

PLCopen Software Creation Guidelines: Creating PLCopen Compliant Libraries

**PLCopen Technical Document
Version 1.0 – Official Release**

DISCLAIMER OF WARRANTIES

The name ‘PLCopen[®]’ is a registered trade mark and together with the PLCopen logos owned by the association PLCopen.

THIS DOCUMENT IS PROVIDED ON AN ‘AS IS’ BASIS AND MAY BE SUBJECT TO FUTURE ADDITIONS, MODIFICATIONS, OR CORRECTIONS. PLCOPEN HEREBY DISCLAIMS ALL WARRANTIES OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, FOR THIS DOCUMENT. IN NO EVENT WILL PLCOPEN BE RESPONSIBLE FOR ANY LOSS OR DAMAGE ARISING OUT OR RESULTING FROM ANY DEFECT, ERROR OR OMISSION IN THIS DOCUMENT OR FROM ANYONE’S USE OF OR RELIANCE ON THIS DOCUMENT. ANY PROGRAM EXAMPLES SHOW HERE ARE JUST LISTED AS AN EXAMPLE, AND ARE NOT TESTED IN PRACTICE AND CAN BE INCORRECT AND NOT BE USEFULL FOR YOUR APPLICATION. THIS MEANS THAT THE CODE SHOULD JUST BE SEEN AS A LEARNING BASIS AND NOT AS IMPLEMENTATION BASIS.

Copyright © 2017 by PLCopen. All rights reserved.

Date: May 4, 2017

The following paper

Creating PLCopen Compliant Libraries

is an official PLCopen white paper.

It summarises the results of the Task Force “*Creating PLCopen Compliant Software Libraries*” under the activity *Software Creation Guidelines* as part of PLCopen Promotional Committee 2 – Training.

The present specification was written thanks to the following members:

Name	Company
Peter Erning	ABB
Andrew Hollom	ABB
Bert van der Linden	ATS
Roland Wagner	B&R Automation
Bernhard Werner	CODESYS
Wolfgang Doll	CODESYS
Rolf Hänisch	Fraunhofer Fokus
Wolfgang Zeller	Hochschule Augsburg
Denis Chalon	Itris
Geert Vanstraelen	Macq
Barry Butcher	Omron
Hiroshi Yoshida	Omron
Andreas Weichert	Phoenix Contact Software
Kevin Hull	Yaskawa
Eelco van der Wal	PLCopen

Change Status List:

Version number	Date	Change comment
V 0.1	October 21, 2015	As results of several proposals and webmeetings
V 0.2	October 23, 2015	As a result of a webmeeting with 3S as preparation
V 0.3	October 28, 2015	As result of input and the webmeeting
V 0.4	November 11, 2015	As result of input and the webmeeting on Nov. 11
V 0.5	June 15, 2016	As result of the webmeeting on June 9 and feedback
V 0.6	June 30, 2016	As input of the webmeeting
V 0.61	July 15, 2016	Document a little more restructured
V 0.7	Aug. 15, 2016	As result of the earlier webmeeting and feedback
V 0.8	Aug. 25, 2016	As a result of the webmeeting and additional feedback
V 0.99	Sept. 30, 2016	As a result of the webmeeting on Sept. 29
V 0.99A	March 14, 2017	As result of the feedback and the webmeeting of Dec 8 as well as inclusion of other feedback material
V 1.0	May 4, 2017	Official release with reference to the source code

Contents

1. INTRODUCTION.....	5
1.1. NAMING CONVENTIONS FOR THIS DOCUMENT	5
1.2. NOTES ON THE EXAMPLES AND USAGE OF EN/ENO.....	5
2. COMMONALITIES IN EXISTING PLCOPEN SPECIFICATIONS.....	6
2.1. MOTION CONTROL, SAFETY AND COMMUNICATION	6
2.2. FUNCTION BLOCK MODELS.....	6
2.3. MOTION CONTROL: GENERAL STRUCTURE	6
<i>Example of a Motion Control Function Block</i>	<i>7</i>
2.4. SAFETY	8
<i>Example of a Safety FB</i>	<i>8</i>
2.5. SPECIFICATIONS IN COMMUNICATION	8
<i>Example of a Communication FB</i>	<i>9</i>
2.6. CONCLUSION.....	9
3. INTRODUCTION OF THE PLCOPEN FUNCTION BLOCK CONCEPTS	10
3.1. RELATION OF EXECUTE AND ENABLE INPUTS TO LEVEL AND TRIGGER INPUTS	10
3.2. INTRODUCTION TO EDGE TRIGGERED FUNCTION BLOCKS	10
3.3. INTRODUCTION TO LEVEL CONTROLLED FUNCTION BLOCKS.....	11
3.4. COMMON PROPERTIES OF THESE TYPES OF FUNCTION BLOCKS	12
3.5. ERROR DOMAINS AND ERROR CODES	13
3.6. HOW TO HANDLE THE STATE ENUM DATA TYPE	15
3.7. COOPERATION OF VARIOUS FUNCTION BLOCKS	15
<i>Extending the Example to a Complete EchoServer.....</i>	<i>16</i>
<i>Transformation to a Multithreaded EchoServer</i>	<i>17</i>
4. INTRODUCTION IN THE OBJECT ORIENTED FEATURES OF IEC 61131-3	19
5. EXPLANATION OF RISING EDGE TRIGGERED FBS.....	20
5.1. THE BASIC FB: ETRIG	20
<i>Example of the ST Program for the FB ETrig with OO.....</i>	<i>21</i>
5.2. ADDING THE ABORTING FUNCTIONALITY TO THE BASIS	25
<i>Example of a SFC Program</i>	<i>27</i>
5.3. ADDING TIMER FUNCTIONALITY.....	28
<i>What does udiTimeLimit do?.....</i>	<i>28</i>
<i>What does udiTimeOut do?</i>	<i>28</i>
<i>Examples with timers without Aborting</i>	<i>28</i>
<i>Examples with Aborting and timers</i>	<i>28</i>
<i>Detailed description of the Function Block ETrigATITo</i>	<i>29</i>
5.4. EXAMPLE OF THE ST PROGRAM FOR ETRIGATLTO	30
6. EXPLANATION OF A LEVEL CONTROLLED FB	31
6.1. BASIC LEVEL CONTROLLED FB	31
<i>State Diagram Basic Level Controlled FB.....</i>	<i>31</i>
6.2. EXAMPLE OF THE ST PROGRAM FOR THE FUNCTION BLOCK LCON IN OO.....	32
6.3. ADDING TIMERS.....	36
<i>Example of LConTl.....</i>	<i>36</i>
<i>State Diagram LConTITo.....</i>	<i>37</i>
<i>Example of an SFC diagram</i>	<i>39</i>
APPENDIX 1 DATASHEETS OF THE EDGE TRIGGERED AND LEVEL CONTROLLED FBS	40
APPENDIX 1.1 GENERAL COMMENTS ABOUT THE SAMPLE PROGRAMS.....	40
APPENDIX 1.2 OVERVIEW OF THE FUNCTIONALITIES	41
APPENDIX 1.3 OVERVIEW EDGE TRIGGERED FBS	42

Appendix 1.3.1	<i>ETrig</i>	43
Appendix 1.3.2	<i>ETrigTl</i>	46
Appendix 1.3.3	<i>ETrigTo</i>	49
Appendix 1.3.4	<i>ETrigTlTo</i>	52
Appendix 1.3.5	<i>ETrigA</i>	55
Appendix 1.3.6	<i>ETrigATl</i>	58
Appendix 1.3.7	<i>ETrigATo</i>	61
Appendix 1.3.8	<i>ETrigATlTo</i>	64
APPENDIX 1.4	OVERVIEW LEVEL CONTROLLED FBS	67
Appendix 1.4.1	<i>LCon</i>	68
Appendix 1.4.2	<i>LConTl</i>	70
Appendix 1.4.3	<i>LConTo</i>	72
Appendix 1.4.4	<i>LConTlTo</i>	74
Appendix 1.4.5	<i>LConC</i>	76
Appendix 1.4.6	<i>LConTlC</i>	78
APPENDIX 2	EXAMPLE WITHOUT USING OBJECT ORIENTED FEATURES	80
APPENDIX 3	EXAMPLE OF AN INTERMEDIATE INTERFACE.....	85
APPENDIX 4	BEHAVIOUR OF INPUTS AND OUTPUTS IN PLCOPEN MOTION CONTROL FBS	86

1. Introduction

One of the best outcomes of the PLCopen specifications for Motion Control, Safety and Communication, (see www.PLCopen.org) is the definition provided for function block input and outputs. This provides a clear and concise shell as a starting point when considering the type of application level function to be created. For this two main function block categories are specified: the Execute and the Enable model. The goal now is to extend these to fit other application areas, and helping users to specify and implement consistent sets of FB libraries for their own usage.

For this reason a working group was started within the PLCopen activity on Software Construction Guidelines defining the guidance for creating PLCopen compliant Function Block Libraries.

By strictly following a few key features of the PLCopen specification, application level function blocks can provide a high degree of robustness, usability and predictability. The behavior described makes it very easy to incorporate and debug functions in an application. Errors and ErrorIDs can be elevated to the calling functions. Interlocks are easier to create. Linking activities becomes easier.

1.1. *Naming conventions for this document*

In line with the PLCopen Coding Guidelines, the following naming conventions are used in this document:

Prefix yes or no	Used both in text (for readability) and in examples (when applicable)
------------------	---

With prefix there is a difference between xError, eError and iError, compared to no prefixing with names like Error and ErrorID. It is decided to use unique names even without pre-fixing. Meaning that in this document xError and eErrorID are used in examples and Error and ErrorID can be used in the text. Also it is advised to use the same capitalization for every object instance, even if the tool/compiler doesn't mandate it. The following guidelines are proposed:

- Use UPPER_SNAKE_CASE for CONSTANTS and user defined datatypes and keywords (like BOOL, FOR, TYPE and END_TYPE).
- Use UpperCamelCase for all other multi-word items

Variable names will be written in Courier New font size 11, while states will be written in the normal font in *italic*.

1.2. *Notes on the examples and usage of EN/ENO*

Any programs listed in this document are just examples, are not tested in practice and can be incorrect and not be useful for your application. This means that the code should just be seen as a learning basis and not as an implementation basis.

Any code in this document should be considered as an example only. There is no need to implement the functionality exactly as proposed, just the interfaces of the function blocks and the state diagrams are necessary for compliance.

The Enable /Execute constructs are on top of the usage of EN/ENO constructs (see Ch. 6.6.1.5. Execution Control (EN, ENO) of the IEC 61131-3 standard 3rd edition).

2. Commonalities in existing PLCopen specifications

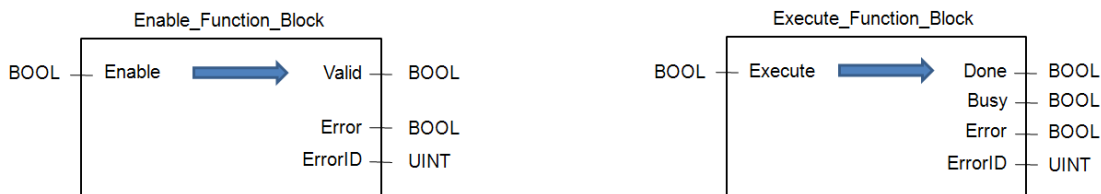
2.1. Motion Control, Safety and Communication

This chapter is created to provide an overview of the current specification of PLCopen Function blocks over the different working groups: Motion, Safety and Communication. This is only done in short form, and for more specific explanation one is referred to the original specifications.

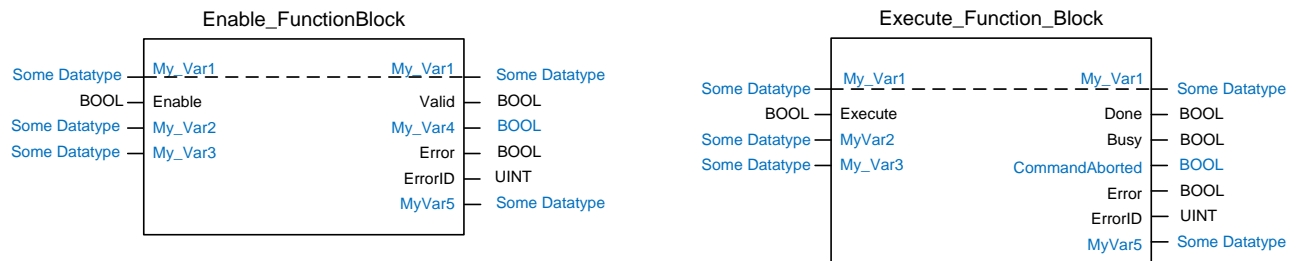
The proposal for the generic function blocks is explained in the next chapters, especially Chapter 5 - Explanation of Rising Edge triggered FBs and Chapter 6 - Explanation of a Level Controlled FB.

2.2. Function Block Models

There are two basic function block models as described below. These functions are shown with the minimum inputs and outputs. Notice that `Execute` pairs with `Done`, and that `Enable` pairs with `Valid`. This aids the visual appeal of the code structure in FBD format when contacts and coils are connected to the function block.



Programming to account for the variants will be described in detail in the following sections. All implementations will include added inputs and outputs for an actual application as shown in blue below.



2.3. Motion Control: General structure

PLCopen has created a suite of specifications for motion control. Within this suite an effort was made to have a consistent specification and layout of the function blocks. This chapter provides an overview of the commonalities in the specifications of the FBs. For more details check Appendix 4 Behaviour of inputs and outputs in PLCopen Motion Control FBs.

The definition of the PLCopen FBs for motion control consists of an activation related section and a status related section.

To activate an FB one originally had two options

1. `Execute` as input, triggering the execution of the FB, and `Done` (or *InVelocity*, *InGear*, *InTorque* or *InSync*) as related output showing when the FB has finalized the command;
2. `Enable` as input, level sensitive, and `Valid` as related output

With the release of Version 2.0 of Part 1, the input 'Continuous Update' was added to combine both.

- If it is TRUE, when the function block is triggered (rising 'Execute'), it will - as long as it stays TRUE – make the function block use the current values of the input variables and apply it to the ongoing movement.

- If 'ContinuousUpdate' is FALSE with the rising edge of the 'Execute' input, a change in the input parameters is ignored during the whole movement and the original behavior of previous versions is applicable.

For the status the following outputs are defined:

- **Busy** - The FB is not finished and new output values are to be expected
- **Active** - Indicates that the FB has control of the axis
- **CommandAborted** - 'Command' is aborted by another command
- **Error** - Signals that an error has occurred within the Function Block
- **ErrorID** - Error identification

Example of a Motion Control Function Block

As an example of a PLCopen Motion Control FB, a graphical representation of the FB MC_MoveAbsolute is shown here.

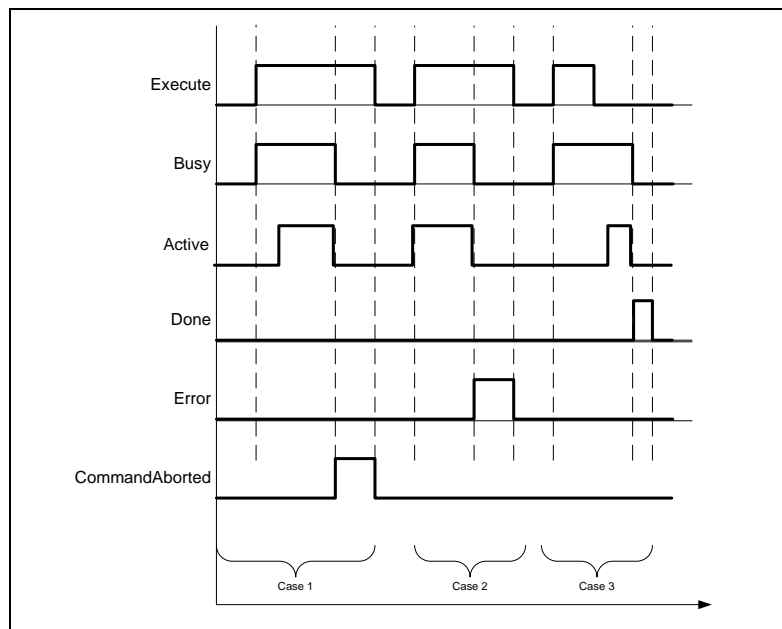
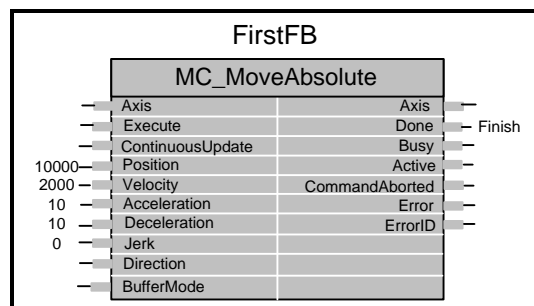


Figure 1: The behavior of the 'Execute' / 'Done' in relevant FBs

The behavior of the 'Execute' / 'Inxx' style FBs is as follows:

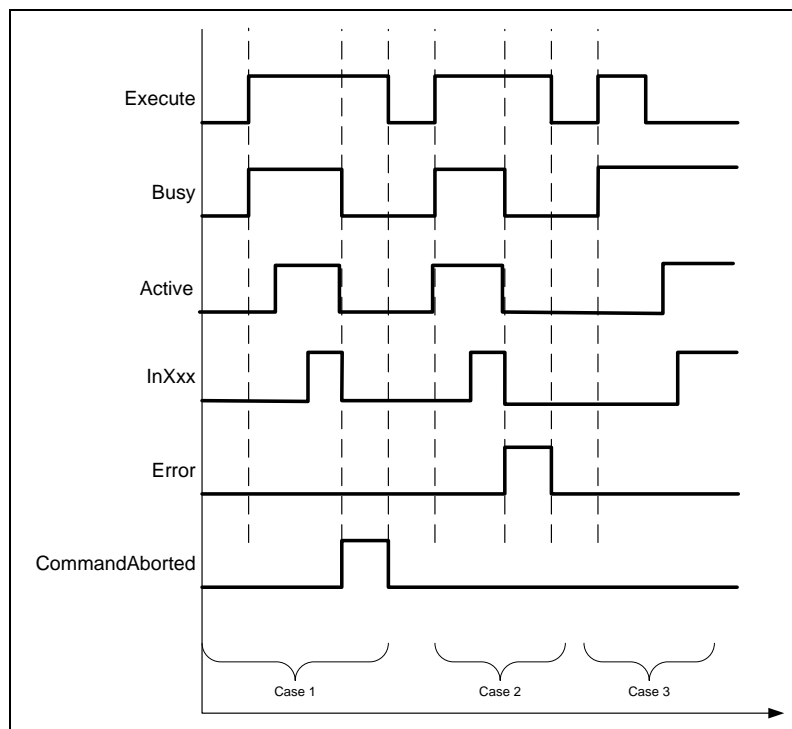
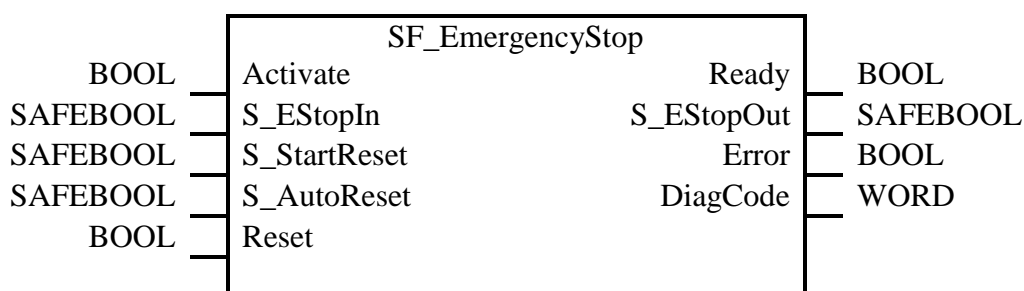


Figure 2: The behavior of the 'Execute' / 'Inxx' in relevant FBs

2.4. Safety

PLCopen has created a suite of specifications for safety. Also within this suite an effort was made to have a consistent specification and layout of the function blocks, in line with motion control. This chapter provides a short overview of the commonalities of the specifications of the safety FBs. Due to the safety related character of this part of the specification a reduction in the datatypes and functionalities is applicable, as well as the introduction of a Safe datatype. This means that there is a smaller overlap in outputs with the motion specification. Also the combination `Execute/Done` is replaced by `Activate/Ready` due to a slightly different behavior.

Example of a Safety FB

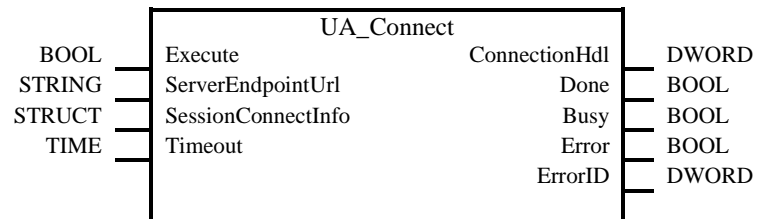


2.5. Specifications in Communication

Together with the OPC Foundation, PLCopen has created FBs for the communication via OPC UA, Unified Architecture. This chapter gives a short overview of the specification.

Also here the combination `Execute/Done` is applicable, as well as `Busy`, `Error` and `ErrorID` with functionalities equal to the PLCopen Motion Control specification.

Example of a Communication FB



2.6. Conclusion

Overall there are 2 levels of commonality specified over the different PLCopen specifications:

- Basic level with **Execute/Done** (or **Activate/Ready**) and **Busy**, **Error** and **ErrorID**, although the **ErrorID** can be an **INTEGER**, a **WORD** or a **DWORD**.
- Extended Level with the addition of **Active** and **CommandAborted**

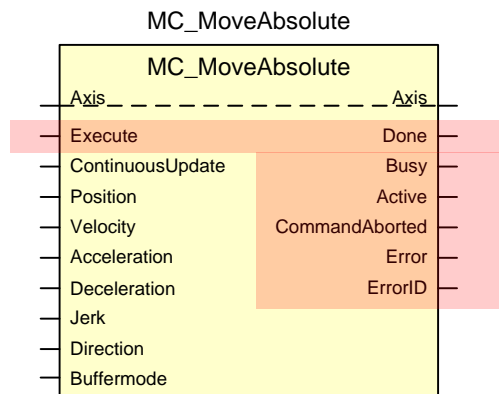


Figure 3: Common behaviour parameters

Overview of related parameters in different libraries.

Library	MC v1.0		MC v2.0	Safety	Communication
	Enable	Execute	Enable/Execute		
Execute		X	X		X
ContinuousUpdate			X		
Enable	X		X		
Activate				X	
Ready				X	
Valid	X		X		
Enabled	X				
Done		X	X		X
Busy	X	X	X		X
Active	X	X	X		
CommandAborted		X	X		
Error	X	X	X	X	X
ErrorID	X	X	X		X
DiagCode				X	

3. Introduction of the PLCopen Function Block concepts

In this chapter the basic concepts of the PLCopen Function Blocks are explained. These concepts can also be called PLCopen Common Behaviour Model. There are 2 main groups identified: Edge Triggered and Level Controlled. The details are listed in Chapter 5 and 6 as well as in Appendix 1.

3.1. Relation of Execute and Enable inputs to Level and Trigger inputs

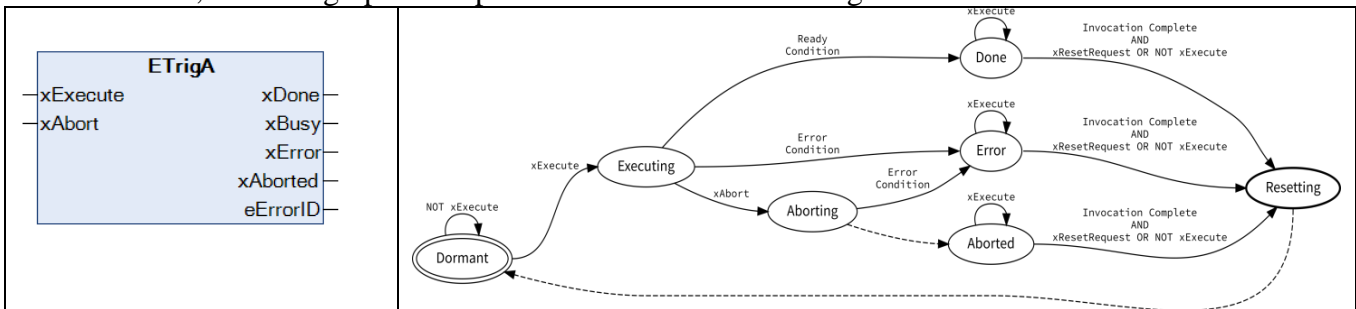
In the specifications of PLCopen FBs as described above, the inputs `Enable` and `Execute` are used to start the FB. However, further in this document a more generic description is used:

- Edge Triggered Function Blocks – which are coupled to the `Execute` input;
- Level Controlled Function Blocks – which are coupled to the `Enable` input.

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.

3.2. Introduction to Edge Triggered function blocks

As an example of an edge triggered function block the ETrigA (Edge Triggered with Abort functionality) is shown here, both the graphical representation and the state diagram.



Edge Triggered function blocks in the context of this document are defined as follows:

- The input variable `xExecute` is the defining feature for this type of function block.
- A rising edge detected at the input variable `xExecute` (*start condition*) starts the operation defined by this particular edge triggered function block.
- All inputs other than `xExecute` and the eventually present variable `xAbort` are sampled with this rising edge. These two inputs will be stored locally. Thus, later changes of these inputs cannot influence the defined operation while it is running [1].
- The input variable `xExecute` can be set to `FALSE` after the status `TRUE` was seen on the output variable `xBusy`.
- A falling edge detected at the input variable `xExecute` will **not** abort the defined operation. The defined operation is running normally to its *ready condition*, *abort condition* or *error condition*.

[1] Sometimes it is necessary to have additional input variables which can influence the internal work flow. In this case, the special behavior of these variables should be clearly documented.

[2] Sometimes it is necessary to have additional output variables with a valid status while `xDone` is not set to `TRUE`. In this case, the special behavior of these variables should be clearly documented.

[3] Sometimes it is necessary to have additional output variables which a valid only in combination with the status of some other output variables. In this case, the special behavior and relationship of these variables should be clearly documented.

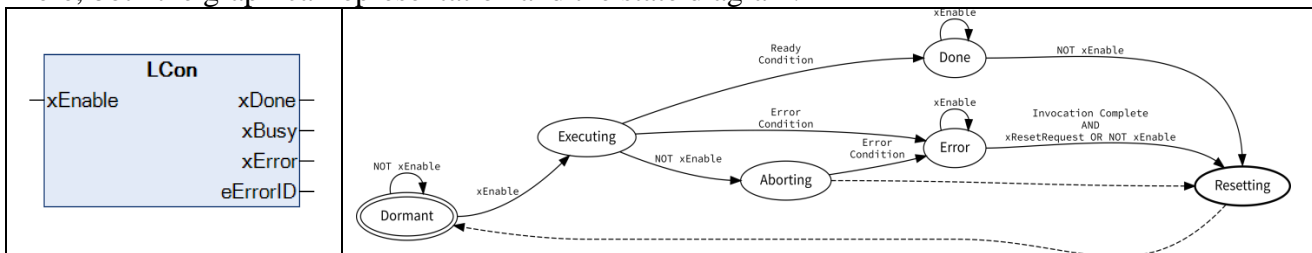
- Only if the status `TRUE` is detected on the (eventually present) input variable `xAbort` the defined operation is cancelled (*abort condition*).
- If the input variable `xAbort` is present and the input variable `xExecute` have the same value `TRUE` the *abort condition* is reached immediately.
- Only one of the output variables `xDone`, `xBusy`, `xError` or the output variable `xAborted` (if present) can have the status `TRUE` at the same time.
- If an *abort condition* was detected, the output variable `xAborted` is set to `TRUE` after setting the output variable `xBusy` to `FALSE`.
- With the falling edge of `xBusy` the input variable `xExecute` is sampled and its inverted value is stored as a *reset request* inside the FB.
- The state of the output variables will be valid for a minimum of one invocation even though the state of the `xExecute` input variable is already set to `FALSE`. In this case (*reset request*) the internal state of the function block is automatically reinitialized. In the other case (`xExecute` is still `TRUE`) the function block is waiting for a falling edge of the input variable `xExecute` before reinitializing the function block (*standard handshake*).
- The status of the output variables other than `xDone`, `xBusy`, `xError`, `xAborted` or `eErrorID` are valid only while `xDone` has the status `TRUE` [2][3].
- With an active *reset request* and after the status `TRUE` of one of the variables `xDone`, `xError` or `xAborted` was seen, the input variable `xExecute` can be set to `TRUE` again and the function block will restart its defined operation (*quick handshake*).

See the detailed descriptions of the reference implementation for the different edge triggered function blocks in the appendix:

ETrig | ETrigTl | ETrigTo | ETrigTlTo | ETrigA | ETrigATl | ETrigATo | ETrigATo | ETrigATlTo

3.3. Introduction to Level Controlled function blocks

As an example of a level controlled function block the LCon (Level Controlled function block) is shown here, both the graphical representation and the state diagram.



Level Controlled function blocks in the context of this document are defined as following:

- The input variable `xEnable` is the defining feature for this type of function block.
- The status `TRUE` detected on the input variable `xEnable` (*start condition*) starts the operation defined by this particular level controlled function block. The defined operation is running to its *ready condition* or *error condition* while the input variable `xEnable` is `TRUE`. The status `FALSE` detected on the input variable `xEnable` is interpreted as an abort (*abort condition*). This means the internal state of the function block and all output variables will be reinitialized and it is then ready for a new start (*standard handshake*).
- The input variables will **not** be stored locally and can so influence the current work flow of the defined operation.
- Only one of the output variables `xDone`, `xBusy` or `xError` can have the status `TRUE` at the same time.
- The status of all output variables is valid as long as the status of the output variables `xBusy` or `xDone` is `TRUE` [3].

- With the falling edge of `xBusy` the input variable `xEnable` is sampled and its inverted value is stored as a *reset request*.
- The state of the output variables will be valid for a minimum of one invocation even though the state of the `xEnable` input variable is already set to `FALSE`. In this case (*reset request* is `TRUE`) the internal state of the function block is automatically reinitialized. This can especially happen after an *error condition* while aborting the defined operation.
- The status of the output variables other than `xDone`, `xBusy`, `xError` or `eErrorID` are valid only while `xDone` has the status `TRUE` [2][3].
- With an active *reset request* and after the status `TRUE` of one of the output variables `xDone` or `xError` was seen, the input variable `xEnable` can be set to `TRUE` again and the function block will restart its defined operation (*quick handshake*).

Sometimes it is necessary to have a behavior model which never reaches its ready condition. An example is the `MC_Power` motion control function block. Also, the `TCPServer` function block as shown in Par. 3.7 Cooperation of various function blocks is an example which needs a clear start condition but will never finishes the defined operation.

Such a Level Controlled function block has no `xDone` output variable and no `Done` state. This behaviour is defined as a *Continuous Behaviour*.

In the context of this document we refer to this kind of behavior models as `LConC` and `LConTIC`.

See the detailed descriptions of the reference implementation for the different level controlled function blocks in the appendix (for `LCon`, `LConTI`, `LConTo`, `LConTITo`, `LConC` and `LConTIC`).

3.4. Common properties of these types of function blocks

- If a specific invocation of a function block detects a *start condition*, the output variable `xBusy` is set immediately to the status `TRUE`.
- As long as the defined operation of a function block is running, the output variable `xBusy` has the value `TRUE`.
- If the *ready condition* is reached, the output variable `xDone` is set to `TRUE` and the output variable `xBusy` is set to `FALSE`.
- If the *error condition* is reached, the output variable `xError` is set to `TRUE` and the output variable `xBusy` is set to `FALSE`. Additionally the output variable `eErrorID` will set to an error code other than `ERROR.NO_ERROR`. The `eErrorID` is defined here as an `ENUM` although users can also define this as `INTEGER`, `WORD`, `DWORD` or other datatype.
- If the defined operation can be fully processed in one invocation, the *ready condition* or the *error condition* is reached immediately and the `TRUE` status of the output variable `xBusy` is never be seen.
- With the rising edge of `xDone`, the status of all output variables will be frozen.
- As long as one of the output variables `xDone`, `xBusy` or `xError` has the status `TRUE` the defined operation of this function block has not yet completed, so a further invocation is necessary.

Timing constraints of these function blocks:

- `udiTimeLimit` ([μ s], 0 \Rightarrow no operating time limit):
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 invocation of this function block. The `udiTimeLimit` parameter can define how much time per invocation is permitted for consumption in the respective function block. Function blocks with an `udiTimeLimit` input variable are implemented in such

a way that the current invocation is exited when the task is complete (*Ready Condition*), or when the consumed time for this invocation has exceeded the settings from `udiTimeLimit`.

- `udiTimeOut` ([μ s], 0 \Rightarrow no operating time limit):
When processing its defined operation, a function block could be forced for example to wait for an external event. It can do this in an internal loop (Busy Wait) or it can check in each invocation whether its task can be completed in full. The `udiTimeOut` parameter can define then how much time is permitted for consumption for the defined operation. Function blocks with the `udiTimeOut` input variable are implemented in such a way that the current invocation is exited towards an *error condition* (`xError` \Rightarrow TRUE and `eErrorID` \Rightarrow ERROR.TIME_OUT) when the time interval as defined by `udiTimeOut` has been exceeded.

3.5. Error Domains and Error Codes

Every library provides its own error domain (Error Domain = Library namespace).

Every library provides a set range of possible values for an `ErrorID` output variable (Error Codes = ERROR enum data type).

All Function Blocks in this document have a Boolean output `xError` to indicate that an *error condition* has been reached. In that case the related information will be signaled with the value of the output `eErrorID`. The `eErrorID` represents an Integer, indicating the reason of the error. In many cases this integer value is used as input for an additional FB which converts the number to a related localized string in an applicable language. The set of values for a specific `eErrorID` are application dependent. In case several libraries are combined (several domains), there can be an overlap in the numbers of the `eErrorID`, meaning that the same number identifies a different error in a different domain. For this reason a value range definition for `eErrorID` per library must be done.

The error handling of a function block should be designed in a way that only error codes are returned, which are documented in the affected library. It is very convenient but not recommended simply to return untreated error codes from sub libraries. This would result in a bad user experience. It is recommended to map foreign error codes to the error range of the affected library.

In the following example we take a closer look to the relationship between two libraries, each with a specific domain of error codes. The first library may be called the “Memory Block Manager library” and is built in the namespace MBM. The second library may be called the “Function Block Factory” and is built in the namespace FBF. Each library defines its own ERROR enum data type.

The ERROR Enum of the library “Memory Block Manager” (MBM)

```

1  {attribute 'qualified_only'}
2  TYPE ERROR : (
3      NO_ERROR := 0, // The defined operation was executed successfully
4      NO_MEMORY := 10 // The memory pool has no further capacity
5      HANDLE_INVALID := 20, // The object was not created properly or has been already released
6      WRONG_ALIGNMENT := 30, // The structure description aligns not properly to the block specification
7      (*...*)
8  END_TYPE

```

The ERROR Enum of the library "Function Block Factory" (FBF)

```

1  {attribute 'qualified_only'}
2  TYPE ERROR : (
3      NO_ERROR := 0, // The defined operation was executed successfully
4      TIMEOUT := 1, // The specified operation time was exceeded
5      INVALID_PARAM := 10, // One or more function parameters have no valid value
6      NO_MEMORY := 20, // The extension of memory pool is not possible
7      (*...*)
8  END_TYPE

```

- Two libraries are isolated with a namespace (in this example FBF and MBM).
- Each ERROR Enum declaration should respect two predefined error codes.
 - NO_ERROR ⇒ 0 (Zero)
 - TIME_OUT ⇒ 1 (One)
- If the TIME_OUT error code has no usage in a specific domain the value should not be reused for another error code.
- Any error code needs a short description about the background of its error condition.
- An enum data type should be isolated from other enum data types with its own namespace ({attribute 'qualified_only'}). FBF.ERROR.NO_MEMORY has a completely different meaning as MBM.ERROR.NO_MEMORY.

