

Техническая документация PLCopen

рабочая группа

Создание руководств по разработке ПО

представляет

Руководство по созданию библиотек

версия 1.0, официальный релиз

Отказ от ответственности

Название 'PLCopen[®]' является зарегистрированным товарным знаком и совместно с логотипом PLCopen является собственностью ассоциации PLCopen.

Данный документ предоставляется "как есть" и в будущем может быть подвергнут изменениям и исправлениям. PLCopen не предоставляет никакие гарантии (явные или подразумеваемые), включая любые гарантии по поводу пригодности использования документа для конкретной цели. Ни при каких обстоятельствах PLCopen не несет ответственности за ущерб или убытки, вызванные ошибками в данном документе или его использованием.

Copyright © 2017 by PLCopen. Все права защищены

Дата публикации: 04.05.2017

Переводчик:	Е.Кислов
Редактор:	А.Осинский
Версия перевода:	1.0
Дата публикации:	14.06.2018

Оглавление

Оглавление.....	2
Список авторов и история версий.....	4
Глоссарий	5
1. Введение	6
1.1. Правила именования переменных в этом документе	6
1.2. Примечания о примерах и EN/ENO.....	7
2. Обзор спецификаций PLCopen	8
2.1. Основная информация.....	8
2.2. Типы функциональных блоков.....	8
2.3. Управление движением	9
2.4. Промышленная безопасность.....	12
2.5. Коммуникация систем управления.....	12
2.6. Заключение	13
3. Основная информация о модели поведения ФБ PLCopen	14
3.1. Основная информация.....	14
3.2. Принцип работы ФБ типа Execute	14
3.3. Принцип работы ФБ типа Enable	16
3.4. Общие свойства ФБ модели PLCopen	17
3.5. Обработка ошибок	18
3.6. Имена шагов	20
3.7. Совместное использование ФБ.....	21
4. Введение в объектно-ориентированные расширения МЭК 61131-3.....	24
5. Описание ФБ типа Execute	25
5.1. Базовый ФБ: ETrig	25
5.2. Добавляем возможность прерывания операции (ETrigA)	30
5.3. Добавляем таймеры.....	34
5.4. Пример реализации ФБ ETrigATiTo на языке ST.....	36
6. Описание ФБ типа Enable	37
6.1. Базовый ФБ: LCon	37
6.2. Пример реализации ФБ LCon на языке ST с использованием ООП	38
6.3. Добавляем таймеры.....	43
Приложение 1. Спецификации ФБ типов Execute и Enable.....	46

Приложение 1.1. Основная информация о примерах	46
Приложение 1.2. Обзор входов/выходов	47
Приложение 1.3. Обзор ФБ типа Execute	48
Приложение 1.3.1. ФБ ETrig	49
Приложение 1.3.2. ФБ ETrigTI	52
Приложение 1.3.3. ФБ ETrigTo	55
Приложение 1.3.4. ФБ ETrigTITo	58
Приложение 1.3.5. ФБ ETrigA	61
Приложение 1.3.6. ФБ ETrigATI	64
Приложение 1.3.7. ФБ ETrigATo	67
Приложение 1.3.8. ФБ ETrigATITo	70
Приложение 1.4. Обзор ФБ типа Enable	73
Приложение 1.4.1. ФБ LCon	74
Приложение 1.4.2. ФБ LConTI	76
Приложение 1.4.3. ФБ LConTo	78
Приложение 1.4.4. ФБ LConTITo	80
Приложение 1.4.5. ФБ LConC	82
Приложение 1.4.6. ФБ LConTIC	84
Приложение 2. Пример реализации ФБ модели PLCopen без ООП	86
Приложение 3. Пример промежуточного интерфейса	90
Приложение 4. Модель поведения ФБ спецификации PLCopen Motion Control.....	91

Список авторов и история версий

Данный документ является официальным документом организации **PLCopen**:

Руководство по созданию библиотек

Он представляет собой результат работы комитета **Разработка руководств по созданию библиотек PLCopen** рабочей группы **Создание руководств по разработке ПО** и включает вклад каждого из участников:

Участник	Компания
Peter Erning	ABB
Andrew Hollom	ABB
Bert van der Linden	ATS International
Roland Wagner	B&R Automation
Bernhard Werner	3S / Codesys
Wolfgang Doll	3S / Codesys
Rolf Hänisch	Fraunhofer FOKUS
Wolfgang Zeller	HS Augsburg
Denis Chalon	Itrix
Geert Vanstraelen	Macq
Barry Butcher	Omron
Hiroshi Yoshida	Omron
Andreas Weichelt	Phoenix Contact
Kevin Hull	Yaskawa
Eelco van der Wal	PLCopen

История версий

Версия	Дата	Описание
0.1	21.10.2015	Первая версия на базе нескольких встреч и видеоконференций.
0.2	23.10.2015	Дополнения по результатам видеоконференции с 3S.
0.3	28.10.2015	Дополнения по результатам видеоконференции.
0.4	11.11.2015	Дополнения по результатам видеоконференции.
0.5	15.06.2016	Дополнения по результатам видеоконференции и обратной связи.
0.6	30.06.2016	Дополнения по результатам видеоконференции.
0.61	15.07.2016	Небольшие изменения в структуре документа.
0.7	15.08.2016	Дополнения по результатам видеоконференции и обратной связи.
0.8	25.08.2016	Дополнения по результатам видеоконференции и обратной связи.
0.99	30.09.2016	Дополнения по результатам видеоконференции.
0.99A	14.03.2017	Дополнения по результатам видеоконференции и обратной связи.
1.0	04.05.2017	Официальный релиз с публикацией исходных кодов.

