0

(empty backward page)

OFB – Approaches for the Object Oriented Function Block OFB Graphic Programming – chapter 7

Dr. Hartmut Schorrig
www.vishia.org 2024-09-14

1 Discussion about graphic presentation approaches

and implementations

Table of Contents

This part of the documentation discusses principles of graphic presentation with Function Blocks with object oriented aspects. It is an add on to the pure handling description (chapter of

A graphical Function Block Diagram (***FBD*** or also FBlock diagram) builds the content and interface of a Function Block type (FBlock type). The top level FBlock diagram is also intrinsically a FBlock type.

The content and interface of a FBlock type can also be described with the textual FBlock syntax given in IEC61499 see [IEC 61499-1/Ed.2] chapter B.2.1 Function block type specification.

This document is related to embedded software more than to automation control software. The difference to automation control is mentioned in some notes.

For embedded software the code generation (C/++) is an important topic. This is the focus of the documentation.

1.1 GBlocks, FBlocks and FBoper - what is a FBlock

In ordinary FBlock diagrams one FBlock instance presents an instance (of a class, using a type from a library) but only with one operation call in one context. For simple stateless FBlocks as ADD or MULT this is not a question. But if the FBlocks contains or accesses to data, it is a question.

If a FBlock type (a class) should have more operations, that is mapped to the OFB concept: You can use the same instance with different data and especially event inputs and outputs, and this is another operation call of the same instance. Other more powerful FBlock tools have non consequently but often similar possibilities: In Simulink S-Functions, *sample time* associations to pins are mapped to several operations).

But the remaining problem is: One FBlock is not only one operation, it is one operation call in only one context. Follow an example:

You have a class and one instance, which holds data to evaluate. Now you have an operation of this class (*"method"*) to store the data. This operation can be called from different threads, under different conditions:

```
MyDataEvaluater eval = new MyDataEvaluater();
  .....
void dataCapature(...){
  eval->storeData(...);
  .....
}
//Other thread or such:
void specificdataCapature2(...){
  eval->storeData(...);
  .....
}
```

This is a little bit ordinary C++ (maybe similar in C). The operation of function `storeData()` is called twice.

For an ordinary FBlock graphic you need two different FBlocks, each for one call of the same operation in different contexts. The FBlocks can be (should be) of the same type, but they are different instances, though it should be one and the same operation call for the same instance. How is it ordinary organized? There is a third FBlock, which is referenced, either internally by the implementing C-Code, or also obviously by address pointer data connections to associate the correct instances. That are ordinary solutions in given FBlock tools with specific implementation tricks. The real existing Object Orientation is often not in focus.

How does it works in OFB FBlock graphic:

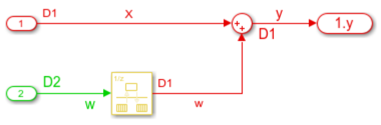
Some things are similar. Each FBlock is not the presentation of an operation, it is the presentation of one operation in one context.

OFB allows to draw the same FBlock with the same name and the same pins more as one time in the graphic. But this does not solve the given problem, because it is only on graphic level.

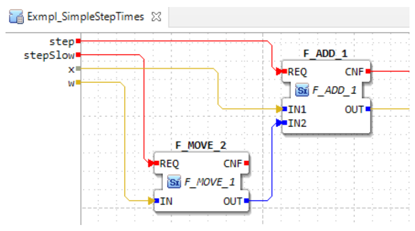
1.2 Data and event flow

The graphical presentation shows the data flow and due to IEC61499 also the event flow. The event flow determines the execution order.

Pure data flow with Sample time designation versus event flow



In comparison to other FBlock diagrams for example from Simulink, usual only the data flow is shown there. It determines the execution order, whereby different step times are used. Each sample time has its data flow. The Figure shows that, the step times are shown here with colors and also with “D1”, “D2”.



This system can be mapped to the system of event flow, whereby each event flow is associated to one sample time in Simulink.

Event flow on the same device → it is a simple execution order of FBlock operations

If all FBlocks or a block of FBlocks with a given event flow are arranged on the same device, one event flow can be code generated to an execution order of one operation of the module per module’s input event, which calls the operation of the FBlocks in the given event order. The operation of the FBlocks are that operations which are associated to one state entry caused by the input event. For that there are some variants, see next chapter FBtype Kinds and their usage

Event queue for execution, also for distributed devices

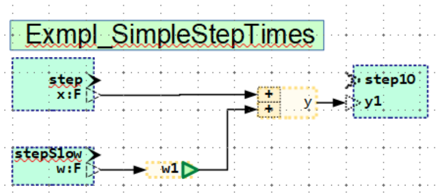
The other general possibility is using an event queue. The execution in the module (and also in sub modules) is determined only by the queueing and dequeuing of events regarding a first-in first out approach: Any execution of a FBlock’s functionality puts the emitted event in the queue, which determines further execution. This event queue approach is necessary and possible, if the FBlocks of the diagram are distributed on several devices. The originally approach for IEC61499 is oriented to several dispersion automation devices, whereby the whole functionality over more as one device is shown in only one diagram (or more diagrams, but not sorted to the devices, sorted to software function module's functionality).

