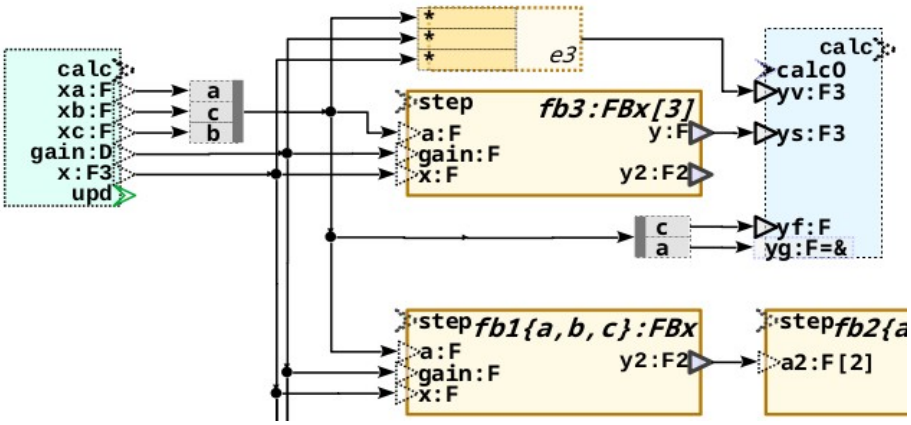


Figure 3: OFB/ArraySlideDemux_Vector_FBlock.png

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

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

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

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

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

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

4.2.4 Built Variants in Modules

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

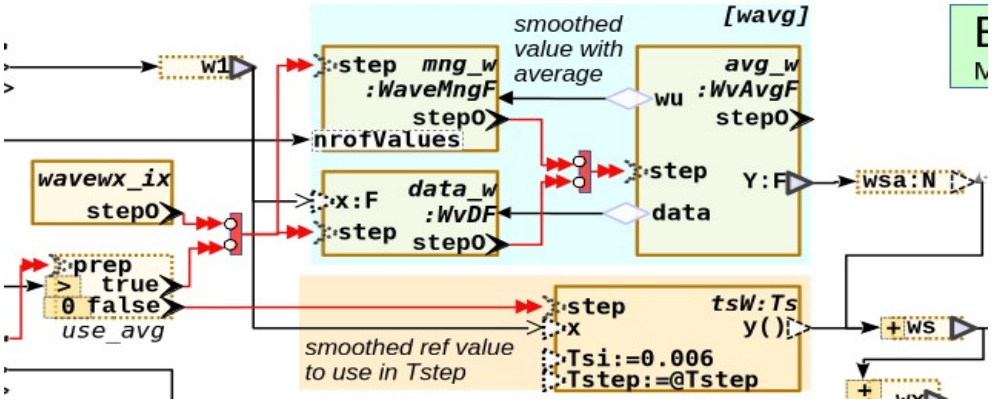


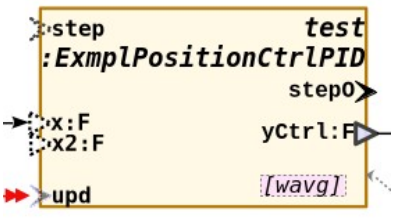
Figure 4: ExmplCtrl/VariantSelPIDctrlAvg.png

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

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

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

Figure 5: ofpOtion-avg-on-PICctrl.png



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

4.3 Graphical Diagram translation to target code

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

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

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

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

The translation separates following steps:

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

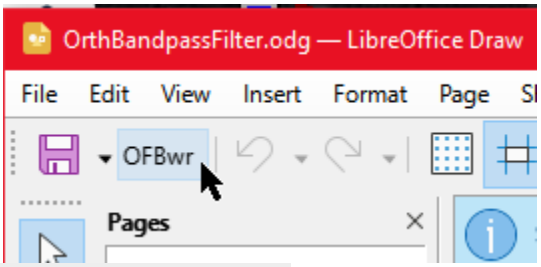


Figure 6: LOffc/IconsTop2.png

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

4.3.1 Read the Graphic Structure

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

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

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

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

4.3.2 Save as Functional Model (FBcl) - IEC61499 compatible

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

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

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

Rationale for IEC 61499 usage:

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

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

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

4.3.3 Template-Based Code Generation

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

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

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

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

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

The generated part looks like

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

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

4.3.4 Work flow till test on target hardware

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

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

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

4.4 Event-Based Execution

4.4.1 Motivation

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

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

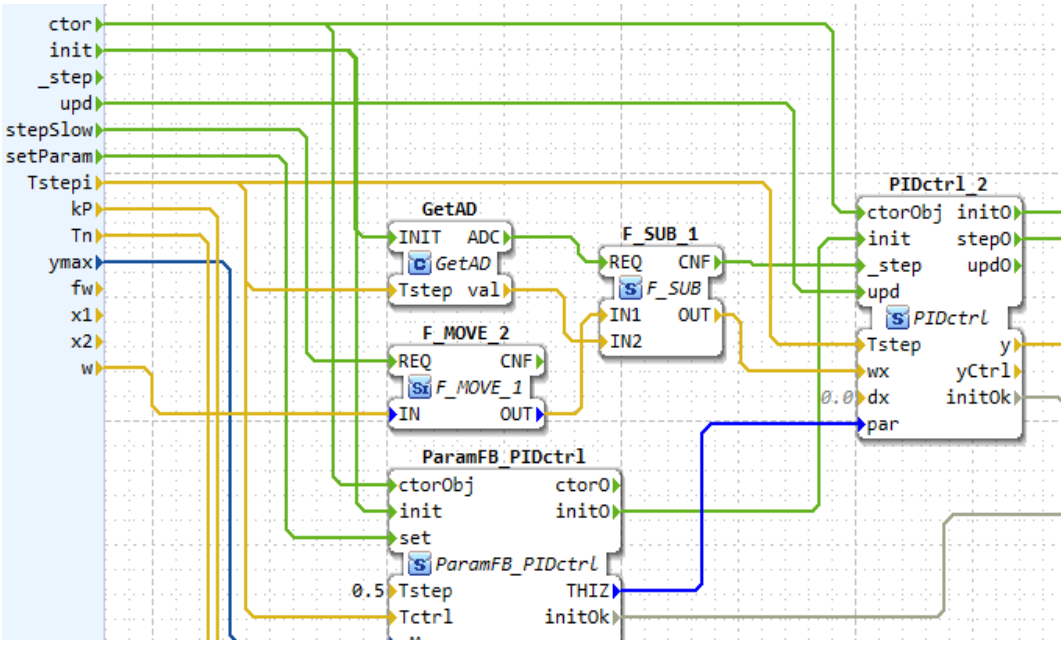


Figure 7: ExmplCtrl/4diac_ExmplGetADCtrl.png

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

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

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

Events therefore generalise cyclic execution rather than replacing it.

4.4.2 Advantages of Event Orientation

Event-driven FBlock execution enables:

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

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

4.4.3 Event Determination from Data Flow (OFB Approach)

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

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

This significantly reduces diagram complexity without losing determinism.