Глоссарий

SFC (*Sequential Function Chart, язык последовательных схем*) – высокоуровневый графический язык программирования ПЛК, представляющий приложение в виде конечного автомата;

ST (*Structured Text, структурированный текст*) – высокоуровневый текстовый язык программирования ПЛК, напоминающий Паскаль;

Модель поведения – спецификация интерфейса ФБ, которая описывает набор основных входов и выходов, а также их взаимное влияние друг на друга. Модель поведения не специфицирует входы и выходы, связанные с конкретной задачей, делая акцент на универсальном интерфейсе, который может быть адаптирован под потребности пользователя;

ООП – объектно-ориентированный подход;

Подтверждение (*стандартное и быстрое*) – условие сброса ФБ. При быстром подтверждении сброс блока происходит после окончания выполнения операции или ее прерывания (к этому моменту вход **xExecute/xEnable** уже имеет значение FALSE). При стандартном подтверждении сброс происходит по заднему фронту на входе **xExecute/xEnable** после окончания выполнения операции;

Тип ФБ – в данном документе под типом ФБ подразумевается один из наборов требований, описанных в модели поведения;

ФБ – функциональный блок.

1. Введение

Одним из важнейших побочных эффектов разработки спецификаций [PLCopen](#) для управления движением (*Motion Control*), промышленной безопасности (*Safety*) и коммуникации систем управления (*Communication*) является создание универсальной модели поведения функциональных блоков. Модель позволяет создать ФБ с простым и понятным интерфейсом, который легко может быть адаптирован под конкретные задачи. Модель определяет две категории ФБ:

- Выполняемые по уровню (**Enable**);
- Выполняемые по фронту (**Execute**).

В настоящий момент ключевыми задачами являются развитие модели для возможности последующего применения в любых приложениях и помощь пользователям в создании библиотек, соответствующих модели. По этой причине один из комитетов рабочей группы **Создание руководств по разработке ПО** приступил к написанию данного руководства.

Следуя основным принципам приведенной в документе спецификации можно создавать ФБ с удобным интерфейсом, предсказуемым поведением, высокой надежностью и связностью кода. Наличие битов и кодов ошибок (**Error** и **ErrorID**) упрощает их обработку в программе. Всё это повышает переносимость кода приложения и облегчает его отладку.

1.1. Правила именования переменных в этом документе

В соответствии с [Руководством по кодированию](#) рекомендуется использовать префиксы в именах переменных. Префиксы позволяют закодировать тип переменной: например, по именам **xError**, **eError** и **iError** можно понять, что соответствующие переменные имеют тип BOOL, ENUM и INT. Определить тип переменных с названиями **Error** и **ErrorID** не представляется возможным. В данном документе имена с префиксами используются в исходном коде примеров, а имена без префиксов могут быть использованы в тексте.

Также в примерах кода используются следующие регистры текста:

- UPPER_SNAKE_CASE для констант, пользовательских типов данных и ключевых слов (таких, как BOOL, FOR и т.д.);
- lowerCamelCase для имен переменных, содержащих префиксы;
- UpperCamelCase для всех остальных составных названий.

Для обозначения названий входов/выходов используется **полуужирный текст**, для названий шагов и методов – *выделенный курсивом*.

1.2. Примечания о примерах и EN/ENO

Любые фрагменты кода, приведенные в данном документе, являются лишь примерами. Они не проверены на практике и могут содержать ошибки. Таким образом, эти примеры можно использовать для ознакомления с описываемыми в документе концепциями, но не следует применять их в конкретном приложении. Пользователю необходимо создать свою реализацию, соответствующую описанным интерфейсам ФБ и диаграммам состояний.

Входы **Enable/Execute** перекрывают функционал дополнительного входа EN, определенного в стандарте МЭК 61131-3 (см. п. 6.6.1.5 в 3-й редакции стандарта).

2. Обзор спецификаций PLCopen

2.1. Основная информация

В этом разделе приводится обзор моделей ФБ из различных спецификаций [PLCopen](#):

- Управление движением (*Motion Control*);
- Промышленная безопасность (*Safety*);
- Коммуникация систем управления (*Communication*).

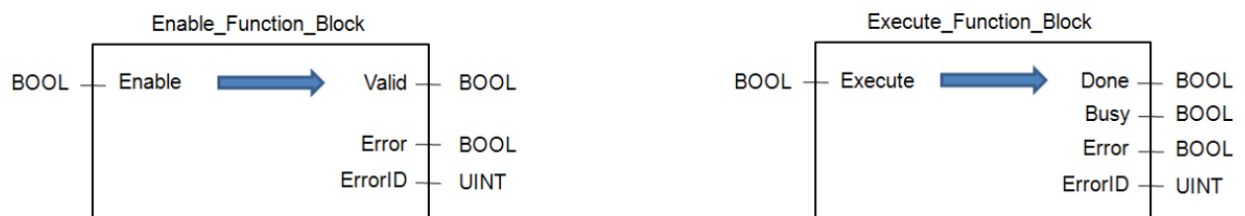
В рамках раздела приводится лишь краткое описание моделей; более подробная информация доступна в исходных текстах спецификаций. Информация об унифицированной модели ФБ приведена в главах [5](#) и [6](#).

2.2. Типы функциональных блоков

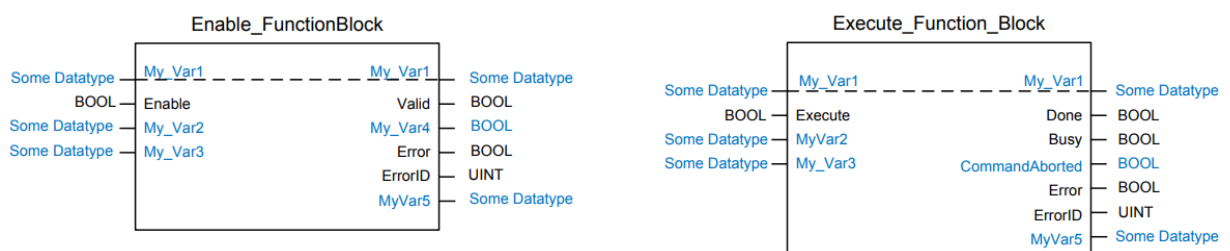
В рамках данного документа рассматриваются два основных типа функциональных блоков:

- Выполняемые по уровню (**Enable**);
- Выполняемые по фронту (**Execute**).