Of course, the event queue is combined with event or message transfer between the devices.

The combination of both is sensible. Often in embedded control one FBlock diagram is really associated to only one device. Then the event queue is not necessary. Code generation can be regard the execution order due to the events. But the possibility to disperse the execution to several devices may be also interesting for embedded software solutions as well as used for automation device software. Emitted events are then put in a transmission queue, the transmission is done via field buses or such, and received events via transmission are also put in the queue. While dequeuing they are processed.

Of course this needs some milliseconds time, not for very fast control parts, but proper for set values, monitoring values, parameter changes and all these stuff.

Automatic detection of event flow



For the Libre-Office Solution The events should be given on the input and output blocks (green), adequate to the given step times in Simulink on the ports. But the connection is done automatically due to the detected data flow. The event flow is written in the textual fbd file with IEC61499 norm due to the here shown graphic. The green triangle, style `ofpZoutRight`, is adequate to the rate transition. it is an output of the `stepSlow` used in the `step` event chain.

1.3 FBtype kinds and their usage (due to IEC61499)

In IEC61499 there are different types of FBlocks:

a) Simple FBlock with one operation: It contains only one function or operation, one input event, one output event. The output data are produced in combinatoric due to the inputs. Examples for such simple FBlocks are mathematic functions, expressions etc. The term "***Simple FBlock***" is also a term in the IEC 61499 norm.

b) Standard FBlocks with more simple operations, as Object Orientation with more events, but with simple association between the input event and output event. The term "***Standard FBlock***" is used in IEC 61499 for FBlocks which have a state machine, named ***ECC*** = "***Execution Control Chart***". Any state can have one or more associated operations, which are executed on state entry, and one or more associated output events, which are activated after the entry operation execution also on state entry.

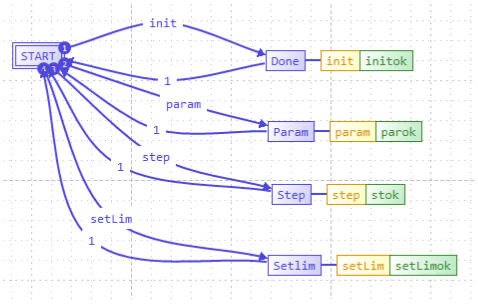


Figure 4: SimpleRegularStmn.png

1.4 Construction, init, run with several step times or events and shutdown

Coming from source code programming (C/++) the life cycle of a running software application can be differ to general three phases:

- **Construction:** Getting memory to run, set initial values. The construction phase is related to the ***constructor*** (ctor) in some programming languages or also with the initializing of memory before entry in `main()` in C language applications. It is the first phase of startup.

- **Initialization:** The initialization should be separated from the construction, because setting the correct initial values to run needs communication between several parts of the application, it presumes the construction. The initialization of one part can depend on finished initialization of another part, which delivers the values for the own initialization. Also a mutual initialization is sometimes necessary, also aggregations of modules each other. For that initialization needs loops. The initialization should be finished in a less number of loops. Any module should check its state of initialization and signal the finished state. If all modules have finished, then the initialization phase can be finished.

  - Often this initialization phase is not proper provided in some platforms. It should be cared about.

- **Run:** This is the working phase till the device is down. It is determined by physical events (timer, signal input) and often organized in fix sample or step times, and also event driven actions. This is also for simple devices with poor controllers and powerful devices.

1.5 Prepare and update actions

In some situations of calculation especially on resolve differential equations first all new values should be calculated starting from the current values (the state). In a second step all new calculated values are set to the current ones, the new state.

In mathematics this is the standard Euler method (from the mathematics Leonhard Euler, 1707 - 1783): https://en.wikipedia.org/wiki/Euler_method.

To calculate the new values, exclusively the old values should be used in all parts of the whole system of equations. Only then the solution is mathematically exact. This is the ***prepare*** phase. After them, or before the next step, the new values should be declared as current state, that is the ***update***

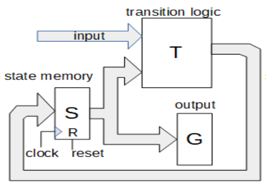
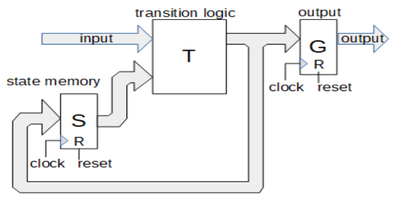


Figure 5: Moore automat 2

Also the theory of digital machines from Moore and Mealy based on this approach. Look on and . The is from https://en.wikipedia.org/wiki/Moore_machine. It shows the ***prepare - update*** concept in a proper kind for the Moore state machine. Here the Block T for Transitions is the ***preparation***, calculates the new state for D-Inputs of the FlipFlops, and the Block S is for ***update***, saves the prepared state as current one. This is classic.