Working together with sub libraries brings up the need for mapping the different error domains to the one local domain. The next example demonstrates the possible design of an error code mapping function. It handles the error codes (from CS.ERROR and CO.ERROR) of two sub libraries and tries to map these to the one local Error Enum (CANOPEN_KERNEL_ERROR) (All enum data types in this example have the base type INT).

```

FUNCTION MapError : CANOPEN_KERNEL_ERROR
VAR_INPUT
    iError : INT;
END_VAR

MapError := CANOPEN_KERNEL_ERROR.CANOPEN_KERNEL_UNKNOWN_ERROR;
IF iError = CS.ERROR.NO_ERROR THEN
    MapError := CANOPEN_KERNEL_ERROR.CANOPEN_KERNEL_NO_ERROR;
ELSIF iError > CS.ERROR.FIRST_ERROR AND iError < CS.ERROR.LAST_ERROR THEN
    CASE iError OF
        CS.ERROR.TIME_OUT      : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_KERNEL_TIMEOUT;
        CS.ERROR.REQUEST_ERROR : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_REQUEST_ERROR;
        CS.ERROR.WRONG_PARAMETER : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_WRONG_PARAMETER;
        CS.ERROR.NODEID_UNKNOWN : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_NODEID_UNKNOWN;
        CS.ERROR.SDOCHANNEL_UNKNOWN : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_SDOCHANNEL_UNKNOWN;
    ELSE
        MapError := CANOPEN_KERNEL_ERROR.CANOPEN_KERNEL_OTHER_ERROR;
    END_CASE
ELSIF iError > CO.ERROR.FIRST_ERROR AND iError < CO.ERROR.LAST_ERROR THEN
    CASE iError OF
        CO.ERROR.TIME_OUT      : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_KERNEL_TIMEOUT;
        CO.ERROR.NO_MORE_MEMORY : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_NO_MORE_MEMORY;
        CO.ERROR.WRONG_PARAMETER : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_WRONG_PARAMETER;
        CO.ERROR.NODEID_UNKNOWN : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_NODEID_UNKNOWN;
        CO.ERROR.NETID_UNKNOWN  : MapError := CANOPEN_KERNEL_ERROR.CANOPEN_NETID_UNKNOWN;
    ELSE
        MapError := CANOPEN_KERNEL_ERROR.CANOPEN_KERNEL_OTHER_ERROR;
    END_CASE
END_IF

```

This design assumes CS.ERROR.NO_ERROR has the same value as CO.ERROR.NO_ERROR and the rest of the value range of CS.ERROR and CO.ERROR is disjunct.

3.6. How to handle the STATE enum data type

The example implementations of the different behaviour models in this document depend strongly on the processing of a central state machine and its handling of the STATE enum data type. For each implementation there is a specific simplified picture of the STATE enum data type included, which is purely for readability and ease of comprehension. In a real library which may be implementing all the different behaviour models as specific function blocks we will find only one STATE enum data type. This data type will define all possible states in relation to all kinds of function blocks in the PLCopen behaviour model function block family.

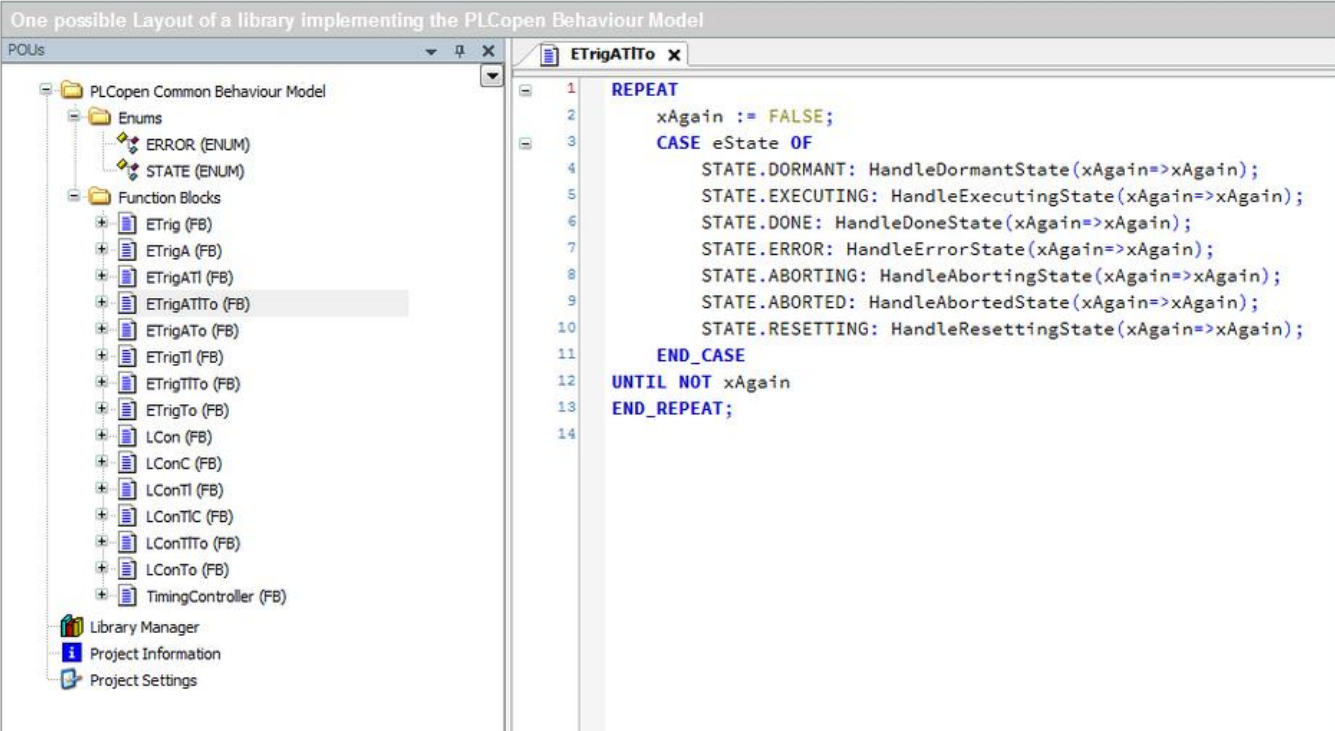
One possible design of the STATE enumeration data type for the PLCopen Behaviour Model

```

1  TYPE STATE :
2  (
3      DORMANT,    // Waiting for Start condition
4      EXECUTING,  // CyclicAction is running
5      DONE,       // Ready condition reached
6      ERROR,      // Error condition reached
7      ABORTING,   // AbortAction is running
8      ABORTED,    // Abort Condition reached
9      RESETTING  // ResetAction is running
10 );
11 END_TYPE

```

One possible Layout of a library implementing the PLCopen Behaviour Model



```

1  REPEAT
2      xAgain := FALSE;
3      CASE eState OF
4          STATE.DORMANT: HandleDormantState(xAgain=>xAgain);
5          STATE.EXECUTING: HandleExecutingState(xAgain=>xAgain);
6          STATE.DONE: HandleDoneState(xAgain=>xAgain);
7          STATE.ERROR: HandleErrorState(xAgain=>xAgain);
8          STATE.ABORTING: HandleAbortingState(xAgain=>xAgain);
9          STATE.ABORTED: HandleAbortedState(xAgain=>xAgain);
10         STATE.RESETTING: HandleResettingState(xAgain=>xAgain);
11     END_CASE
12 UNTIL NOT xAgain
13 END_REPEAT;
14

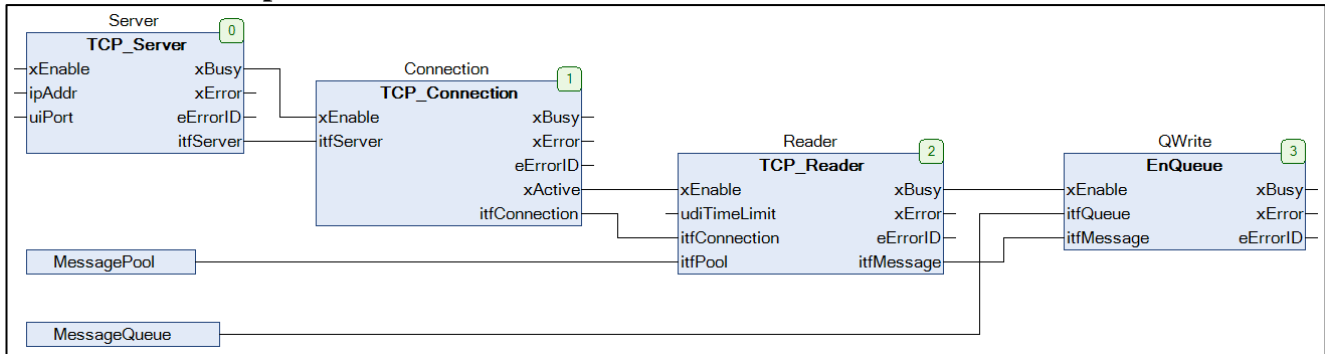
```

3.7. Cooperation of various function blocks

The cooperation of various function blocks each follow one of the previous described behaviour models. Here an example is shown of this cooperation.

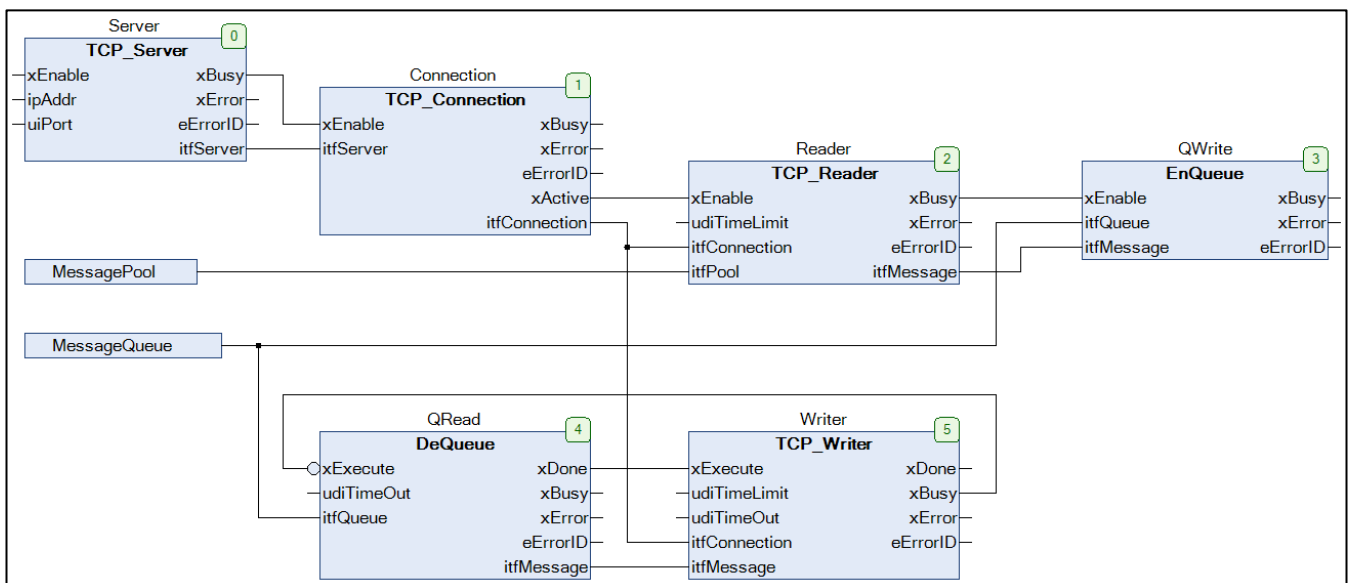
If the design of function block interfaces consistently takes into account this PLCopen Common Behaviour Model then complex relationships between instances of them can easily be expressed and understood, in particular in graphical languages. This example demonstrates the opportunities which are resulting from the use of the PLCopen Common Behaviour Model.

TCP Reader Example



A *TCP Server* (LConC) listens at a specific endpoint. This endpoint is defined with a tuple consisting of an IP address and a related port number. After a connection to a TCP client has been established, the related *Connection* (LConC) will be activated. With the help of a *Reader* (LConTlC), a message structure is allocated out of a *MessagePool*. This structure is then used to store the information received from the client. For further processing, the message is then handed over to function block *QWrite* (LCon) that interfaces a queue.

Extending the Example to a Complete EchoServer



A *Writer* (ETrigTlTo) is working now at the end of the original Queue and is requesting (dequeuing) the stored messages. This message is sent back over the original *Connection* to the client. So every received message will be sent back to the original sender, the client. With this small modification, a complete *Echo Server* is created.

Note:

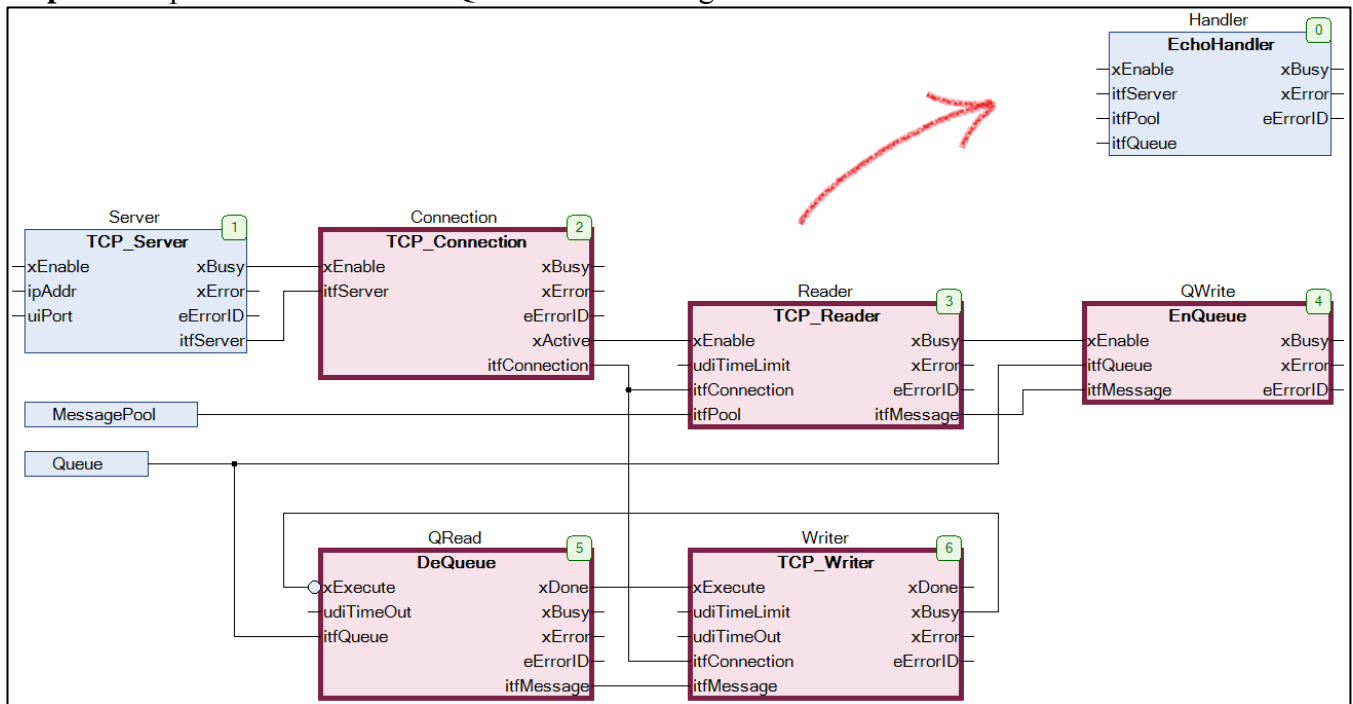
The Reader function block in the example above must have the possibility to provide a new message in each cycle. If there is no new message available in the current cycle, the output variable is set to null. So in this case, a level-control model was chosen. Sending a message with the help of the *Writer* function block in the example above can last more than one cycle. So it is important to choose an edge-triggered model.

In order to combine these two function blocks into an efficient working team, a queue mechanism is necessary.

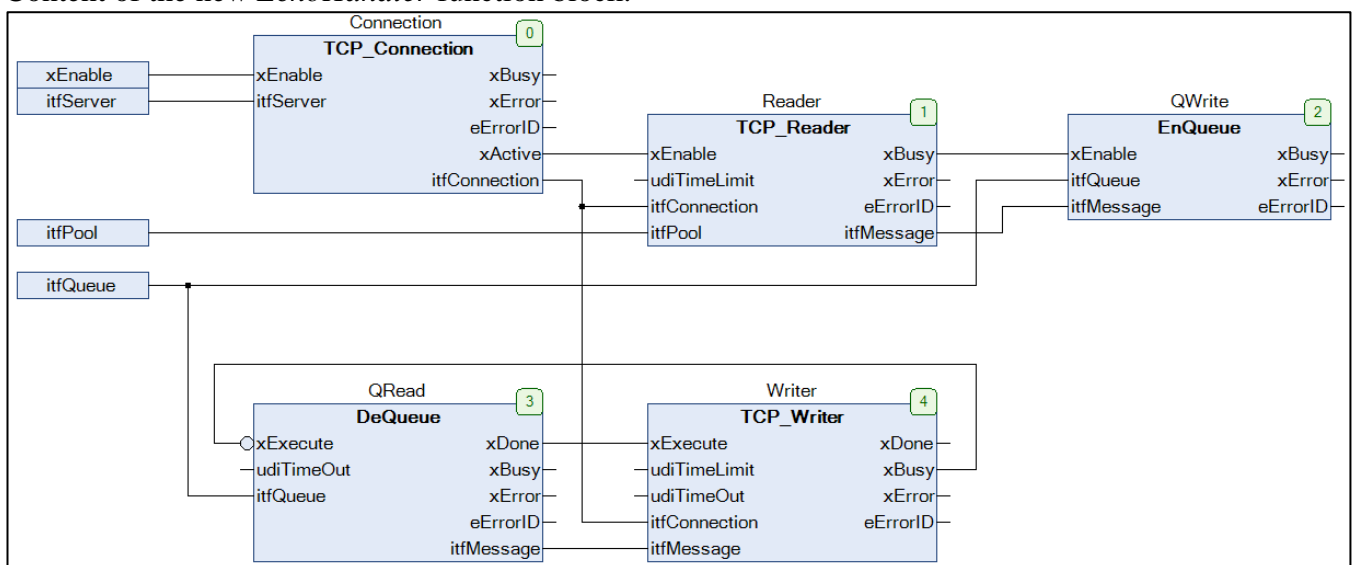
Note the connection between the `xBusy` output variable of the Writer and the negated `xExecute` input variable of the QReader as handshake: in this way no extra cycle is necessary and no data will be lost, making it very efficient.

Transformation to a Multithreaded EchoServer

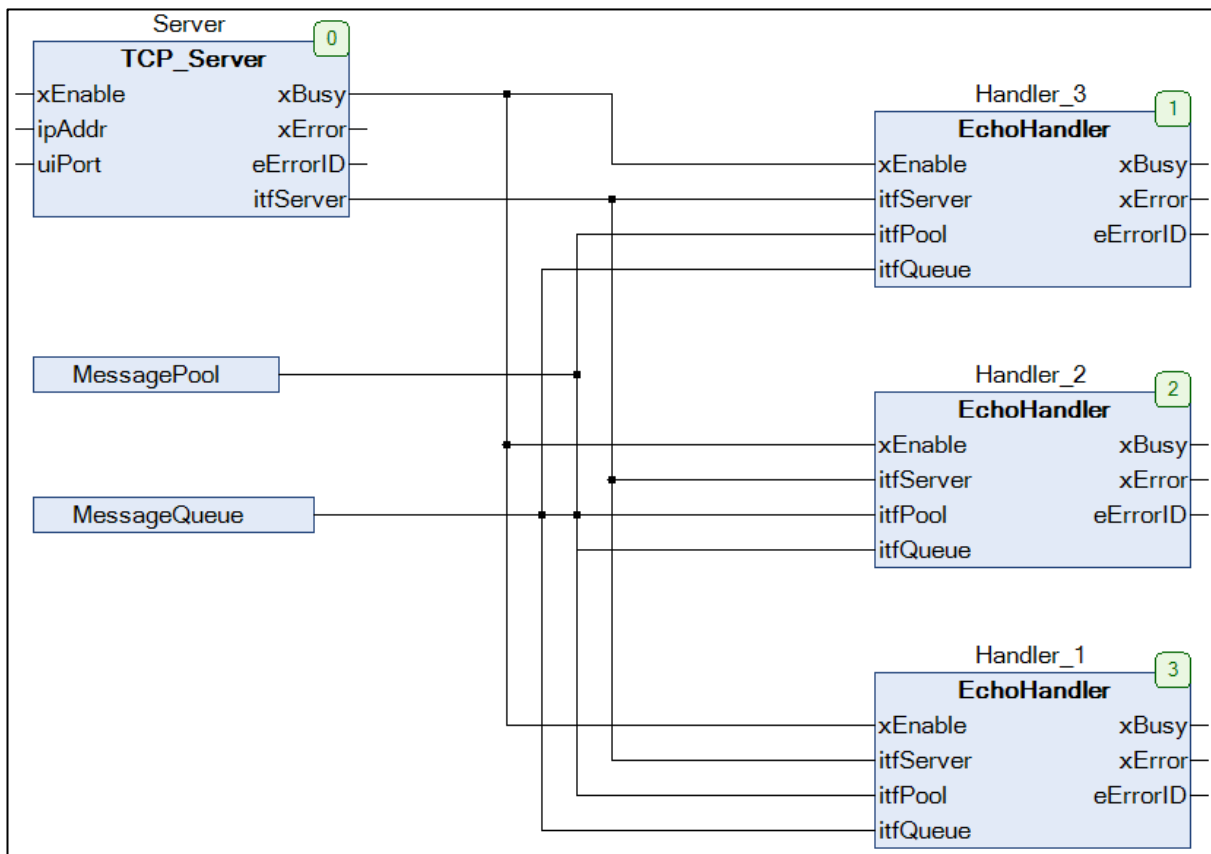
Step 1: Encapsulate the “Reader – Queue – Writer” logic into a function block called *EchoHandler*



Content of the new *EchoHandler* function block:



Step 2: Declare a number of *EchoHandler* instances and assign these instances to different tasks.



The result is a complete multi-threaded EchoServer.

4. Introduction in the object oriented features of IEC 61131-3

The 3rd Edition of the IEC 61131-3 introduces object oriented features to the standard. In particular, the standard defines methods, classes, interfaces, inheritance, etc. In this document those features are not further explained in detail. Those concepts are considered to be known to the reader.

This document makes careful use of those features. Only protected and private methods are used to structure the code of the resulting function blocks.

These function blocks define a set of state machines, that are considered to be very common in their behaviour. These function blocks are not supposed to be used directly in an application, but as base function blocks for further usage. To make use of such a function block, a new function block could be derived from the appropriate base function block by inheritance. The protected methods should be overridden to implement the specific behaviour.

However, no feature of the 3rd Edition is really necessary to create compliant function blocks. Every method call in the code of this document can be replaced by the code of the method. The base function blocks can then be considered as templates. To make use of the base function block, a new function block could be created by copying the base function block. The calls to the protected methods should be replaced to implement the specific behaviour.

The basic FBs ETrig and LCon can be extended via inheritance to the FBs ETrigA, ETrigATl, ETrigATo, ETrigATITo, ETrigTITo, ETRigTo, ETRigTl and LConTl, LConTo, LConTITo respectively.

5. Explanation of Rising Edge triggered FBs

5.1. The basic FB: ETrig

This is the 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
<pre> FUNCTION_BLOCK ETrig VAR_INPUT xExecute: BOOL; END_VAR VAR_OUTPUT xDone: BOOL; xBusy: BOOL; xError: BOOL; eErrorID: INT; END_VAR </pre>	

To describe the functionality and behaviour of this FB one can make use of a state diagram, describing the different states as well as the transitions as a result of an activity.

Basically there are 4 states: *Dormant*, *Busy* and *Done*, combined with *Reset* and *Error*. See hereunder. Listed are also the transitions including error behaviours’.

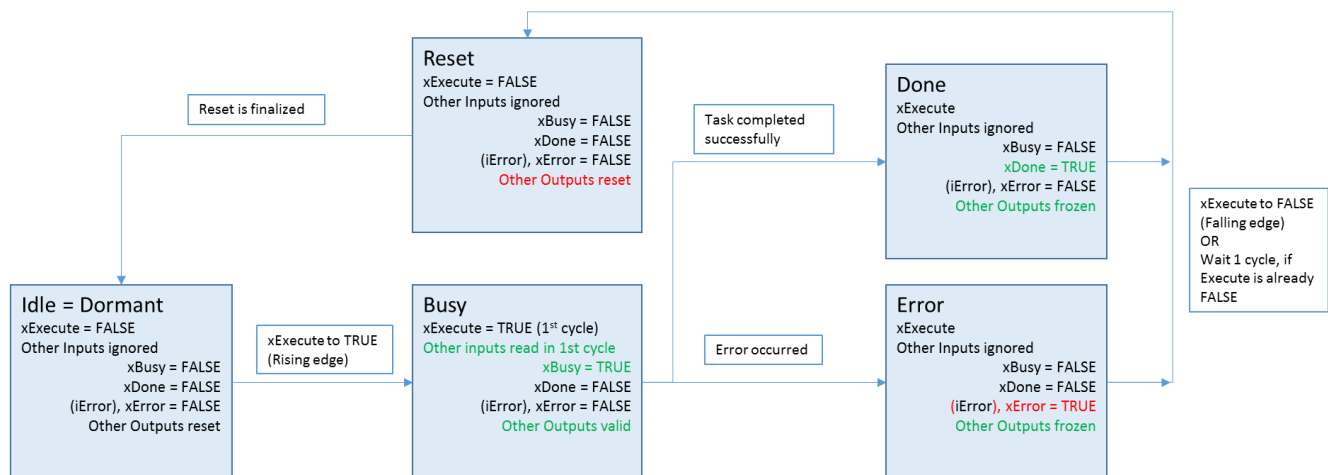


Figure 4: Basic State Diagram

After a rising edge was detected at the input *xExecute* the internal state is switched from *Dormant* to *Busy* while in that invocation all the inputs are sampled and stored. The output *xBusy* will be set to *TRUE*. The defined operation will be started.

While working on the defined operation, a number of conditions can appear that lead to the exit from the *Busy* state. This means the value of the output variable *xBusy* will be set to *FALSE* and the internal state will be switched from *Busy* to one of the states *Done* or *Error*. This change will be mirrored to one of the output variables *xDone* or *xError*. Only one output variable of this set of variables can have the status *TRUE* at the same time.

Ready Condition: If the operation has reached its ready condition without any error and timing constraints the output variable *xDone* is set to *TRUE*. This means the internal state is switched from *Busy* to *Done*.

Error Condition: If an error condition was detected, the output variable *xError* is set to *TRUE*. This means the internal state is switched from *Busy* to *Error*. In addition, one of the defined error

codes (one value out of the local enumeration type `ERROR`) will be assigned to the output variable `eError`.

The values `TRUE` of one of the output variables `xDone` or `xError` has to be stable for minimum one cycle.

After a `FALSE` status for the input variable `xExecute` is detected, the internal state will be switched to `Reset`.

All Outputs will be initialized to their default status (Reset Outputs). All claimed resources will be freed. Especially the output variables `xDone` and `xError` will be set to `FALSE`.

After doing this reinitialization work, the internal state will be switched from *Reset* to *Dormant*.

(Note: One has to make sure that the state `TRUE` of one of the output variables `xDone` or `xError` must be stable for a minimum of one cycle, e.g. add a sub-state `Sync`.)

Example of the ST Program for the FB ETrig with OO

For this example the Object Oriented features are used. For an example in the classical approach refer to Appendix 2 Example without using Object Oriented features.

The following methods are used in the code: `prvResetOutputs()`, `prvStart()` and `prvCyclicAction()`

The STATE Enumeration:

```
TYPE STATE :
(
    DORMANT,    // Waiting for Start
    EXECUTING,  // CyclicAction is running
    DONE,       // CyclicAction is complete
    ERROR,      // Error condition reached
    RESETTING  // ResetAction is running
);
END_TYPE
```

The ERROR Enumeration:

```
TYPE ERROR :
(
    NO_ERROR := 0,
    TIME_OUT := 1
    (*...*)
);
END_TYPE
```

Implementation of the Function Block ETrig:

```
FUNCTION_BLOCK ETrig
VAR_INPUT
    // Rising edge starts defined operation
    // FALSE ⇒ reset the defined operation
    // after ready condition was reached
    xExecute: BOOL;
END_VAR

VAR_OUTPUT
    // ready condition reached
    xDone: BOOL;
    // operation is running
    xBusy: BOOL;
    // error condition reached
    xError: BOOL;
    // error code describing error condition
    eErrorID : ERROR;
END_VAR

VAR
```

```
        eState : STATE;
        xFIRSTINVOCATION : BOOL := TRUE;
        xResetRequest : BOOL;
    END_VAR

    VAR_TEMP
        xAgain : BOOL;
    END_VAR

    REPEAT
        xAgain := FALSE;
        CASE eState OF
            STATE.DORMANT:    HandleDormantState(xAgain=>xAgain);
            STATE.EXECUTING:  HandleStartState(xAgain=>xAgain);
            STATE.DONE:       HandleDoneState(xAgain=>xAgain);
            STATE.ERROR:      HandleErrorState(xAgain=>xAgain);
            STATE.RESETTING:  HandleResettingState(xAgain=>xAgain);
        END_CASE

    UNTIL NOT xAgain
    END_REPEAT;
```

The Handler for the Dormant State

```
METHOD PRIVATE FINAL HandleDormantState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR

IF xExecute THEN
    xBusy := TRUE;
    eState := STATE.EXECUTING;
    xAgain := TRUE;
END_IF
```

The Handler for the Executing State

```
METHOD PRIVATE FINAL HandleExecutingState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR
VAR
    xComplete : BOOL;
END_VAR

CyclicAction(
    xComplete=>xComplete,
    eErrorID=>eErrorID
);

IF eErrorID <> ERROR.NO_ERROR THEN
    eState := STATE.ERROR;
    xAgain := TRUE;
ELSIF xComplete THEN
    eState := STATE.DONE;
    xAgain := TRUE;
END_IF
```

The Handler for the Done State

```
METHOD PRIVATE FINAL HandleDoneState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR

IF xDone AND (xResetRequest OR NOT xExecute) THEN
    eState := STATE.RESETTING;
    xAgain := TRUE;
```

```
ELSE
    xBusy := FALSE;
    xDone := TRUE;
    xResetRequest := NOT xExecute;
    xAgain := FALSE; (* !!! *)
END_IF
```

The Handler for the Error State

```
METHOD PRIVATE FINAL HandleErrorState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR

IF xError AND (xResetRequest OR NOT xExecute) THEN
    eState := STATE.RESETTING;
    xAgain := TRUE;
ELSE
    xBusy := FALSE;
    xError := TRUE;
    xResetRequest := NOT xExecute;
    xAgain := FALSE; (* !!! *)
END_IF
```

The Handler of the Resetting State

```
METHOD PRIVATE FINAL HandleResettingState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR
VAR
    xComplete : BOOL;
END_VAR

ResetAction(xComplete=>xComplete);

IF xComplete THEN
    xBusy := FALSE;
    xDone := FALSE;
    xError := FALSE;
    eErrorID := ERROR.NO_ERROR;
    eState := STATE.DORMANT;
    xFirstInvocation := TRUE;
    xAgain := xResetRequest; (* !!! *)
    xResetRequest := FALSE;
END_IF
```

Example implementation of the application specific Methods

This code is listed here just as an example. The content needs to be adapted to the real requirements of a specific application.

The Implementation of the CyclicAction

```
METHOD PROTECTED CyclicAction
VAR_OUTPUT
    xComplete : BOOL;
    eErrorID : ERROR;
END_VAR

IF xFirstInvocation THEN
    (* Starting *)
    // for the first (!) invocation,
    // sample the input variables
    xFirstInvocation := FALSE;
END_IF

(* Executing *)
```

```
// working to reach the ready condition
// ⇒ xComplete := TRUE
// if an error condition is reached
// ⇒ set eErrorID to a value other than ERROR.NO_ERROR

xComplete := TRUE;
eErrorID := ERROR.NO_ERROR;

IF xComplete OR eErrorID <> ERROR.NO_ERROR THEN
    (* Cleaning *)
    // if possible free as much allocated resources
    // as possible
END_IF
```

The Implementation of the ResetAction

```
METHOD PROTECTED ResetAction
VAR_OUTPUT
    xComplete : BOOL;
END_VAR

// free all allocated resources
// reinitialize instance variables

xComplete := TRUE;
```

5.2. Adding the Aborting Functionality to the basis

The next step is to add the aborting functionality to the basic function block. With aborting the on-going functionality is interrupted with another command. For this we need an input (`xAbort`), to initiate the aborting functionality, and an output (`xAborted`) to reflect the status: it is `SET` when the FB is aborted. (Note that this representation is different from the PLCopen Motion Control Specification where the aborting functionality is hidden while shown here as an input)

Textual representation	Graphical representation
<pre> FUNCTION_BLOCK ETrigA VAR_INPUT xExecute: BOOL; xAbort: BOOL; END_VAR VAR_OUTPUT xDone: BOOL; xBusy: BOOL; xError: BOOL; xAborted: BOOL; eErrorID: INT END_VAR </pre>	

With this added functionality, the state diagram gets more complex since there is a state Abort added:

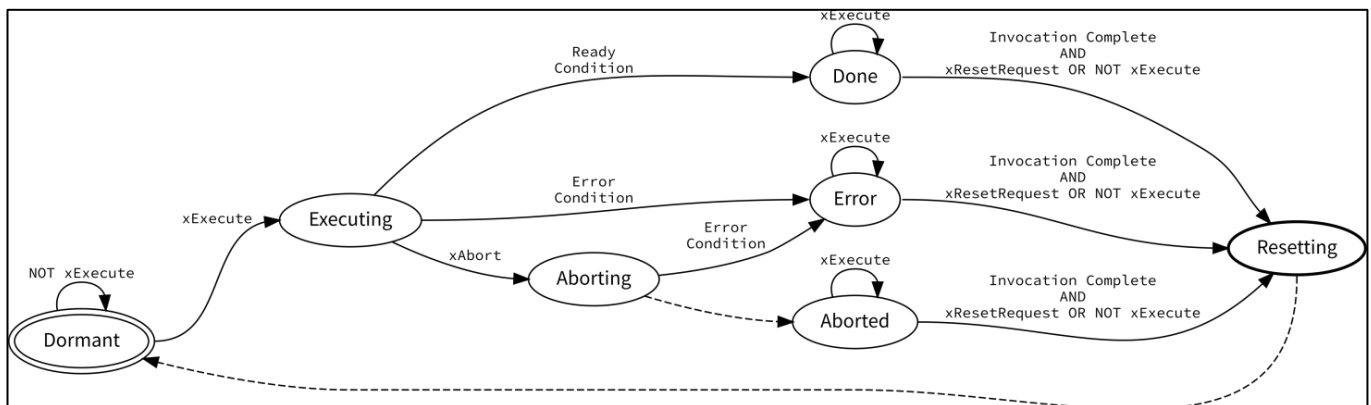


Figure 5: The State Diagram of ETrigA

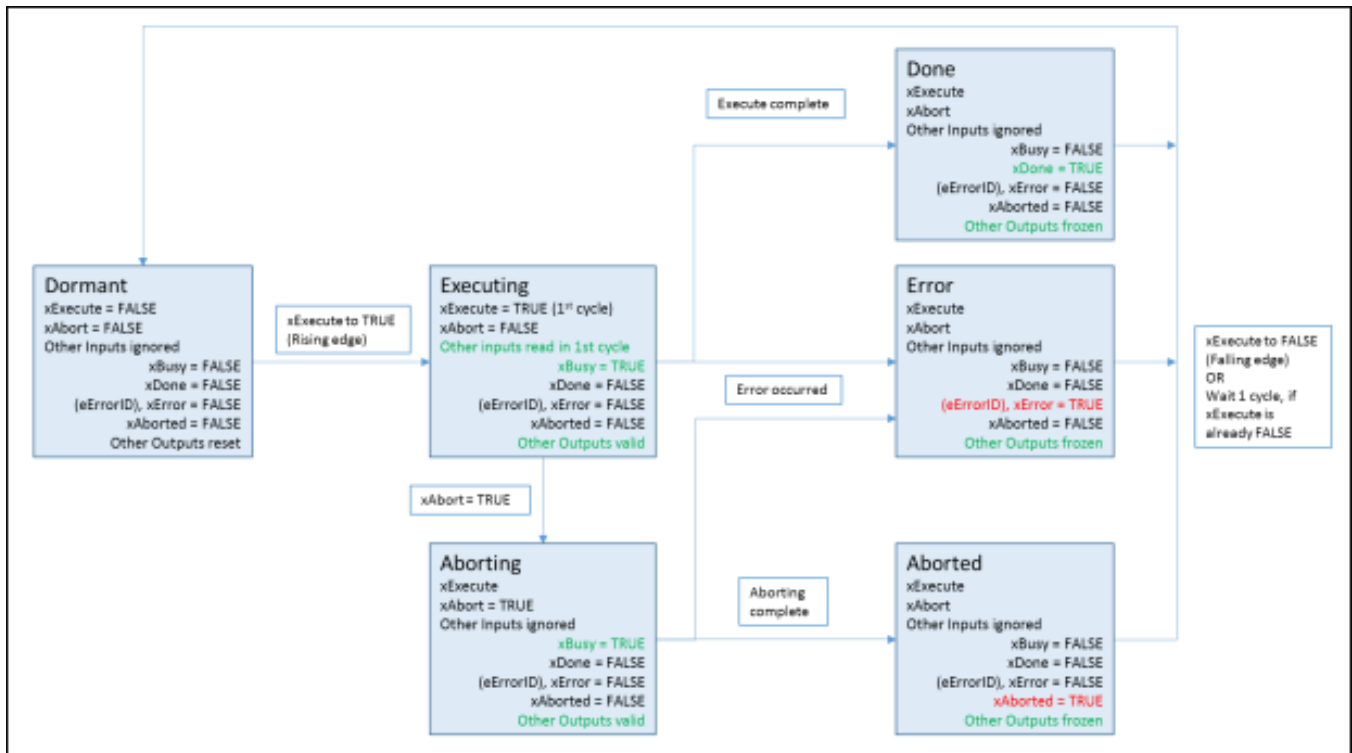


Figure 6: State Diagram of ETrigA with output status

Note that the Resetting State in this picture is included in the Dormant state.