На рисунке ниже приведено графическое представление обоих типов с минимально необходимым числом входов и выходов. **Обратите внимание**, что входу **Execute** всегда соответствует выход **Done**, а входу **Enable** – выход **Valid**¹. Это упрощает использование ФБ на графических языках программирования (LD, FBD).



В зависимости от конкретной задачи число входов и выходов ФБ может быть расширено. Более подробная информация о доступных входах/выходах приведена в следующих главах. На рисунке ниже эти входы и выходы выделены голубым цветом.



¹ Это справедливо для ФБ из спецификаций управления движением. В рамках данного документа соответствующий выход имеет название Busy (прим. пер.)

2.3. Управление движением

Организация [PLCopen](#) разработала набор спецификаций для решения задач управления движением (*Motion Control*). В рамках этих спецификаций использовалась общая модель функциональных блоков, основные сведения о которой приведены в этой главе. Более подробная информация доступна в [Приложении 4](#).

Интерфейс ФБ управления движением можно разделить на две концептуальные части: вход активации и статусные выходы.

Существует два варианта активации ФБ:

1. По фронту (через вход **Execute**). В этом случае происходит однократное выполнение ФБ, которое в случае отсутствия ошибок завершается сигналом на выходе **Done**. Вместо **Done** ФБ может иметь выход **InXxx (InVelocity, InGear)** – он принимает значение TRUE, пока выполняется заданное действие (перемещение, вращение и т.д.).
2. По уровню (через вход **Enable**). В этом случае активность блока характеризует значение TRUE на выходе **Valid**.

С релизом версии 2.0 для первой части спецификации также был добавлен вход **Continuous Update**, который объединяет функционал обоих вариантов активации:

- Если этот вход имеет значение TRUE, то по переднему фронту на входе **Execute** начинается выполнение ФБ и при этом чтение значений аргументов ФБ происходит в каждом цикле (т.е. параметры блока могут меняться динамически в процессе выполнения);
- Если этот вход имеет значение FALSE, то по переднему фронту на входе **Execute** начинается выполнение ФБ со значениями аргументов, считанными в момент запуска (т.е. применение новых значений аргументов произойдет только при следующем запуске ФБ).

В спецификации определены следующие статусные выходы ФБ:

- **Busy** – блок находится в процессе работы, значения выходов еще не достоверны;
- **Active** – блок управляет движением по выбранной оси;
- **CommandAborted** – работа блока была прервана сигналом на входе **Abort**;
- **Error** – в процессе работы блока произошла ошибка;
- **ErrorID** – код возникшей ошибки.

Пример ФБ управления движением

В качестве примера ФБ управления движением приведем графическое представление и циклограмму функционального блока **MC_MoveAbsolute**.

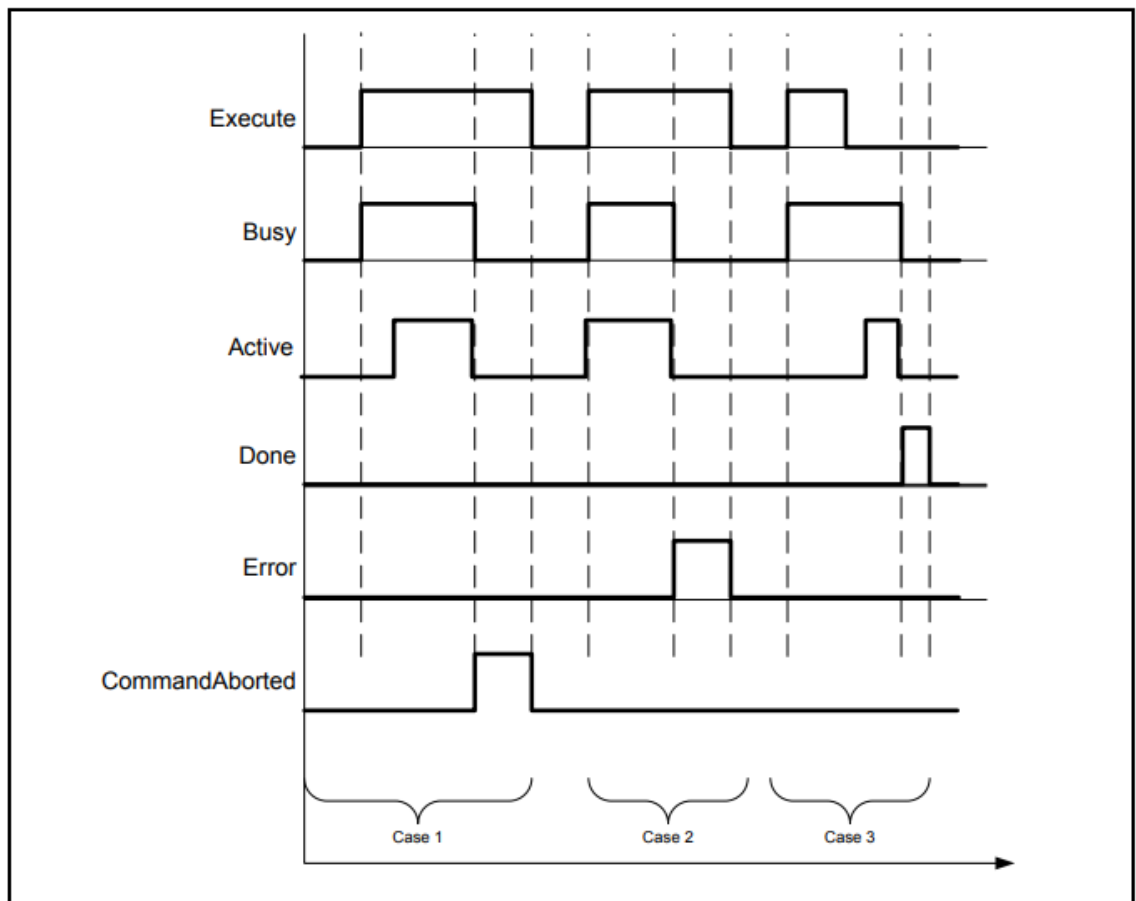
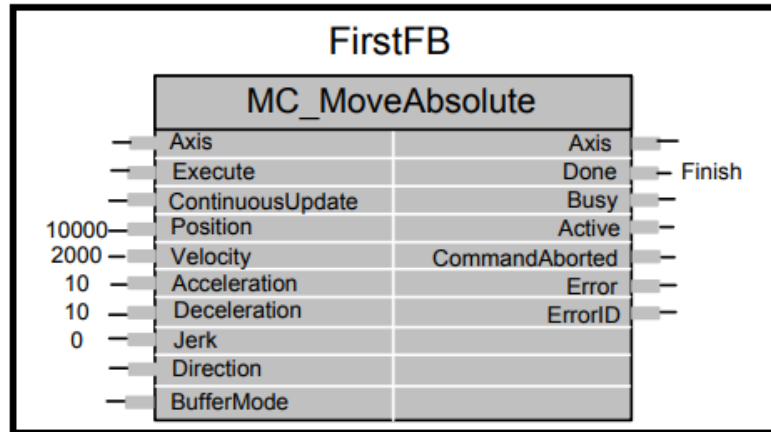