Exact the same is drawn in right side.. Only the positions are a little bit changed. But compare it with the next image:

Figure 6: data flow with qout



In difference to the image Moore automat 2 above only the FlipFlops which presents the output state are separated from the other FlipFlops for the inner state. But also output logic is removed, the output functionality is built immediately from the logic block. This is a special more simple case of the Moore automat (sometimes named as '***<:n:Medwedew-Automat.***>'). If states are necessary also for output as also as inner state, the FlipFlops are twice.

The opens up an understanding of what happens during signal processing in control technology for analog variables or for automation processing. It is primarily the same

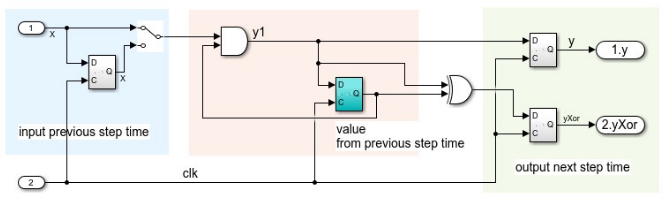
But the output registers are formed by the physical output, the digital-to-analog converter, also the transfer of information to another device that outputs or processes it, or setting a new pulse width for electrical converters, etc. These outputs are assigned to the next step time, just as the outputs of the flip-flops in the digital automaton are the state of the next clock period.

This is a general approach, separating between ***prepare*** and ***update***. This general approach can be subverted for certain solutions.

- All operations to calculate a new state from the old state are done in ***prepare***.
- The ***update*** refreshes all current values of all FBlocks to the before calculated prepared values.

For that it is to difference between FBlocks, which are only combinatory and state less. That FBlocks are used in ***prepare*** chains, or also in calculations for the update. FBlocks with a state can have also a prepare event input, but have also an update event input which updates the new prepared state to the outputs.

1.5.1 Example prepare and update for boolean logic



Exact the same approach is also used for boolean logic with D-Flip-Flops: The next value (as booleans) is ***prepared*** by logic on the D-inputs of Flipflops, and then all together on the same time are ***updated*** to the Q-output with a clock edge.The image above shows any processing signals (with the AND) which uses a value from the previous step time. One result of preparation is the signal `y1` which is output as `y` valid for the next step time. For that the outputs on an IC (for example FPGA) have DFF on the pins. The signal is '***clocked***', it comes time synchronous to a central clock. But the same signal is also used in a module after, where it is compared with the previous state of the same signal. It means the difference of the output in time can be built, here evaluated with a XOR to detect changes.

1.5.2 State of the art, ignoring prepare and update concept

Outside of boolean logic and FPGA usual a proper order of calculation is often found to regard the correct relations between the current (old) and new values for solving differential equations. This is often so in ordinary C/++ development, as also for example in the event driven 4diac tool for IEC61499. Because the execution sequence can be determined with tricky precision of the event connections, an appropriate solution will usually be found for the modeling approaches.

See the next examples.

1.5.3 Example prepare and update in source text languages (C/++)

What about ***update*** and the state variables: Usual, in C++ language programming and also in automaton programming the output of the prepared values are stored in variables anyway. If this variables are just used as current values for the next step then the ***update*** process is already done with store values and used for the next step. Look at the simple solution in C programming for an integrate:

Cpp: Simple integrate

```
yIntg += fIntg * x;
```

All is done with one statement, maybe with one machine code instruction. The old value is used, the difference is added as expression here from input and multiply the integrate factor, and the result is stored back to the only one integrate variable.

Because the proper solution is usual solved individually inside a module, regarding data dependencies and the correct calculation order, the ***prepare - update*** concept is not usually in focus. But sometimes small errors occurs which are not so obviously.