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

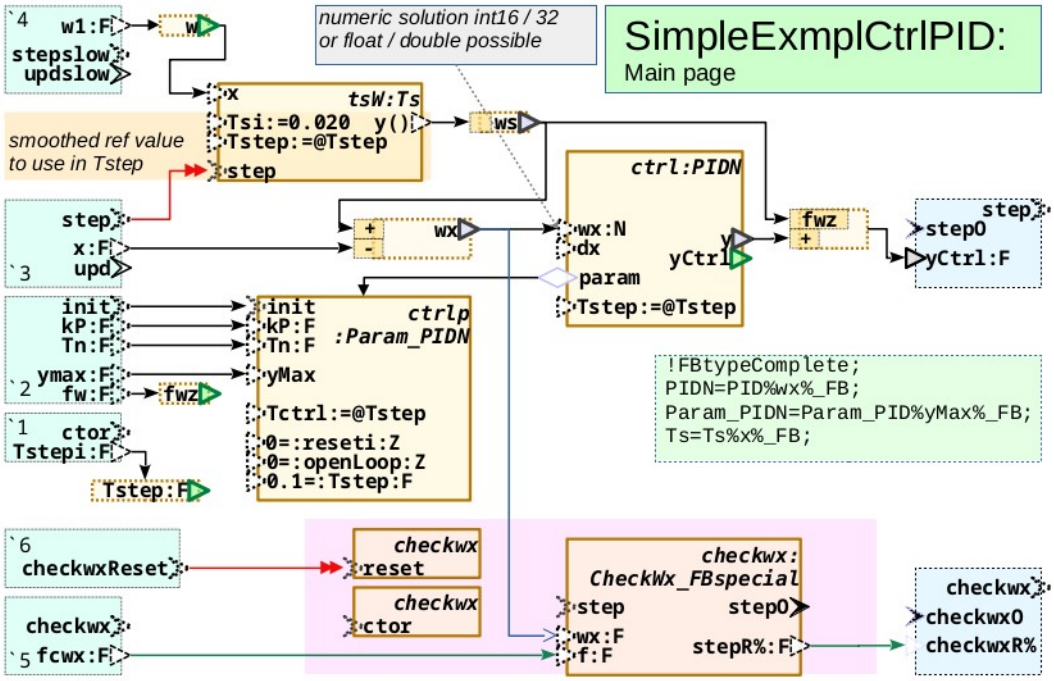


Figure 8: ExmplCtrl/SimpleExmplCtrlPID.png-2

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

4.4.4 Sequential Target Code Generation

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

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

4.4.5 Conditional Execution via Events

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

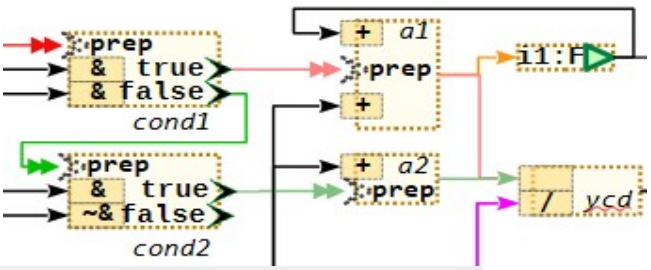


Figure 9: OFB/ExmpTrueFalseSimple.png

4.5 Capabilities of state diagrams / StateMachines - UML compatible

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

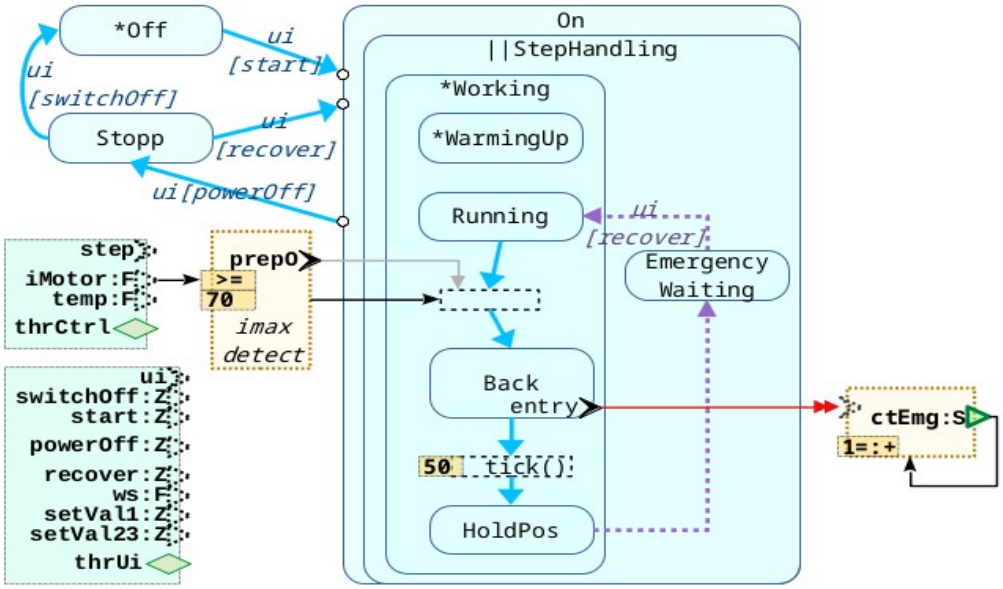


Figure 10: ExmplCtrl/Stm1

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

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

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

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

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

StateMachines in OFB

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

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

Dispersing of state execution

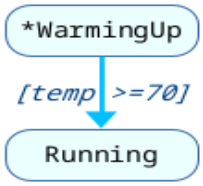
One essential capability is: Regions (parts of State switching) can be executed in different threads, or even on distributed devices. In the shown example, the position control is in the Composite State `Working`, its inner State switching is executed in a fast interrupt to recognise the overcurrent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