Рис. 1. Циклограмма ФБ модели Execute/Done

Пример циклограммы ФБ с выходом InXxx (InVelocity, InGear и т.п.):

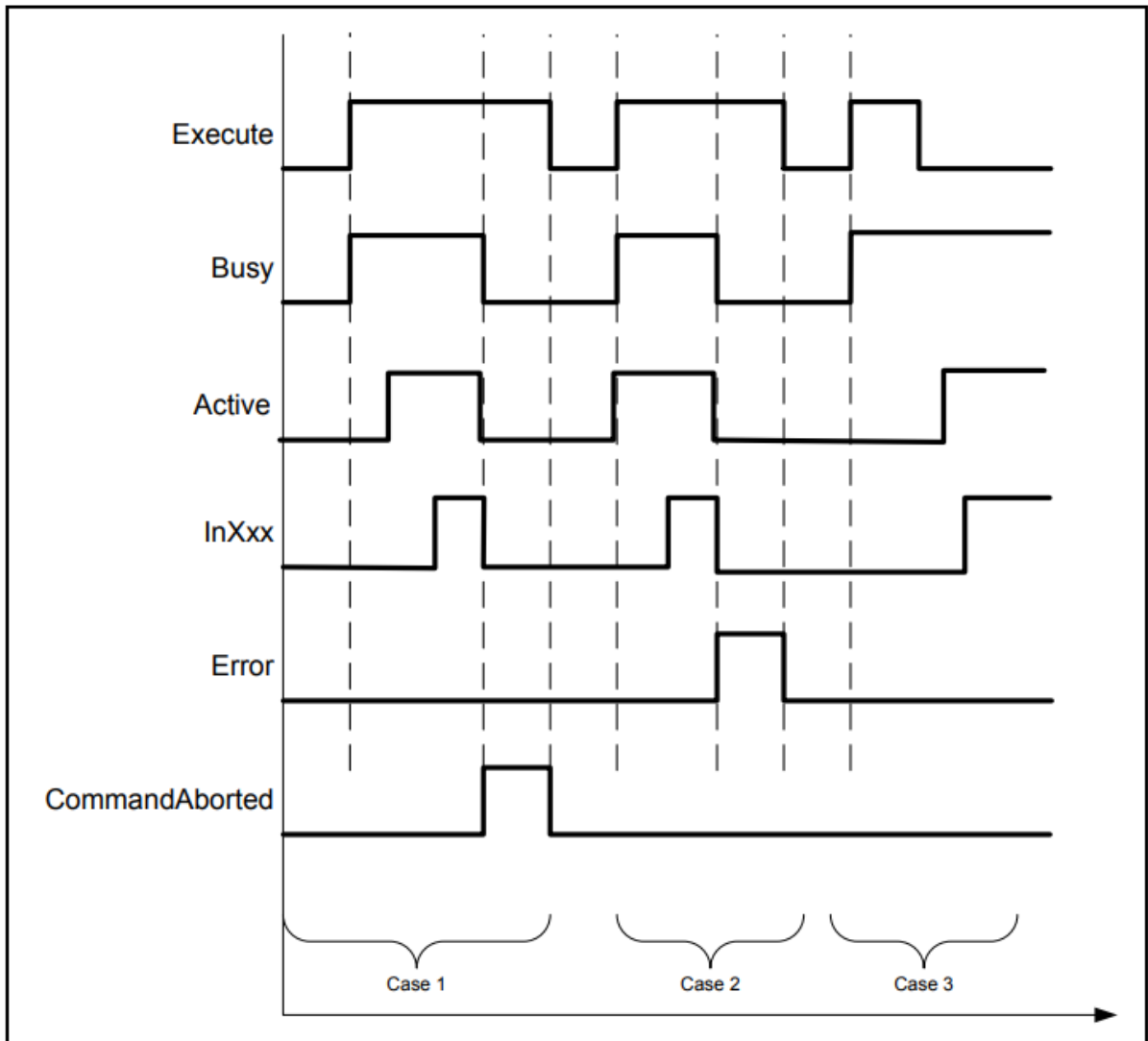
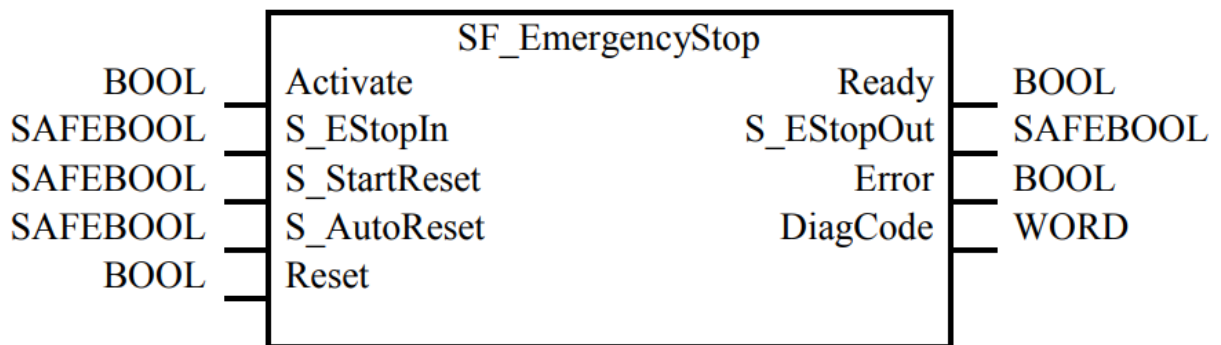