The simple form above is only possible if the old integrate value is no more necessary for any other operation later, after this operation the previous current value in no more existing. That’s why look on a little bit more complex integrate process, the solving of a differential equation for a bandpass filter. As example you can visit www.vishia.org/emc/html/Ctrl/OrthBandpass.html, chapter equations This is a filter algorithm. The equations in C are programmed firstly as:

Filter algorithm in C integrates dependent two values

```
 1: static inline void step_OrthBandpassF_Ctrl_emC
(OrthBandpassF_Ctrl_emC_s* thiz, float xAdiff, float xBdiff)
2: {
3: Param_OrthBandpassF_Ctrl_emC_s* par = thiz->par;
4: float a = thiz->yab.re; // store the current value of component yab.re
5: thiz->yab.re = par->fI_own * thiz->yab.re;
6: + par->fI_oth * ( thiz->kA * xAdiff - thiz->yab.im); // integrate .re
7: thiz->yab.im = par->fI_own * thiz->yab.im;
8: + par->fI_oth * ( thiz->kB * xBdiff + a); // integrate .im
9: }
```

The `yab.re` and `yab.im` are the both the current and also the new values after solving the differential equations. For an exact result it is very important to use the previous value `a` in line 4 instead the already new calculated value `yab.re` for calculation of `yab.im`. This is a simple solution. ***prepare and update*** are done also in one step, but the current value for the second equation is stored immediately in an individually variable.

But what is happen for this solution if the current values of the integrate variables are need for more operations, in this example for a more complex filter for harmonics. Then it is better to have a systematic solution, which looks like:

Filter algorithm in C consequently with prepare and update

```
static inline void step_OrthBandpassF_Ctrl_emC(OrthBandpassF_Ctrl_emC_s* thiz
, float xAdiff, float xBdiff
) {
Param_OrthBandpassF_Ctrl_emC_s* par = thiz->par;
thiz->xadiff = xAdiff; //store for evaluating (phase) and debug view
thiz->yab.re = par->fown * thiz->yabz.re + par->foth * ( thiz->kA * xAdiff - thiz->yabz.im);
thiz->yab.im = par->fown * thiz->yabz.im + par->foth * ( thiz->kB * xBdiff + thiz->yabz.re);
}
static inline void upd_OrthBandpassF_Ctrl_emC(OrthBandpassF_Ctrl_emC_s* thiz) {
thiz->yabz = thiz->yab; // update the current state z
}
```

For that two calls are necessary, first `step_…` to prepare the new values whereby the new values are stored here in `thiz→yab`. Right side in all equations this `thiz→yab` should never be used to build `thiz→yab` itself, don't mix old and new values, access always `thiz→yabz`. But for further operation the `thiz→yab` is accessible if necessary (as also the D-inputs of FlipFlops can be used to calculate further preparation phase D-values).

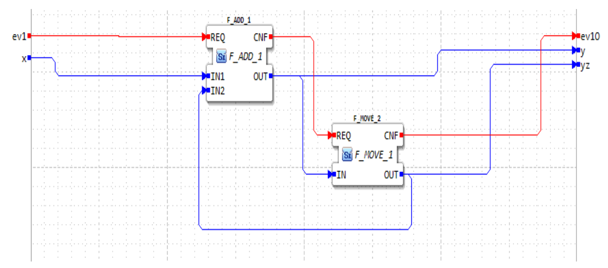
The `upd…` is the ***update*** operation. It stores the new state as current state for the next step. This assignment is intrinsically a fast `memcpy` from view of machine code.

The ***prepare - update approach*** needs two variables more, more memory, and the second update call is necessary. But the solution is more obviously and better able to review.

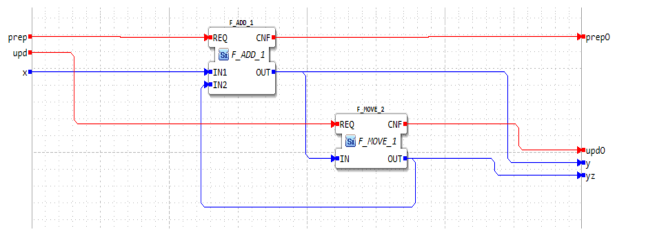
It is to decide which is more important, a very fast algorithm or obviously sources. Unfortunately the compiler optimization does not solve here this problem.

1.5.4 Example prepare and update in 4diac with MOVE-FBlock

The example of the simple integrate is also solvable by the simple calculation order controlled by the event flow:



Here the MOVE block is executed immediately after ADD and stores the output from the ADD FBlock for the next event occurrence which is the next step time. The previous value after integrate is no more existing after the event flow.



This is almost the same as image Example 4diac prep & update. But here the update is an extra event chain with `ùpd` and `updO`. The prepared result of ADD is available for further preparation which can also use the current (previous) value of the ADD, present in `y` and `yz`, for example to build a difference, the growth of the integrate between two step times, similar as the XOR in the boolean logic image Example binary logic prep & update