After a rising edge was detected at the input `xExecute` the internal state is switched from *Dormant* to *Executing* while in that invocation all the inputs are sampled and stored.

The output `xBusy` will be set to `TRUE`. This means the internal state is switched to *Busy*.

The defined operation will be started (CyclicAction).

While working on the defined operation a number of conditions can appear, that lead to the exit from the *Busy* state. This means the value of the output variable `xBusy` will be set to `FALSE` and the internal state will be switched from *Busy* to one of the states *Done*, *Error* or *Aborted*. This change will be mirrored to one of the output variables `xDone`, `xError` or `xAborted`. Only one output variable of this set of variables can have the status `TRUE` at the same time.

Ready Condition: If the operation has reached its ready condition without any error and timing constraints the output variable `xDone` is set to `TRUE`. This means the internal state is switched from *Busy* to *Done*.

Error Condition: If an error condition was detected, the output variable `xError` is set to `TRUE`. This means the internal state is switched from *Busy* to *Error*. In addition, one of the defined error codes (one value out of the local enumeration type `ERROR`) will be assigned to the output variable `eErrorID`.

Abort Condition: If a status of `TRUE` was detected for the `xAbsort` input variable, the abort condition is reached. This means the internal state is switched from *Busy* to *Abort*. Any current action of the defined operation will be aborted. After this is done the output variable `Aborted` will be set to `TRUE` and the internal state is switched from *Abort* to *Aborted*.

The value of `TRUE` of one of the output variables `xDone`, `xError` or `xAborted` has to be stable for minimum one cycle.

After a `FALSE` status for the input variable `xExecute` is detected, the internal state will be switched to *Reset*.

All Outputs will be initialized to their default status (Reset Outputs). All claimed resources will be freed.

Especially the output variables `xDone`, `xError` and `xAborted` will be set to `FALSE`.

After doing this reinitialization work, the internal state will be switched from *Reset* to *Dormant*.

Example of a SFC Program

Hereunder an example of the implementation in Sequential Function Chart, SFC of the state diagram of Figure 5: The State Diagram of ETrigA. The different states are clearly identifiable in the Steps, linked to the related Action Blocks with the Action Qualifiers. The transition conditions between the Steps are linked to the transitions in the State Diagram.

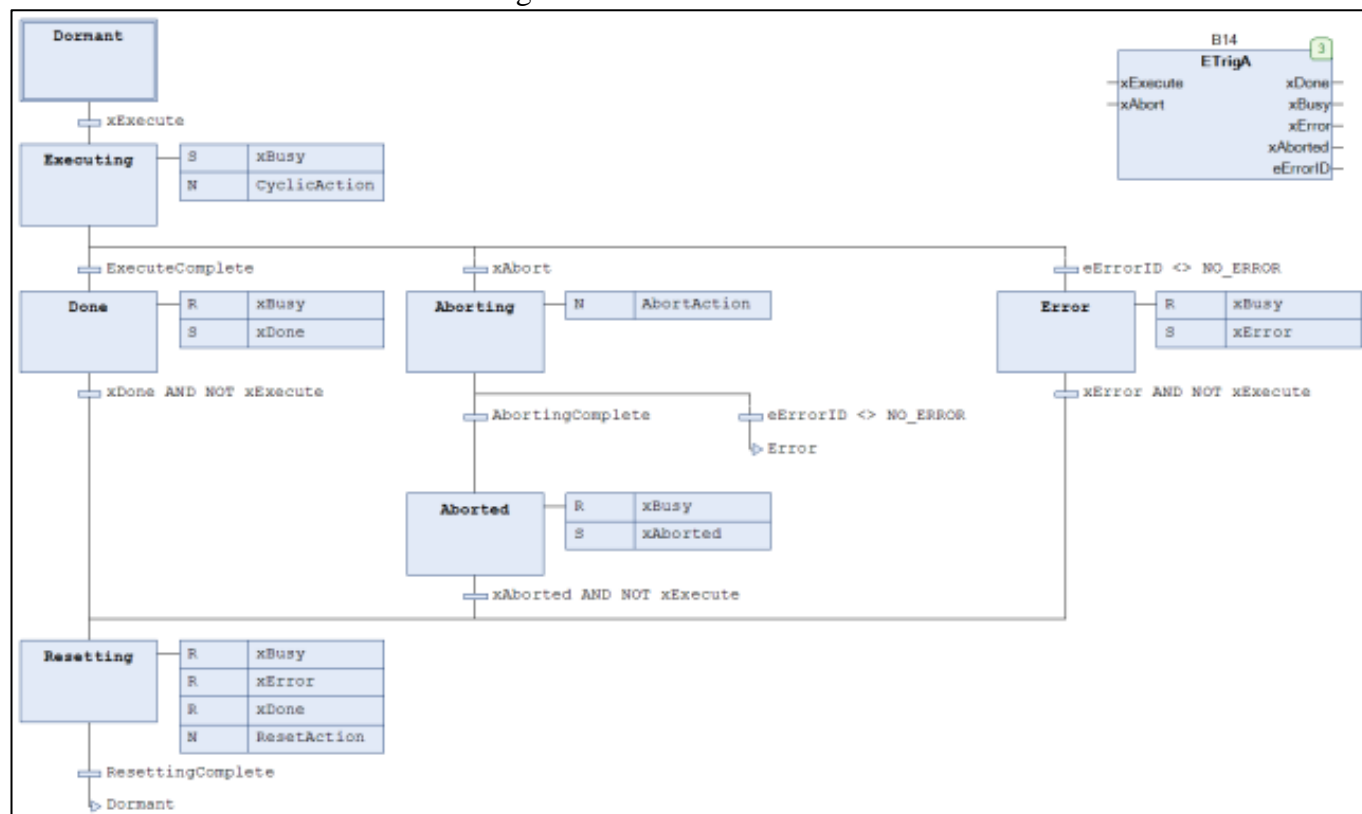


Figure 7: Example of the ETrigA State Diagram in SFC

Note to the used Action qualifiers:

N	Non-stored, executes while associated step is active
R	Resets a stored action
S	Sets an action active, i.e. stored

For more details refer to the IEC 61131-3 standard.

For an example of the code and the timing diagrams, refer to Appendix 1 Datasheets of the Edge Triggered and Level Controlled FBs Par. Appendix 1.3.5 ETrigA.

5.3. Adding timer functionality

On top of the abort functionality or 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 μ s) as specified by the input value `udiTimeOut`;
2. **TimeLimit (TI)**: 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 (**TITo**).

What does `udiTimeLimit` do?

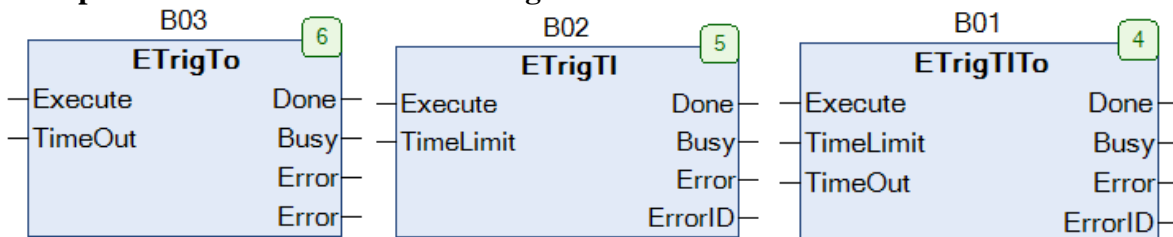
A function block could, for example, complete a complex task in a loop. The larger the task is, the more time that is consumed in the current task of this function block. The `udiTimeLimit` parameter can define how much time per invocation is permitted for consumption in the respective function block. Function blocks with `udiTimeLimit` input must implement their `CyclicAction` in such a way that this method is exited when the task is complete (`ReadyCondition`, `xComplete:=TRUE`) or when the consumed time in this cycle has exceeded the settings from `udiTimeLimit` (`xComplete:=FALSE`).

What does `udiTimeOut` do?

When processing its cycle action, a function block for example could 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 *Busy* state.

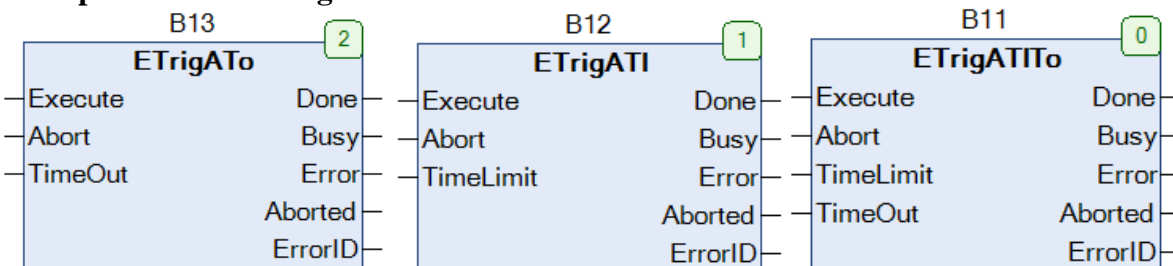
Function blocks with the `udiTimeOut` input must implement their `CyclicAction` in such a way that this method is exited towards `xError` (`eError := ERROR.TIME_OUT`) when the time interval as defined by `udiTimeOut` has been exceeded.

Examples with timers without Aborting



Note: the inputs and outputs are listed here without prefixes in the names

Examples with Aborting and timers



Note: the inputs and outputs are listed here without prefixes in the names

For the state diagram let us look at the most extended version: `ETrigATITo`. The condition `TimeOut` results in a transition to the state *Error*. The condition `TimeLimit` is applicable in *Executing*.

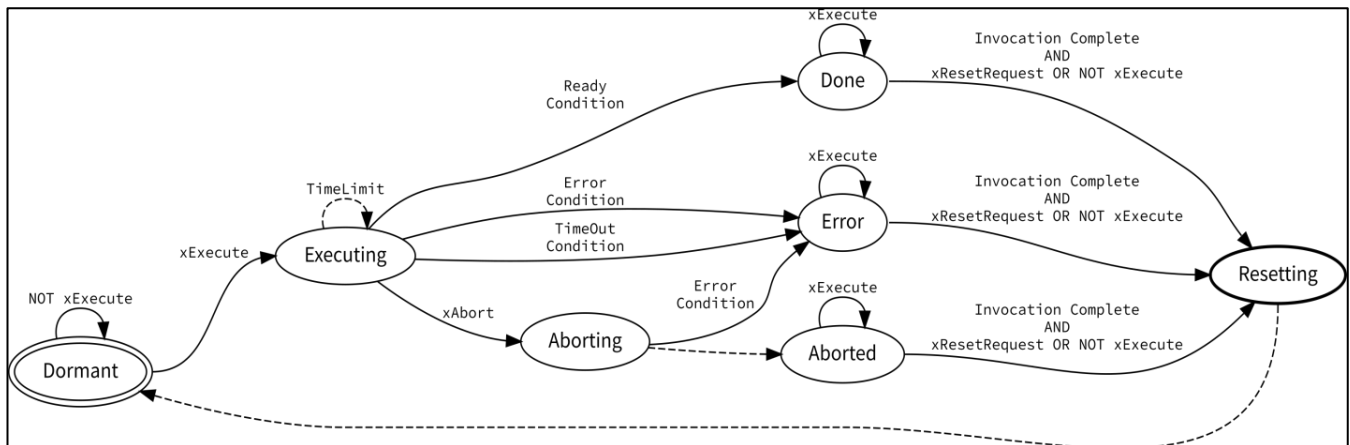


Figure 8: State Diagram for ETrigATTo

Note: there are no R_Trig or F_Trig in this state diagram. The values of the parameters are relevant here. For example from *Dormant* to *Executing* one checks the value of *xExecute*

Detailed description of the Function Block ETrigATTo

1. The function block is called inside of a POU one time per task cycle without any conditions. This is called an invocation.
2. After a rising edge has been detected at the input variable *xExecute*, the internal state is switched from *Dormant* to *Executing*.
3. The status of all other input variables then *xExecute* and *xAbort* will be sampled and stored locally (see *xFirstInvocation* inside *CyclicAction*). Thus, later changes of these inputs cannot influence the defined operation while it is running 0.
4. The output *xBusy* will be set to TRUE.
At this point in time it would be possible to set the input variable *xExecute* to the status FALSE (*quick handshake*).
5. The defined operation will be started (see the comments inside *CyclicAction*).
6. If the operating time for the current invocation is higher than the time (in μ s) as specified by the input value *xTimeLimit*, the operation will be interrupted and continued in the next invocation (see *xTimeLimit* in *CyclicAction*).
7. While working on the defined operation, a number of conditions can appear that lead to the exit from the *Executing* state. This means the value of the output variable *xBusy* will be set to FALSE and the internal state will be switched from *Executing* to one of the states *Done*, *Error* or *Aborted*. This change will be mirrored to one of the output variables *xDone*, *xError* or *xAborted*. Only one output variable of this set of variables can have the status TRUE at the same time. With the falling edge of *xBusy* the input variable *xExecute* is sampled and its inverted value is stored as a *reset request* (see *xResetRequest* in the methods *HandleDoneState*, *HandleErrorState* and *HandleAbortedState*).
 - a. Ready Condition:
If the operation has reached its ready condition without any error and timing constraints (see *xComplete* in *CyclicAction*), the output variable *xDone* is set to TRUE. This means the internal state is switched from *Executing* to *Done*.
 - b. Error Condition:
If an error condition was detected (see *eErrorID* in *CyclicAction*), the output variable *xError* is set to TRUE. This means the internal state is switched from *Executing* to *Error*. In addition, one of the defined error codes (one value out of the local enumeration type *ERROR*) will be assigned to the output variable *eErrorID*.

- c. **Timeout Condition:**
If the overall operating time of the defined operation is higher than the time (in μ s) as specified by the input value `udiTimeOut`, a timeout condition will be detected causing the output variable `xError` to be set to `TRUE`. This means the internal state is switched from *Executing* to *Error*. Furthermore, the output variable `eErrorID` is set to a special error code (`ERROR.TIME_OUT`).
 - d. **Abort Condition:**
If a status of `TRUE` was detected for the `xAbort` input variable, the abort condition is reached. This means the internal state is switched from *Executing* to *Aborting*. Any current action of the defined operation will be aborted. After this is done (see `xComplete` inside `AbortAction`), the output variable `xAborted` will be set to `TRUE` and the internal state is switched from *Aborting* to *Aborted*. If an error condition was detected (see `eErrorID` in `AbortAction`), the output variable `xError` is set to `TRUE`. This means the internal state is switched from *Aborting* to *Error*.
- 8. As a reaction to the rising edge of one of the output variables `xDone`, `xError` or `xAborted` it would be possible to set the status of the input variable `xExecute` again to `TRUE` (*quick handshake*).
 - 9. The value of `TRUE` of one of the output variables `xDone`, `xError` or `xAborted` must be stable for a minimum of one invocation. This means the internal states *Done*, *Error* or *Aborted* must be active for a minimum of one invocation. This property guarantees that the values of the output variables are valid and stable for a minimum of one invocation.
 - 10. The status of output variables other than `xDone`, `xBusy`, `xError`, `xAborted` or `eErrorID` are valid only while `xDone` has the status `TRUE`.
 - 11. After a `FALSE` status for the input variable `xExecute` is detected (*standard handshake*) or a *reset request* is active (*quick handshake*), the internal state will be switched from *Done*, *Error* or *Aborted* to *Resetting*.
 - 12. All outputs will be initialized to their default statuses (`ResetAction`). All claimed resources will be freed. Specifically, the output variables `xDone`, `xError` and `xAborted` will be set to `FALSE`. After executing the code of the `ResetAction` the function block should be prepared for a new switch of the internal state from *Dormant* to *Executing*.
 - 13. After doing this reinitialization work, the internal state will be switched from *Resetting* to *Dormant* (see `xComplete` inside `ResetAction`).

5.4. Example of the ST Program for ETrigATITo

An example of a possible ST program is shown in Appendix 1.3.8 ETrigATITo.

6. Explanation of a Level Controlled FB

6.1. Basic Level Controlled FB

This is the equivalent of the previous most simple FB, but now as level controlled.

Textual representation	Graphical representation
<pre> FUNCTION_BLOCK LCon VAR_INPUT xEnable: BOOL; END_VAR VAR_OUTPUT xDone: BOOL; xBusy: BOOL; xError: BOOL; eErrorID: INT; END_VAR </pre>	

State Diagram Basic Level Controlled FB

This state diagram also has 5 states: *Dormant*, *Executing*, *Done*, *Error* and *Aborting*, although for clarity we added one state in Figure 9: Overview State Diagram Level Controlled FB to show clearly the transition back to the state *Resetting*. In Figure 10: Example of the state diagram for a basic level controlled FB this state is again incorporated in the *Dormant* state.

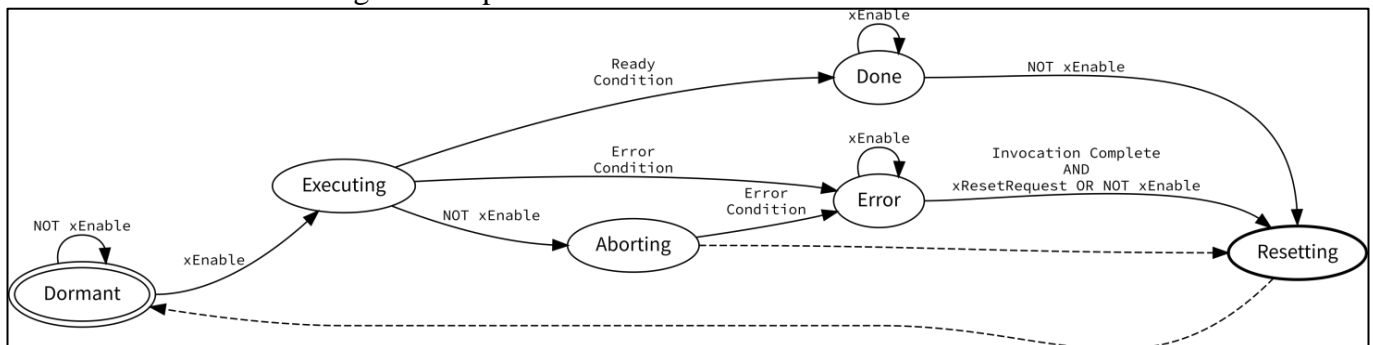


Figure 9: Overview State Diagram Level Controlled FB (LCon)

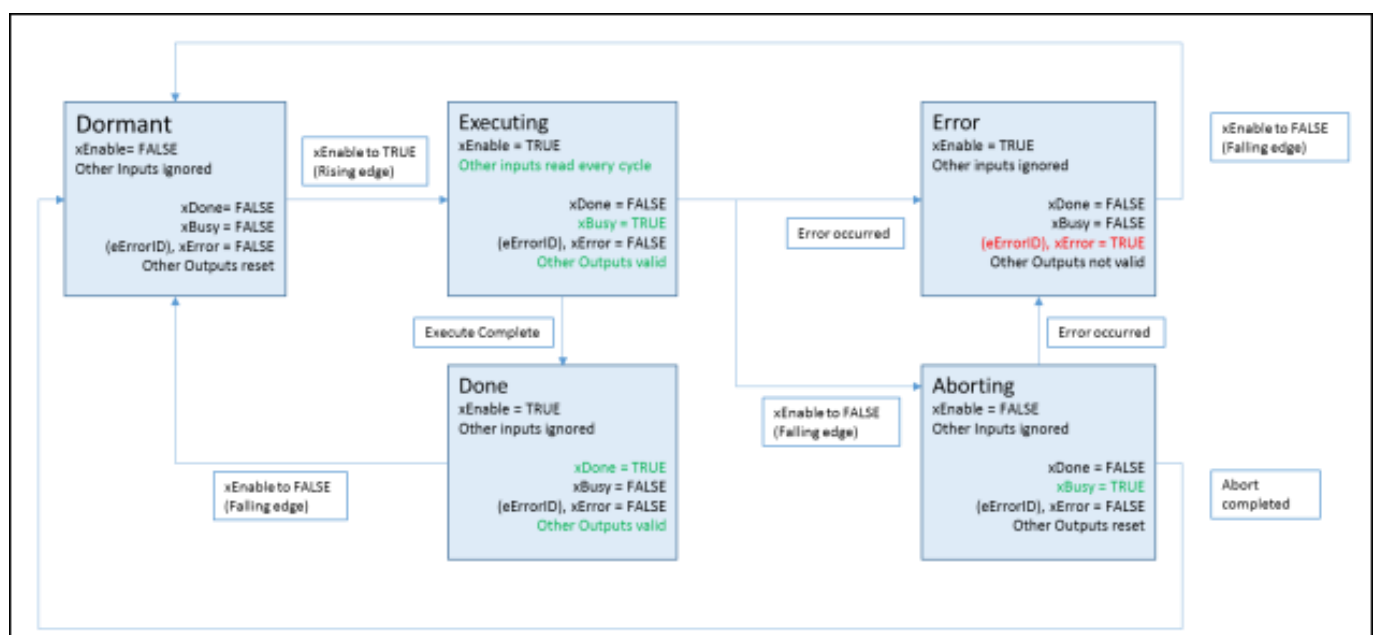


Figure 10: Example of the state diagram for a basic level controlled FB (LCon)

6.2. Example of the ST Program for the Function Block LCon in OO

The STATE Enumeration:

```
TYPE STATE :
(
    DORMANT,    // Waiting for xEnable
    EXECUTING,  // CyclicAction is running
    ABORTING,   // AbortAction is running
    DONE,       // Ready condition reached
    ERROR,      // Error condition reached
    RESETTING  // ResetAction is running
);
END_TYPE
```

The ERROR Enumeration

```
TYPE ERROR :
(
    NO_ERROR := 0,
    TIME_OUT := 1
    (* ... *)
);
END_TYPE
```

Implementation of the Function Block LCon:

```
FUNCTION_BLOCK LCon
VAR_INPUT
    // TRUE ⇒ activate the defined operation
    // FALSE ⇒ abort/reset the defined operation
    xEnable: BOOL;
END_VAR
VAR_OUTPUT
    // ready condition reached
    xDone: BOOL;
    // operation is running
    xBusy: BOOL;
    // error condition reached
    xError: BOOL;
    // error code describing error condition
    eErrorID : ERROR;
END_VAR
VAR
    eState : STATE;
    xResetRequest : BOOL;
END_VAR
VAR_TEMP
    xAgain :BOOL;
END_VAR

REPEAT
    xAgain := FALSE;
    CASE eState OF
        STATE.DORMANT: HandleDormantState(xAgain=>xAgain);
        STATE.EXECUTING: HandleExecutingState(xAgain=>xAgain);
        STATE.DONE: HandleDoneState(xAgain=>xAgain);
        STATE.ERROR: HandleErrorState(xAgain=>xAgain);
        STATE.ABORTING: HandleAbortingState(xAgain=>xAgain);
        STATE.RESETTING: HandleResettingState(xAgain=>xAgain);
    END_CASE
UNTIL NOT xAgain
END_REPEAT;
```

The Handler for the *Dormant* State:

```
METHOD PRIVATE FINAL HandleDormantState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR

IF xExecute THEN
    xBusy := TRUE;
    eState := STATE.EXECUTING;
    xAgain := TRUE;
END_IF
```

The Handler for the *Executing* State:

```
METHOD PRIVATE FINAL HandleExecutingState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR
VAR
    xComplete : BOOL;
    xTimeOut : BOOL;
END_VAR

IF xEnable THEN
    CyclicAction(
        xComplete=>xComplete,
        eErrorID=>eErrorID
    );
END_IF

IF eErrorID <> ERROR.NO_ERROR THEN
    eState := STATE.ERROR;
    xAgain := TRUE;
ELSIF NOT xEnable THEN
    eState := STATE.ABORTING;
    xAgain := TRUE;
ELSIF xComplete THEN
    eState := STATE.DONE;
    xAgain := TRUE;
END_IF
```

The Handler for the *Aborting* State:

```
METHOD PRIVATE FINAL HandleAbortingState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR
VAR
    xComplete : BOOL;
END_VAR

AbortAction(
    xComplete=>xComplete,
    eErrorID=>eErrorID
);

IF eErrorID <> ERROR.NO_ERROR THEN
    eState := STATE.ERROR;
    xAgain := TRUE;
ELSIF xComplete THEN
    eState := STATE.RESETTING;
    xAgain := TRUE;
END_IF
```

The Handler for the *Done* State:

```
METHOD PRIVATE FINAL HandleDoneState
VAR_OUTPUT
```

```
        xAgain : BOOL;
END_VAR

IF xDone AND NOT xEnable THEN
    eState := STATE.RESETTING;
    xAgain := TRUE;
ELSE
    xBusy := FALSE;
    xDone := TRUE;
    xAgain := FALSE; (* !!! *)
END_IF
```

The Handler for the *Error* State:

```
METHOD PRIVATE FINAL HandleErrorState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR

IF xError AND (xResetRequest OR NOT xEnable) THEN
    eState := STATE.RESETTING;
    xAgain := TRUE;
ELSE
    xBusy := FALSE;
    xError := TRUE;
    xResetRequest := NOT xEnable;
    xAgain := FALSE; (* !!! *)
END_IF
```

The Handler of the *Resetting* State:

```
METHOD PRIVATE FINAL HandleResettingState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR
VAR
    xComplete : BOOL;
END_VAR

ResetAction(xComplete=>xComplete);

IF xComplete THEN
    xBusy := FALSE;
    xDone := FALSE;
    xError := FALSE;
    eErrorID := ERROR.NO_ERROR;
    eState := STATE.DORMANT;
    xAgain := xResetRequest; (* !!! *)
    xResetRequest := FALSE;
END_IF
```

Exemplary Implementation of the Application specific Methods

This code is listed here just as an example. The content needs to be adapted to the real requirements of a specific application.

The Implementation of the *CyclicAction*:

```
METHOD PROTECTED CyclicAction
VAR_OUTPUT
    xComplete : BOOL;
    eErrorID : ERROR;
END_VAR

IF xEnable THEN
    (* Executing *)
```

```
// for every invocation,
// sample the input variables

// working to reach the ready condition
// ⇒ xComplete := TRUE
// if an error condition is reached set
// eErrorID to a value other than ERROR.NO_ERROR

xComplete := TRUE;
eErrorID := ERROR.NO_ERROR;
END_IF

IF NOT xEnable OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN
    (* Cleaning *)
    // if possible free as much allocated resources
    // as possible
END_IF
```

The Implementation of the AbortAction:

```
METHOD PROTECTED AbortAction
VAR_OUTPUT
    xComplete : BOOL;
    eErrorID : ERROR;
END_VAR

// abort all running operations
// if an error condition is reached set
// eErrorID to a value other than ERROR.NO_ERROR

xComplete := TRUE;
eErrorID := ERROR.NO_ERROR;
```

The Implementation of the ResetAction:

```
METHOD PROTECTED ResetAction
VAR_OUTPUT
    xComplete : BOOL;
END_VAR

// free all allocated resources
// reinitialize instance variables

xComplete := TRUE;
```

6.3. Adding Timers

Also here one can add timer functionalities:

1. **TimeOut (To)**: the overall operating time of the defined operation should be lower than the time (in μ s) as specified by the input value `udiTimeOut`;
2. **TimeLimit (TI)** (and **TIC – TimeLimit without Done output**): 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 (**TITo**)

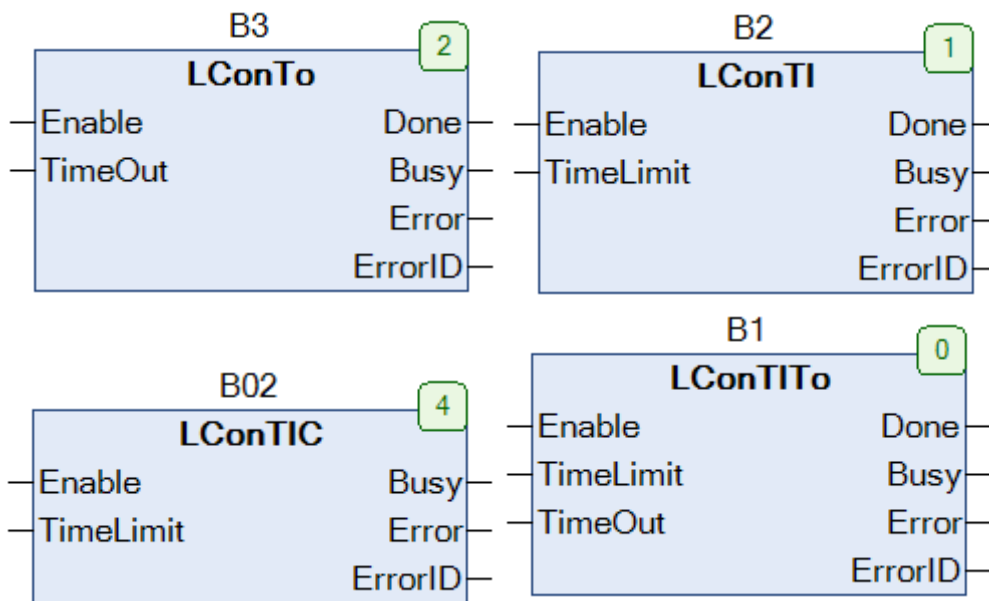


Figure 11: Representation of the added Timers to the level controlled FB

Note: the inputs and outputs are listed here without prefixes in the names

Example of LConTI

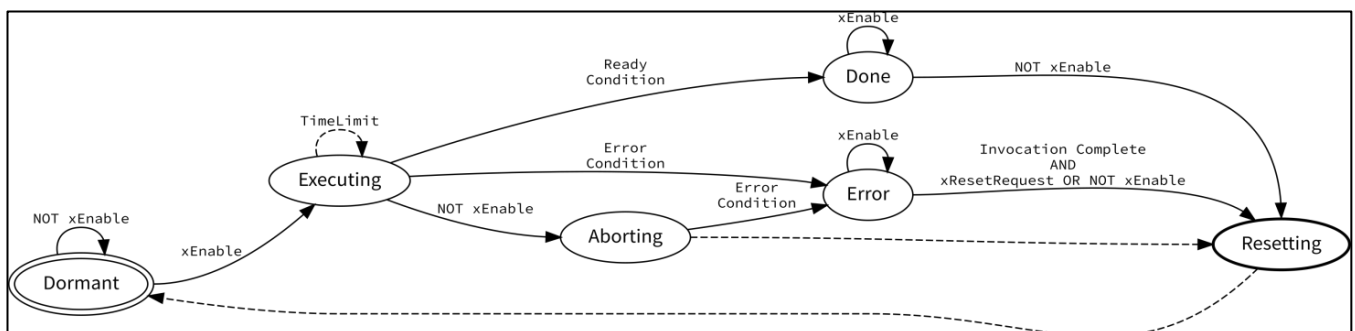


Figure 12: State diagram LConTI

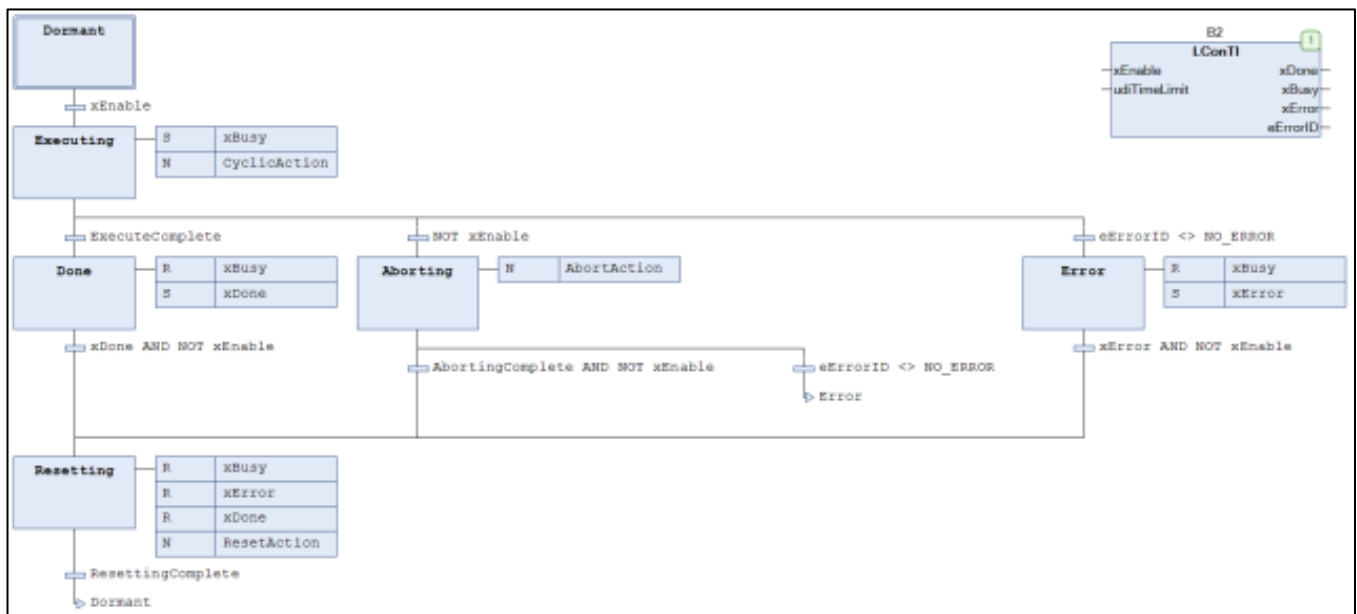


Figure 13: Example of implementation of the state diagram LConTI in SFC

The implementation of LConTI in Sequential Function Chart, SFC, is straightforward. The different states are clearly identifiable in the Steps, linked to the related Action Blocks with the Action Qualifiers. The transition conditions between the Steps are linked to the transitions in the State Diagram.

State Diagram LConTITo

An example of the state diagram for LConTITo, so a level controlled FB with time limit and time out, is shown hereunder:

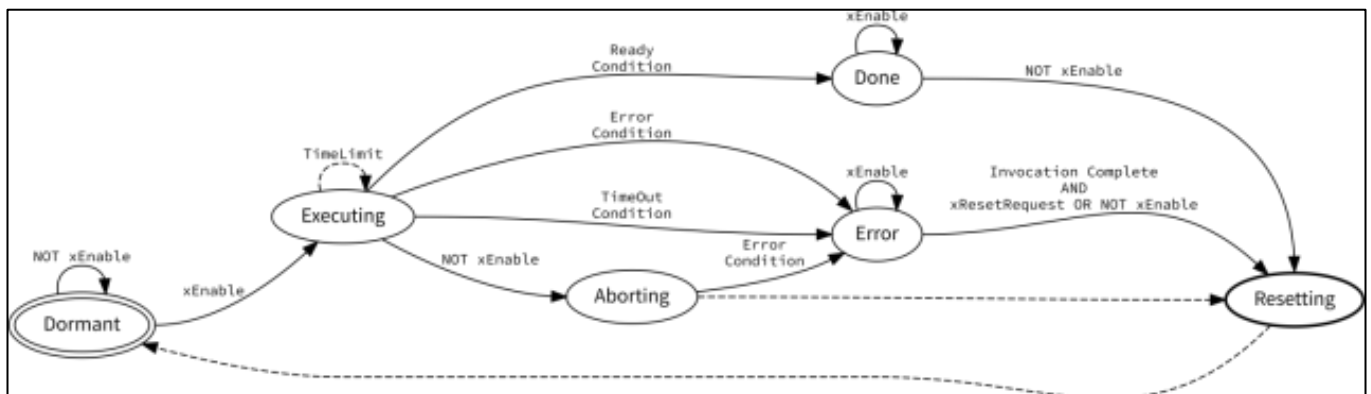


Figure 14: State Diagram for LConTITo

Detailed description of the Function Block LConTITo

1. The function block is called inside of a POU one time per task cycle without any conditions. This is called an invocation.
2. After a TRUE status has been detected at the input xEnable, the internal state is switched from *Dormant* to *Executing*.
3. The status of all inputs will **not** be sampled. They influence the current operation in every invocation (see *CyclicAction*).
4. The output xBusy will be set to TRUE.
5. The defined operation will be started (see the REPEAT-Loop inside *CyclicAction*).
6. If the operating time for the current invocation is higher than the time (in µs) as specified by the input value xTimeLimit, the operation will be interrupted and continued in the next invocation (see xTimeLimit in *CyclicAction*).