Рис. 2. Циклограмма ФБ модели Execute/InXxx

2.4. Промышленная безопасность

Организация [PLCopen](#) разработала набор спецификаций для создания безопасного (*safety*) промышленного ПО. При работе над спецификациями было решено использовать в них модели ФБ, максимально близкие к моделям, применявшимся в [спецификациях по управлению движением](#). Из-за особенностей предметной области число используемых типов данных и выполняемых блоками функций сведены до минимально необходимого. Также спецификация вводит понятие безопасных (SAFE) типов данных. Из-за этого набор доступных входов и выходов ФБ меньше, чем в спецификациях по управлению движением. Существует также небольшая разница между типом **Activate/Ready**, который представлен в спецификациях по безопасности, и рассматриваемым в предыдущих пунктах типом **Execute/Done**.

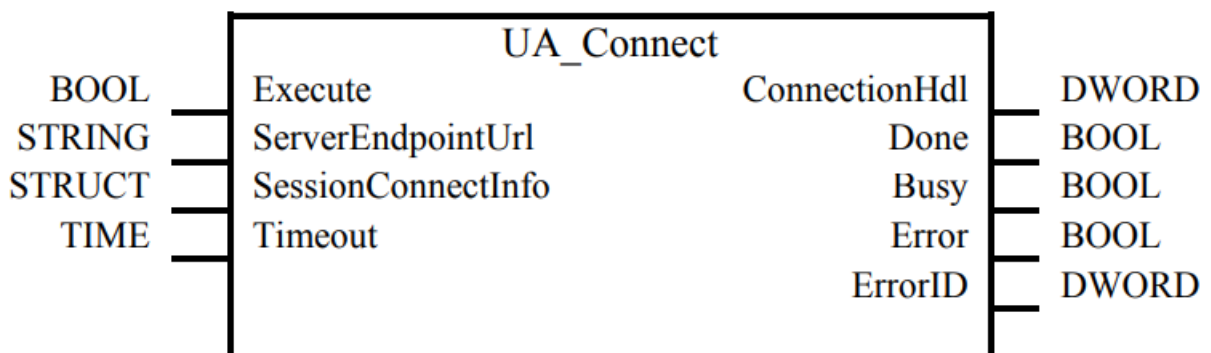
Пример безопасного ФБ



2.5. Коммуникация систем управления

Организация [PLCopen](#) совместно с промышленным консорциумом [OPC Foundation](#) разработала набор ФБ для организации обмена по [OPC UA](#). Функциональные блоки соответствуют модели **Execute/Done** и имеют выходы **Busy**, **Error**, **ErrorID** (аналогично ФБ из [спецификаций управления движением](#)).

Пример коммуникационного ФБ



2.6. Заключение

В спецификациях [PLCopen](#) используются два типа интерфейса ФБ:

- **Основной** – соответствующий типу **Execute/Done** (или **Activate/Ready**) с выходами **Busy**, **Error** и **ErrorID** (при этом тип переменной **ErrorID** может быть разным);
- **Расширенный** – с дополнительными выходами **Active** и **CommandAborted**.

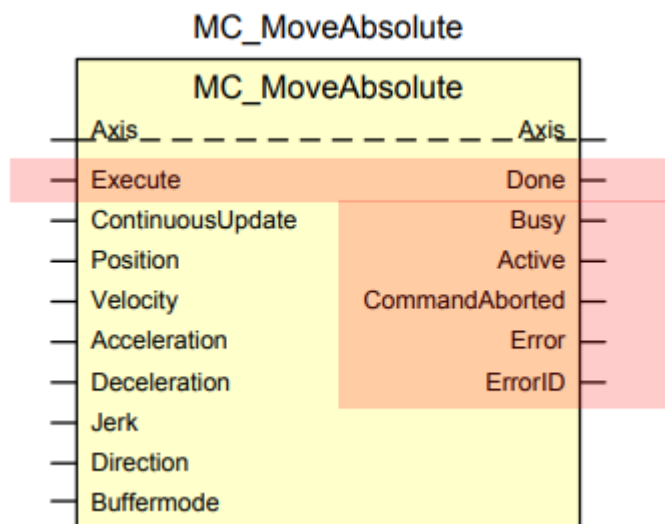


Рис. 3. Входы и выходы расширенного типа интерфейса ФБ **PLCopen**

Список входов и выходов, используемых в различных спецификациях **PLCopen**:

Библиотека	MC v1.0		MC v2.0	Safety	Communication
Модель ФБ	Enable	Execute	Enable/Execute	Activate	Execute
Доступные входы и выходы					
Execute		+	+		+
ContinuousUpdate			+		
Enable	+		+		
Activate				+	
Ready				+	
Valid	+		+		
Enabled	+				
Done		+	+		+
Busy	+	+	+		+
Active	+	+	+		
CommandAborted		+	+		
Error	+	+	+	+	+
ErrorID	+	+	+		+
DiagCode				+	

3. Основная информация о модели поведения ФБ PLCopen

В этой главе описываются основные особенности интерфейса функциональных блоков из спецификации **PLCopen**. Эти особенности в совокупности можно назвать **моделью поведения PLCopen**. Модель включает в себя два типа ФБ: выполняемые по уровню и выполняемые по фронту. Более подробная информация о реализации ФБ, соответствующих модели, приведена в главах [5](#) и [6](#), а также [Приложении 1](#).

3.1. Основная информация

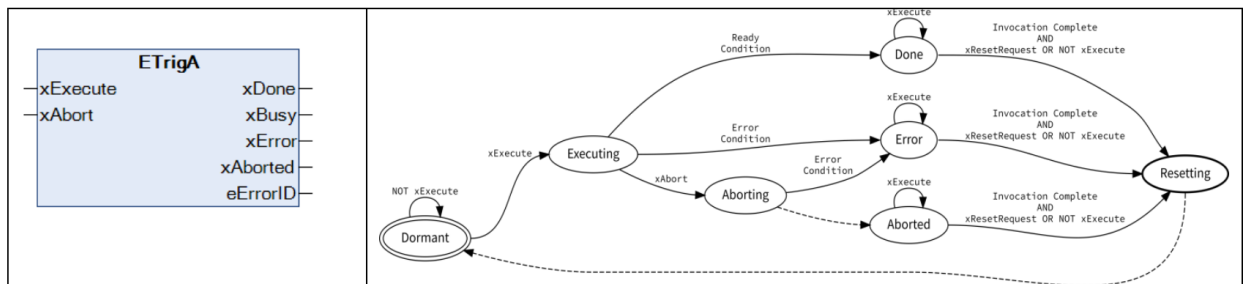
Как упоминалось выше, в спецификациях PLCopen используются два основных типа функциональных блоков:

- Выполняемые по уровню (со входом **Enable**);
- Выполняемые по фронту (со входом **Execute**).

При создании пользовательского ФБ его тип должен соответствовать требованиям решаемой задачи. Например, следует отметить, что для детектирования фронта требуется два цикла ПЛК. Таким образом, при необходимости обрабатывать значения входов ФБ каждый цикл тип **Execute** не подойдет; следует использовать тип **Enable**.

3.2. Принцип работы ФБ типа Execute

В качестве примера ФБ типа **Execute** рассмотрим функциональный блок **ETrigA** (Edge Triggered with Abort functionality). Ниже приведены его графическое представление и диаграмма состояний.



Блок работает по следующему принципу:

- по переднему фронту на входе **xExecute** начинается выполнение заданной операции;
- значения всех входов ФБ (кроме **xExecute** и **xAbort**) копируются в блок при его запуске, и их изменение в процессе работы блока не произведет никакого эффекта²;
- входу **xExecute** может быть присвоено значение FALSE после появления значения TRUE на выходе **xBusy**;

² В некоторых ситуациях могут потребоваться дополнительные входы, которые могут влиять на состояние ФБ в процессе его работы. Эти переменные должны быть хорошо документированы.

- по заднему фронту на входе **xExecute** работа ФБ не прекращается. Работа блока может быть завершена только в случае успешного окончания заданной операции (**xDone**), принудительного прерывания операции (**xAborted**) или же возникновения ошибки (**xError**);
- по переднему фронту на входе **xAbort** выполнение операции прекращается;
- если при вызове блока и вход **xExecute**, и вход **xAbort** имеют значение TRUE, то блок переходит в состояние *Aborting*;
- в пределах цикла ПЛК только один из статусных выходов блока (**xDone**, **xBusy**, **xError** или **xAborted**) может иметь значение TRUE^{3,4};
- после детектирования прерывания операции (**xAbort**) выход **xBusy** принимает значение **FALSE**, а выход **xAborted** – значение **TRUE**;
- по заднему фронту на выходе **xBusy** инвертированное значение входа **xExecute** копируется внутрь ФБ и будет использоваться как условие сброса ФБ. Как следует из диаграммы состояний (см. выше), сброс ФБ производится:
 - при *быстром подтверждении* – сразу же после завершения операции;
 - при *стандартном подтверждении* – по заднему фронту входа **xExecute**.
- значения выходов ФБ являются действительными⁵ как минимум один цикл ПЛК, даже если вход **xExecute** к этому времени уже имеет значение **FALSE**. В этом случае локальные переменные ФБ будут переинициализированы. Если же **xExecute** к моменту завершения операции имеет значение **TRUE**, то переинициализация локальных переменных произойдет по его заднему фронту (*стандартное подтверждение*);
- значения нестатусных выходов ФБ (т.е. всех выходов, кроме **xDone**, **xBusy**, **xError**, **eErrorID** и **xAborted**) являются действительными только в том случае, если выход **xDone** имеет значение **TRUE**;
- после появления на одном из перечисленных выходов (**xDone**, **xError** или **xAborted**) значения **TRUE** вход **xExecute** может быть опять установлен в значение **TRUE** для повторного выполнения операции (*быстрое подтверждение*).

