Creating PLCopen compliant Function Block libraries

Introduction

One of the problems with software development is that the increase in efficiency is very low. No Moore's law here, no doubling of the efficiency every 18 months. And it is even worse: with increasing functionality the complexity increases exponentially. To solve this one needs modern software development tools and modern development methodologies.
Part of a modern software development methodology is creating re-usable components, called function blocks, which contain part of the application software functionality. This process is called encapsulation. The next step is to integrate these tested and documented components in a library, from which everybody in and outside the organization can make use, and in this way create extended application software faster and with fewer errors. This is where the association PLCopen contributed with their document on how to create PLCopen compliant libraries.

The association PLCopen specified these kinds of re-usable components already for many application areas, like the function blocks for motion control, safety and communication. Now PLCopen wants to reach out to the software engineers that create their own libraries on top of this. These software engineers can be found in areas like machine builders and system integrators.
To ease the development of user libraries, PLCopen together with its members created guidelines on how to create PLCopen compliant function blocks.

1. Level controlled versus Edge triggering

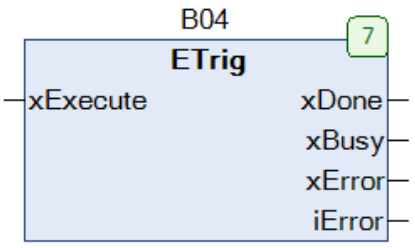
Basically there are 2 types of function blocks, those that are initialized (started) at a rising input and stay active till finalized or aborted, and those that are active as long as the corresponding input is high. The first are called edge-triggered, and the second are called level-triggered. An example of edge-triggered functionality is a move-to-position command which, after being initialized via the input *Execute*, runs till finalized. An example of a level triggered functionality is a PowerOn function, where the power is switched on as long as the relevant input (*Enable)* is set. Notice that the *Enable* input pairs with *Valid* output, and that *Execute* pairs with *Done*.

Sometimes it is important to choose a level-controlled model rather than an edge-triggered model. For the detection of a rising edge in a function block, two PLC cycles are necessary. Thus, if the requirement is to be able to process a new value in each cycle, an edge-triggered model cannot serve as a solution. In this case, a level-controlled function block model is the preferred way to implement the required functionality.

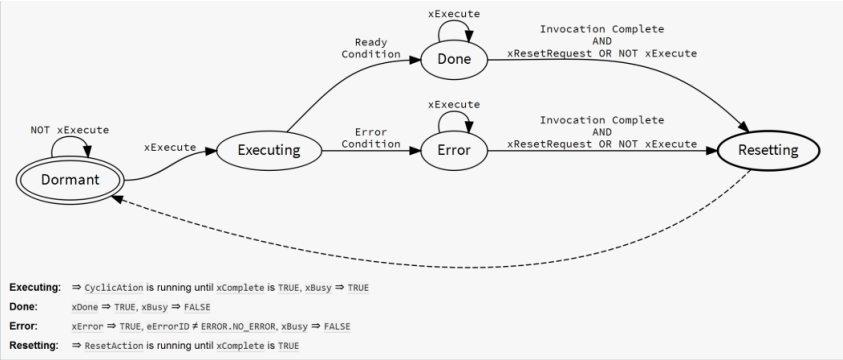
2. Example of edge triggered

The basic function block is the Edge Triggered Etrig. This is the edge triggered functionality in its most simple form with only an *Execute* as input, both in textual as well as in graphical representation:

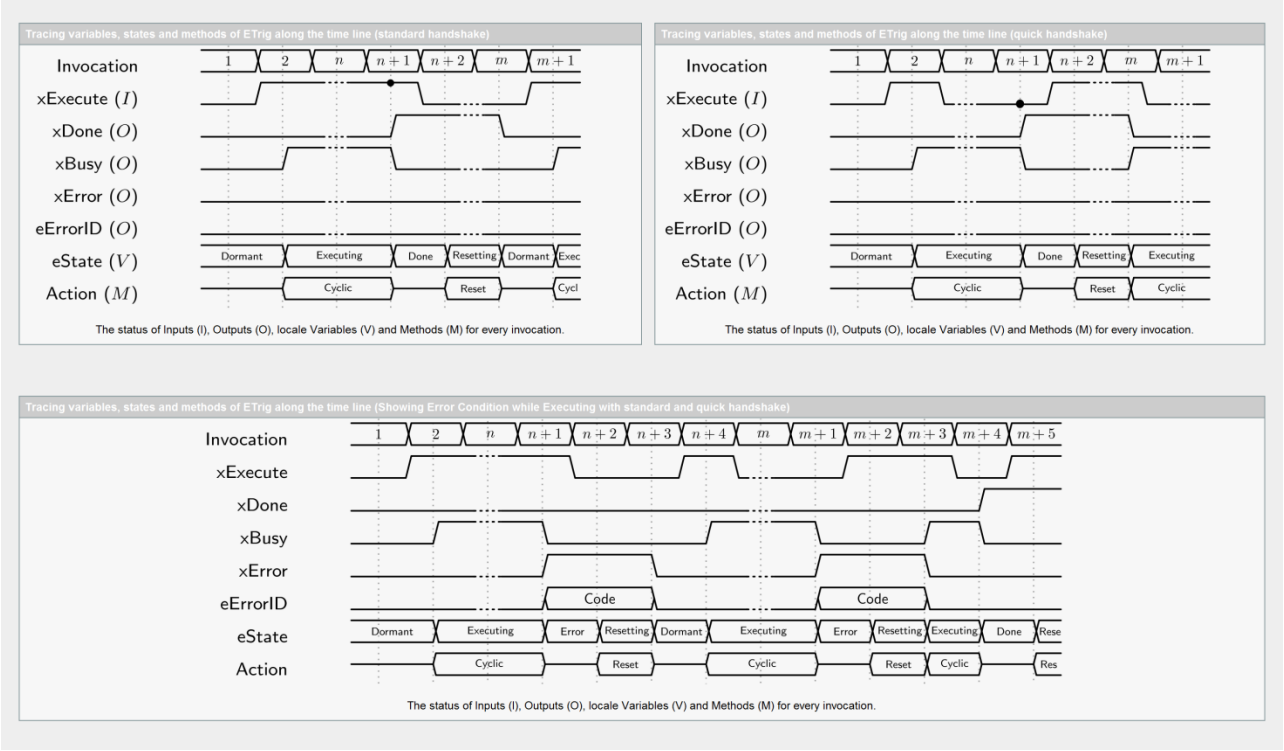
Textual representation	***Graphical representation***

`FUNCTION_BLOCK ETrig`
`VAR_INPUT`
`  xExecute: BOOL;`
`END_VAR`
`VAR_OUTPUT`
`  xDone: BOOL;`
`  xBusy: BOOL;`
`  xError: BOOL;`
`  iErrorID: INT;`
`END_VAR`
	

The corresponding state diagram is shown hereunder, and consists of the *Dormant*, *Executing*, and *Done* linked to *Resetting* and back to *Dormant* for the normal behavior, and via *Error* for the irregular behavior. For implementation such as state diagram can be easily converted to SFC or ST. Examples of this are included in the specification.

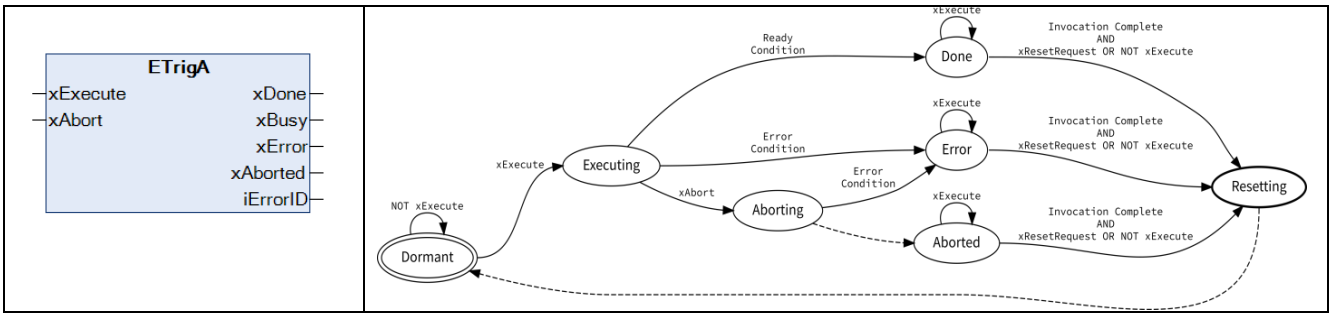


The corresponding timing diagrams for Etrig are shown here, where the 3rd diagram shows the behaviour in case of an error.