7. While working on the defined operation, a number of conditions can appear that lead to the exit from the *Executing* state. This means the value of the output variable `xBusy` will be set to `FALSE` and the internal state will be switched from *Executing* to one of the states *Done* or *Error*. This change will be mirrored to one of the output variables `xDone` or `xError`. Only one output variable of this set of variables can have the status `TRUE` at the same time. With the falling edge of `xBusy` the input variable `xEnable` is sampled and its inverted value is stored as a *reset request* (see `xResetRequest` in the method `HandleErrorState`).
 - a. Ready Condition ²:
If the operation has reached its ready condition without any error and timing constraints (see `xComplete` in `CyclicAction`), the output variable `xDone` is set to `TRUE`. This means the internal state is switched from *Executing* to *Done*.
 - b. Error Condition:
If an error condition was detected (see `eErrorID` in `CyclicAction`), the output variable `xError` is set to `TRUE`. This means the internal state is switched from *Executing* to *Error*. In addition, one of the defined error codes (one value out of the local enumeration type `ERROR`) will be assigned to the output variable `eErrorID`.
 - c. Timeout Condition:
If the overall operating time of the defined operation is higher than the time (in μ s) as specified by the input value `udiTimeOut`, a timeout condition will be detected causing the output variable `xError` to be set to `TRUE`. This means the internal state is switched from *Executing* to *Error*. Furthermore, the output variable `eErrorID` is set to a special error code (`ERROR.TIME_OUT`).
 - d. Abort Condition:
If a status of `FALSE` was detected for the `xEnable` input variable, the abort condition is reached. This means the internal state is switched from *Executing* to *Aborting*. Any current action of the defined operation will be aborted. After this is done (see `xComplete` inside `AbortAction`), the internal state is switched from *Aborting* to *Resetting*. If an error condition was detected (see `eErrorID` in `AbortAction`), the output variable `xError` is set to `TRUE`. This means the internal state is switched from *Aborting* to *Error* (see *error condition*).
8. As a reaction to the rising edge of the output variable `xError` it would be possible to set the status of the input variable `xEnable` again to `TRUE` (*quick handshake*).
9. The value `TRUE` of one of the output variables `xDone` or `xError` must be stable for a minimum of one invocation. This means the internal states *Done* or *Error* must be active for a minimum of one invocation. This property guarantees that the values of the output variables are valid and stable for a minimum of one invocation.
10. After a `FALSE` status for the input variable `xEnable` is detected (*standard handshake*) or a *reset request* is active (*quick handshake*), the internal state will be switched from *Done* or *Error* to *Resetting*.
11. All outputs will be initialized to their default statuses (`ResetAction`). All claimed resources will be freed. Specifically, the output variables `xDone` and `xError` will be set to `FALSE`. After executing the code of the `ResetAction` the function block should be prepared for a new switch of the internal state from *Dormant* to *Executing*.

² Sometimes a function block without an `xDone` output variable is required. In this case, please select the `LConC` or the `LConTlC` types. The state machine of this kind of function blocks will never switch to the *Done* state. An example of this is the `MC_Power` function block for motion control or the `TCPServer` function block as used in 3.7 Cooperation of various function blocks.

12. After doing this reinitialization work, the internal state will be switched from *Resetting* to *Dormant* (see xComplete inside ResetAction).

Example of an SFC diagram

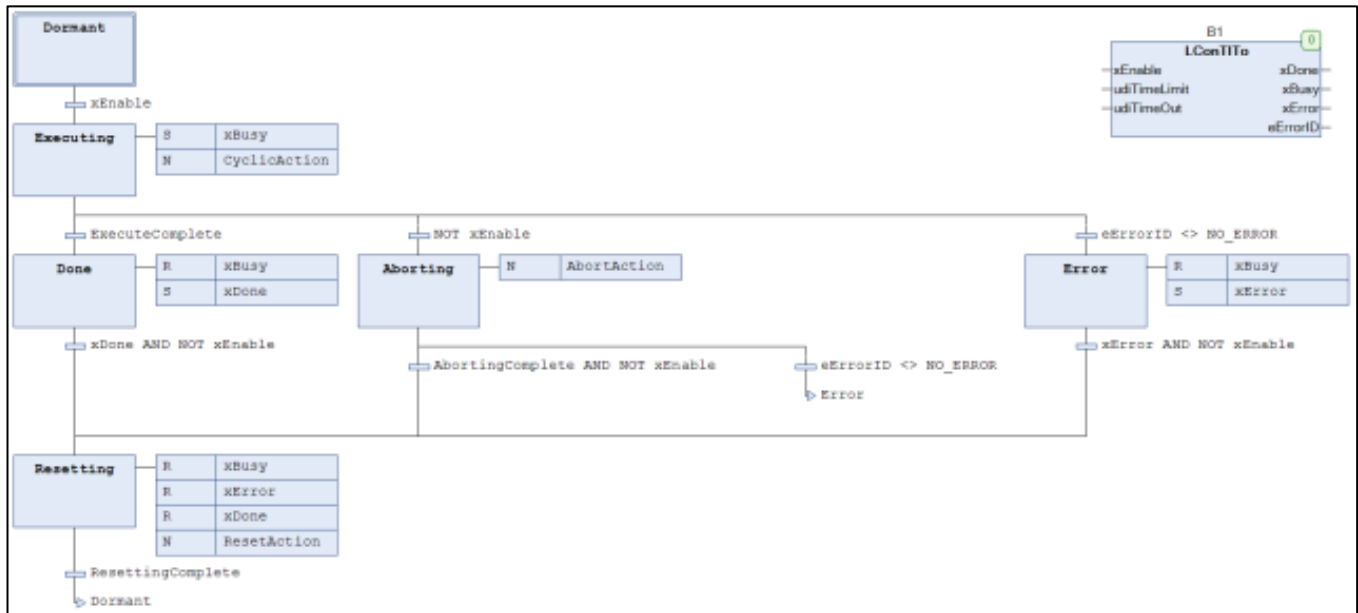


Figure 15: Example of a state diagram of LConTito in SFC

An example of the implementation of LConTito in ST is shown in Appendix 1.4.4 LConTito.

Appendix 1 Datasheets of the Edge Triggered and Level Controlled FBs

In this chapter an overview of examples is provided, including source code, of possible implementations of both the edge as well as level controlled function blocks as part of the PLCopen Behaviour Model. The source code itself is available on the PLCopen website www.PLCopen.org in the same section and in different formats (with and without prefixes and with enum) and the files can be opened in any text editor and with a simple copy&paste added to your own environment.

Appendix 1.1 General comments about the sample programs

Some parts of the code for the different function blocks are very similar. Some parts are critical for a proper functionality according to the specification other parts are provided just as an example and should be adapted to the real requirements of a specific application.

Methods marked with “PRIVATE FINAL” are a crucial part of the implementation.

Methods marked with “PROTECTED” are a kind of template, they should be adapted to the real requirements of a specific application.

Here is a list of the critical parts:

- The order of the evaluation of the variables `xExecute`, `xComplete`, `eErrorID` and `xAbort` defines the behaviour in the case that these variables are set at the same time. A different order results in a different behaviour.
- The handling of the variable `xResetRequest` defines the behaviour of the function block if a quick handshake operation is necessary.
- The handling of the `TimingController` determines the behaviour of the function block in case an `ERROR.TIME_OUT` has to occur or at which exact point in time a specific function block will suspend the processing of its `CyclicAction` and return to its caller (`xTimeLimit`).
- If an Abort Condition or an Error Condition was reached then it is not possible to return to a normal Ready Condition. An Abort Condition can be changed to an Error Condition. In an Error Condition only the `eErrorID` can be modified but not to the value `ERROR.NO_ERROR`.
- An Error Condition that is not caused by a Timeout Condition (`eErrorID ≠ ERROR.TIME_OUT`) has the higher priority and must never be overwritten by `ERROR.TIME_OUT`.
- Every state with a name ending with -ing like `Executing` or `Resetting` can run more than one invocation. In fact they will be executed as long as they need to reach their local Ready Condition, Abort Condition, Timeout Condition or Error Condition.

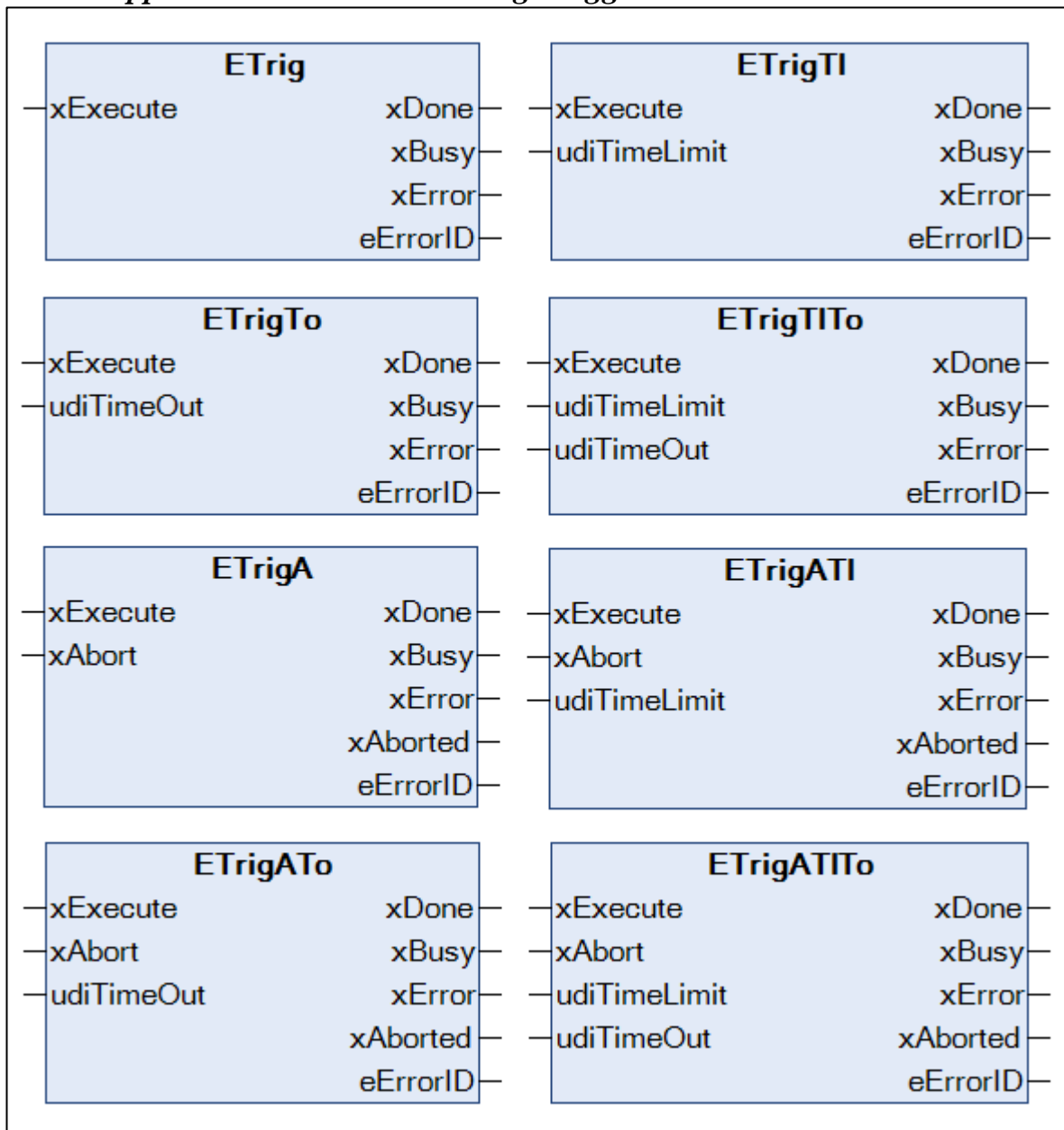
Any state machine implementation of the PLCopen Behaviour Model has to consider these kinds of things to keep the conformance according to this document!

In order to restrict any further interpretation as much as possible, in addition to the source code of each function block a set of timing diagrams for each function block was provided.

Appendix 1.2 Overview of the functionalities

									Description
	Inputs				Outputs				
	Execute	TimeOut	TimeLimit	Abort	Done	Busy	Error / ErrorID	Aborted	
ETrig	x				x	x	x		Rising Edge triggered FB
ETrigTI	x		x		x	x	x		Rising Edge triggered FB with TimeLimit
ETrigTo	x	x			x	x	x		Rising Edge triggered FB with TimeOut
ETrigTITo	x	x	x		x	x	x		Rising Edge triggered FB with TimeOut and TimeLimit
ETrigA	x			x	x	x	x	x	Rising Edge triggered FB with Aborting
ETrigATI	x		x	x	x	x	x	x	Rising Edge triggered FB with Aborting and TimeLimit
ETrigATo	x	x		x	x	x	x	x	Rising Edge triggered FB with Aborting and TimeOut
ETrigATITo	x	x	x	x	x	x	x	x	Rising Edge triggered FB with Aborting and TimeOut and TimeLimit
	Inputs				Outputs				
	Enable	TimeOut	TimeLimit	Busy	Error / ErrorID	Done			
LCon	x			x	x	x			Level Controlled FB
LConTI	x		x	x	x	x			Level Controlled FB with TimeLimit
LConTo	x	x		x	x	x			Level Controlled FB with TimeOut
LConTITo	x	x	x	x	x	x			Level Controlled FB with TimeOut and TimeLimit
LConC	x			x	x				Level Controlled FB, Continuous (never done)
LConTIC	x		x	x	x				Level Controlled FB with TimeLimit, Continuous (never done)

Appendix 1.3 Overview edge triggered FBs

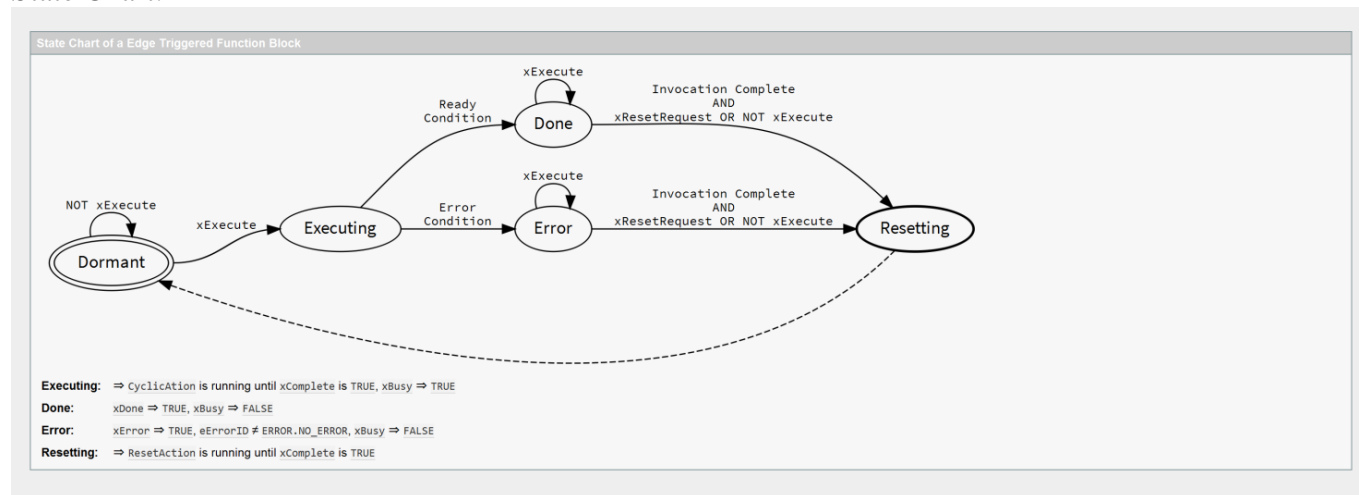


Note: In practice the types STATE and ERROR will be defined only once in a library and not in every FB as is done in the examples shown here.

Appendix 1.3.1 ETrig

ETrig (Edge Triggered | Not Abortable | Not Time Limited | Not Time Out Constraint)

State Chart:



ETrig Implementation

Overview - The Content of ETrig

The STATE Enumeration

```

1  TYPE STATE :
2  (
3    DORMANT, // Waiting for xExecute
4    EXECUTING, // CyclicAction is running
5    DONE, // Ready condition reached
6    ERROR, // Error condition reached
7    RESETTING // ResetAction is running
8  );
9  END_TYPE
    
```

The ERROR Enumeration

```

1  TYPE ERROR :
2  (
3    NO_ERROR := 0,
4    TIME_OUT := 1
5    (* ... *)
6  );
7  END_TYPE
    
```

Implementation of the Function Block ETrig

```

1  FUNCTION_BLOCK ETrig
2  VAR_INPUT
3    // Rising edge starts defined operation
4    // FALSE ⇒ resets the defined operation
5    // after ready condition was reached
6    xExecute: BOOL;
7  END_VAR
8  VAR_OUTPUT
9    // ready condition reached
10   xDone: BOOL;
11   // operation is running
12   xBusy: BOOL;
13   // error condition reached
14   xError: BOOL;
15   // error code describing error condition
16   eErrorID: ERROR;
17 END_VAR
18 VAR
19   eState: STATE;
20   xFirstInvocation: BOOL := TRUE;
21   xResetRequest: BOOL;
22 END_VAR
23 VAR_TEMP
24   xAgain: BOOL;
25 END_VAR
26 REPEAT
27   xAgain := FALSE;
28   CASE eState OF
29     STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
30     STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
31     STATE.DONE: HandleDoneState(xAgain⇒xAgain);
32     STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
33     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
34   END_CASE
35   UNTIL NOT xAgain
36 END_REPEAT;
37
    
```

The Handler for the Dormant State

```
1  METHOD PRIVATE FINAL HandleDormantState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xExecute THEN
7    xBusy := TRUE;
8    eState := STATE.EXECUTING;
9    xAgain := TRUE;
10 END_IF
```

The Handler for the Executing State

```
1  METHOD PRIVATE FINAL HandleExecutingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  CyclicAction(
10 xComplete=>xComplete,
11 eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15   eState := STATE.ERROR;
16   xAgain := TRUE;
17 ELSEIF xComplete THEN
18   eState := STATE.DONE;
19   xAgain := TRUE;
20 END_IF
```

The Handler for the Done State

```
1  METHOD PRIVATE FINAL HandleDoneState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xDone AND (xResetRequest OR NOT xExecute) THEN
7    eState := STATE.RESETTING;
8    xAgain := TRUE;
9  ELSE
10   xBusy := FALSE;
11   xDone := TRUE;
12   xResetRequest := NOT xExecute;
13   xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Error State

```
1  METHOD PRIVATE FINAL HandleErrorState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xError AND (xResetRequest OR NOT xExecute) THEN
7    eState := STATE.RESETTING;
8    xAgain := TRUE;
9  ELSE
10   xBusy := FALSE;
11   xError := TRUE;
12   xResetRequest := NOT xExecute;
13   xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1  METHOD PRIVATE FINAL HandleResettingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12   xBusy := FALSE;
13   xDone := FALSE;
14   xError := FALSE;
15   eErrorID := ERROR.NO_ERROR;
16   eState := STATE.DORMANT;
17   xFirstInvocation := TRUE;
18   xAgain := xResetRequest; (* !!! *)
19   xResetRequest := FALSE;
20 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

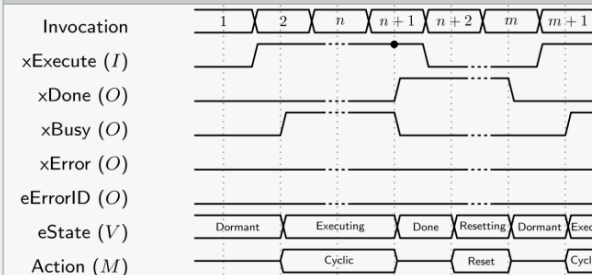
```
1  METHOD PROTECTED CyclicAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6
7  IF xFirstInvocation THEN
8    (* Starting *)
9    // for the first (!) invocation,
10   // sample the input variables
11   xFirstInvocation := FALSE;
12 END_IF
13
14 (* Executing *)
15 // working to reach the ready condition
16 // => xComplete := TRUE
17 // if an error condition is reached
18 // => set eErrorID to a value other than ERROR.NO_ERROR
19
20 xComplete := TRUE;
21 eErrorID := ERROR.NO_ERROR;
22
23 IF xComplete OR eErrorID <> ERROR.NO_ERROR THEN
24   (* Cleaning *)
25   // if possible free as much allocated resources
26   // as possible
27 END_IF
```

The Implementation of the Reset Action (Exemplary Implementation)

```
1  METHOD PROTECTED ResetAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  END_VAR
5
6  // free all allocated resources
7  // reinitialize instance variables
8
9  xComplete := TRUE;
```

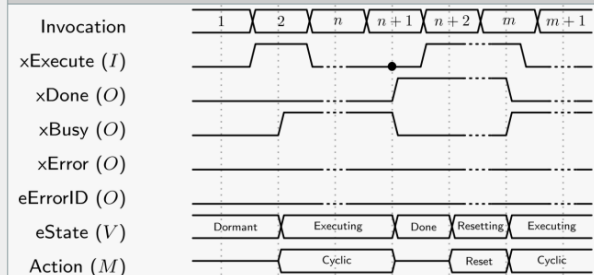
ETrig Timing Diagram

Tracing variables, states and methods of ETrig along the time line (standard handshake)

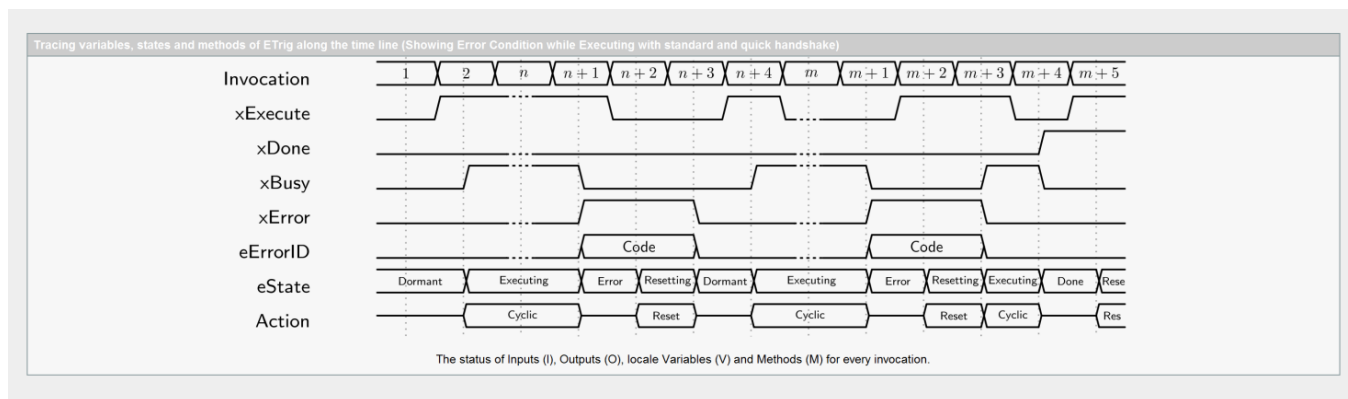


The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Tracing variables, states and methods of ETrig along the time line (quick handshake)



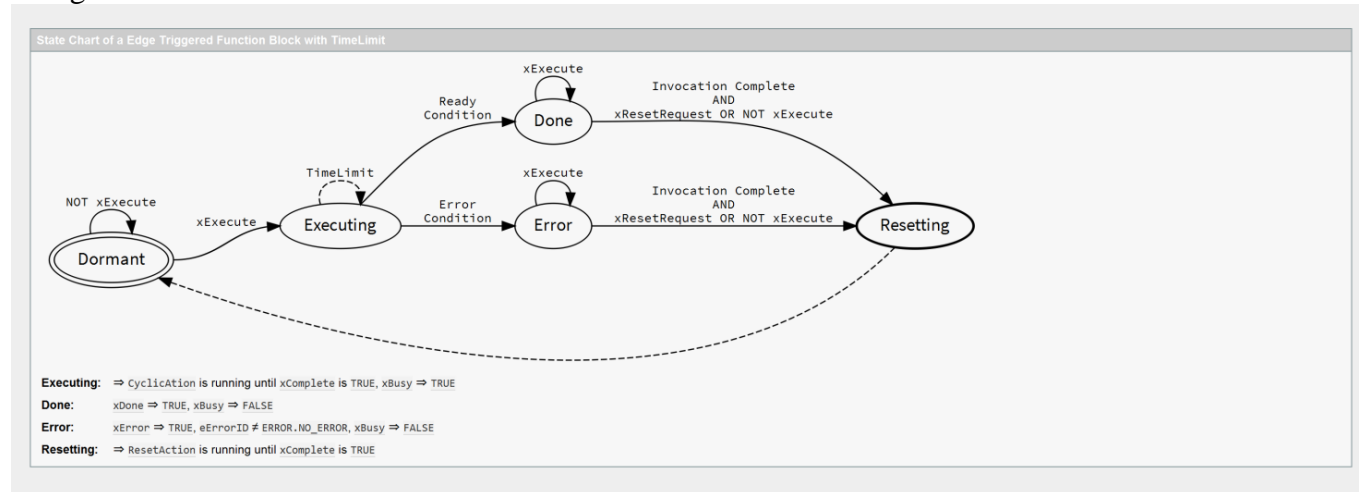
The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.



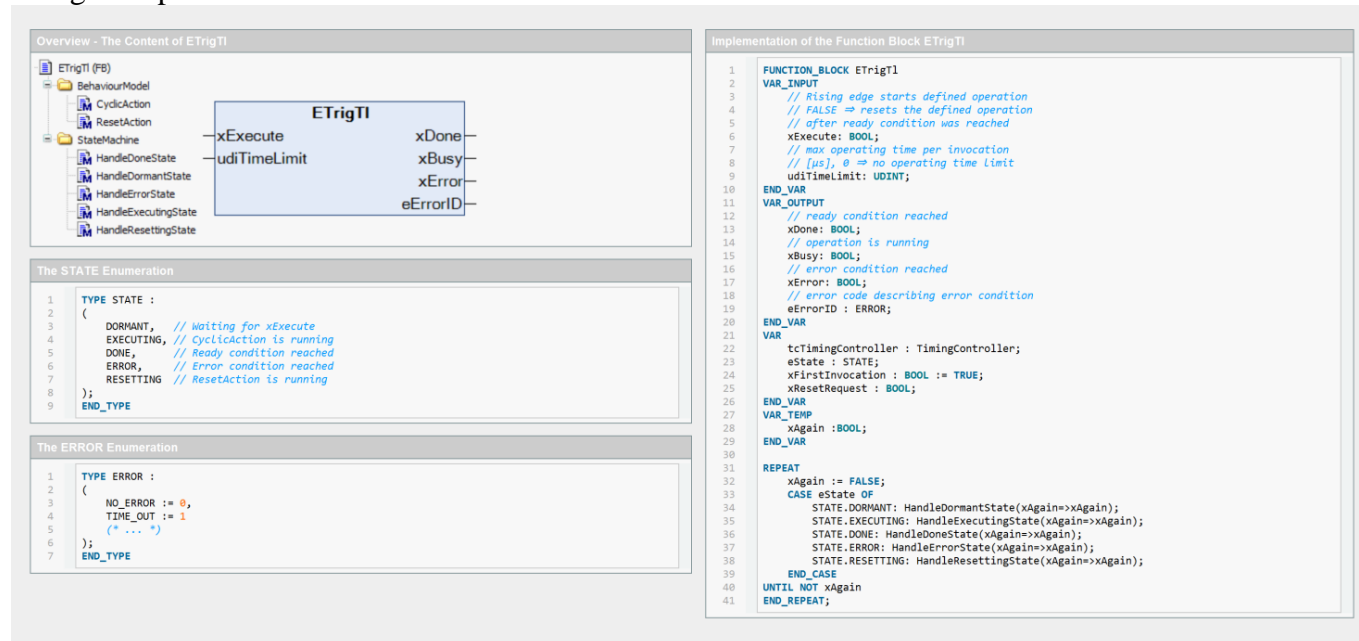
Appendix 1.3.2 ETrigTl

ETrigTl (Edge Triggered | Not Abortable | Time Limited | Not Time Out Constraint)

ETrigTl State Chart



ETrigTl Implementation



The Handler for the Dormant State

```
1 METHOD PRIVATE FINAL HandleDormantState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xExecute THEN
7   xBusy := TRUE;
8   eState := STATE.EXECUTING;
9   xAgain := TRUE;
10 END_IF
```

The Handler for the Executing State

```
1 METHOD PRIVATE FINAL HandleExecutingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7   xTimeLimit : BOOL;
8 END_VAR
9
10 tcTimingController.StartInvocationTimer();
11
12 CyclicaAction(
13   xComplete=>xComplete,
14   eErrorID=>eErrorID
15 );
16
17 IF eErrorID <> ERROR.NO_ERROR THEN
18   eState := STATE.ERROR;
19   xAgain := TRUE;
20 ELSEIF xComplete THEN
21   eState := STATE.DONE;
22   xAgain := TRUE;
23 END_IF
```

The Handler for the Done State

```
1 METHOD PRIVATE FINAL HandleDoneState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xDone AND (xResetRequest OR NOT xExecute) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xDone := TRUE;
12  xResetRequest := NOT xExecute;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Error State

```
1 METHOD PRIVATE FINAL HandleErrorState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xError AND (xResetRequest OR NOT xExecute) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xError := TRUE;
12  xResetRequest := NOT xExecute;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler of the Resetting State

```
1 METHOD PRIVATE FINAL HandleResettingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12   xBusy := FALSE;
13   xDone := FALSE;
14   xError := FALSE;
15   eErrorID := ERROR.NO_ERROR;
16   eState := STATE.DORMANT;
17   xFirstInvocation := TRUE;
18   xAgain := xResetRequest; (* !!! *)
19   xResetRequest := FALSE;
20 END_IF
```

The Implementation of the Cyclica Action (Exemplary Implementation)

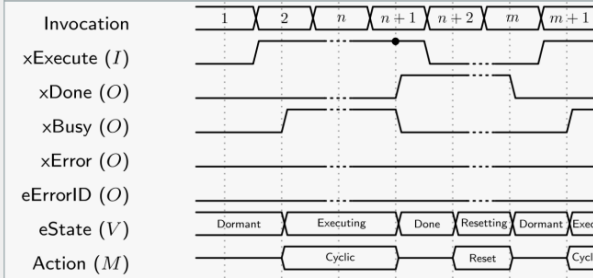
```
1 METHOD PROTECTED CyclicaAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6 VAR
7   xTimeLimit : BOOL;
8 END_VAR
9
10 IF xFirstInvocation THEN
11   (* Starting *)
12   // for the first (!) invocation,
13   // sample the input variables
14   tcTimingController.TimeLimit := udiTimeLimit;
15   xFirstInvocation := FALSE;
16 END_IF
17
18 REPEAT
19   (* Executing *)
20   // working to reach the ready condition
21   // => xComplete := TRUE
22   // if the maximum invocation time is reached
23   // => xTimeLimit := TRUE
24   // if an error condition is reached set
25   // eErrorID to a value other than ERROR.NO_ERROR
26   tcTimingController.CheckTiming(
27     xTimeLimit->xTimeLimit
28   );
29
30   xComplete := TRUE;
31   eErrorID := ERROR.NO_ERROR;
32
33 UNTIL xComplete OR xTimeLimit OR
34   eErrorID <> ERROR.NO_ERROR
35 END_REPEAT
36
37 IF xComplete OR eErrorID <> ERROR.NO_ERROR THEN
38   (* Cleaning *)
39   // if possible free as much allocated resources
40   // as possible
41 END_IF
```

The Implementation of the Reset Action (Exemplary Implementation)

```
1 METHOD PROTECTED ResetAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4 END_VAR
5
6 // free all allocated resources
7 // reinitialize instance variables
8
9 xComplete := TRUE;
```

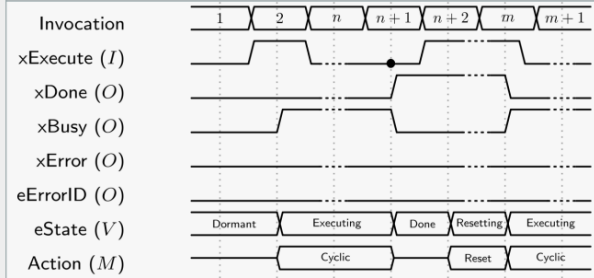
ETrigTI Timing Diagram

Tracing variables, states and methods of ETrigTI along the time line (standard handshake)



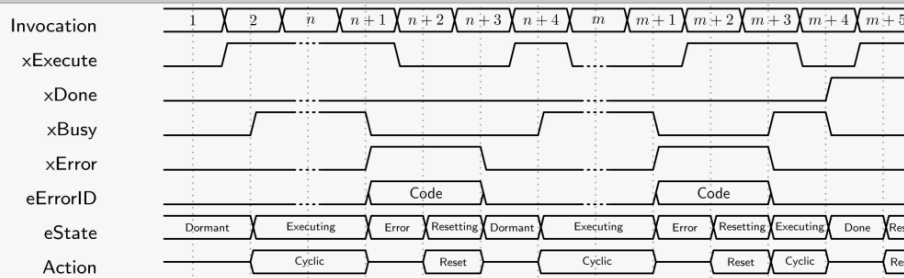
The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Tracing variables, states and methods of ETrigTI along the time line (quick handshake)



The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Tracing variables, states and methods of ETrigTI along the time line (Showing Error Condition while Executing with standard and quick handshake)

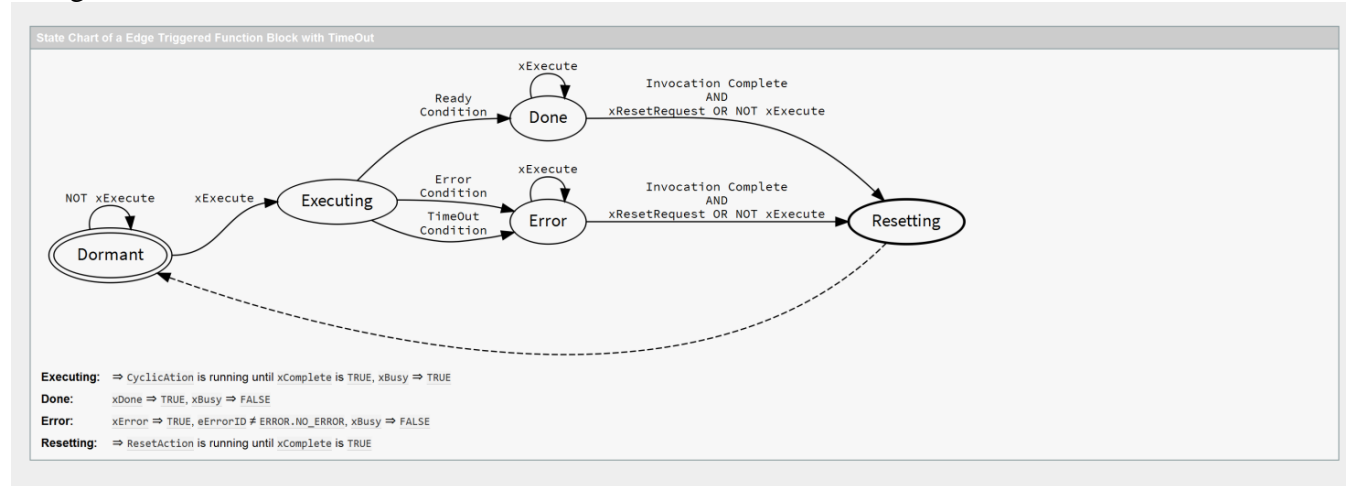


The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Appendix 1.3.3 ETrigTo

ETrigTo (Edge Triggered | Not Abortable | Not Time Limited | Time Out Constraint)

ETrigTo State Chart



ETrigTo Implementation

Overview - The Content of ETrigTo

The STATE Enumeration

```

1  TYPE STATE :
2  (
3    DORMANT, // Waiting for xExecute
4    EXECUTING, // Cyclication is running
5    DONE, // Ready condition reached
6    ERROR, // Error condition reached
7    ABORTING, // AbortAction is running
8    ABORTED, // Abort Condition reached
9    RESETTING // ResetAction is running
10 );
11 END_TYPE
  
```

The ERROR Enumeration

```

1  TYPE ERROR :
2  (
3    NO_ERROR := 0,
4    TIME_OUT := 1
5    (* ... *)
6  );
7  END_TYPE
  