Более подробная информация о реализации блоков типа **Execute** приведена в [Приложении 1.3](#).

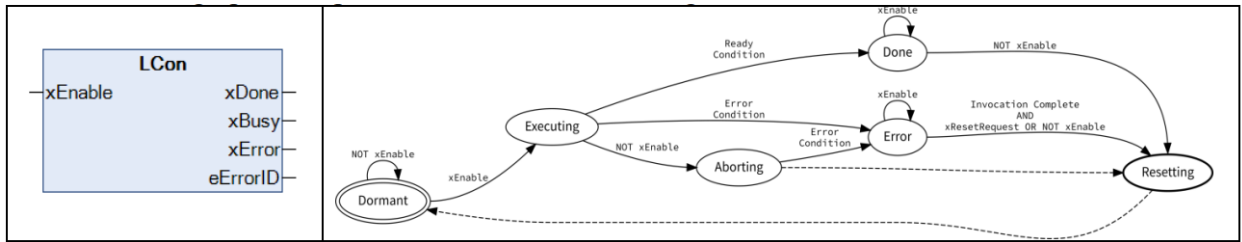
³ В некоторых ситуациях могут потребоваться дополнительные статусные выходы, которые имеют действительные значения, даже если выход **xDone=FALSE**. Эти переменные должны быть хорошо документированы.

⁴ В некоторых ситуациях могут потребоваться дополнительные статусные выходы, которые имеют действительные значения при конкретных значениях других статусных выходов. Эти переменные и их зависимость от других переменных должна быть хорошо документирована.

⁵ Под «действительным» значением в данном случае подразумевается корректное значение выхода, допускающее его использование другими программными модулями (прим. пер.).

3.3. Принцип работы ФБ типа Enable

В качестве примера ФБ типа **Enable** рассмотрим функциональный блок **LCon** (Level Controlled). Ниже приведены его графическое представление и диаграмма состояний.



Блок работает по следующему принципу:

- пока вход **xEnable** имеет значение TRUE, блок выполняет заданную операцию. Результатом работы блока является успешное выполнение операции или возникновение ошибки. Если вход **xEnable** принимает значение FALSE, то выполнение операции прерывается и происходит переинициализация выходов блока, после чего он ожидает следующего запуска (*стандартное подтверждение*);
- значения входов блока не копируются, а используются напрямую. Таким образом, изменение значений входов в процессе работы блока влияет на его поведение;
- в пределах цикла ПЛК только один из статусных выходов блока (**xDone**, **xBusy** или **xError**) может иметь значение TRUE^{6,7};
- значения выходов ФБ являются действительными⁸ только в том случае, если выход **xBusy** или **xDone** имеет значение TRUE;
- по заднему фронту на выходе **xBusy** инвертированное значение входа **xEnable** копируется внутрь ФБ и будет использоваться как условие сброса ФБ;
- значения выходов ФБ являются действительными как минимум один цикл ПЛК, даже если вход **xEnable** к этому времени уже имеет значение FALSE. В этом случае ФБ будет автоматически переинициализирован. Чаще всего такая ситуация происходит, если возникла ошибка в процессе прерывания (*Aborting*) работы блока.
- значения нестатусных выходов ФБ (т.е. всех выходов, кроме **xDone**, **xBusy**, **xError** и **eErrorID**) являются действительными только в том случае, если выход **xDone** имеет значение TRUE^{5,6};
- после появления на одном из перечисленных выходов (**xDone** или **xError**) значения TRUE вход **xEnable** может быть опять установлен в значение TRUE для повторного выполнения операции (*быстрое подтверждение*).

⁶ В некоторых ситуациях могут потребоваться дополнительные статусные выходы, которые имеют действительные значения, даже если выход **xDone**=FALSE. Эти переменные должны быть хорошо документированы.

⁷ В некоторых ситуациях могут потребоваться дополнительные статусные выходы, которые имеют действительные значения при конкретных значениях других статусных выходов. Эти переменные их зависимость от других переменных должна быть хорошо документирована.

⁸ Под «действительным» значением в данном случае подразумевается корректное значение выхода, допускающее его использование другими программными модулями (прим. пер.).

В некоторых случаях требуется модель поведения, которая не подразумевает завершение операции. В качестве примеров можно привести блок **MC_Power** из спецификации по управлению движением и блок TCP-сервера, рассматриваемый в [п. 3.7](#). Такие блоки не имеют выхода **xDone** и относятся к типу **Continuous Behaviour**. В рамках данного документа будут рассмотрены блоки [LConC](#) и [LConTIC](#), принадлежащие данному типу.

Более подробная информация о реализации блоков типа **Enable** приведена в [Приложении 1.4](#).

3.4. Общие свойства ФБ модели PLCopen

- если при вызове ФБ выполняется условие начала операции, то выход **xBusy** в этом же цикле принимает значение TRUE;
- пока ФБ выполняет заданную операцию, выход **xBusy** имеет значение TRUE;
- если операция успешно завершена, то выход **xDone** принимает значение TRUE, а выход **xBusy** – значение FALSE;
- если во время выполнения операции возникает ошибка, то выход **xError** принимает значение TRUE, выход **xBusy** – значение FALSE, а выход **eErrorID** – значение, отличное от *ERROR.NO_ERROR*. Выход **eErrorID** имеет тип *Перечисление (ENUM)*, но пользователь может определить его как INT, WORD и т.д.;
- если операция успешно завершается в течение первого цикла после начала ее выполнения, то выход **xDone** сразу принимает значение TRUE, а выход **xBusy** остается в состоянии FALSE;
- в момент появления переднего фронта на выходе **xDone** значения остальных выходов фиксируются;
- ФБ необходимо вызывать до тех пор, пока один из перечисленных выходов (**xDone**, **xBusy** или **xError**) имеет значение FALSE.