3. **Extending the basis with aborting**

In addition to this one can add the aborting functionality to abort the process, which includes additional states *Aborting* and *Aborted*.



Also this state diagram can be easily converted to SFC or ST, for which examples are included in the document.

4. Including timer functionalities

On top of the abort functionality (or even without this functionality) one can add timers to make the functionality more robust. Basically there are 3 options for timers:

1. TimeOut (To): the overall operating time of the defined operation should be lower than the time (in µs) as specified by the input value udiTimeOut;
2. TimeLimit (Tl): here the time limit is set that the operation stays within the cycle time. In that way a longer operation can be divided over several cycles;
3. And the combination of them both (TlTo).

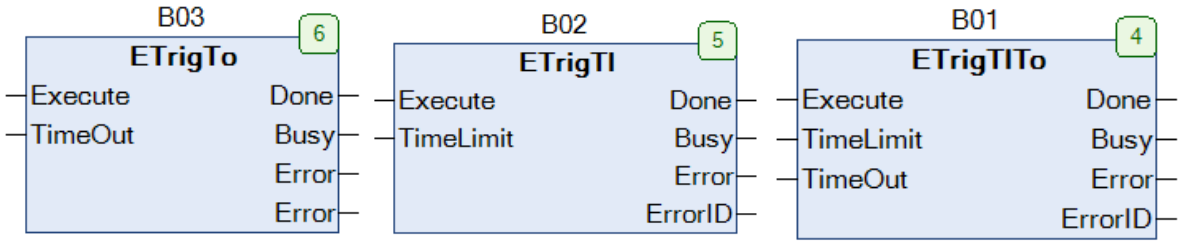
4.1. What does udiTimeLimit do?

A function block could, for example, complete a complex task in a loop. The larger the task is, the more time that is consumed in the current task of this function block. The udiTimeLimit parameter can define how much time per invocation is permitted for consumption in the respective function block.

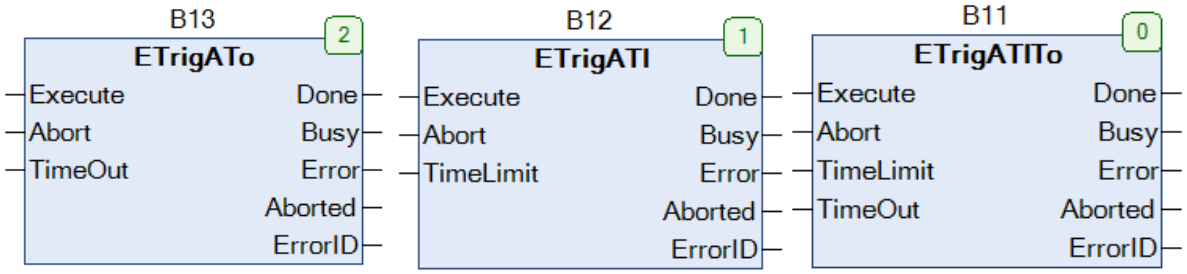
4.2. What does udiTimeOut do?

When processing its cycle action, a function block can be forced to wait for an external event. It can do this in an internal loop (BusyWait) or it can check in each cycle whether its task can be completed in full. The udiTimeOut parameter can define then how much time is permitted for consumption in the xBusy state.