```

Implementation of the Function Block ETrigTo

```

1  FUNCTION_BLOCK ETrigTo
2  VAR_INPUT
3    // Rising edge starts defined operation
4    // FALSE ⇒ resets the defined operation
5    // after ready condition was reached
6    xExecute: BOOL;
7    // max operating time for executing
8    // [µs], 0 ⇒ no operating time limit
9    udiTimeOut: UDINT;
10 END_VAR
11 VAR_OUTPUT
12 // ready condition reached
13 xDone: BOOL;
14 // operation is running
15 xBusy: BOOL;
16 // error condition reached
17 xError: BOOL;
18 // error code describing error condition
19 eErrorID: ERROR;
20 END_VAR
21 VAR
22   tcTimingController : TimingController;
23   eState : STATE;
24   xFirstInvocation : BOOL := TRUE;
25   xResetRequest : BOOL;
26 END_VAR
27 VAR_TEMP
28   xAgain : BOOL;
29 END_VAR
30
31 REPEAT
32   xAgain := FALSE;
33   CASE eState OF
34     STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
35     STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
36     STATE.DONE: HandleDoneState(xAgain⇒xAgain);
37     STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
38     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
39   END_CASE
40 UNTIL NOT xAgain
41 END_REPEAT;
  
```

The Handler for the Dormant State

```
1  METHOD PRIVATE FINAL HandleDormantState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xEnable THEN
7      tcTimingController.StartOperationTimer();
8      xBusy := TRUE;
9      eState := STATE.EXECUTING;
10     xAgain := TRUE;
11 END_IF
```

The Handler for the Executing State

```
1  METHOD PRIVATE FINAL HandleExecutingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6      xComplete : BOOL;
7      xTimeout : BOOL;
8  END_VAR
9
10 CyclicAction(
11     xComplete=>xComplete,
12     eErrorID=>eErrorID
13 );
14
15 tcTimingController.CheckTiming(xTimeout=>xTimeout);
16
17 IF xTimeout AND eErrorID = ERROR.NO_ERROR THEN
18     eErrorID := ERROR.TIME_OUT;
19 END_IF
20
21 IF eErrorID <> ERROR.NO_ERROR THEN
22     eState := STATE.ERROR;
23     xAgain := TRUE;
24 ELSEIF xComplete THEN
25     eState := STATE.DONE;
26     xAgain := TRUE;
27 END_IF
```

The Handler for the Done State

```
1  METHOD PRIVATE FINAL HandleDoneState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xDone AND (xResetRequest OR NOT xExecute) THEN
7      eState := STATE.RESETTING;
8      xAgain := TRUE;
9  ELSE
10     xBusy := FALSE;
11     xDone := TRUE;
12     xResetRequest := NOT xExecute;
13     xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Error State

```
1  METHOD PRIVATE FINAL HandleErrorState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xError AND (xResetRequest OR NOT xExecute) THEN
7      eState := STATE.RESETTING;
8      xAgain := TRUE;
9  ELSE
10     xBusy := FALSE;
11     xError := TRUE;
12     xResetRequest := NOT xExecute;
13     xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1  METHOD PRIVATE FINAL HandleResettingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6      xComplete : BOOL;
7  END_VAR
8
9  ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12     xBusy := FALSE;
13     xDone := FALSE;
14     xError := FALSE;
15     eErrorID := ERROR.NO_ERROR;
16     eState := STATE.DORMANT;
17     xFirstInvocation := TRUE;
18     xAgain := xResetRequest; (* !!! *)
19     xResetRequest := FALSE;
20 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

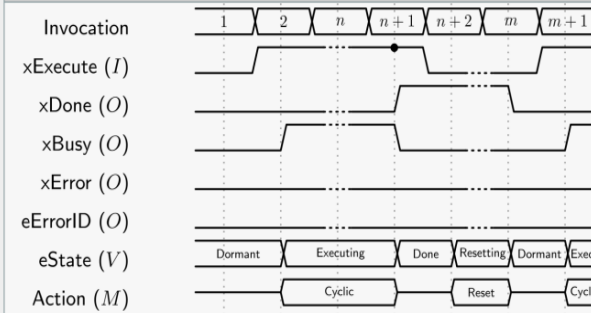
```
1  METHOD PROTECTED CyclicAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6  VAR
7      xTimeout : BOOL;
8  END_VAR
9
10 IF xFirstInvocation THEN
11     (* Starting *)
12     // for the first (!) invocation,
13     // sample the input variables
14     tcTimingController.Timeout := udTimeout;
15     xFirstInvocation := FALSE;
16 END_IF
17
18 (* Executing *)
19 // working to reach the ready condition
20 // => xComplete := TRUE
21 // if the maximum operating time is reached
22 // => xTimeout := TRUE
23 // if an error condition is reached
24 // => set eErrorID to a value other than ERROR.NO_ERROR
25 tcTimingController.CheckTiming(
26     xTimeout=>xTimeout,
27 );
28
29 xComplete := TRUE;
30 eErrorID := ERROR.NO_ERROR;
31
32 IF xTimeout AND eErrorID = ERROR.NO_ERROR THEN
33     eErrorID := ERROR.TIME_OUT;
34 END_IF
35
36 IF xComplete OR eErrorID <> ERROR.NO_ERROR THEN
37     (* Cleaning *)
38     // if possible free as much allocated resources
39     // as possible
40 END_IF
```

The Implementation of the Reset Action (Exemplary Implementation)

```
1  METHOD PROTECTED ResetAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  END_VAR
5
6  // free all allocated resources
7  // reinitialize instance variables
8
9  xComplete := TRUE;
```

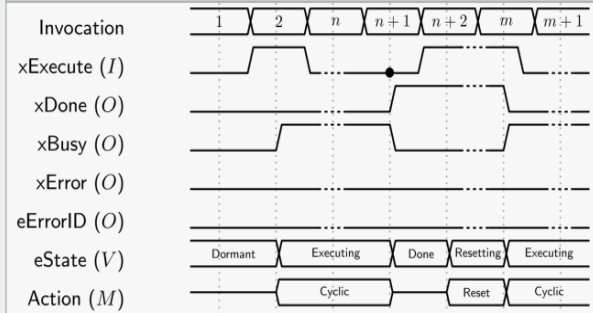
ETrigTo Timing Diagram

Tracing variables, states and methods of ETrigTo along the time line (standard handshake)



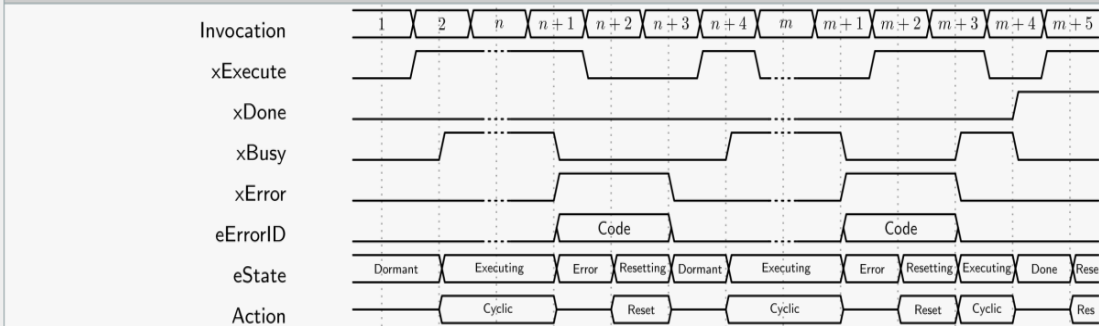
The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Tracing variables, states and methods of ETrigTo along the time line (quick handshake)



The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Tracing variables, states and methods of ETrigTo along the time line (Showing Error Condition while Executing with standard and quick handshake)

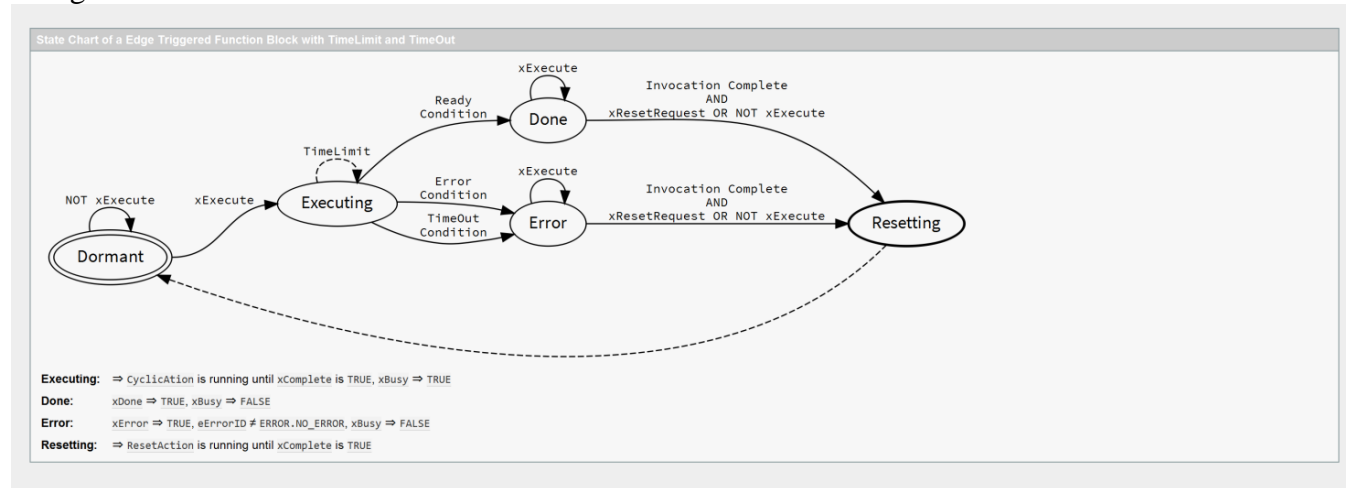


The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Appendix 1.3.4 ETrigTlTo

ETrigTlTo (Edge Triggered | Not Abortable | Time Limited | Time Out Constraint)

ETrigTlTo State Chart



ETrigTlTo Implementation

Overview - The Content of ETrigTlTo

The STATE Enumeration

```

1  TYPE STATE :
2  (
3    DORMANT, // Waiting for xExecute
4    EXECUTING, // Cyclication is running
5    DONE, // Ready condition reached
6    ERROR, // Error condition reached
7    RESETTING // ResetAction is running
8  );
9  END_TYPE
  
```

The ERROR Enumeration

```

1  TYPE ERROR :
2  (
3    NO_ERROR := 0,
4    TIME_OUT := 1
5  (* ... *)
6  );
7  END_TYPE
  
```

Implementation of the Function Block ETrigTlTo

```

1  FUNCTION_BLOCK ETrigTlTo
2  VAR_INPUT
3    // Rising edge starts defined operation
4    // FALSE ⇒ resets the defined operation
5    // after ready condition was reached
6    xExecute: BOOL;
7    // max operating time per invocation
8    // [µs], 0 ⇒ no operating time limit
9    udiTimeLimit: UDINT;
10   // max operating time for executing
11   // [µs], 0 ⇒ no operating time limit
12   udiTimeOut: UDINT;
13 END_VAR
14 VAR_OUTPUT
15   // ready condition reached
16   xDone: BOOL;
17   // operation is running
18   xBusy: BOOL;
19   // error condition reached
20   xError: BOOL;
21   // error code describing error condition
22   eErrorID: ERROR;
23 END_VAR
24 VAR
25   tcTimingController : TimingController;
26   eState: STATE;
27   xFirstInvocation: BOOL := TRUE;
28   xResetRequest: BOOL;
29 END_VAR
30 VAR_TEMP
31   xAgain: BOOL;
32 END_VAR
33 REPEAT
34   xAgain := FALSE;
35   CASE eState OF
36     STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
37     STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
38     STATE.DONE: HandleDoneState(xAgain⇒xAgain);
39     STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
40     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
41   END_CASE
42 UNTIL NOT xAgain
43 END_REPEAT;
  
```


The Handler for the Dormant State

```
1  METHOD PRIVATE FINAL HandleDormantState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xEnable THEN
7      tcTimingController.StartOperationTimer();
8      xBusy := TRUE;
9      eState := STATE.EXECUTING;
10     xAgain := TRUE;
11 END_IF
```

The Handler for the Executing State

```
1  METHOD PRIVATE FINAL HandleExecutingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6      xComplete : BOOL;
7      xTimeout : BOOL;
8  END_VAR
9
10 tcTimingController.StartInvocationTimer();
11
12 CyclicAction(
13     xComplete=>xComplete,
14     eErrorID=>eErrorID
15 );
16
17 tcTimingController.CheckTiming(xTimeout=>xTimeout);
18
19 IF xTimeout AND eErrorID = ERROR.NO_ERROR THEN
20     eErrorID := ERROR.TIME_OUT;
21 END_IF
22
23 IF eErrorID <> ERROR.NO_ERROR THEN
24     eState := STATE.ERROR;
25     xAgain := TRUE;
26 ELSEIF xComplete THEN
27     eState := STATE.DONE;
28     xAgain := TRUE;
29 END_IF
```

The Handler for the Done State

```
1  METHOD PRIVATE FINAL HandleDoneState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xDone AND (xResetRequest OR NOT xExecute) THEN
7      eState := STATE.RESETTING;
8      xAgain := TRUE;
9  ELSE
10     xBusy := FALSE;
11     xDone := TRUE;
12     xResetRequest := NOT xExecute;
13     xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Error State

```
1  METHOD PRIVATE FINAL HandleErrorState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xError AND (xResetRequest OR NOT xExecute) THEN
7      eState := STATE.RESETTING;
8      xAgain := TRUE;
9  ELSE
10     xBusy := FALSE;
11     xError := TRUE;
12     xResetRequest := NOT xExecute;
13     xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler of the Resetting State

```
1  METHOD PRIVATE FINAL HandleResettingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6      xComplete : BOOL;
7  END_VAR
8
9  ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12     xBusy := FALSE;
13     xDone := FALSE;
14     xError := FALSE;
15     eErrorID := ERROR.NO_ERROR;
16     eState := STATE.DORMANT;
17     xFirstInvocation := TRUE;
18     xAgain := xResetRequest; (* !!! *)
19     xResetRequest := FALSE;
20 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

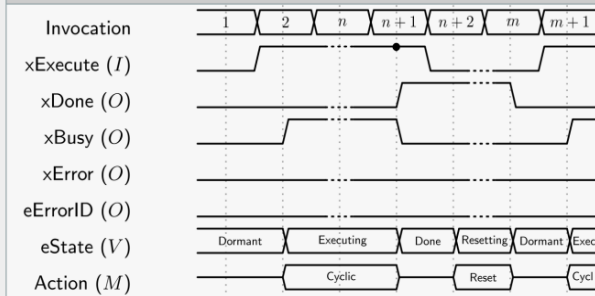
```
1  METHOD PROTECTED CyclicAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6  VAR
7      xTimeout : BOOL;
8      xTimeLimit : BOOL;
9  END_VAR
10
11 IF xFirstInvocation THEN
12     (* Starting *)
13     // for the first (!) invocation,
14     // sample the input variables
15     tcTimingController.TimeLimit := udiTimeLimit;
16     tcTimingController.TimeOut := udiTimeOut;
17     xFirstInvocation := FALSE;
18 END_IF
19
20 REPEAT
21     (* Executing *)
22     // working to reach the ready condition
23     // => xComplete := TRUE
24     // if the maximum invocation time is reached
25     // => xTimeLimit := TRUE
26     // if the maximum operating time is reached
27     // => xTimeout := TRUE
28     // if an error condition is reached
29     // => set eErrorID to a value other than ERROR.NO_ERROR
30     tcTimingController.CheckTiming(
31         xTimeout=>xTimeout,
32         xTimeLimit=>xTimeLimit
33     );
34
35     xComplete := TRUE;
36     eErrorID := ERROR.NO_ERROR;
37
38 UNTIL xComplete OR xTimeout OR xTimeLimit OR
39     eErrorID <> ERROR.NO_ERROR
40 END_REPEAT
41
42 IF xTimeout AND eErrorID = ERROR.NO_ERROR THEN
43     eErrorID := ERROR.TIME_OUT;
44 END_IF
45
46 IF xComplete OR eErrorID <> ERROR.NO_ERROR THEN
47     (* Cleaning *)
48     // if possible free as much allocated resources
49     // as possible
50 END_IF
```

The Implementation of the Reset Action (Exemplary Implementation)

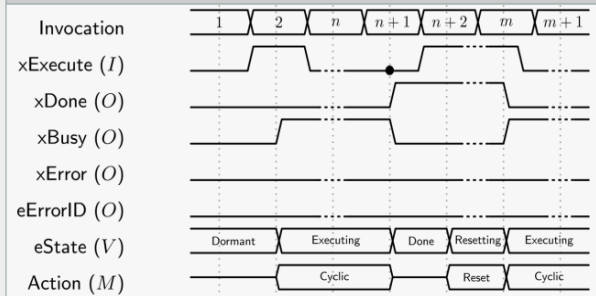
```
1  METHOD PROTECTED ResetAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  END_VAR
5
6  // free all allocated resources
7  // reinitialize instance variables
8
9  xComplete := TRUE;
```

ETrigTITo Timing Diagram

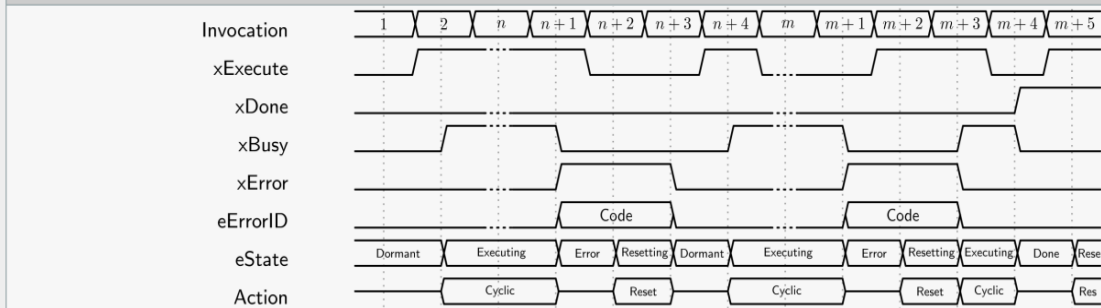
Tracing variables, states and methods of ETrigTITo along the time line (standard handshake)



Tracing variables, states and methods of ETrigTITo along the time line (quick handshake)



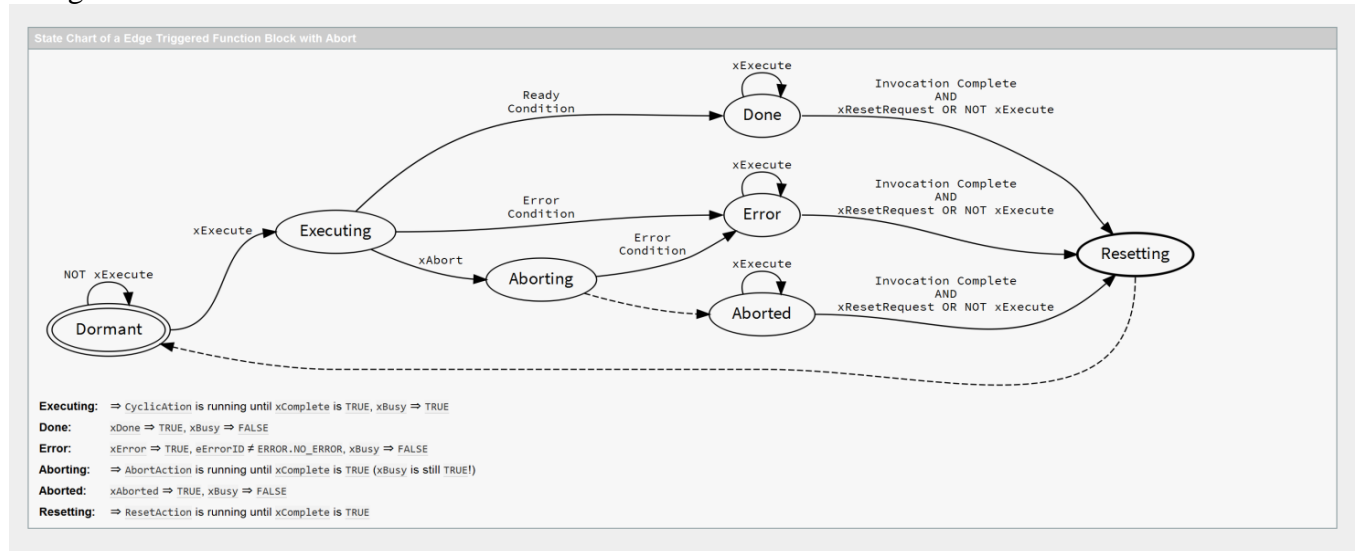
Tracing variables, states and methods of ETrigTITo along the time line (Showing Error Condition while Executing with standard and quick handshake)



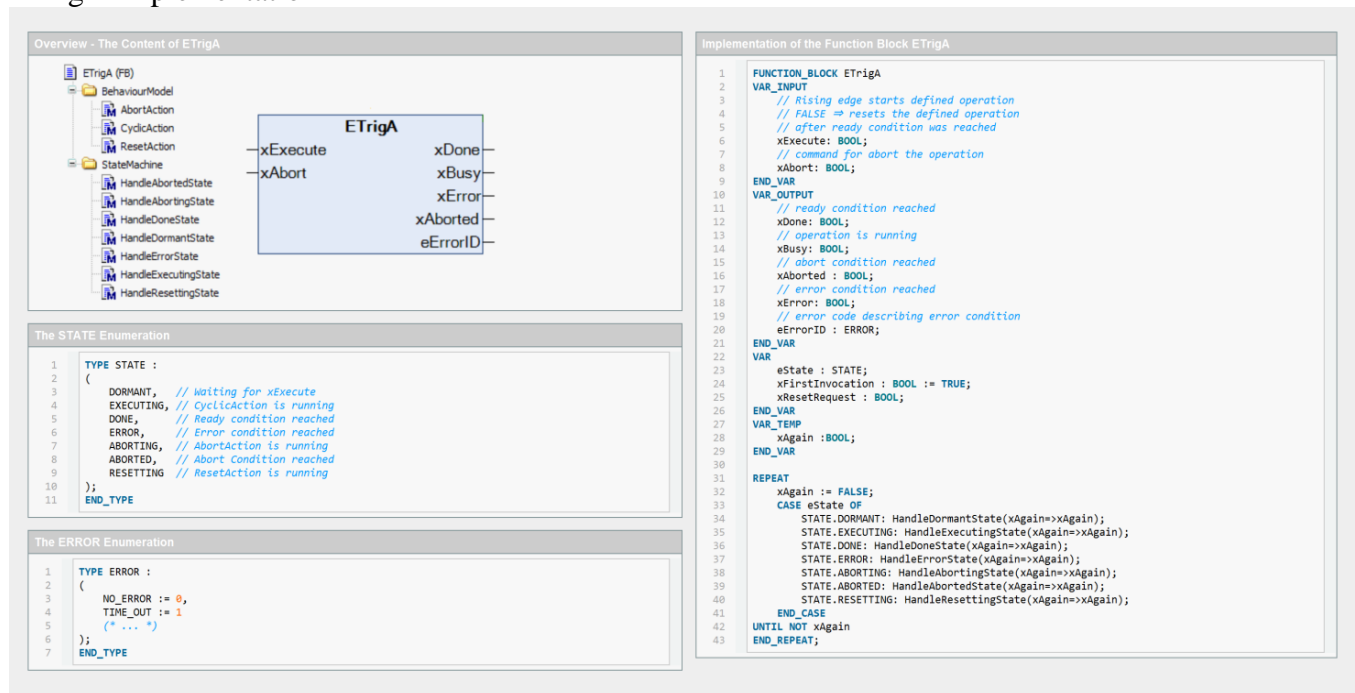
Appendix 1.3.5 ETrigA

ETrigA (Edge Triggered | Abortable | Not Time Limited | Not Time Out Constraint)

ETrigA State Chart



ETrigA Implementation



The Handler for the Dormant State

```
1  METHOD PRIVATE FINAL HandleDormantState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xExecute THEN
7    xBusy := TRUE;
8    eState := STATE.EXECUTING;
9    xAgain := TRUE;
10 END_IF
```

The Handler for the Executing State

```
1  METHOD PRIVATE FINAL HandleExecutingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  IF NOT xAbort THEN
10   CyclicAction(
11     xComplete=>xComplete,
12     eErrorID=>eErrorID
13   );
14 END_IF
15
16 IF eErrorID <> ERROR.NO_ERROR THEN
17   eState := STATE.ERROR;
18   xAgain := TRUE;
19 ELSEIF xAbort THEN
20   eState := STATE.ABORTING;
21   xAgain := TRUE;
22 ELSEIF xComplete THEN
23   eState := STATE.DONE;
24   xAgain := TRUE;
25 END_IF
```

The Handler for the Aborting State

```
1  METHOD PRIVATE FINAL HandleAbortingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  AbortAction(
10   xComplete=>xComplete,
11   eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15   eState := STATE.ERROR;
16   xAgain := TRUE;
17 ELSEIF xComplete THEN
18   eState := STATE.ABORTED;
19   xAgain := TRUE;
20 END_IF
```

The Handler for the Done State

```
1  METHOD PRIVATE FINAL HandleDoneState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xDone AND (xResetRequest OR NOT xExecute) THEN
7    eState := STATE.RESETTING;
8    xAgain := TRUE;
9  ELSE
10   xBusy := FALSE;
11   xDone := TRUE;
12   xResetRequest := NOT xExecute;
13   xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Error State

```
1  METHOD PRIVATE FINAL HandleErrorState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF eError AND (xResetRequest OR NOT xExecute) THEN
7    eState := STATE.RESETTING;
8    xAgain := TRUE;
9  ELSE
10   xBusy := FALSE;
11   xError := TRUE;
12   xResetRequest := NOT xExecute;
13   xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Aborted State

```
1  METHOD PRIVATE FINAL HandleAbortedState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xAborted AND (xResetRequest OR NOT xExecute) THEN
7    eState := STATE.RESETTING;
8    xAgain := TRUE;
9  ELSE
10   xBusy := FALSE;
11   xAborted := TRUE;
12   xResetRequest := NOT xExecute;
13   xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1  METHOD PRIVATE FINAL HandleResettingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12   xBusy := FALSE;
13   xDone := FALSE;
14   xError := FALSE;
15   xAborted := FALSE;
16   eErrorID := ERROR.NO_ERROR;
17   eState := STATE.DORMANT;
18   xFirstInvocation := TRUE;
19   xAgain := xResetRequest; (* !!! *)
20   xResetRequest := FALSE;
21 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1  METHOD PROTECTED CyclicAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6
7  IF NOT xAbort THEN
8    IF xFirstInvocation THEN
9      (* Starting *)
10     // for the first (!) invocation,
11     // sample the input variables
12     xFirstInvocation := FALSE;
13   END_IF
14
15   (* Executing *)
16   // working to reach the ready condition
17   // => xComplete := TRUE
18   // if an error condition is reached
19   // => set eErrorID to a value other than ERROR.NO_ERROR
20
21   xComplete := TRUE;
22   eErrorID := ERROR.NO_ERROR;
23 END_IF
24
25 IF xAbort OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN
26   (* Cleaning *)
27   // if possible free as much allocated resources
28   // as possible
29 END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

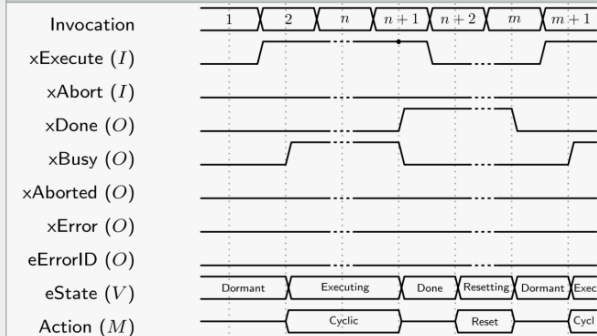
```
1  METHOD PROTECTED AbortAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6
7  // abort all running operations
8  // if an error condition is reached set
9  // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

The Implementation of the Reset Action (Exemplary Implementation)

```
1  METHOD PROTECTED ResetAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  END_VAR
5
6  // free all allocated resources
7  // reinitialize instance variables
8
9  xComplete := TRUE;
```

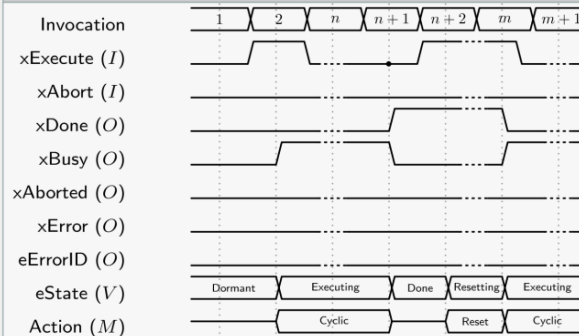
ETrigA Timing Diagram

Tracing variables, states and methods of ETrigA along the time line (standard handshake)



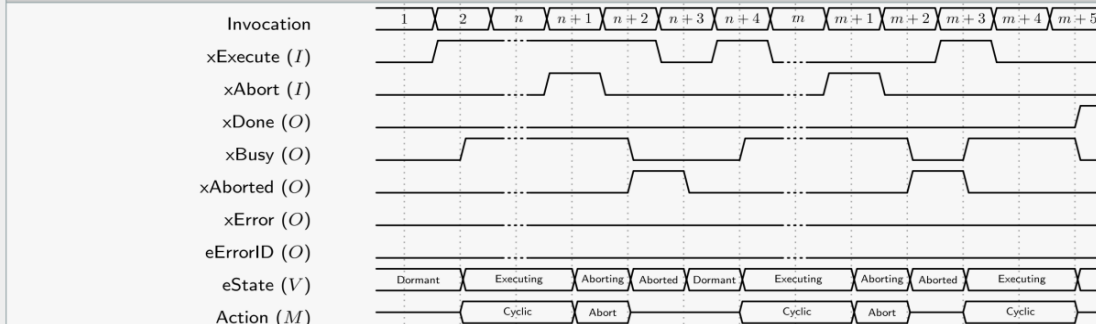
The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Tracing variables, states and methods of ETrigA along the time line (quick handshake)



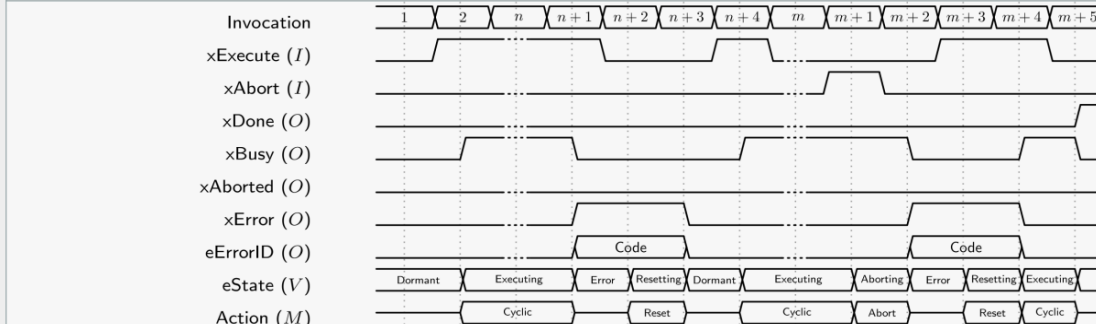
The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Tracing variables, states and methods of ETrigA along the time line (Showing Abort Condition with standard and quick handshake)



The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

Tracing variables, states and methods of ETrigA along the time line (Showing Error Condition while Executing and Aborting with standard and quick handshake)

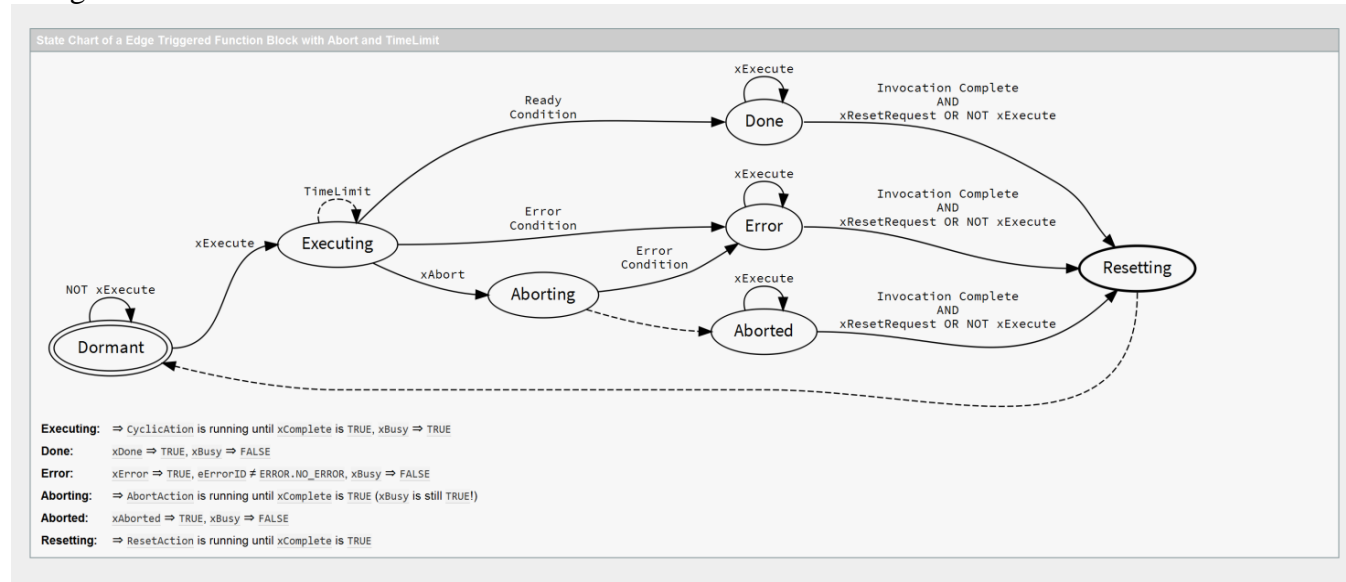


The status of Inputs (I), Outputs (O), locale Variables (V) and Methods (M) for every invocation.

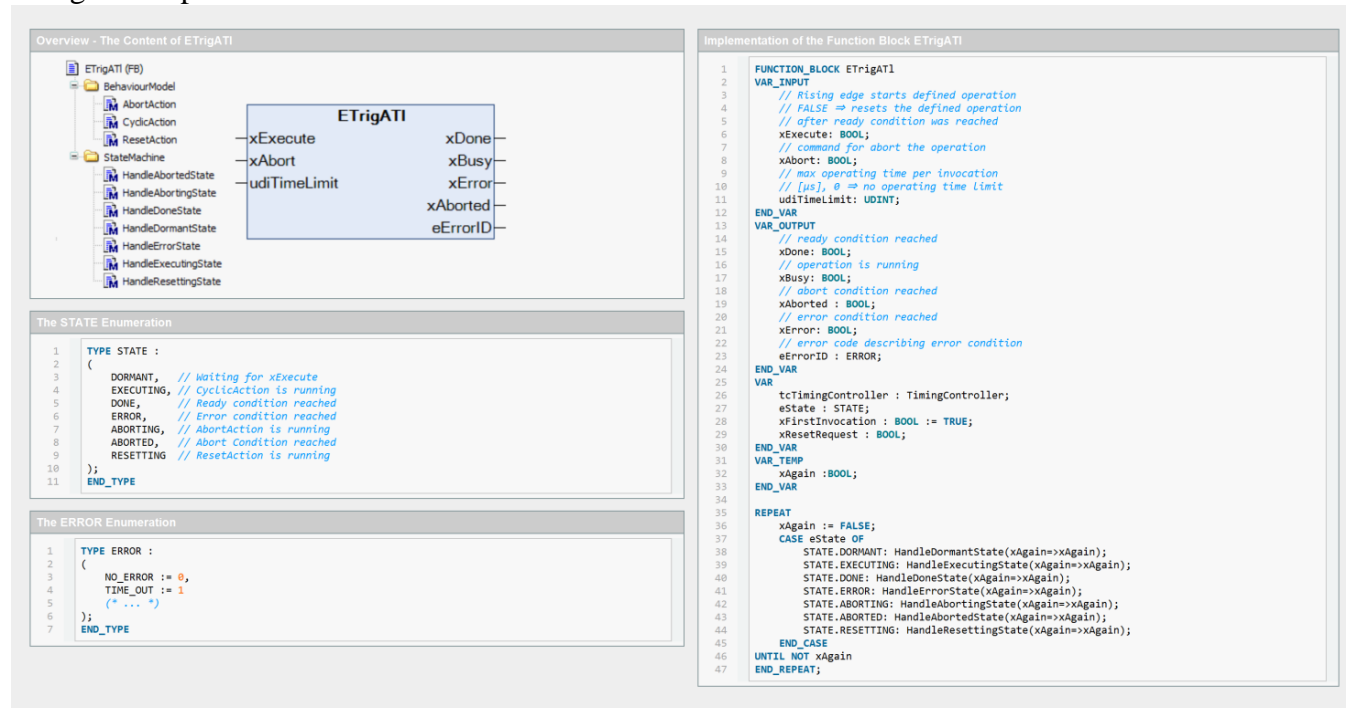
Appendix 1.3.6 ETrigATI

ETrigATI (Edge Triggered | Abortable | Time Limited | Not Time Out Constraint)

ETrigATI State Chart



ETrigATI Implementation



The Handler for the Dormant State