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

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

Repeated presentation of States in graphic

Figure 11: ExmplCtrl/Stm3



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

Additional possibilities for state drawing

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

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

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

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

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

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

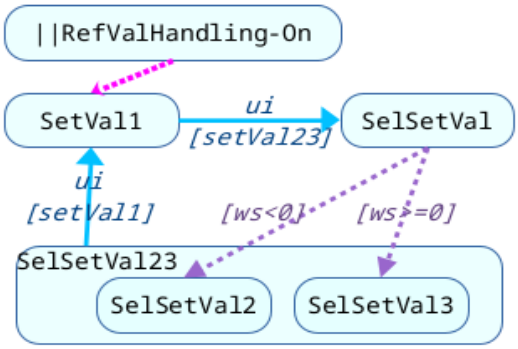


Figure 12: ExmplCtrl/Stm2

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

4.6 Comprehensive Example with Variants in Runtime

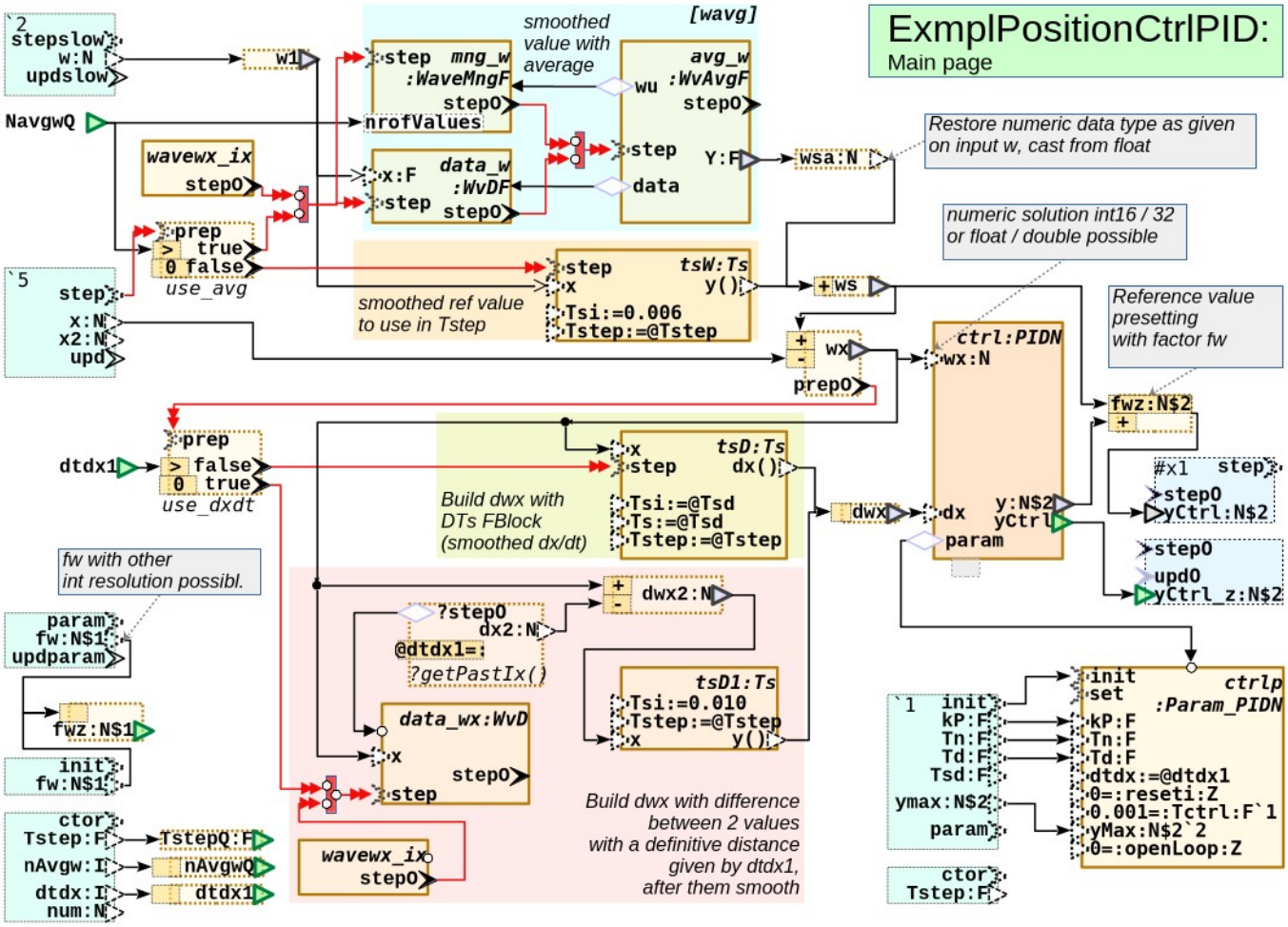


Figure 13: ExmplCtrl/PIDCtrl-Variants.png

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

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

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

- The set value `w` comes with a slower sampling time `stepSlow`. It is stored in a local variable. There are two ways designed to smooth this value for a continuous process when changes occur. First a simple smoothing block explained already on chapter *4.4.3 Event Determination from Data Flow (OFB Approach)* page *11* is used.
- The second solution is an **average filter**. This results in a really continual value.

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