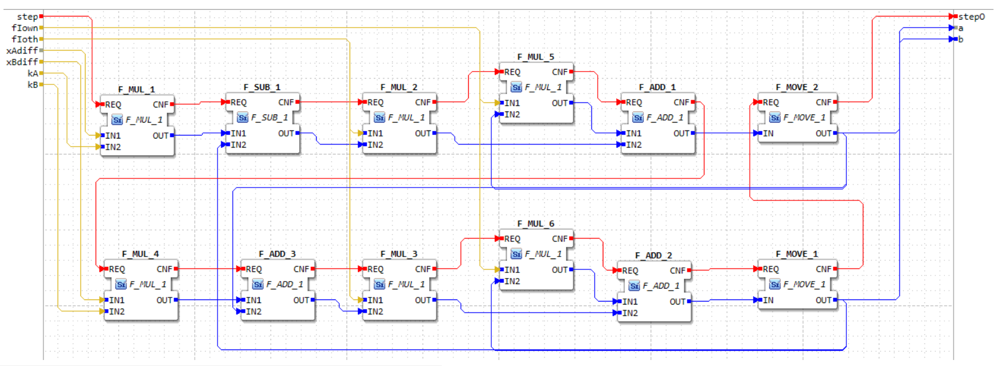


Figure 10: OrthBandpass without update event

This image shows the bandpass filter algorithm in 4diac similar as in Filter algorithm in C integrates dependent two values. The current previous values for integrate are used from the `F_MOVE_1` FBlock right side, but after calculate the filter the bothe `F_MOVE_1` FBlocks are also updated immediately in the same event chain. This works exact for the filter algorithm for one filter, but it gives slightly wrong results if more than one filter is used, for example for harmonics. Look for this usage of the image Example binary logic prep & update:

Figure 10. OrthBandpass with update event

There are two differences, first is the `upd` and `updO` event for update, but also a `ya` and `yb` is given which presents the calculated new outputs. This may be important because if the outputs are used as process outputs, they become active in the next step time because of course, it should be first give to the output device. If only the `yaz` and `ybz` are given, then they are the old values, one time back, which causes an additional dead time for control.

Figure 11. OrthBandpass in a filter application

The image above just shows an application where two OrthBandpass without update event are used, one for the fundamental oscillation, and one for an harmonic. Both values are output, `yfilt` is the filtered output of `x` and `y2harm` is the detected harmonic. That is the mission and possibility of this filter stuff. Also more as one harmonic is possible to filter. The principle is, all detected waves are added and compared with the input. The difference input for all ***OrthBandpass*** is equal, but each ***OrthBandpass*** has the resonance for its own frequency. If all frequencies are summarized and this is sufficient then the difference is 0 and the signals are stable.

But back to the event topics. The events are connected in that kind, that the resulting signals from the filter are presented in the outputs. The `F_ADD_1` is calculated firstly, takes the ***old*** current values from the step time before, put it in the feedback, and last the both ***OrthBandpass*** FBlocks are calculated. This is tricky. But what about if for more harmonic parts or other evaluations outside of this module the ***old*** current values are necessary. Then the logic becomes more complex.

Using the ***prepare and update*** concept is more obviously.

Figure 12. OrthBandpass in a filter application

Using the base variant of the filter with update, now also an filter application is possible and simple understandable, which outputs the filtered signal as new one for output on physic, and delivers also signals for further evaluation, here both components of fundamental and harmonic oscillation and the magnitude of the harmonics. The last one is calculated in the `upd` event chain.

Figure 13. OrthBandpass in a filter application

The interface shows the assignment of `yfilt` to the `prepO` output event, and the other signals to the `updO` event. The `prep` event queue is for ordinary evaluation of calculations, the end signal may be output to hardware or transmit, and the `upd` event queue delivers ***signals as state of another event updO*** to use it in the `prep` calculation (in the comprehensive superior module). But of course both event chains are related, not formally, but semantically. The event source should organize the proper order of `prep` and `update`.

#prepUpdSmlk

1.5.5 Example prepare and update in Simulink

in Simulink (© Mathworks) also an ***prepare - update*** concept is used. Simulink knows S-Functions, so named System-Functions which are not programmed graphically, instead textual. This S-Functions can be written in C language. The S-Functions can be used to understand the calculation principles of Simulink, it is obviously. The Standard FBlocks should have (expectable) the same principles. See especially the ***unit delay*** in this chapter below.

In the SFunction implementation two different operations should be called: `mdlUpdate(…)` and `mdlOutputs(…)`. The original text from the Mathworks help is

https://www.mathworks.com/help/simulink/sfg/mdloutputs.html: ***<:n:The Simulink® engine invokes this required method at each simulation time step.**.> **<:n:The method should compute the S-function's outputs at the current time step**.> **<:n:and store the results in the S-function's output signal arrays.**.>*

https://www.mathworks.com/help/simulink/sfg/mdlupdate.html: ***<:n:The Simulink® engine invokes this optional method at each major simulation time step.**.> **<:n:The method should compute the S-function's states at the current time step**.> **<:n:and store the states in the S-function's state vector.**.>*

The `mdlOutputs(…)` operation can process inputs of the FBlock, and sets of course the outputs of the FBlock. If the FBlock is only combinatoric (an expression), then this is the only need operation, `mdlUpdate(…)` has no sense.