```
1 METHOD PRIVATE FINAL HandleDormantState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xExecute THEN
7   xBusy := TRUE;
8   eState := STATE.EXECUTING;
9   xAgain := TRUE;
10 END_IF
```

The Handler for the Executing State

```
1 METHOD PRIVATE FINAL HandleExecutingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 IF NOT xAbort THEN
10  tcTimingController.StartInvocationTimer();
11
12  CyclicAction(
13    xComplete=>xComplete,
14    eErrorID=>eErrorID
15  );
16 END_IF
17
18 IF eErrorID <> ERROR.NO_ERROR THEN
19   eState := STATE.ERROR;
20   xAgain := TRUE;
21 ELSEIF xAbort THEN
22   eState := STATE.ABORTING;
23   xAgain := TRUE;
24 ELSEIF xComplete THEN
25   eState := STATE.DONE;
26   xAgain := TRUE;
27 END_IF
```

The Handler for the Aborting State

```
1 METHOD PRIVATE FINAL HandleAbortingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 AbortAction(
10  xComplete=>xComplete,
11  eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15   eState := STATE.ERROR;
16   xAgain := TRUE;
17 ELSEIF xComplete THEN
18   eState := STATE.ABORTED;
19   xAgain := TRUE;
20 END_IF
```

The Handler for the Done State

```
1 METHOD PRIVATE FINAL HandleDoneState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xDone AND (xResetRequest OR NOT xExecute) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xDone := TRUE;
12  xResetRequest := NOT xExecute;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Error State

```
1 METHOD PRIVATE FINAL HandleErrorState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xError AND (xResetRequest OR NOT xExecute) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xError := TRUE;
12  xResetRequest := NOT xExecute;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Aborted State

```
1 METHOD PRIVATE FINAL HandleAbortedState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xAborted AND (xResetRequest OR NOT xExecute) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xAborted := TRUE;
12  xResetRequest := NOT xExecute;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1 METHOD PRIVATE FINAL HandleResettingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12   xBusy := FALSE;
13   xDone := FALSE;
14   xError := FALSE;
15   xAborted := FALSE;
16   eErrorID := ERROR.NO_ERROR;
17   eState := STATE.DORMANT;
18   xFirstInvocation := TRUE;
19   xAgain := xResetRequest; (* !!! *)
20   xResetRequest := FALSE;
21 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1 METHOD PROTECTED CyclicAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6 VAR
7   xTimeLimit : BOOL;
8 END_VAR
9
10 IF NOT xAbort THEN
11   IF xFirstInvocation THEN
12     (* Starting *)
13     // for the first (!) invocation,
14     // sample the input variables
15     tcTimingController.TimeLimit := udiTimeLimit;
16     xFirstInvocation := FALSE;
17   END_IF
18
19   REPEAT
20     (* Executing *)
21     // working to reach the ready condition
22     // => xComplete := TRUE
23     // if the maximum invocation time is reached
24     // => xTimeLimit := TRUE
25     // if an error condition is reached
26     // => set eErrorID to a value other than ERROR.NO_ERROR
27     tcTimingController.CheckTiming(
28       xTimeLimit=>xTimeLimit
29     );
30
31     xComplete := TRUE;
32     eErrorID := ERROR.NO_ERROR;
33
34     UNTIL xAbort OR xComplete OR xTimeLimit OR
35           eErrorID <> ERROR.NO_ERROR
36     END_REPEAT
37   END_IF
38
39   IF xAbort OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN
40     (* Cleaning *)
41     // if possible free as much allocated resources
42     // as possible
43   END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

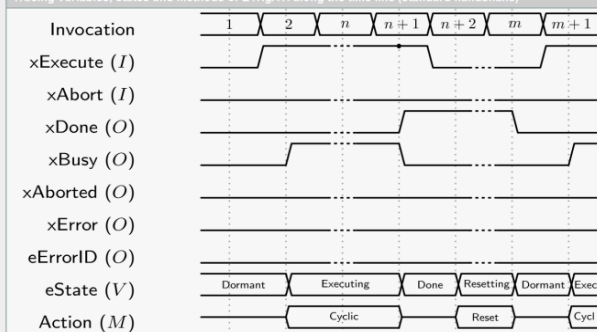
```
1 METHOD PROTECTED AbortAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6
7 // abort all running operations
8 // if an error condition is reached set
9 // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

The Implementation of the Reset Action (Exemplary Implementation)

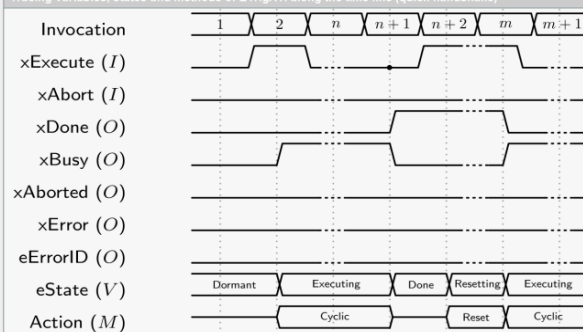
```
1 METHOD PROTECTED ResetAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4 END_VAR
5
6 // free all allocated resources
7 // reinitialize instance variables
8
9 xComplete := TRUE;
```

ETrigATI Timing Diagram

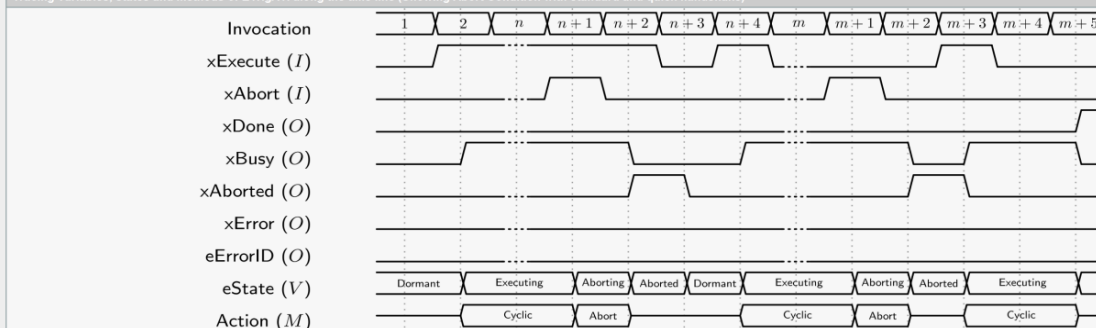
Tracing variables, states and methods of ETrigATI along the time line (standard handshake)



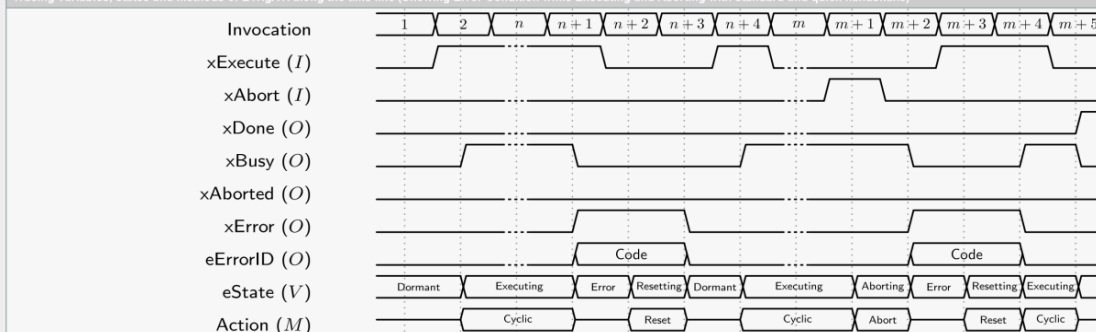
Tracing variables, states and methods of ETrigATI along the time line (quick handshake)



Tracing variables, states and methods of ETrigATI along the time line (Showing Abort Condition with standard and quick handshake)



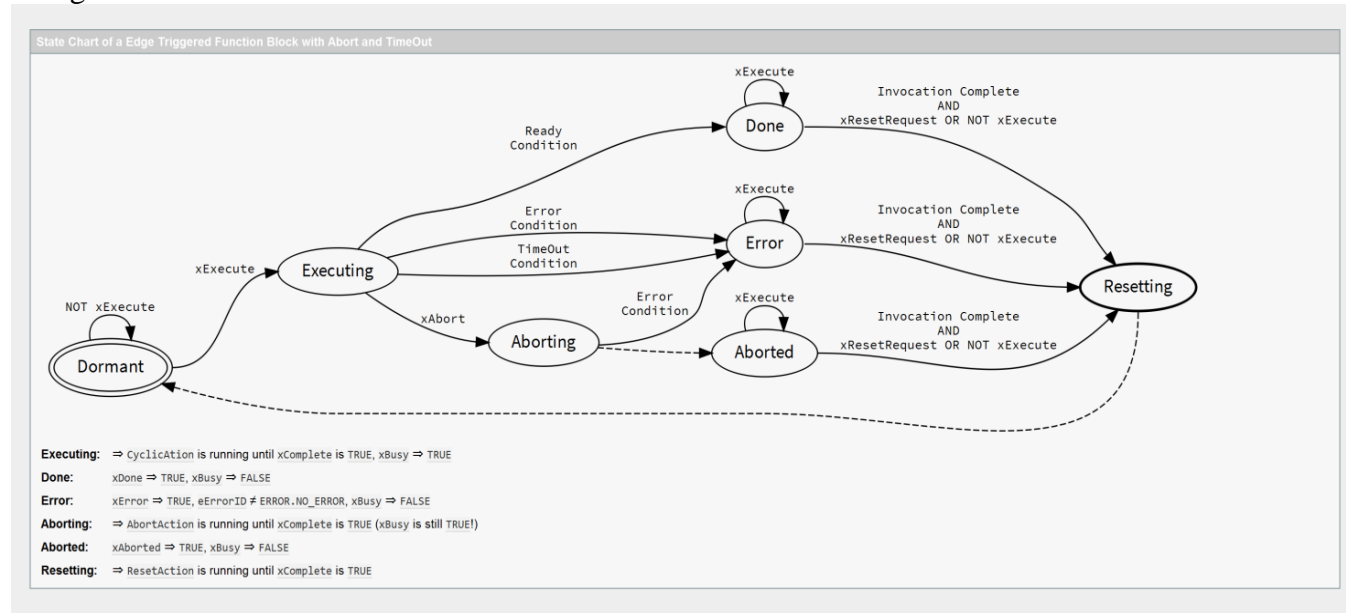
Tracing variables, states and methods of ETrigATI along the time line (Showing Error Condition while Executing and Aborting with standard and quick handshake)



Appendix 1.3.7 EtrigATo

ETrigATo (Edge Triggered | Abortable | Not Time Limited | Time Out Constraint)

ETrigATo State Chart



ETrigATo Implementation

Overview - The Content of ETrigATo

The STATE Enumeration

```

1 TYPE STATE :
2 (
3     DORMANT, // Waiting for xExecute
4     EXECUTING, // CyclicAction is running
5     DONE, // Ready condition reached
6     ERROR, // Error condition reached
7     ABORTING, // AbortAction is running
8     ABORTED, // Abort Condition reached
9     RESETTING // ResetAction is running
10 );
11 END_TYPE
    
```

The ERROR Enumeration

```

1 TYPE ERROR :
2 (
3     NO_ERROR := 0,
4     TIME_OUT := 1,
5     (* ... *)
6 );
7 END_TYPE
    
```

Implementation of the Function Block ETrigATo

```

1 FUNCTION_BLOCK ETrigATo
2 VAR_INPUT
3     // Rising edge starts defined operation
4     // FALSE ⇒ resets the defined operation
5     // after ready condition was reached
6     xExecute: BOOL;
7     // command for abort the operation
8     xAbort: BOOL;
9     // max operating time for executing
10    // [µs], 0 ⇒ no operating time limit
11    udiTimeOut: UDINT;
12 END_VAR
13 VAR_OUTPUT
14     // ready condition reached
15     xDone: BOOL;
16     // operation is running
17     xBusy: BOOL;
18     // abort condition reached
19     xAborted: BOOL;
20     // error condition reached
21     xError: BOOL;
22     // error code describing error condition
23     eErrorID: ERROR;
24 END_VAR
25 VAR
26     tcTimingController: TimingController;
27     eState: STATE;
28     xFirstInvocation: BOOL := TRUE;
29     xResetRequest: BOOL;
30 END_VAR
31 VAR_TEMP
32     xAgain: BOOL;
33 END_VAR
34 REPEAT
35     xAgain := FALSE;
36     CASE eState OF
37         STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
38         STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
39         STATE.DONE: HandleDoneState(xAgain⇒xAgain);
40         STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
41         STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
42         STATE.ABORTED: HandleAbortedState(xAgain⇒xAgain);
43         STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
44     END_CASE
45 UNTIL NOT xAgain
46 END_REPEAT;
    
```

The Handler for the Dormant State

```
1  METHOD PRIVATE FINAL HandleDormantState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xExecute THEN
7      tcTimingController.StartOperationTimer();
8      xBusy := TRUE;
9      eState := STATE.EXECUTING;
10     xAgain := TRUE;
11 END_IF
```

The Handler for the Executing State

```
1  METHOD PRIVATE FINAL HandleExecutingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  xTimeOut : BOOL;
8  END_VAR
9
10 IF NOT xAbort THEN
11     CyclicAction(
12         xComplete=>xComplete,
13         eErrorID=>eErrorID
14     );
15     tcTimingController.CheckTiming(xTimeOut=>xTimeOut);
16 END_IF
17
18 IF xTimeOut AND eErrorID = ERROR.NO_ERROR THEN
19     eErrorID := ERROR.TIME_OUT;
20 END_IF
21
22 IF eErrorID <> ERROR.NO_ERROR THEN
23     eState := STATE.ERROR;
24     xAgain := TRUE;
25 ELSEIF xAbort THEN
26     eState := STATE.ABORTING;
27     xAgain := TRUE;
28 ELSEIF xComplete THEN
29     eState := STATE.DONE;
30     xAgain := TRUE;
31 END_IF
32
```

The Handler for the Aborting State

```
1  METHOD PRIVATE FINAL HandleAbortingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  AbortAction(
10     xComplete=>xComplete,
11     eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15     eState := STATE.ERROR;
16     xAgain := TRUE;
17 ELSEIF xComplete THEN
18     eState := STATE.ABORTED;
19     xAgain := TRUE;
20 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1  METHOD PROTECTED CyclicAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6  VAR
7  xTimeOut : BOOL;
8  END_VAR
9
10 IF xAbort THEN
11     IF xFirstInvocation THEN
12         (* Starting *)
13         // for the first (!) invocation,
14         // sample the input variables
15         tcTimingController.TimeOut := udiTimeOut;
16         xFirstInvocation := FALSE;
17     END_IF
18
19     (* Executing *)
20     // working to reach the ready condition
21     // => xComplete := TRUE
22     // if the maximum operating time is reached
23     // => xTimeOut := TRUE
24     // if an error condition is reached
25     // => set eErrorID to a value other than ERROR.NO_ERROR
26     tcTimingController.CheckTiming(
27         xTimeOut=>xTimeOut,
28     );
29
30     xComplete := TRUE;
31     eErrorID := ERROR.NO_ERROR;
32 END_IF
33
34 IF xTimeOut AND eErrorID = ERROR.NO_ERROR THEN
35     eErrorID := ERROR.TIME_OUT;
36 END_IF
37
38 IF xAbort OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN
39     (* Cleaning *)
40     // if possible free as much allocated resources
41     // as possible
42 END_IF
```

The Handler for the Done State

```
1  METHOD PRIVATE FINAL HandleDoneState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xDone AND (xResetRequest OR NOT xExecute) THEN
7      eState := STATE.RESETTING;
8      xAgain := TRUE;
9  ELSE
10     xBusy := FALSE;
11     xDone := TRUE;
12     xResetRequest := NOT xExecute;
13     xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Error State

```
1  METHOD PRIVATE FINAL HandleErrorState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xError AND (xResetRequest OR NOT xExecute) THEN
7      eState := STATE.RESETTING;
8      xAgain := TRUE;
9  ELSE
10     xBusy := FALSE;
11     xError := TRUE;
12     xResetRequest := NOT xExecute;
13     xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Aborted State

```
1  METHOD PRIVATE FINAL HandleAbortedState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xAborted AND (xResetRequest OR NOT xExecute) THEN
7      eState := STATE.RESETTING;
8      xAgain := TRUE;
9  ELSE
10     xBusy := FALSE;
11     xAborted := TRUE;
12     xResetRequest := NOT xExecute;
13     xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1  METHOD PRIVATE FINAL HandleResettingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12     xBusy := FALSE;
13     xDone := FALSE;
14     xError := FALSE;
15     xAborted := FALSE;
16     eErrorID := ERROR.NO_ERROR;
17     eState := STATE.DORMANT;
18     xFirstInvocation := TRUE;
19     xAgain := xResetRequest; (* !!! *)
20     xResetRequest := FALSE;
21 END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

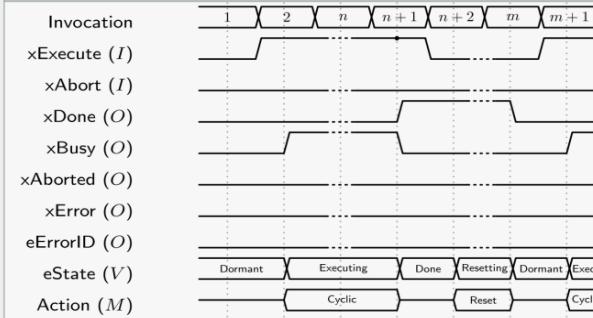
```
1  METHOD PROTECTED AbortAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6
7  // abort all running operations
8  // if an error condition is reached set
9  // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

The Implementation of the Reset Action (Exemplary Implementation)

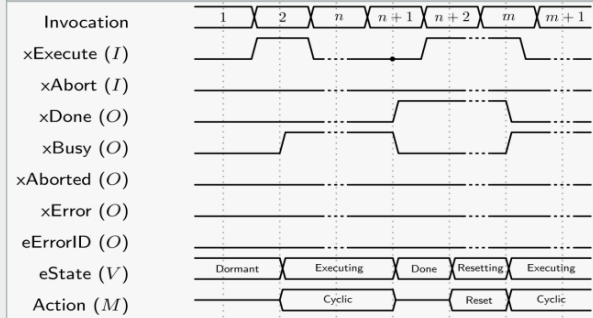
```
1  METHOD PROTECTED ResetAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  END_VAR
5
6  // free all allocated resources
7  // reinitialize instance variables
8
9  xComplete := TRUE;
```

ETrigATo Timing Diagram

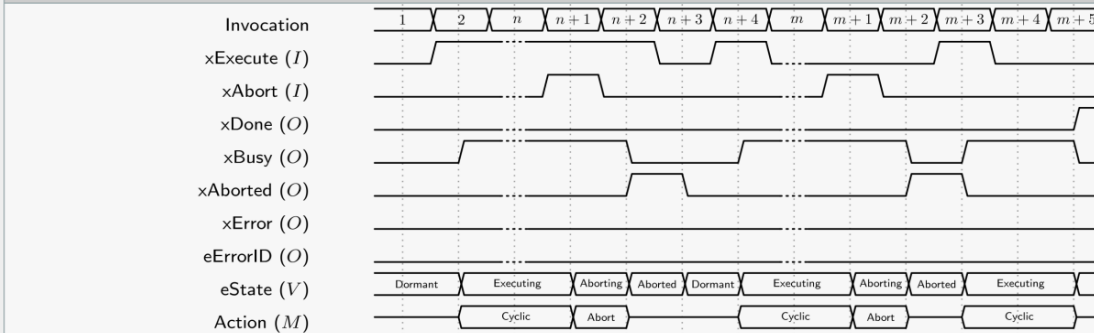
Tracing variables, states and methods of ETrigATo along the time line (standard handshake)



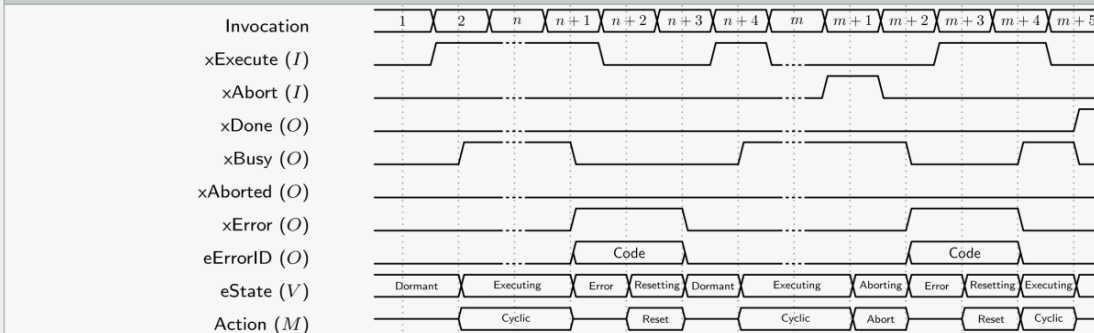
Tracing variables, states and methods of ETrigATo along the time line (quick handshake)



Tracing variables, states and methods of ETrigATo along the time line (Showing Abort Condition with standard and quick handshake)



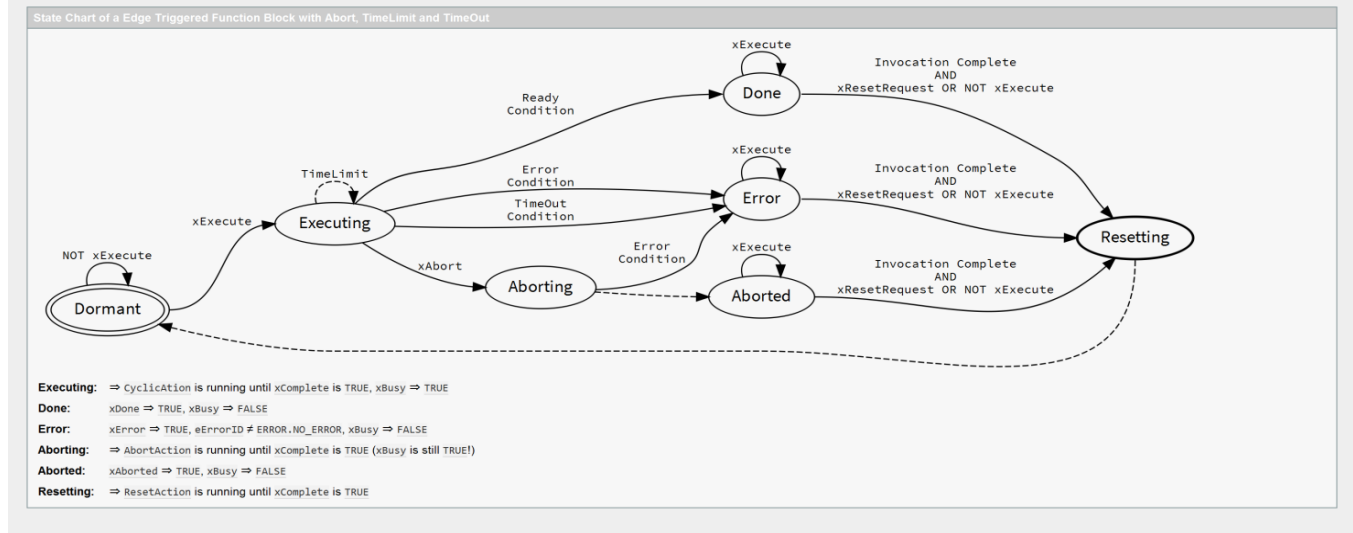
Tracing variables, states and methods of ETrigATo along the time line (Showing Error Condition while Executing and Aborting with standard and quick handshake)



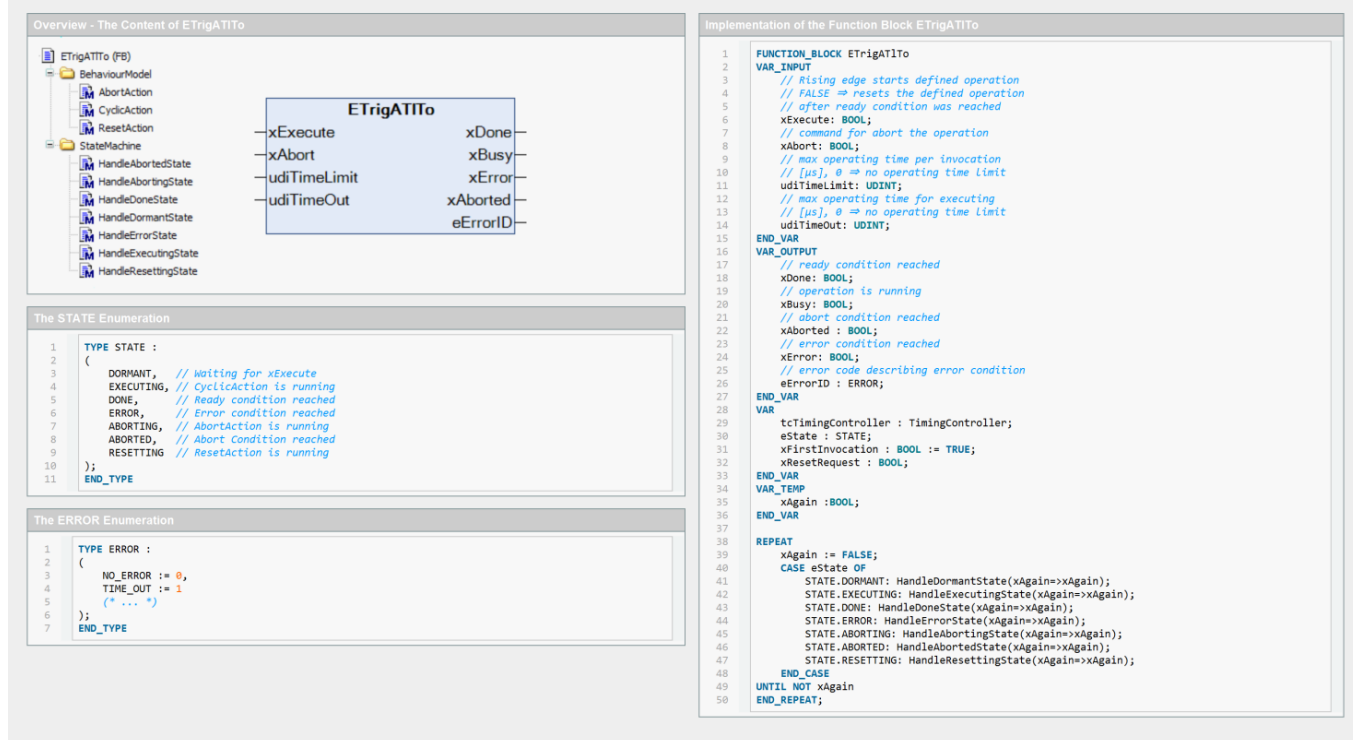
Appendix 1.3.8 ETrigATITo

ETrigATITo (Edge Triggered | Abortable | Time Limited | Time Out Constraint)

ETrigATITo State Chart



ETrigATITo Implementation



The Handler for the Dormant State

```
1 METHOD PRIVATE FINAL HandleDormantState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xExecute THEN
7   tcTimingController.StartOperationTimer();
8   xBusy := TRUE;
9   eState := STATE.EXECUTING;
10  xAgain := TRUE;
11 END_IF
```

The Handler for the Executing State

```
1 METHOD PRIVATE FINAL HandleExecutingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7   xTimeOut : BOOL;
8 END_VAR
9
10 IF NOT xAbort THEN
11   tcTimingController.StartInvocationTimer();
12
13   CyclicAction(
14     xComplete=>xComplete,
15     eErrorID=>eErrorID
16   );
17
18   tcTimingController.CheckTiming(xTimeOut=>xTimeOut);
19 END_IF
20
21 IF xTimeOut AND eErrorID = ERROR.NO_ERROR THEN
22   eErrorID := ERROR.TIME_OUT;
23 END_IF
24
25 IF eErrorID <> ERROR.NO_ERROR THEN
26   eState := STATE.ERROR;
27   xAgain := TRUE;
28 ELSEIF xAbort THEN
29   eState := STATE.ABORTING;
30   xAgain := TRUE;
31 ELSEIF xComplete THEN
32   eState := STATE.DONE;
33   xAgain := TRUE;
34 END_IF
```

The Handler for the Aborting State

```
1 METHOD PRIVATE FINAL HandleAbortingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 AbortAction(
10  xComplete=>xComplete,
11  eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15   eState := STATE.ERROR;
16   xAgain := TRUE;
17 ELSEIF xComplete THEN
18   eState := STATE.ABORTED;
19   xAgain := TRUE;
20 END_IF
```

The Handler for the Done State

```
1 METHOD PRIVATE FINAL HandleDoneState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xDone AND (xResetRequest OR NOT xExecute) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xDone := TRUE;
12  xResetRequest := NOT xExecute;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Error State

```
1 METHOD PRIVATE FINAL HandleErrorState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xError AND (xResetRequest OR NOT xExecute) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xError := TRUE;
12  xResetRequest := NOT xExecute;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Aborted State

```
1 METHOD PRIVATE FINAL HandleAbortedState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xAborted AND (xResetRequest OR NOT xExecute) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xAborted := TRUE;
12  xResetRequest := NOT xExecute;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler of the Resetting State

```
1 METHOD PRIVATE FINAL HandleResettingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12   xBusy := FALSE;
13   xDone := FALSE;
14   xError := FALSE;
15   xAborted := FALSE;
16   eErrorID := ERROR.NO_ERROR;
17   eState := STATE.DORMANT;
18   xFirstInvocation := TRUE;
19   xAgain := xResetRequest; (* !!! *)
20   xResetRequest := FALSE;
21 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1 METHOD PROTECTED CyclicAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6 VAR
7   xTimeOut : BOOL;
8   xTimeLimit : BOOL;
9 END_VAR
10
11 IF NOT xAbort THEN
12   IF xFirstInvocation THEN
13     (* Starting *)
14     // for the first (!) invocation,
15     // sample the input variables
16     tcTimingController.TimeLimit := udiTimeLimit;
17     tcTimingController.TimeOut := udiTimeOut;
18     xFirstInvocation := FALSE;
19   END_IF
20
21   REPEAT
22     (* Executing *)
23     // working to reach the ready condition
24     // => xComplete := TRUE
25     // if the maximum invocation time is reached
26     // => xTimeLimit := TRUE
27     // if the maximum operating time is reached
28     // => xTimeOut := TRUE
29     // if an error condition is reached
30     // => set eErrorID to a value other than ERROR.NO_ERROR
31     tcTimingController.CheckTiming(
32       xTimeOut=>xTimeOut,
33       xTimeLimit=>xTimeLimit
34     );
35
36     xComplete := TRUE;
37     eErrorID := ERROR.NO_ERROR;
38   UNTIL xAbort OR xComplete OR
39     xTimeOut OR xTimeLimit OR
40     eErrorID <> ERROR.NO_ERROR
41   END_REPEAT
42 END_IF
43
44 IF xTimeOut AND eErrorID = ERROR.NO_ERROR THEN
45   eErrorID := ERROR.TIME_OUT;
46 END_IF
47
48 IF xAbort OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN
49   (* Cleaning *)
50   // if possible free as much allocated resources
51   // as possible
52 END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

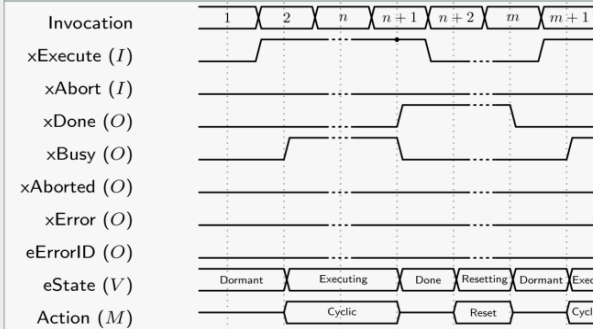
```
1 METHOD PROTECTED AbortAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6
7 // abort all running operations
8 // if an error condition is reached set
9 // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

The Implementation of the Reset Action (Exemplary Implementation)

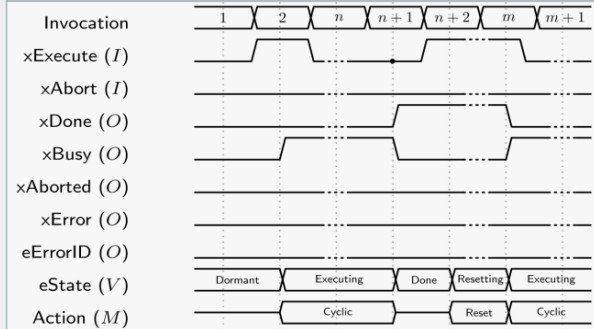
```
1 METHOD PROTECTED ResetAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4 END_VAR
5
6 // free all allocated resources
7 // reinitialize instance variables
8
9 xComplete := TRUE;
```

ETrigATITo Timing Diagram

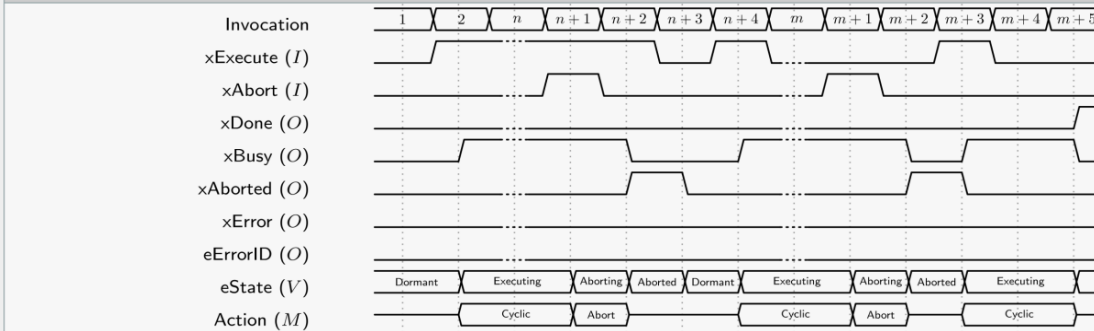
Tracing variables, states and methods of ETrigATITo along the time line (standard handshake)



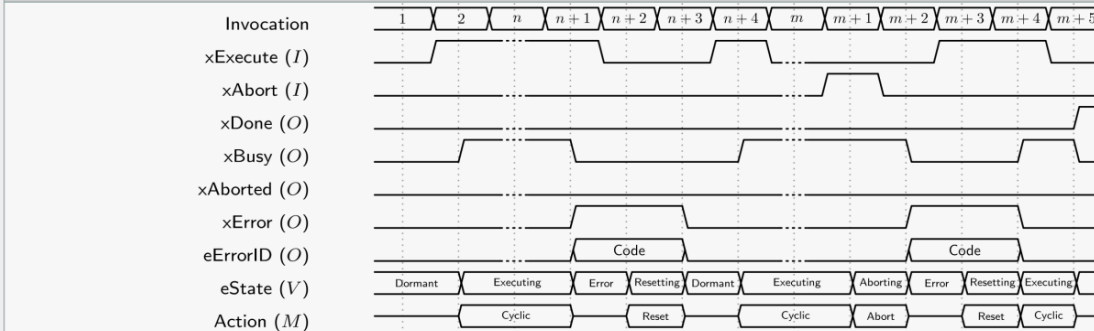
Tracing variables, states and methods of ETrigATITo along the time line (quick handshake)