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

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

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

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

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

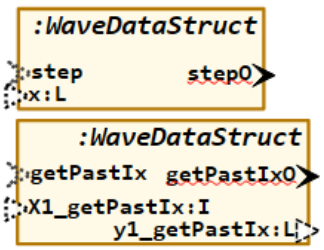
- Using a smoothing block for a DT1 D-part. This is a simple smoothed difference signal.
- Calculating a difference of the signal with a definitely period.

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

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

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

Figure 14: LibCtrl_emC/ WaveDataStruct-getPastIx.png

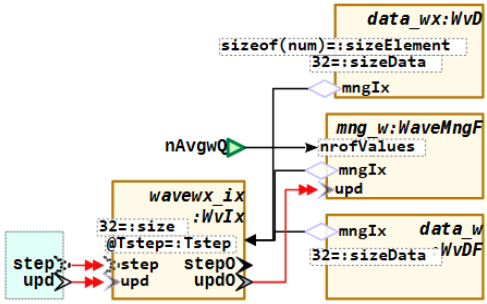


The access operation is defined by the specific input event `getpastIx` with its input and output data.

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

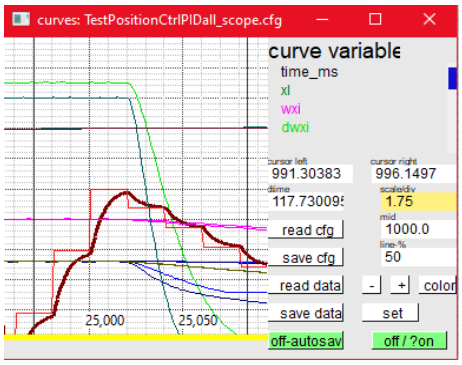
One detail is drawn with second view of this same FBlocks:

Figure 15: ExmplCtrl/ WaveMng-Aggr.png



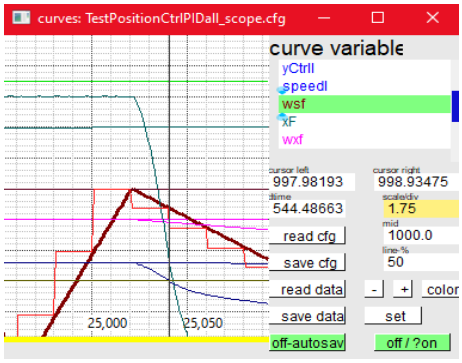
This is an overview which data buffers and average periods are used in this module and how they are aggregated and parametrised.

Figure 16: CurveView-w-Smooth-6ms.png



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

Figure 17: CurveView-w-Average.png



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

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

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

mailto: hartmut.schorrig@vishia.de