If the FBlock has states, then the output can be calculated from states and inputs. These input pins should be marked as `ssSetInputPortDirectFeedThrough(…)`. Then the engine of Simulink detects loops in the data flow with these pins which is shown normally as error. It means these input pins should be used only straight forward with the outputs for combinatoric. Note: A Moore automaton would not process inputs for the outputs, uses only the states. But this is not a Mealy-automaton, because due to figure data flow with qout the outputs are further used in prepare-calculation or are the inputs for the physical output. The view of Mealy and Moore is inappropriate here. It is in mid of the transition or just prepare logic.

The `mdlUpdate(…)` operation can have inputs of the FBlock to calculate the new state from input and the state before, or it can also used internal variables calculate on the `mdlOutputs(…)` to set the state. It does not change outputs of the FBlock.

In the graphical Simulink model first all `mdlOutputs(…)` operations of all FBlocks are called. It means the current states (of the step time before) are presented on the outputs and the data flow for combinatorics are calculated, offer to inputs for further processing.

If all `mdlOutputs(…)` are called and the combinatoric data flow is done, then all `mdlUpdate(…)` are called. They may use values on inputs, but do not change outputs, and calculate the internal state for the next step time.

It means the `mdlOutputs(…)` with the combinatoric calculation is exact the ***prepare*** phase, and the `mdlUpdate(…)` is the ***update***. For ***update*** a few combinatorics inside the FBlock can be also calculated. That makes it a little bit more powerful for some special desires, but also more complicated. The state can also be set only from internal variables calculated on `mdlOutputs(…)` due to the image data flow with qout.

Because the programming of user - S-Functions in C/++ language can be done in any kind in responsibility to the user, it is also possible to omit the `mdlUpdate(…)`, do all in `mdlOutputs(…)` and consider the order of statements. The result of one FBlock can then be exactly, but the mix of ***prepare*** and ***update*** both done in one operation `mdlOutputs(…)` can cause small mathematically errors in differential equation solving over more FBlocks. Note that the order of calculation is other, `mdlUpdate(…)` of **all** FBlocks is called **after all** `mdlOutputs(…)` are processed.

The *unit delay* FBlock

Now look on the working example for the bandpass filter above with pure Simulink graphic.

Figure 14. Bandpass filter base FBlock in Simulink

Figure 15. setable unit delay in Simulink

The FBlocks A and B are a simple store FBlocks able to set as shown right. The important one FBlock here inside is the ***unit delay*** marked with `1/z`. It stores the value on input as current value for the next step. It means the first called `mdlOutputs(…)` outputs the current value, also for the own integrate, and also for use for further calculations with the current state (set from the previous step time). The later called `mdlUpdate(…)` then stores the input inside, to output it in the next step time.

If you look now to the whole module Bandpass filter base FBlock in Simulink then you see the `Yz`outputs of the both storage FBlocks as `Yz` or `Yaz` for this module. This is the current state from the previous step time whereas `Y` is the new state also usable for example for immediately output, which becomes currently in the next step because of physical device properties. But also for example the difference between `Yz` and `Y` can be built to get the growth (differential) of the outputs.

If you look on a usage of this module, you see that the `Yz` is used for a feedback to compare the input value with the current state, not the `Y`. Because both FBlocks have the ***unit delay*** inside with exact usage of `mdlOutputs(…)` and `mdlUpdate(…)` the solution is correct. This is a bandpass filter with high resolution, so small errors are seen in a bigger abbreviation of phases or resonance frequencies.