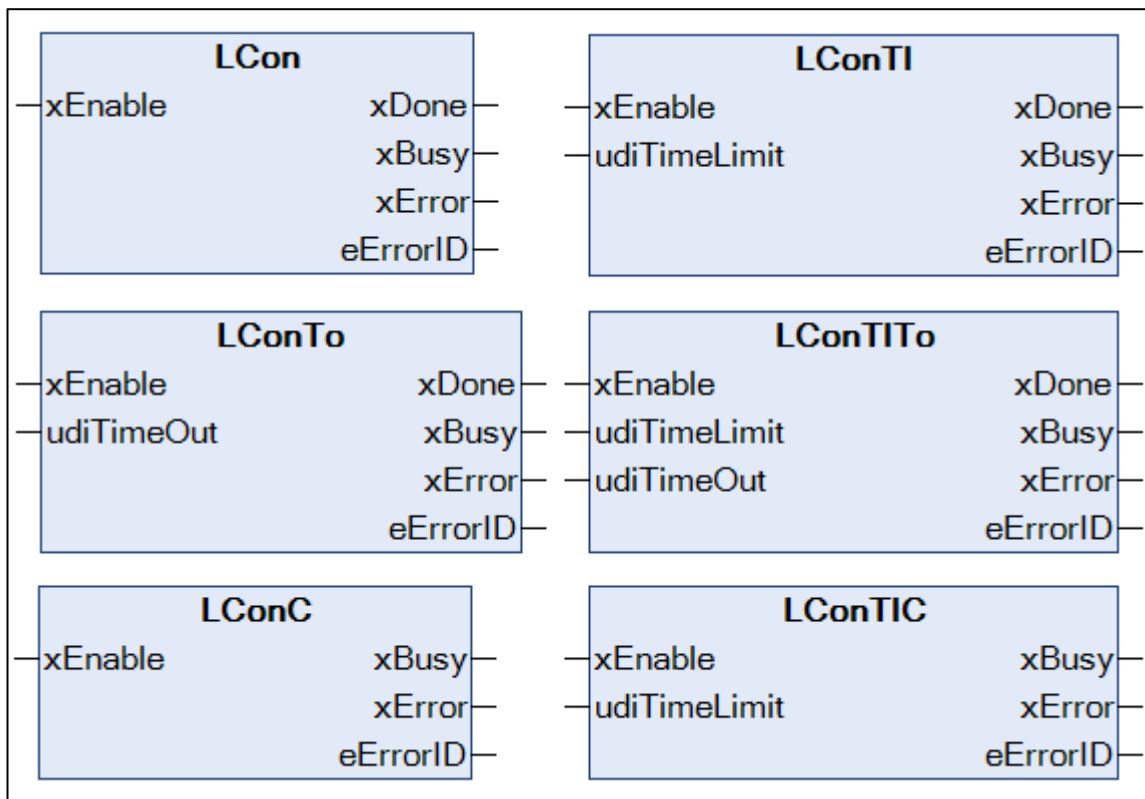
Tracing variables, states and methods of ETrigATITo along the time line (Showing Abort Condition with standard and quick handshake)



Tracing variables, states and methods of ETrigATITo along the time line (Showing Error Condition while Executing and Aborting with standard and quick handshake)



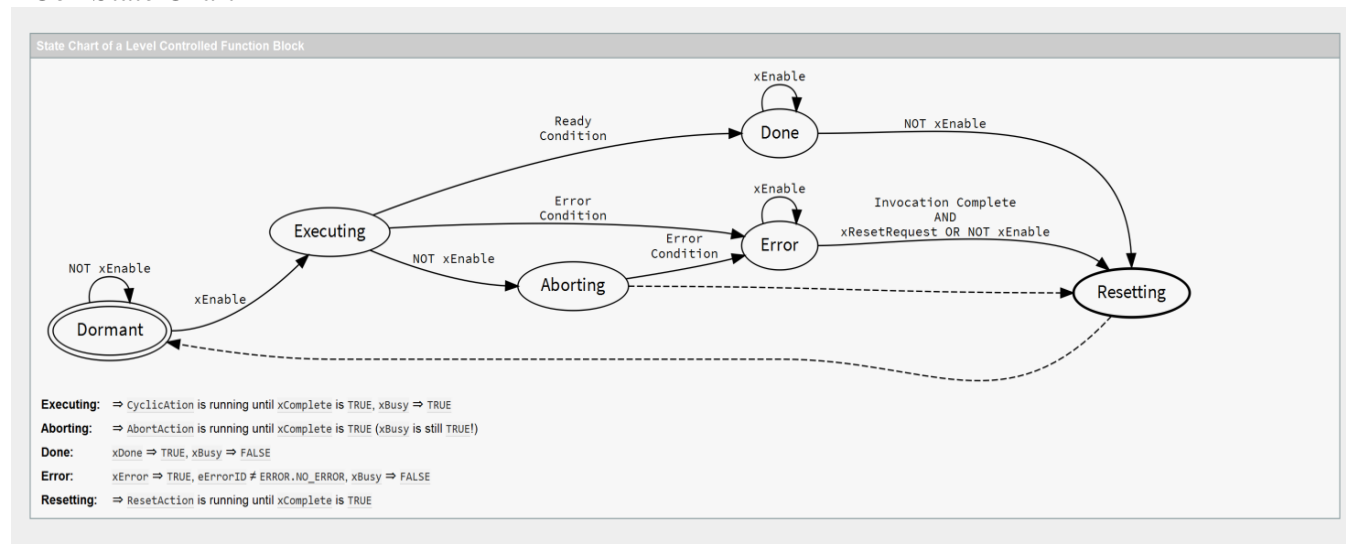
Appendix 1.4 Overview Level Controlled FBs



Appendix 1.4.1 LCon

LCon (Level Controlled | Not Time Limited | No Time Out Constraint | No Continuous Behaviour)

LCon State Chart



LCon Implementation

Overview - The Content of LCon

The STATE Enumeration

```

1 TYPE STATE :
2 {
3     DORMANT, // Waiting for xEnable
4     EXECUTING, // CyclicAction is running
5     ABORTING, // AbortAction is running
6     DONE, // Ready condition reached
7     ERROR, // Error condition reached
8     RESETTNG, // ResetAction is running
9 }
10 END_TYPE

```

The ERROR Enumeration

```

1 TYPE ERROR :
2 {
3     NO_ERROR := 0,
4     TIME_OUT := 1
5     (* ... *)
6 }
7 END_TYPE

```

Implementation of the Function Block LCon

```

1 FUNCTION_BLOCK LCon
2 VAR_INPUT
3     // TRUE ⇒ activates the defined operation
4     // FALSE ⇒ aborts/resets the defined operation
5     xEnable: BOOL;
6 END_VAR
7 VAR_OUTPUT
8     // ready condition reached
9     xDone: BOOL;
10    // operation is running
11    xBusy: BOOL;
12    // error condition reached
13    xError: BOOL;
14    // error code describing error condition
15    eErrorID : ERROR;
16 END_VAR
17 VAR
18     eState : STATE;
19     xResetRequest : BOOL;
20 END_VAR
21 VAR_TEMP
22     xAgain : BOOL;
23 END_VAR
24 REPEAT
25     xAgain := FALSE;
26     CASE eState OF
27         STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
28         STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
29         STATE.DONE: HandleDoneState(xAgain⇒xAgain);
30         STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
31         STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
32         STATE.RESETTNG: HandleResettingState(xAgain⇒xAgain);
33     END_CASE
34     UNTIL NOT xAgain
35     END_REPEAT;
36 
```


The Handler for the Dormant State

```
1 METHOD PRIVATE FINAL HandleDormantState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xEnable THEN
7   xBusy := TRUE;
8   eState := STATE.EXECUTING;
9   xAgain := TRUE;
10 END_IF
```

The Handler for the Executing State

```
1 METHOD PRIVATE FINAL HandleExecutingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7   xTimeOut : BOOL;
8 END_VAR
9
10 IF xEnable THEN
11   CyclicAction(
12     xComplete=>xComplete,
13     eErrorID=>eErrorID
14   );
15 END_IF
16
17 IF eErrorID <> ERROR.NO_ERROR THEN
18   eState := STATE.ERROR;
19   xAgain := TRUE;
20 ELSEIF NOT xEnable THEN
21   eState := STATE.ABORTING;
22   xAgain := TRUE;
23 ELSEIF xComplete THEN
24   eState := STATE.DONE;
25   xAgain := TRUE;
26 END_IF
```

The Handler for the Aborting State

```
1 METHOD PRIVATE FINAL HandleAbortingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 AbortAction(
10   xComplete=>xComplete,
11   eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15   eState := STATE.ERROR;
16   xAgain := TRUE;
17 ELSEIF xComplete THEN
18   eState := STATE.RESETTING;
19   xAgain := TRUE;
20 END_IF
```

The Handler for the Done State

```
1 METHOD PRIVATE FINAL HandleDoneState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xDone AND NOT xEnable THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xDone := TRUE;
12  xAgain := FALSE; (* !!! *)
13 END_IF
```

The Handler for the Error State

```
1 METHOD PRIVATE FINAL HandleErrorState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xError AND (xResetRequest OR NOT xEnable) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xError := TRUE;
12  xResetRequest := NOT xEnable;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1 METHOD PRIVATE FINAL HandleResettingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12   xBusy := FALSE;
13   xDone := FALSE;
14   xError := FALSE;
15   eErrorID := ERROR.NO_ERROR;
16   eState := STATE.DORMANT;
17   xAgain := xResetRequest; (* !!! *)
18   xResetRequest := FALSE;
19 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1 METHOD PROTECTED CyclicAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6
7 IF xEnable THEN
8   (* Executing *)
9   // for every invocation,
10  // sample the input variables
11
12  // working to reach the ready condition
13  // => xComplete := TRUE
14  // if an error condition is reached set
15  // eErrorID to a value other than ERROR.NO_ERROR
16
17  xComplete := TRUE;
18  eErrorID := ERROR.NO_ERROR;
19 END_IF
20
21 IF NOT xEnable OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN
22   (* Cleaning *)
23   // if possible free as much allocated resources
24   // as possible
25 END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

```
1 METHOD PROTECTED AbortAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6
7 // abort all running operations
8 // if an error condition is reached set
9 // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

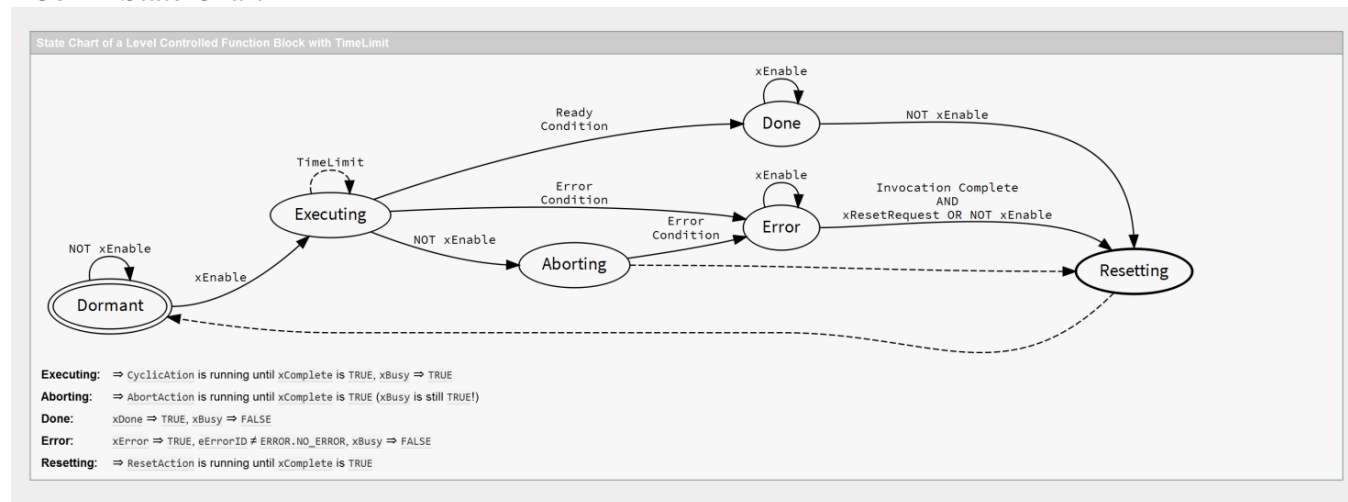
The Implementation of the Reset Action (Exemplary Implementation)

```
1 METHOD PROTECTED ResetAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4 END_VAR
5
6 // free all allocated resources
7 // reinitialize instance variables
8
9 xComplete := TRUE;
```

Appendix 1.4.2 LConTl

LConTl (Level Controlled | Time Limited | Not Time Out Constraint | No Continuous Behaviour)

LConTl State Chart



LConTl Implementation

Overview - The Content of LConTl

The STATE Enumeration

```

1  TYPE STATE :
2  (
3    DORMANT, // waiting for xEnable
4    EXECUTING, // CyclicAction is running
5    ABORTING, // AbortAction is running
6    DONE, // Ready condition reached
7    ERROR, // Error condition reached
8    RESETTING // ResetAction is running
9  );
10 END_TYPE

```

The ERROR Enumeration

```

1  TYPE ERROR :
2  (
3    NO_ERROR := 0,
4    TIME_OUT := 1
5  );
6  (* ... *)
7  END_TYPE

```

Implementation of the Function Block LConTl

```

1  FUNCTION_BLOCK LConTl
2  VAR_INPUT
3    // TRUE ⇒ activates the defined operation
4    // FALSE ⇒ aborts/resets the defined operation
5    xEnable: BOOL;
6    // max operating time per invocation
7    // [µs], 0 ⇒ no operating time limit
8    udiTimeLimit: UDINT;
9  END_VAR
10 VAR_OUTPUT
11   // ready condition reached
12   xDone: BOOL;
13   // operation is running
14   xBusy: BOOL;
15   // error condition reached
16   xError: BOOL;
17   // error code describing error condition
18   eErrorID: ERROR;
19 END_VAR
20 VAR
21   tcTimingController: TimingController;
22   eState: STATE;
23   xResetRequest: BOOL;
24 END_VAR
25 VAR_TEMP
26   xAgain: BOOL;
27 END_TEMP
28 END_VAR
29 REPEAT
30   xAgain := FALSE;
31   CASE eState OF
32     STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
33     STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
34     STATE.DONE: HandleDoneState(xAgain⇒xAgain);
35     STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
36     STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
37     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
38   END_CASE
39   UNTIL NOT xAgain
40 END_REPEAT;

```

The Handler for the Dormant State

```
1  METHOD PRIVATE FINAL HandleDormantState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xEnable THEN
7    xBusy := TRUE;
8    eState := STATE.EXECUTING;
9    xAgain := TRUE;
10 END_IF
```

The Handler for the Executing State

```
1  METHOD PRIVATE FINAL HandleExecutingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  IF xEnable THEN
10   tcTimingController.StartInvocationTimer();
11
12   CyclicAction(
13     xComplete=>xComplete,
14     eErrorID=>eErrorID
15   );
16 END_IF
17
18 IF eErrorID <> ERROR.NO_ERROR THEN
19   eState := STATE.ERROR;
20   xAgain := TRUE;
21 ELSEIF NOT xEnable THEN
22   eState := STATE.ABORTING;
23   xAgain := TRUE;
24 ELSEIF xComplete THEN
25   eState := STATE.DONE;
26   xAgain := TRUE;
27 END_IF
```

The Handler for the Aborting State

```
1  METHOD PRIVATE FINAL HandleAbortingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  AbortAction(
10   xComplete=>xComplete,
11   eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15   eState := STATE.ERROR;
16   xAgain := TRUE;
17 ELSEIF xComplete THEN
18   eState := STATE.RESETTING;
19   xAgain := TRUE;
20 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1  METHOD PROTECTED CyclicAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6  VAR
7  xTimeLimit : BOOL;
8  END_VAR
9
10 IF xEnable THEN
11   (* Executing *)
12   // for every invocation,
13   // sample the input variables
14   tcTimingController.TimeLimit := udiTimeLimit;
15
16   REPEAT
17     // working to reach the ready condition
18     // => xComplete := TRUE
19     // if the maximum invocation time is reached
20     // => xTimeLimit := TRUE
21     // if an error condition is reached set
22     // eErrorID to a value other than ERROR.NO_ERROR
23     tcTimingController.CheckTiming(
24       xTimeLimit=>xTimeLimit
25     );
26
27     xComplete := TRUE;
28     eErrorID := ERROR.NO_ERROR;
29
30   UNTIL NOT xEnable OR xComplete OR xTimeLimit OR
31     eErrorID <> ERROR.NO_ERROR
32   END_REPEAT
33 END_IF
34
35 IF NOT xEnable OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN
36   (* Cleaning *)
37   // if possible free as much allocated resources
38   // as possible
39 END_IF
```

The Handler for the Done State

```
1  METHOD PRIVATE FINAL HandleDoneState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xDone AND NOT xEnable THEN
7    eState := STATE.RESETTING;
8    xAgain := TRUE;
9  ELSE
10   xBusy := FALSE;
11   xDone := TRUE;
12   xAgain := FALSE; (* !!! *)
13 END_IF
```

The Handler for the Error State

```
1  METHOD PRIVATE FINAL HandleErrorState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5
6  IF xError AND (xResetRequest OR NOT xEnable) THEN
7    eState := STATE.RESETTING;
8    xAgain := TRUE;
9  ELSE
10   xBusy := FALSE;
11   xError := TRUE;
12   xResetRequest := NOT xEnable;
13   xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1  METHOD PRIVATE FINAL HandleResettingState
2  VAR_OUTPUT
3  xAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12   xBusy := FALSE;
13   xDone := FALSE;
14   xError := FALSE;
15   eErrorID := ERROR.NO_ERROR;
16   eState := STATE.DORMANT;
17   xAgain := xResetRequest; (* !!! *)
18   xResetRequest := FALSE;
19 END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

```
1  METHOD PROTECTED AbortAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6
7  // abort all running operations
8  // if an error condition is reached set
9  // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

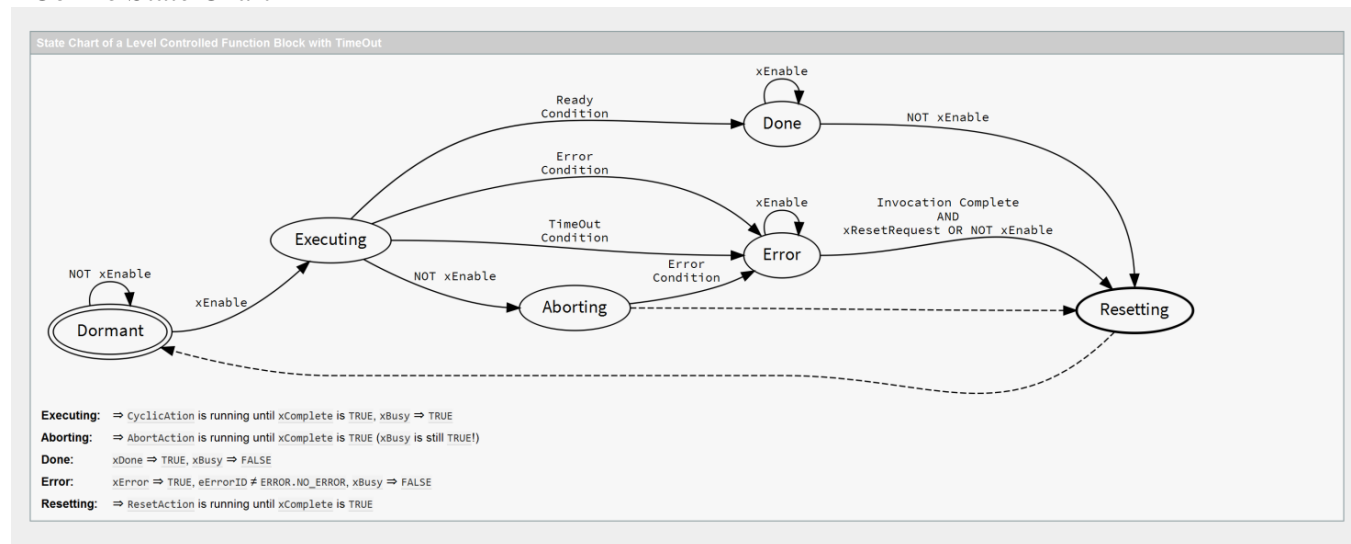
The Implementation of the Reset Action (Exemplary Implementation)

```
1  METHOD PROTECTED ResetAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  END_VAR
5
6  // free all allocated resources
7  // reinitialize instance variables
8
9  xComplete := TRUE;
```

Appendix 1.4.3 LConTo

LConTo (Level Controlled | Not Time Limited | Time Out Constraint | No Continuous Behaviour)

LConTo State Chart



LConTo Implementation

Overview - The Content of LConTo

The STATE Enumeration

```

1  TYPE STATE :
2  (
3    DORMANT, // Waiting for xEnable
4    EXECUTING, // CyclicAction is running
5    ABORTING, // AbortAction is running
6    DONE, // Ready condition reached
7    ERROR, // Error condition reached
8    RESETTING // ResetAction is running
9  );
10 END_TYPE

```

The ERROR Enumeration

```

1  TYPE ERROR :
2  (
3    NO_ERROR := 0,
4    TIME_OUT := 1
5  );
6  (* ... *)
7  END_TYPE

```

Implementation of the Function Block LConTo

```

1  FUNCTION_BLOCK LConTo
2  VAR_INPUT
3    // TRUE ⇒ activates the defined operation
4    // FALSE ⇒ aborts/resets the defined operation
5    xEnable: BOOL;
6    // max operating time for executing
7    // [µs], 0 ⇒ no operating time limit
8    udiTimeOut: UDINT;
9  END_VAR
10 VAR_OUTPUT
11   // ready condition reached
12   xDone: BOOL;
13   // operation is running
14   xBusy: BOOL;
15   // error condition reached
16   xError: BOOL;
17   // error code describing error condition
18   eErrorID: ERROR;
19 END_VAR
20 VAR
21   tcTimingController: TimingController;
22   eState: STATE;
23   xResetRequest: BOOL;
24 END_VAR
25 VAR_TEMP
26   xAgain: BOOL;
27 END_VAR
28 REPEAT
29   xAgain := FALSE;
30   CASE eState OF
31     STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
32     STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
33     STATE.DONE: HandleDoneState(xAgain⇒xAgain);
34     STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
35     STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
36     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
37   END_CASE
38 UNTIL NOT xAgain
39 END_REPEAT;
40

```

The Handler for the Dormant State

```
1 METHOD PRIVATE FINAL HandleDormantState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xEnable THEN
7   tcTimingController.StartOperationTimer();
8   xBusy := TRUE;
9   eState := STATE.EXECUTING;
10  xAgain := TRUE;
11 END_IF
```

The Handler for the Executing State

```
1 METHOD PRIVATE FINAL HandleExecutingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xTimeout : BOOL;
7 END_VAR
8
9 IF xEnable THEN
10  CyclicAction(
11    eErrorID=>eErrorID
12  );
13
14  tcTimingController.CheckTiming(xTimeout=>xTimeout);
15 END_IF
16
17 IF xTimeout AND eErrorID = ERROR.NO_ERROR THEN
18   eErrorID := ERROR.TIME_OUT;
19 END_IF
20
21 IF eErrorID <> ERROR.NO_ERROR THEN
22   eState := STATE.ERROR;
23   xAgain := TRUE;
24 ELSEIF NOT xEnable THEN
25   eState := STATE.ABORTING;
26   xAgain := TRUE;
27 END_IF
```

The Handler for the Aborting State

```
1 METHOD PRIVATE FINAL HandleAbortingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 AbortAction(
10  xComplete=>xComplete,
11  eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15   eState := STATE.ERROR;
16   xAgain := TRUE;
17 ELSEIF xComplete THEN
18   eState := STATE.RESETTING;
19   xAgain := TRUE;
20 END_IF
```

The Handler for the Done State

```
1 METHOD PRIVATE FINAL HandleDoneState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xDone AND NOT xEnable THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xDone := TRUE;
12  xAgain := FALSE; (* !!! *)
13 END_IF
```

The Handler for the Error State

```
1 METHOD PRIVATE FINAL HandleErrorState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xError AND (xResetRequest OR NOT xEnable) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xError := TRUE;
12  xResetRequest := NOT xEnable;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1 METHOD PRIVATE FINAL HandleResettingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12  xBusy := FALSE;
13  xDone := FALSE;
14  xError := FALSE;
15  eErrorID := ERROR.NO_ERROR;
16  eState := STATE.DORMANT;
17  xAgain := xResetRequest; (* !!! *)
18  xResetRequest := FALSE;
19 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1 METHOD PROTECTED CyclicAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6 VAR
7   xTimeout : BOOL;
8 END_VAR
9
10 IF xEnable THEN
11  (* Executing *)
12  // for every invocation,
13  // sample the input variables
14  tcTimingController.Timeout := udTimeout;
15
16  // working to reach the ready condition
17  // => xComplete := TRUE
18  // if the maximum operating time is reached
19  // => xTimeout := TRUE
20  // if an error condition is reached set
21  // eErrorID to a value other than ERROR.NO_ERROR
22  tcTimingController.CheckTiming(
23    xTimeout=>xTimeout
24  );
25
26  xComplete := TRUE;
27  eErrorID := ERROR.NO_ERROR;
28 END_IF
29
30 IF xTimeout AND eErrorID = ERROR.NO_ERROR THEN
31   eErrorID := ERROR.TIME_OUT;
32 END_IF
33
34 IF NOT xEnable OR eErrorID <> ERROR.NO_ERROR THEN
35  (* Cleaning *)
36  // if possible free as much allocated resources
37  // as possible
38 END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

```
1 METHOD PROTECTED AbortAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6
7 // abort all running operations
8 // if an error condition is reached set
9 // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

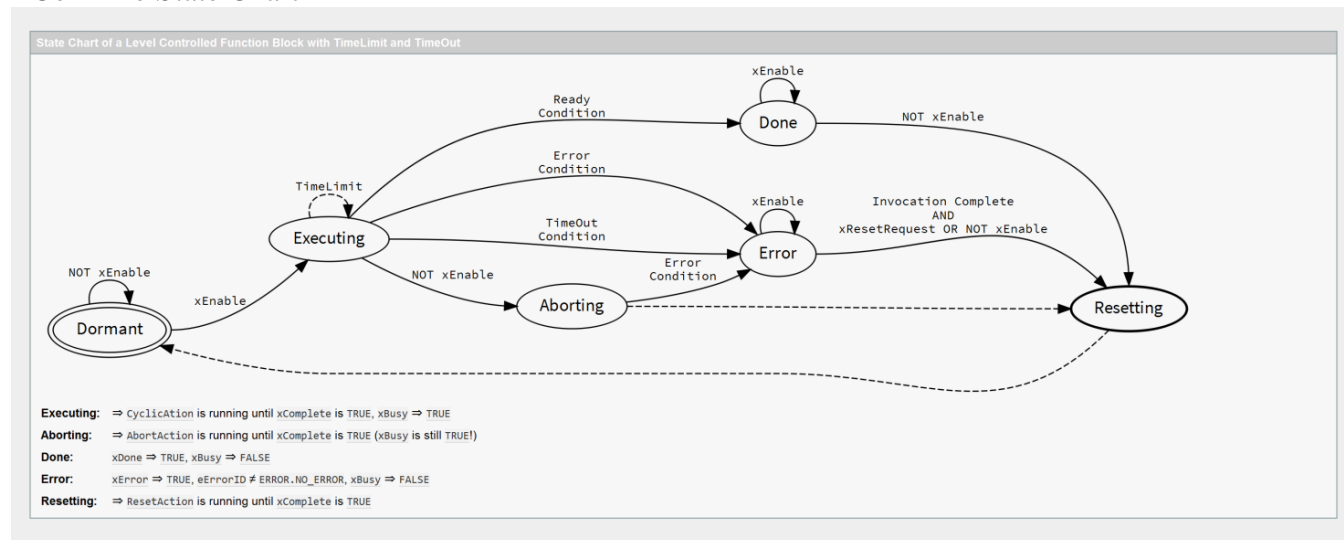
The Implementation of the Reset Action (Exemplary Implementation)

```
1 METHOD PROTECTED ResetAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4 END_VAR
5
6 // free all allocated resources
7 // reinitialize instance variables
8
9 xComplete := TRUE;
```

Appendix 1.4.4 LConTITo

LConTITo (Level Controlled | Time Limited | Time Out Constraint | No Continuous Behaviour)

LConTITo State Chart



LConTITo Implementation

Overview - The Content of LConTITo

LConTITo

Inputs: xEnable, udiTimeLimit, udiTimeOut
 Outputs: xDone, xBusy, xError, xErrorID

The STATE Enumeration

```

1  TYPE STATE :
2  (
3      DORMANT, // waiting for xenable
4      EXECUTING, // CyclicAction is running
5      ABORTING, // AbortAction is running
6      DONE, // Ready condition reached
7      ERROR, // Error condition reached
8      RESETTING // ResetAction is running
9  );
10 END_TYPE
    
```

The ERROR Enumeration

```

1  TYPE ERROR :
2  (
3      NO_ERROR := 0,
4      TIME_OUT := 1
5      (* ... *)
6  );
7  END_TYPE
    
```

Implementation of the Function Block LConTITo

```

1  FUNCTION_BLOCK LConTITo
2  VAR_INPUT
3      // TRUE ⇒ activates the defined operation
4      // FALSE ⇒ aborts/resets the defined operation
5      xEnable: BOOL;
6      // max operating time per invocation
7      // [µs], 0 ⇒ no operating time limit
8      udiTimeLimit: UDINT;
9      // max operating time for executing
10     // [µs], 0 ⇒ no operating time limit
11     udiTimeOut: UDINT;
12 END_VAR
13 VAR_OUTPUT
14     // ready condition reached
15     xDone: BOOL;
16     // operation is running
17     xBusy: BOOL;
18     // error condition reached
19     xError: BOOL;
20     // error code describing error condition
21     eErrorID: ERROR;
22 END_VAR
23 VAR
24     tcTimingController: TimingController;
25     eState: STATE;
26     xResetRequest: BOOL;
27 END_VAR
28 VAR_TEMP
29     xAgain: BOOL;
30 END_TEMP
31 END_VAR
32 REPEAT
33     xAgain := FALSE;
34     CASE eState OF
35         STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
36         STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
37         STATE.DONE: HandleDoneState(xAgain⇒xAgain);
38         STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
39         STATE.ABORTING: HandleAbortState(xAgain⇒xAgain);
40         STATE.RESETTING: HandleResetState(xAgain⇒xAgain);
41     END_CASE
42 UNTIL NOT xAgain
43 END_REPEAT;
    
```

The Handler for the Dormant State

```
1  METHOD PRIVATE FINAL HandleDormantState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5
6  IF xEnable THEN
7      tcTimingController.StartOperationTimer();
8      xBusy := TRUE;
9      eState := STATE.EXECUTING;
10     XAgain := TRUE;
11 END_IF
```

The Handler for the Executing State

```
1  METHOD PRIVATE FINAL HandleExecutingState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  xTimeout : BOOL;
8  END_VAR
9
10 IF xEnable THEN
11     tcTimingController.StartInvocationTimer();
12
13     CyclicAction(
14         xComplete=>xComplete,
15         eErrorID=>eErrorID
16     );
17
18     tcTimingController.CheckTiming(xTimeout=>xTimeout);
19 END_IF
20
21 IF xTimeout AND eErrorID = ERROR.NO_ERROR THEN
22     eErrorID := ERROR.TIME_OUT;
23 END_IF
24
25 IF eErrorID <> ERROR.NO_ERROR THEN
26     eState := STATE.ERROR;
27     XAgain := TRUE;
28 ELSEIF NOT xEnable THEN
29     eState := STATE.ABORTING;
30     XAgain := TRUE;
31 ELSEIF xComplete THEN
32     eState := STATE.DONE;
33     XAgain := TRUE;
34 END_IF
```

The Handler for the Aborting State

```
1  METHOD PRIVATE FINAL HandleAbortingState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  AbortAction(
10     xComplete=>xComplete,
11     eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15     eState := STATE.ERROR;
16     XAgain := TRUE;
17 ELSEIF xComplete THEN
18     eState := STATE.RESETTING;
19     XAgain := TRUE;
20 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1  METHOD PROTECTED CyclicAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6  VAR
7  xTimeout : BOOL;
8  xTimeLimit : BOOL;
9  END_VAR
10
11 IF xEnable THEN
12     (* Executing *)
13     // for every invocation,
14     // sample the input variables
15     tcTimingController.TimeLimit := udiTimeLimit;
16     tcTimingController.Timeout := udiTimeout;
17
18     REPEAT
19         // working to reach the ready condition
20         // => xComplete := TRUE
21         // if the maximum invocation time is reached
22         // => xTimeLimit := TRUE
23         // if the maximum operating time is reached
24         // => xTimeout := TRUE
25         // if an error condition is reached set
26         // eErrorID to a value other than ERROR.NO_ERROR
27         tcTimingController.CheckTiming(
28             xTimeout=>xTimeout,
29             xTimeLimit=>xTimeLimit
30         );
31
32         xComplete := TRUE;
33         eErrorID := ERROR.NO_ERROR;
34
35     UNTIL NOT xEnable OR xComplete OR
36           xTimeout OR xTimeLimit OR
37           eErrorID <> ERROR.NO_ERROR
38     END_REPEAT
39 END_IF
40
41 IF xTimeout AND eErrorID = ERROR.NO_ERROR THEN
42     eErrorID := ERROR.TIME_OUT;
43 END_IF
44
45 IF NOT xEnable OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN
46     (* Cleaning *)
47     // if possible free as much allocated resources
48     // as possible
49 END_IF
```

The Handler for the Done State

```
1  METHOD PRIVATE FINAL HandleDoneState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5
6  IF xDone AND NOT xEnable THEN
7      eState := STATE.RESETTING;
8      XAgain := TRUE;
9  ELSE
10     xBusy := FALSE;
11     xDone := TRUE;
12     XAgain := FALSE; (* !!! *)
13 END_IF
```

The Handler for the Error State

```
1  METHOD PRIVATE FINAL HandleErrorState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5
6  IF xError AND (xResetRequest OR NOT xEnable) THEN
7      eState := STATE.RESETTING;
8      XAgain := TRUE;
9  ELSE
10     xBusy := FALSE;
11     xError := TRUE;
12     xResetRequest := NOT xEnable;
13     XAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1  METHOD PRIVATE FINAL HandleResettingState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12     xBusy := FALSE;
13     xDone := FALSE;
14     xError := FALSE;
15     eErrorID := ERROR.NO_ERROR;
16     eState := STATE.DORMANT;
17     XAgain := xResetRequest; (* !!! *)
18     xResetRequest := FALSE;
19 END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

```
1  METHOD PROTECTED AbortAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6
7  // abort all running operations
8  // if an error condition is reached set
9  // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

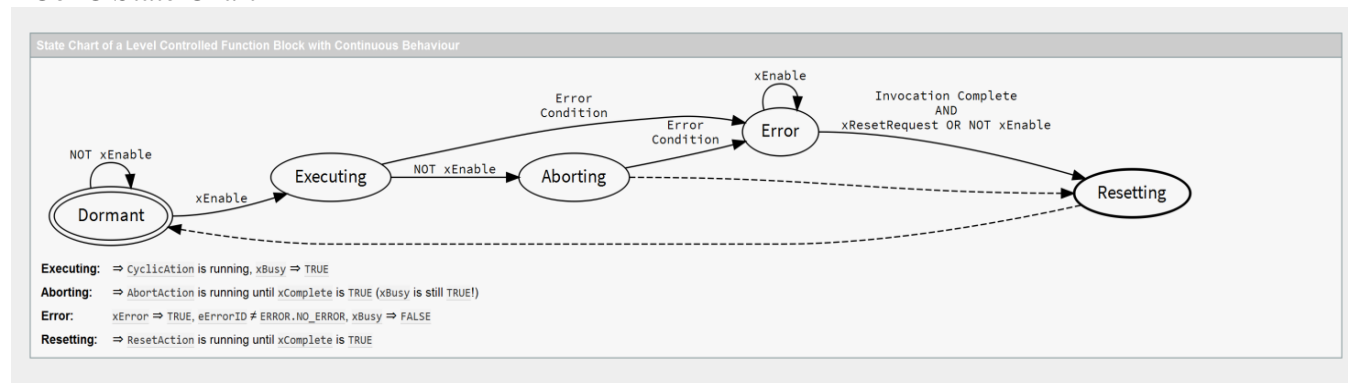
The Implementation of the Reset Action (Exemplary Implementation)

```
1  METHOD PROTECTED ResetAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  END_VAR
5
6  // free all allocated resources
7  // reinitialize instance variables
8
9  xComplete := TRUE;
```