4.3. Examples with timers without Aborting

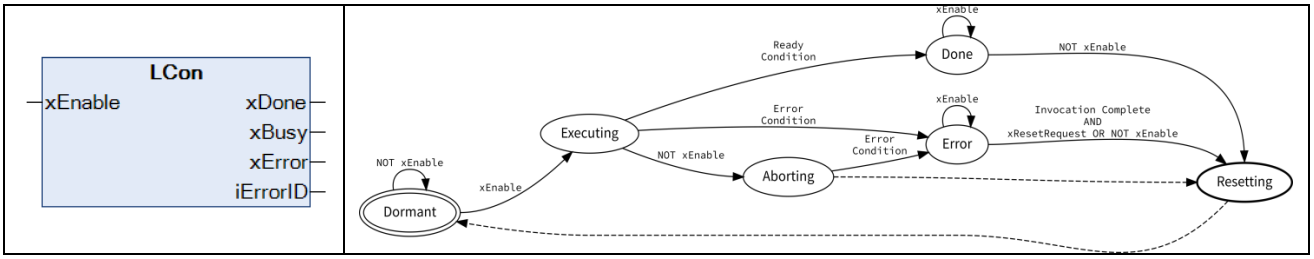


4.4. Examples with Aborting and timers

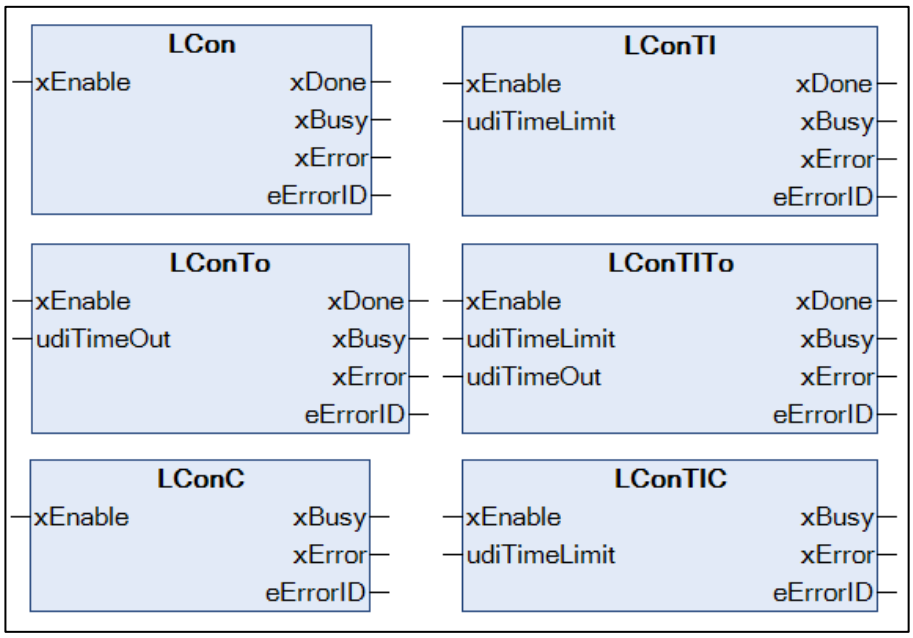


5. Level controlled function blocks

For level-controlled functionalities a similar set can be identified with similar states:



Adding the timers Time Out, To, and Time Limit, Tl, and the combination TlTo is similar. However there are 2 additional blocks for continuous behaviour, meaning it does not stop so there is no *Done* state: LConC for Continuous Behaviour and LConTlC, including the time limit:



Overview of the level controlled function blocks

6. Datasheets of Edge triggered and Level Controller function blocks

Included in the specification itself are the datasheets for all the listed function blocks. This includes the state chart, the implementation and for the edge triggered FBs the timing diagrams.
Also for the LCon functionality, an example of the corresponding ST code is included with object orientation. In the appendix an ST code example for the ETrigATlTo Function Block according to IEC 61131–3 2nd Edition, meaning no object orientation added.

7. Conclusion

With the document “Creating PLCopen Compliant Libraries” the organization shows how users can create own libraries within the PLCopen concept. This concept includes both level and edge controlled functionalities, which are extended by aborting and timer functionalities. To complete this specification examples in ST are shown, both in the classical as well as in the object oriented approach.
PLCopen released this document version 1.0 in May 2017 and it can be downloaded from the PLCopen website.

info@PLCopen.org
www.PLCopen.org