#prepUpdExmplPIDctrl

1.5.6 Example prepare and update for odg Graphic code generation (Libre Office)

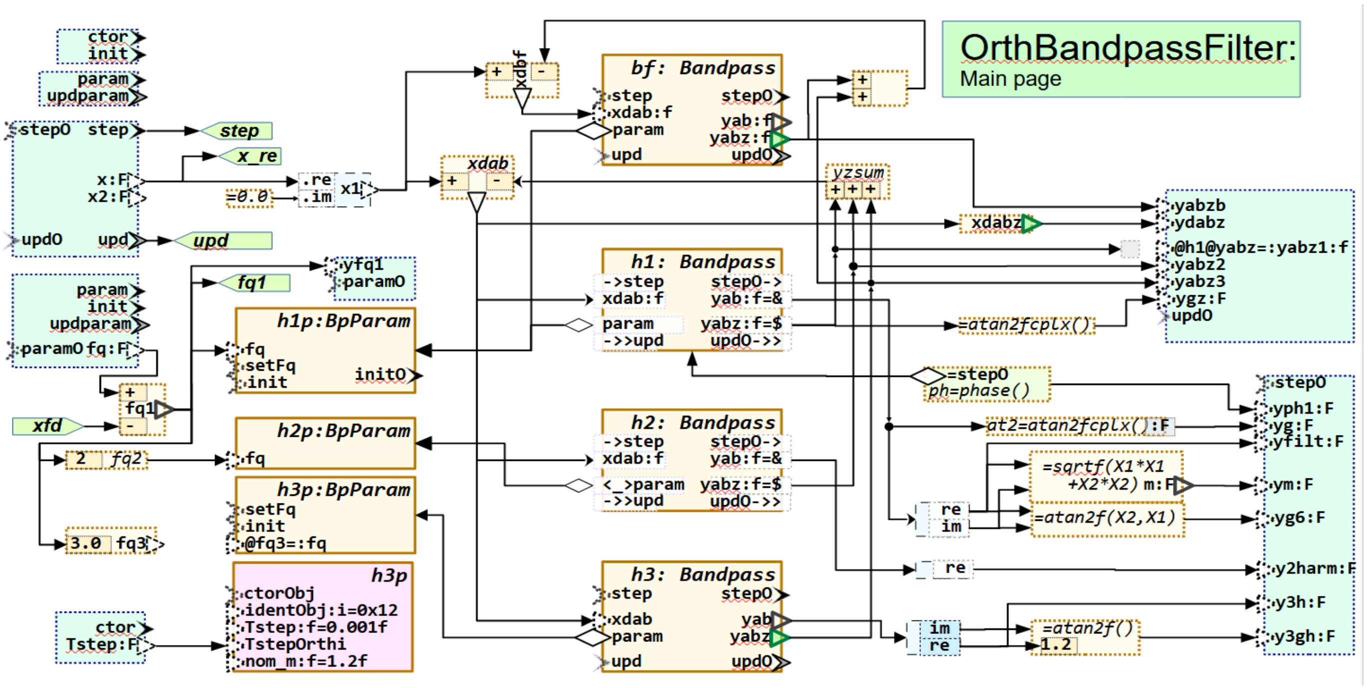


Figure 11: OrthBandpassFilter.odg.png

The image above shows the application of a bandpass filter, the same as shown also in C, 4diac and Simulink, drawn in LibreOffice graphic. This is the approach of ../pdf/UML-FBCL-Diagrams-Libreoffice-2023-09-23.pdf. From this graphic both a IEC61499 module should be generated as well as also execution code in C (this is in progress, not ready yet). The event connections are all gray, because they don't need to be drawn, they are established by the data flow exploration. Only the data flow connections should be drawn. But the event pins and the event to data associations should be known. For that the green dashed blocks shows input and outputs of the module, whereby always one prepare event pin is contained in the module's pin block, and also the associated update event pin and the associated output pins. With this information and with the adequate information in the used FBlocks the event connections can be determined.

The image contains also an ***aggregation*** `param`, to a `BpParam`FBlock which is filled with the `param` event.

The used modules are given as C language routines with a wrapper in IEC61499 as textual.fbd The wrapper for the `OrthBandpassF_Ctrl_emC` is given as following (manually written following the C operations):

Wrapper for OrthBandpassF_Ctrl_emC in IEC61499 to adapt to C

```
FUNCTION_BLOCK OrthBandpassF_Ctrl_emC
EVENT_INPUT
  ctor WITH OTHIS, Tstep;
  init WITH param;
  step WITH xab;
  upd WITH step; (* Note: Association of upd to the step dataflow *)
END_EVENT
EVENT_OUTPUT
  initO WITH initOk;
  stepO WITH yab;
  updO WITH upd, yabz; (*Note: Assoc upd input event)
END_EVENT
```

```
VAR_INPUT
  OTHIS: OrthBandpassF_Ctrl_emC__REF;
  xab : CREAL; (* Difference to adjust *)
  param: Param_OrthBandpassF_Ctrl_emC__REF; (* reference to parameter *)
  Tstep: REAL; (* Step time for calculations *)
END_VAR
VAR_OUTPUT
  yab: CREAL; (* new calculated value *)
  yabz : CREAL; (* state value from last update *)
  initOk: BOOLEAN;
END_VAR
```