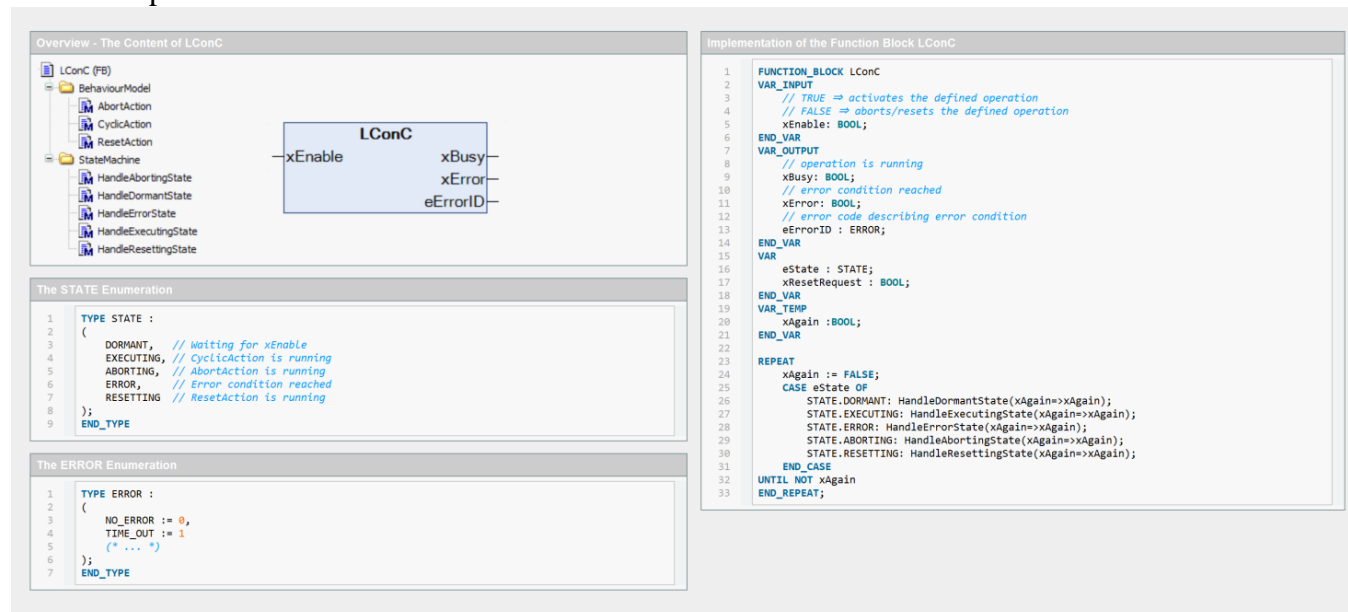
Appendix 1.4.5 LConC

LConC (Level Controlled | Not Time Limited | Continuous Behaviour)

LConC State Chart



LConC Implementation



The Handler for the Dormant State

```
1  METHOD PRIVATE FINAL HandleDormantState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5
6  IF xEnable THEN
7    xBusy := TRUE;
8    eState := STATE.EXECUTING;
9    XAgain := TRUE;
10 END_IF
```

The Handler for the Executing State

```
1  METHOD PRIVATE FINAL HandleExecutingState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5
6  IF xEnable THEN
7    CyclicAction(
8      eErrorID=>eErrorID
9    );
10 END_IF
11
12 IF eErrorID <> ERROR.NO_ERROR THEN
13   eState := STATE.ERROR;
14   XAgain := TRUE;
15 ELSEIF NOT xEnable THEN
16   eState := STATE.ABORTING;
17   XAgain := TRUE;
18 END_IF
```

The Handler for the Aborting State

```
1  METHOD PRIVATE FINAL HandleAbortingState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  AbortAction(
10   xComplete=>xComplete,
11   eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15   eState := STATE.ERROR;
16   XAgain := TRUE;
17 ELSEIF xComplete THEN
18   eState := STATE.RESETTING;
19   XAgain := TRUE;
20 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1  METHOD PROTECTED CyclicAction
2  VAR_OUTPUT
3  eErrorID : ERROR;
4  END_VAR
5
6  IF xEnable THEN
7    (* Executing *)
8    // for every invocation,
9    // sample the input variables
10
11    // if an error condition is reached set
12    // eErrorID to a value other than ERROR.NO_ERROR
13
14    eErrorID := ERROR.NO_ERROR;
15  END_IF
16
17 IF NOT xEnable OR eErrorID <> ERROR.NO_ERROR THEN
18   (* Cleaning *)
19   // if possible free as much allocated resources
20   // as possible
21 END_IF
```

The Handler for the Error State

```
1  METHOD PRIVATE FINAL HandleErrorState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5
6  IF xError AND (xResetRequest OR NOT xEnable) THEN
7    eState := STATE.RESETTING;
8    XAgain := TRUE;
9  ELSE
10   xBusy := FALSE;
11   xError := TRUE;
12   xResetRequest := NOT xEnable;
13   XAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1  METHOD PRIVATE FINAL HandleResettingState
2  VAR_OUTPUT
3  XAgain : BOOL;
4  END_VAR
5  VAR
6  xComplete : BOOL;
7  END_VAR
8
9  ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12   xBusy := FALSE;
13   xError := FALSE;
14   eErrorID := ERROR.NO_ERROR;
15   eState := STATE.DORMANT;
16   XAgain := xResetRequest; (* !!! *)
17   xResetRequest := FALSE;
18 END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

```
1  METHOD PROTECTED AbortAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  eErrorID : ERROR;
5  END_VAR
6
7  // abort all running operations
8  // if an error condition is reached set
9  // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

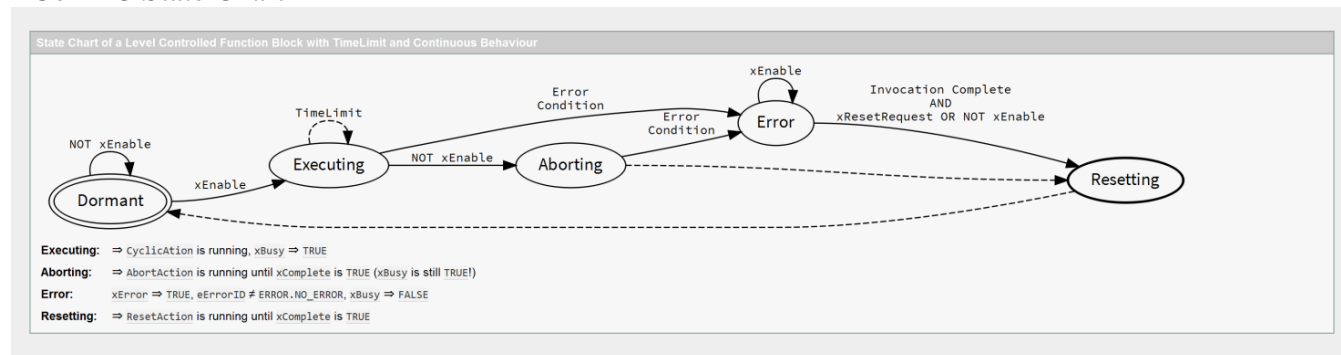
The Implementation of the Reset Action (Exemplary Implementation)

```
1  METHOD PROTECTED ResetAction
2  VAR_OUTPUT
3  xComplete : BOOL;
4  END_VAR
5
6  // free all allocated resources
7  // reinitialize instance variables
8
9  xComplete := TRUE;
```

Appendix 1.4.6 LConTIC

LConTIC (Level Controlled | Time Limited | Continuous Behaviour)

LConTIC State Chart



LConTIC Implementation

Overview - The Content of LConTIC

Implementation of the Function Block LConTIC

```

1 FUNCTION_BLOCK LConTIC
2 VAR_INPUT
3   // TRUE ⇒ activates the defined operation
4   // FALSE ⇒ aborts/resets the defined operation
5   xEnable: BOOL;
6   // max operating time per invocation
7   // [µs], 0 ⇒ no operating time limit
8   udiTimeLimit: UDINT;
9
10 END_VAR
11 VAR_OUTPUT
12   // operation is running
13   xBusy: BOOL;
14   // error condition reached
15   xError: BOOL;
16   // error code describing error condition
17   eErrorID: ERROR;
18
19 END_VAR
20 VAR
21   tcTimingController : TimingController;
22   eState : STATE;
23   xResetRequest : BOOL;
24
25 END_VAR
26
27 REPEAT
28   xAgain := FALSE;
29   CASE eState OF
30     STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
31     STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
32     STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
33     STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
34     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
35   END_CASE
36 UNTIL NOT xAgain
37 END_REPEAT;
  
```

The STATE Enumeration

```

1 TYPE STATE :
2 (
3   DORMANT, // Waiting for xEnable
4   EXECUTING, // CyclicAction is running
5   ABORTING, // AbortAction is running
6   ERROR, // Error condition reached
7   RESETTING // ResetAction is running
8 );
9 END_TYPE
  
```

The ERROR Enumeration

```

1 TYPE ERROR :
2 (
3   NO_ERROR := 0,
4   TIME_OUT := 1
5   (* ... *)
6 );
7 END_TYPE
  
```

The Handler for the Dormant State

```
1 METHOD PRIVATE FINAL HandleDormantState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xEnable THEN
7   xBusy := TRUE;
8   eState := STATE.EXECUTING;
9   xAgain := TRUE;
10 END_IF
```

The Handler for the Executing State

```
1 METHOD PRIVATE FINAL HandleExecutingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xEnable THEN
7   tcTimingController.StartInvocationTimer();
8   CyclicAction(
9     eErrorID=>eErrorID
10  );
11 END_IF
12
13 IF eErrorID <> ERROR.NO_ERROR THEN
14   eState := STATE.ERROR;
15   xAgain := TRUE;
16 ELSEIF NOT xEnable THEN
17   eState := STATE.ABORTING;
18   xAgain := TRUE;
19 END_IF
```

The Handler for the Aborting State

```
1 METHOD PRIVATE FINAL HandleAbortingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 AbortAction(
10  xComplete=>xComplete,
11  eErrorID=>eErrorID
12 );
13
14 IF eErrorID <> ERROR.NO_ERROR THEN
15   eState := STATE.ERROR;
16   xAgain := TRUE;
17 ELSEIF xComplete THEN
18   eState := STATE.RESETTING;
19   xAgain := TRUE;
20 END_IF
```

The Implementation of the Cyclic Action (Exemplary Implementation)

```
1 METHOD PROTECTED CyclicAction
2 VAR_OUTPUT
3   eErrorID : ERROR;
4 END_VAR
5 VAR
6   xTimeLimit : BOOL;
7 END_VAR
8
9 IF xEnable THEN
10  (* Executing *)
11  // for every invocation,
12  // sample the input variables
13  tcTimingController.TimeLimit := udiTimeLimit;
14
15  REPEAT
16    // if the maximum invocation time is reached
17    // => xTimeLimit := TRUE
18    // if an error condition is reached set
19    // eErrorID to a value other than ERROR.NO_ERROR
20    tcTimingController.CheckTiming(
21      xTimeLimit=>xTimeLimit
22    );
23    eErrorID := ERROR.NO_ERROR;
24
25    UNTIL NOT xEnable OR
26          xTimeLimit OR udiTimeLimit = 0 OR
27          eErrorID <> ERROR.NO_ERROR
28    END_REPEAT
29  END_IF
30
31 IF NOT xEnable OR eErrorID <> ERROR.NO_ERROR THEN
32  (* Cleaning *)
33  // if possible free as much allocated resources
34  // as possible
35 END_IF
```

The Handler for the Error State

```
1 METHOD PRIVATE FINAL HandleErrorState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5
6 IF xError AND (xResetRequest OR NOT xEnable) THEN
7   eState := STATE.RESETTING;
8   xAgain := TRUE;
9 ELSE
10  xBusy := FALSE;
11  xError := TRUE;
12  xResetRequest := NOT xEnable;
13  xAgain := FALSE; (* !!! *)
14 END_IF
```

The Handler for the Resetting State

```
1 METHOD PRIVATE FINAL HandleResettingState
2 VAR_OUTPUT
3   xAgain : BOOL;
4 END_VAR
5 VAR
6   xComplete : BOOL;
7 END_VAR
8
9 ResetAction(xComplete=>xComplete);
10
11 IF xComplete THEN
12   xBusy := FALSE;
13   xError := FALSE;
14   eErrorID := ERROR.NO_ERROR;
15   eState := STATE.DORMANT;
16   xAgain := xResetRequest; (* !!! *)
17   xResetRequest := FALSE;
18 END_IF
```

The Implementation of the Abort Action (Exemplary Implementation)

```
1 METHOD PROTECTED AbortAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4   eErrorID : ERROR;
5 END_VAR
6
7 // abort all running operations
8 // if an error condition is reached set
9 // eErrorID to a value other than ERROR.NO_ERROR
10
11 xComplete := TRUE;
12 eErrorID := ERROR.NO_ERROR;
```

The Implementation of the Reset Action (Exemplary Implementation)

```
1 METHOD PROTECTED ResetAction
2 VAR_OUTPUT
3   xComplete : BOOL;
4 END_VAR
5
6 // free all allocated resources
7 // reinitialize instance variables
8
9 xComplete := TRUE;
```

Appendix 2 Example without using Object Oriented features

It is not necessary to use object oriented features for creating function block libraries. With existing and classical environments this is very well possible and was always an approach in IEC 61131-3.

For this reason an example of how to do that is shown here.

If one can use the functionality of an edge triggered FB, one can define the basic ST code as listed in Chapter 5.1 The basic FB: ETrig, however without the object oriented features. This code could look like:

The ETrigATlTo Function Block coded according to IEC 61131-3 2nd Edition
(Exemplary Implementation)

```
FUNCTION_BLOCK ETrigATlTo
VAR_INPUT
    // Rising edge starts defined operation
    // FALSE ? resets the defined operation
    // after ready condition was reached
    xExecute: BOOL;
    // command for abort the operation
    xAbort: BOOL;
    // max operating time per invocation
    // [µs], 0 ? no operating time limit
    udiTimeLimit: UDINT;
    // max operating time per invocation
    // [µs], 0 ? no operating time limit
    udiTimeOut: UDINT;
END_VAR

VAR_OUTPUT
    // ready condition reached
    xDone: BOOL;
    // operation is running
    xBusy: BOOL;
    // error condition reached
    xError: BOOL;
    // abort condition reached
    xAborted : BOOL;
    // error code describing error condition
    eErrorID : ERROR;
END_VAR

VAR
    tcTimingController : TimingController;
    eState : STATE := STATE.DORMANT;
    xFirstInvocation : BOOL := TRUE;
    xAbortProposed : BOOL;
    eErrorIDProposed : ERROR;
    xResetRequest : BOOL;
END_VAR

VAR_TEMP
    xAgain :BOOL;
    xComplete : BOOL;
    xTimeLimit : BOOL;
    xTimeOut : BOOL;
```

```
xLocalAbort : BOOL;
eLocalErrorID : ERROR;
END_VAR

REPEAT
  xAgain := FALSE;
  CASE eState OF
    STATE.DORMANT:
      IF xExecute THEN
        tcTimingController(xStartOperationTimer:=TRUE);
        tcTimingController.xStartOperationTimer := FALSE;
        xBusy := TRUE;
        eState := STATE.STARTING;
        xAgain := TRUE;
      END_IF

    STATE.STARTING:
      IF NOT xAbort THEN (* StartAction *)
        IF xFirstInvocation THEN
          // sample the input variables
          tcTimingController.udiTimeLimit := udiTimeLimit;
          tcTimingController.udiTimeOut := udiTimeOut;
          xFirstInvocation := FALSE;
        END_IF

        // working to reach the locale ready condition
        // ? xComplete := TRUE
        // if an error condition is reached
        // ? set eLocalErrorID to a value other than
        // ERROR.NO_ERROR

        xComplete := TRUE;
        eLocalErrorID := ERROR.NO_ERROR;

      ELSE
        xAbortProposed := TRUE;
      END_IF

      tcTimingController(xTimeOut=>xTimeOut);

      IF xTimeOut AND eLocalErrorID = ERROR.NO_ERROR THEN
        eLocalErrorID := ERROR.TIME_OUT;
      END_IF

      IF eLocalErrorID <> ERROR.NO_ERROR OR xAbortProposed THEN
        eState := STATE.CLEANING;
        xAgain := TRUE;
      ELSIF xComplete THEN
        eState := STATE.EXECUTING;
        xAgain := TRUE;
      END_IF

    STATE.EXECUTING:
      IF NOT (xAbort OR xAbortProposed) THEN
        tcTimingController(xStartInvocationTimer:=TRUE);
        tcTimingController.xStartInvocationTimer := FALSE;
```

```
REPEAT (* CyclicAction *)
    // working to reach the ready condition
    // ? xComplete := TRUE
    // if the maximum invocation time is reached
    // ? xTimeLimit := TRUE
    // if the maximum operating time is reached
    // ? xTimeOut := TRUE
    // if an error condition is reached
    // ? set eLocalErrorID to a value other than
    // ERROR.NO_ERROR

    tcTimingController(
        xTimeOut=>xTimeOut,
        xTimeLimit=>xTimeLimit
    );

    xComplete := TRUE;
    eLocalErrorID := ERROR.NO_ERROR;

    UNTIL xComplete OR
        xTimeOut OR xTimeLimit OR
        eLocalErrorID <> ERROR.NO_ERROR
    END_REPEAT

ELSE
    xAbortProposed := TRUE;
END_IF

tcTimingController(xTimeOut=>xTimeOut);
IF xTimeOut AND eLocalErrorID = ERROR.NO_ERROR THEN
    eLocalErrorID := ERROR.TIME_OUT;
END_IF

IF xComplete OR eLocalErrorID <> ERROR.NO_ERROR OR
    xAbortProposed THEN
    eErrorIDProposed := eLocalErrorID;
    eState := STATE.CLEANING;
    xAgain := TRUE;
END_IF

STATE.CLEANING: (* CleanupAction *)
    IF xAbortProposed THEN
        // abort all running operations
        // if an error condition is reached
        // ? set eErrorID to a value other than ERROR.NO_ERROR
        xLocalAbort := xAbortProposed;
    END_IF

    // if possible free as much allocated resources
    // as possible
    // working to reach the locale ready condition
    // ? xComplete := TRUE
    // if an error condition is reached
    // ? set eLocalErrorID to a value other than ERROR.NO_ERROR
    xComplete := TRUE;
```

```
eLocalErrorID := eErrorIDProposed;

IF xAbortProposed THEN
    xComplete := FALSE;
ELSE
    xLocalAbort := FALSE;
END_IF

IF eErrorIDProposed <> ERROR.NO_ERROR THEN
    xComplete := FALSE;
    xAbort := FALSE;
END_IF

IF eLocalErrorID <> ERROR.NO_ERROR THEN
    eErrorIDProposed := eLocalErrorID;
END_IF

IF eLocalErrorID <> ERROR.NO_ERROR THEN
    eState := STATE.ERROR;
    xAgain := TRUE;
ELSIF xLocalAbort THEN
    eState := STATE.ABORTED;
    xAgain := TRUE;
ELSIF xComplete THEN
    eState := STATE.DONE;
    xAgain := TRUE;
END_IF

STATE.DONE:
    IF xDone AND (xResetRequest OR NOT xExecute) THEN
        eState := STATE.RESETTING;
        xAgain := TRUE;
    ELSE
        xBusy := FALSE;
        xDone := TRUE;
        xResetRequest := NOT xExecute;
        xAgain := FALSE; (* !!! *)
    END_IF

STATE.ERROR:
    IF xError AND (xResetRequest OR NOT xExecute) THEN
        eState := STATE.RESETTING;
        xAgain := TRUE;
    ELSE
        xBusy := FALSE;
        xError := TRUE;
        eErrorID := eErrorIDProposed;
        xResetRequest := NOT xExecute;
        xAgain := FALSE; (* !!! *)
    END_IF

STATE.ABORTED:
    IF xAborted AND (xResetRequest OR NOT xExecute) THEN
        eState := STATE.RESETTING;
        xAgain := TRUE;
    ELSE
```

```
        xBusy := FALSE;
        xAborted := TRUE;
        xResetRequest := NOT xExecute;
        xAgain := FALSE; (* !!! *)
    END_IF

STATE.RESETTING: (* ResetAction *)
    // free all residual allocated resources
    // reinitialize instance variables
    // working to reach the locale ready condition
    // ? xComplete := TRUE
    xComplete := TRUE;

    IF xComplete THEN
        xBusy := FALSE;
        xDone := FALSE;
        xError := FALSE;
        xAborted := FALSE;
        xAbortProposed := FALSE;
        eErrorIDProposed := ERROR.NO_ERROR;
        eErrorID := ERROR.NO_ERROR;
        eState := STATE.DORMANT;
        xAgain := xResetRequest; (* !!! *)
        xResetRequest := FALSE;
        xFirstInvocation := TRUE;
    END_IF
END_CASE
UNTIL NOT xAgain
END_REPEAT;
```

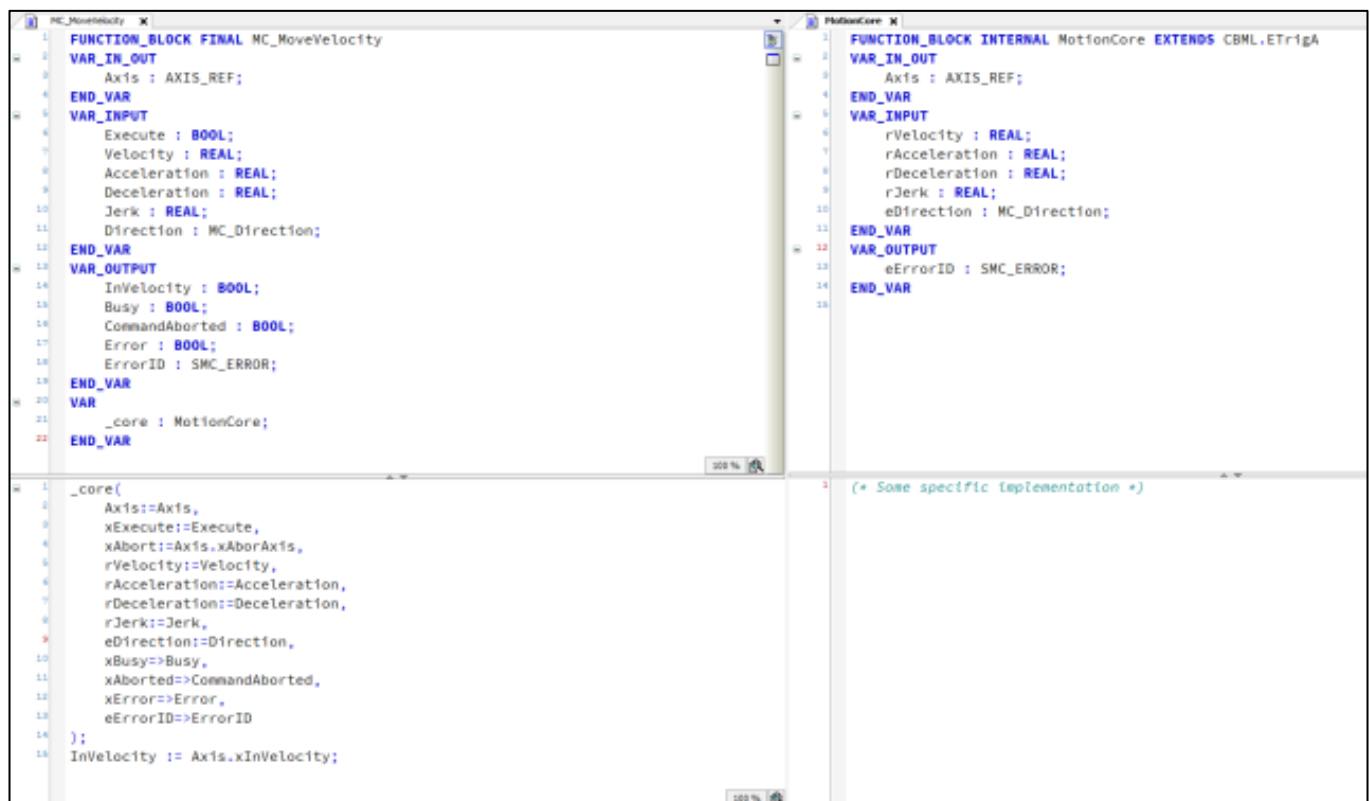

Appendix 3 Example of an intermediate interface

In practice there are more than one naming convention standards. This can result in different layers of naming conventions: for the creation of the function block libraries, for the creation of the functional application, and for the mapping at the application level to the conventions of the user.

These different levels can cooperate by encapsulating the functionalities on the different levels, or by using aliases, where one maps one name to another.

One can call these intermediate interfaces, like an interface between the function block library and the application itself, both using different areas and so using different naming conventions while the application is building on top of the library. The different naming conventions can be linked (“alias”).

One example is shown in the picture below. The public final function block MC_MoveVelocity on the left is based on the internal FB MotionCore on the right. By creating these two function blocks it is possible to encapsulate the complete implementation details. Using this technique the layout of the public function block is completely decoupled from the layout of the internal function block (see the different naming conventions for variable names). Because of the FINAL keyword it is not possible to extend this function block in another context. So no problems can occur after changing some implementation details (for example adding some local variables). Because of the INTERNAL keyword nobody can use a function block marked in this way outside of its defined context, its original library. This decouples these two layers.



Appendix 4 Behaviour of inputs and outputs in PLCopen Motion Control FBs

For the relevant inputs and outputs

Input parameters	<p><i>With 'Execute' without 'ContinuousUpdate'</i>: The parameters are used with the rising edge of the 'Execute' input. To modify any parameter it is necessary to change the input parameter(s) and to trigger the 'Execute' input again.</p> <p><i>With 'Execute' combined with 'ContinuousUpdate'</i>: The parameters are used with the rising edge of the 'Execute' input. The parameters can be modified continuously as long as the 'ContinuousUpdate' is SET.</p> <p><i>With 'Enable'</i>: The parameters are used with the rising edge of the enable input and can be modified continuously.</p>
Inputs exceeding application limits	If a FB is commanded with parameters which result in a violation of application limits, the inputs are limited by the system or the instance of the FB generates an error. The consequences of this error for the axis are application specific and thus should be handled by the application program.
Missing input parameters	According to IEC 61131-3, if any parameter of a function block input is missing ("open") then the value from the previous invocation of this instance will be used. In the first invocation the initial value is applied.
Acceleration, Deceleration and Jerk inputs	If the input 'Deceleration', 'Acceleration' or 'Jerk' is set to 0, the result is implementation dependent. There are several implementations possible, like one goes to the error state, one signals a warning (via a supplier specific output), one inhibits this in the editor, one takes the value as either specified in AxisRef or in the drive itself, or one takes a maximum value. Even if the 0 value input is accepted by the system, please use with caution especially if compatibility is targeted.
Output exclusivity	<p><i>With 'Execute'</i>: The outputs 'Busy', 'Done', 'Error', and 'CommandAborted' are mutually exclusive: only one of them can be TRUE on one FB. If 'Execute' is TRUE, one of these outputs has to be TRUE.</p> <p>Only one of the outputs 'Active', 'Error', 'Done' and 'CommandAborted' is set at the same time, except in MC_Stop where 'Active' and 'Done' can be set both at the same time</p> <p><i>With 'Enable'</i>: The outputs 'Valid' and 'Error' are mutually exclusive: only one of them can be TRUE on one FB.</p>
Output status	<p><i>With 'Execute'</i>: The 'Done', 'Error', 'ErrorID' and 'CommandAborted' outputs are reset with the falling edge of 'Execute'. However the falling edge of 'Execute' does not stop or even influence the execution of the actual FB. It must be guaranteed that the corresponding outputs are set for at least one cycle if the situation occurs, even if execute was reset before the FB completed.</p> <p>If an instance of a FB receives a new execute before it finished (as a series of commands on the same instance), the FB won't return any feedback, like 'Done' or 'CommandAborted', for the previous action.</p> <p><i>With 'Enable'</i>: The 'Valid', 'Enabled', 'Busy', 'Error', and 'ErrorID' outputs are reset with the falling edge of 'Enable' as soon as possible.</p>

Behavior of Done output	<p>The 'Done' output is set when the commanded action has been completed successfully.</p> <p>With multiple Function Blocks working on the same axis in a sequence, the following applies: when one movement on an axis is interrupted with another movement on the same axis without having reached the final goal, 'Done' of the first FB will not be set.</p>
Behavior of Busy output	<p><i>With 'Execute':</i> Every FB can have an output 'Busy', reflecting that the FB is not finished and new output values can be expected. 'Busy' is SET at the rising edge of 'Execute' and RESET when one of the outputs 'Done', 'Aborted', or 'Error' is set.</p> <p><i>With 'Enable':</i> Every FB can have an output 'Busy', reflecting that the FB is working and new output values can be expected. 'Busy' is SET at the rising edge of 'Enable' and stays SET as long as the FB is performing any action.</p> <p>It is recommended that the FB should be kept in the active loop of the application program for at least as long as 'Busy' is true, because the outputs may still change.</p>
Behavior of InVelocity, InGear, InTorque and InSync	<p>The outputs 'InVelocity', 'InGear', 'InTorque', and 'InSync' (from now on referred to as 'Inxxx') have a different behavior than the 'Done' output. As long as the FB is Active, 'Inxxx' is SET when the set value equals the commanded value, and will be RESET when at a later time they are unequal. For example, the InVelocity output is SET when the set velocity is equal to the commanded velocity. This is similar for 'InGear', 'InTorque', and 'InSync' outputs in the applicable FBs.</p> <p>'Inxxx' is updated even if 'Execute' is low as long as the FB has control of the axis ('Active' and 'Busy' are SET).</p> <p>The behavior of 'Inxxx' directly after 'Execute' is SET again while the condition of 'Inxxx' is already met, is implementation specific.</p> <p>'Inxxx' definition does not refer to the actual axis value, but must refer to the internal instantaneous setpoint.</p>
Output 'Active'	<p>The 'Active' output is required on buffered Function Blocks. This output is set at the moment the function block takes control of the motion of the according axis. For un-buffered mode the outputs 'Active' and 'Busy' can have the same value.</p> <p>For one axis, several Function Blocks might be busy, but only one can be active at a time. Exceptions are FBs that are intended to work in parallel, like MC_MoveSuperimposed and MC_Phasing's, where more than one FB related to one axis can be active.</p>
Behavior of CommandAborted output	<p>'CommandAborted' is set, when a commanded motion is interrupted by another motion command.</p> <p>The reset-behavior of 'CommandAborted' is like that of 'Done'. When 'CommandAborted' occurs, the other output-signals such as 'InVelocity' are reset.</p>

Enable and Valid	<p>The 'Enable' input is coupled to a 'Valid' output. 'Enable' is level sensitive and 'Valid' shows that a valid set of outputs is available at the FB.</p> <p>The 'Valid' output is TRUE as long as a valid output value is available and the 'Enable' input is TRUE. The relevant output value can be refreshed as long as the input 'Enable' is TRUE.</p> <p>If there is a FB error, the output is not valid ('Valid' set to FALSE). When the error condition disappears, the values will reappear and 'Valid' output will be set again.</p>
Position versus distance	<p>'Position' is a value defined within a coordinate system. 'Distance' is a relative measure related to technical units. 'Distance' is the difference between two positions.</p>
Sign rules	<p>The 'Acceleration', 'Deceleration' and 'Jerk' are always positive values. 'Velocity', 'Position' and 'Distance' can be both positive and negative.</p>
Error Handling Behavior	<p>All blocks can have two outputs, which deal with errors that can occur while executing that Function Block. These outputs are defined as follow:</p> <p>Error Rising edge of 'Error' informs that an error occurred during the execution of the Function Block.</p> <p>ErrorID Error identification (Extended parameter)</p> <p>'Done', 'InVelocity', 'InGear', 'InTorque', and 'InSync' mean successful completion so these signals are logically exclusive to 'Error'.</p> <p>Types of errors:</p> <ul style="list-style-type: none"> • Function Blocks (e.g. parameters out of range, state machine violation attempted) • Communication • Drive <p>Instance errors do not always result in an axis error (bringing the axis to 'ErrorStop')</p> <p>The error outputs of the relevant FB are reset with falling edge of 'Execute' and 'Enable'. The error outputs at FBs with 'Enable' can be reset during operation (without a reset of 'Enable').</p>
FB Naming	<p>In case of multiple libraries within one system (to support multiple drive / motion control systems), the FB naming may be changed to "MC_FBname_SupplierID".</p>
Naming conventions ENUM types	<p>Due to the naming constraints in the IEC standard on the uniqueness of variable names, the 'mc' reference to the PLCopen Motion Control namespace is used for the ENUMs.</p> <p>In this way we avoid the conflict that using the ENUM types 'positive' and 'negative' for instance with variables with these names throughout the rest of the project since they are called mcPositive and mcNegative respectively.</p>

Table 1: General Rules

The behavior of the 'Execute' / 'Done' style FBs is as follows:

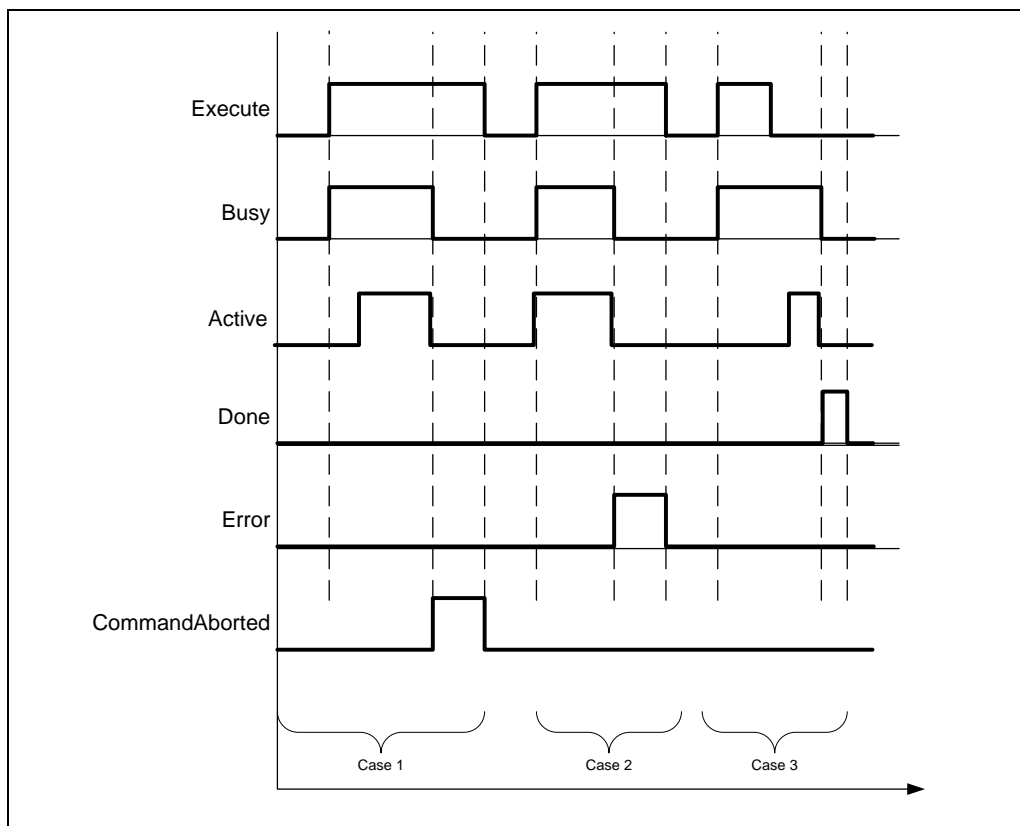


Figure 16: The behavior of the 'Execute' / 'Done' in relevant FBs

The behaviour of the 'Execute' / 'Inxx' style FBs is as follows:

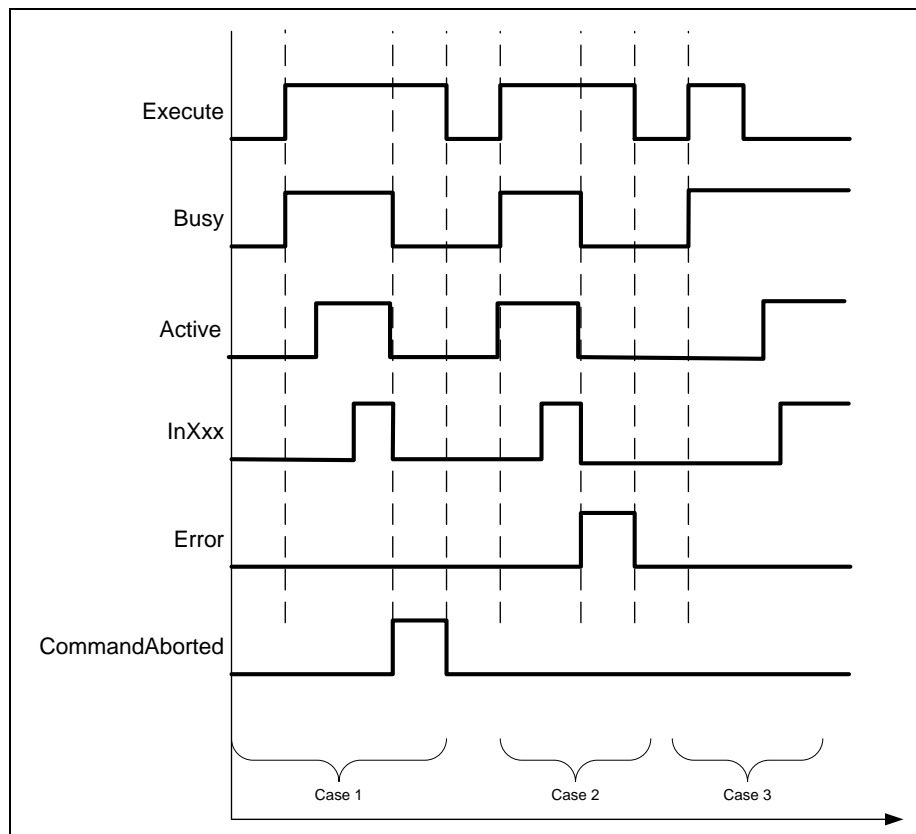


Figure 17: The behavior of the 'Execute' / 'Inxx' in relevant FBs