Wrapper for OrthBandpassF_Ctrl_emC in IEC61499 to adapt to C

```
VAR
  THIS: OrthBandpassF_Ctrl_emC__REF;
END_VAR
EC_STATES
  IDLE; (* EC idle state *)
  CTOR: CTOR; (* Constructor *)
  INIT:INIT -> init0; (* EC State with Algorithm and EC Action *)
  STEP: STEP -> step0, ->step2;
  UPD: UPDATE -> upd0;
END_STATES
EC_TRANSITIONS
  IDLE TO CTOR:= ctor; (* constructor call *)
  IDLE TO INIT:= init; (* An EC Transition with event*)
  IDLE TO STEP:= step;
  IDLE TO UPD:= upd;
  CTOR TO IDLE:= 1;
  INIT TO IDLE:= 1;
  STEP TO IDLE:= 1;
  UPD TO IDLE:= 1;
END_TRANSITIONS
ALGORITHM CTOR IN ST:
  THIS := ctor_OrthBandpassF_Ctrl_emC(othiz:=OTHIS, Tstep:=Tstep);
END_ALGORITHM
ALGORITHM INIT IN ST:
  initOk := init_OrthBandpassF_Ctrl_emC(thiz:=THIS, param:=param);
END_ALGORITHM
ALGORITHM STEP IN ST:
  step_OrthBandpassF_Ctrl_emC(thiz:=THIS, xAdiff:=xab.real, xBdiff:=xab.imag);
  yab := THIS.yab;
END_ALGORITHM
ALGORITHM UPDATE IN ST:
  upd_OrthBandpassF_Ctrl_emC(thiz:=THIS);
  yabz := THIS.yabz;
END_ALGORITHM
END_FUNCTION_BLOCK
```

In words of Simulink, this is a S-Function.

In the graphic you see outputs green with dark borders for `yabz`. This outputs have a graphic style of `ofpZoutRight`. This identifies it as an output of a value from the last steptime as current state, similar as a ***unit delay*** in Simulink or as an output without `ssSetInputPortDirectFeedThrough(…)` for a Simulink S-Function. This output is related to the `upd` event in the FBlock.

For the data flow it means that this outputs are given, can be used without preparation.

The data flow goes forward to the adder, then to the subtraction, and to the inputs of the ***Bandpass*** modules. Also the input of the module is processed. Due to this data flow the `prep` event is calculated starting from the module's input, first through the adder, then to the ***Bandpass*** FBlocks, whereby all three can be calculated parallel. Any Bandpass `yab` output is then taken through the ***complex to real*** access and put to the step output, related to the output `stepO` event. That is the preparation.

The ***update*** of the ***Bandpass*** FBlocks is necessary because they have an update event input `upd` which is related to the `step` event input. Hence they need connected to that event from the module, which is related to the same prepare event. This is the `step` event chain, and the `upd` of the module is associated.

The outputs `yabz1` and `yabz2` of the modules are designated again with the graphic style `ofpZoutLeft`, but it needs to be related to an update event which renews the value. This is explored due to the event-data relation `yabz` to `updO` in the `OrthBandpassF_Ctrl_emC` module and the data flow.

1.5.7 How to associate the prepare to the update event

`prepare` (in the example `step`) und `update` are related. If the events are given manually in the graphic, then it is not a quest. But in the graphic above Wrapper for OrthBandpassF_Ctrl_emC in IEC61499 to adapt to C only the data flow is given. The event flow, here drawn in gray, can be missed, should be supplement automatically. This is as usual for FBlock diagrams, where often only the data flow is drawn.

To determine the correct event connections as shown here in gray, the data should determine which update event is associated to a step event. Also it should be known from all used FBlock types, which data in- and outputs are related to the events. In the image and in this way in LibreOffice FBlock diagrams the relation between prepare and update event is given in the input box (style `ofbMdlPins`). Such an module pin box contains exact one prepare event, the associated update event, associated prepare and update output events (left side) and the data associated to the prepare event. The module pin box right side with `yCtrl` associates this pin with the `updO` event.

The FBlock PID itself is given as ready to used SFunction in C language with all these events regarded in implementation. The interface of this FBlock type is given as fbd file in the textual notation of IEC61499:

Step and update association in FBD

```
EVENT_INPUT
  ctor .>WITH<:cF: OTHIS, Tstep;
  init WITH param;
  step WITH xab;
  upd WITH step; (* Note: Association of upd to the step dataflow *)
END_EVENT
EVENT_OUTPUT
  initO WITH initOk;
  stepO WITH yab;
  updO WITH upd, yabz; (*Note: Assoc upd input event)
END_EVENT
VAR_INPUT
  .....
```

Here in line 5 the `upd` event is declared using another event `WITH step`. Normally for IEC61499 textual notation only a data association to events should be noted here. But the syntax is not changed by this approach, only the semantic. On evaluation of the source it is detect: `upd` is related `WITH step`, `step` is an event, and hence `upd` is an update event related to the `step`. This is the only one enhancement of IEC61499 textual notation, without syntax change.

With this information, and the information in the state machine (ECC) about associated output events to inputs (see link TODO) the necessary event connections can be determined. See chapter TODO other html document to write

1.6 Extern labels

Docu file: Internals_LibreOffcZMarkup

1 Internals page 2 (#internal)