Функциональные блоки могут иметь входы, ограничивающие время их работы:

- **udiTimeLimit** (в мкс, 0 – время выполнения операции не ограничено).
ФБ может выполнять сложную операцию, и это займет длительное время. Вход **udiTimeLimit** позволяет определить максимальное время, которое блок может потратить на выполнение операции в пределах одного цикла ПЛК. Таким образом, сложные операции будут выполняться несколько циклов, но при этом не будут блокировать работу других задач приложения;
- **udiTimeOut** (в мкс, 0 – время ожидания не ограничено).
При выполнении некоторых операций функциональному блоку могут потребоваться сигналы о возникновении определенных внешних событий (например, получения ответа от другого устройства). Блок ожидает их либо в своем внутреннем цикле (состояние *BUSY WAIT*), либо выполняет проверку их возникновения при каждом своем вызове. Вход **udiTimeOut** позволяет ограничить время этого ожидания. При его превышении блок детектирует ошибку – выход **xError** принимает значение TRUE, а выход **eErrorID** – значение *ERROR.TIME_OUT*.

3.5. Обработка ошибок

В состав каждой библиотеки, соответствующей модели **PLCopen**, входит перечисление **ERROR**. Это перечисление скрыто за пространством имен библиотеки и содержит наименования и коды ошибок, которые могут возникнуть при работе функциональных блоков, входящих в библиотеку.

Каждый ФБ, рассмотренный в данном документе, имеет логический выход **xError**, который позволяет детектировать факт возникновения ошибки при работе блока. Если этот выход имеет значение TRUE, то на выход **eErrorID** транслируется код конкретной ошибки, описанной в перечислении **ERROR**. Фактически перечисление представляет собой численный тип данных, в котором можно связать значения с символьными именами (см. рис. ниже). Во многих случаях удобно конвертировать код ошибки в ее строковое описание на нужном языке с помощью дополнительного ФБ. Набор конкретных кодов уникален для каждой библиотеки. При объединении нескольких библиотек может возникнуть ситуация, когда одному коду ошибки будет соответствовать несколько разных символьных имен (т.е. произойдет перекрытие). В этом случае следует либо создать отдельные «конвертирующие» ФБ для каждой вложенной библиотеки, либо распределить между ними коды ошибок таким образом, чтобы избежать перекрытия.

Обработка ошибок должна быть реализована таким образом, чтобы на выход блока возвращались только те ошибки, которые задокументированы в библиотеке (т.е. присутствуют в перечислении **ERROR**). Если внутри ФБ происходит вызов блоков других библиотек, то кажется очень подходящим решением транслировать на его выходы коды ошибок этих блоков; тем не менее, это будет крайне неудобно для пользователя, который увидит недокументированные коды ошибок. Поэтому в данном случае рекомендуется скопировать нужные коды вложенных ФБ в перечисление **ERROR** разрабатываемого блока.

Рассмотрим приведенный выше материал на конкретном примере. Пусть у нас есть две библиотеки – **Memory Block Manager library** с пространством имен **MBM** и **Function Block Factory** с пространством имен **FBF**. Каждая библиотека содержит перечисление **ERROR** с кодами ошибок.

```
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
```

- каждая библиотека имеет пространство имен (**MBM** и **FBF**);
- каждое перечисление **ERROR** должно содержать код *NO_ERROR* (0) и может содержать код *TIME_OUT* (1);
- если код *TIME_OUT* не требуется, то значение 1 не может соответствовать другой ошибке;
- для каждой ошибки требуется комментарий с коротким, но ясным описанием;
- перечисление **ERROR** должно быть изолировано от других перечислений с помощью атрибута *'qualified_only'*. Таким образом, *MBM.ERROR.NO_MEMORY* и *FBF.ERROR.NO_MEMORY* будут представлять собой разные ошибки с разными кодами.

При работе с вложенными библиотеками возникает потребность в создании общего перечисления, в которое будут «проброшены» ошибки из перечислений отдельных библиотек. Ниже приведен пример функции, которая транслирует ошибки из перечислений **ERROR** библиотек с пространствами имен **CS** и **CO** в общее перечисление **CANOPEN_KERNEL_ERROR**. Следует отметить, что все перечисления в данном случае должны принадлежать одинаковому целочисленному типу данных (например, 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

```

В рамках функции предполагается, что ошибки *CS.ERROR.NO_ERROR* и *CO.ERROR.NO_ERROR* имеют одинаковые коды, а коды остальных ошибок перечислений никак не связаны между собой.

3.6. Имена шагов

Реализация ФБ, соответствующего конкретной модели поведения, тесно связана с разработкой машины состояний, имена шагов которой содержатся в перечислении **STATE**. В рамках данного документа в каждом примере используется свое перечисление **STATE**, которое содержит необходимые для этого примера шаги. В настоящих библиотеках присутствует только одно перечисление **STATE**, которые содержит все шаги, необходимые для реализации ФБ библиотеки.

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

POUs

- PLCopen Common Behaviour Model
 - Enums
 - ERROR (ENUM)
 - STATE (ENUM)
 - Function Blocks
 - ETrig (FB)
 - ETrigA (FB)
 - ETrigATI (FB)
 - ETrigATo (FB)
 - ETrigTI (FB)
 - ETrigTTo (FB)
 - ETrigTo (FB)
 - LCon (FB)
 - LConC (FB)
 - LConTI (FB)
 - LConTIC (FB)
 - LConTTo (FB)
 - LConTo (FB)
 - TimingController (FB)
- Library Manager
- Project Information
- Project Settings

ETrigATo x

```

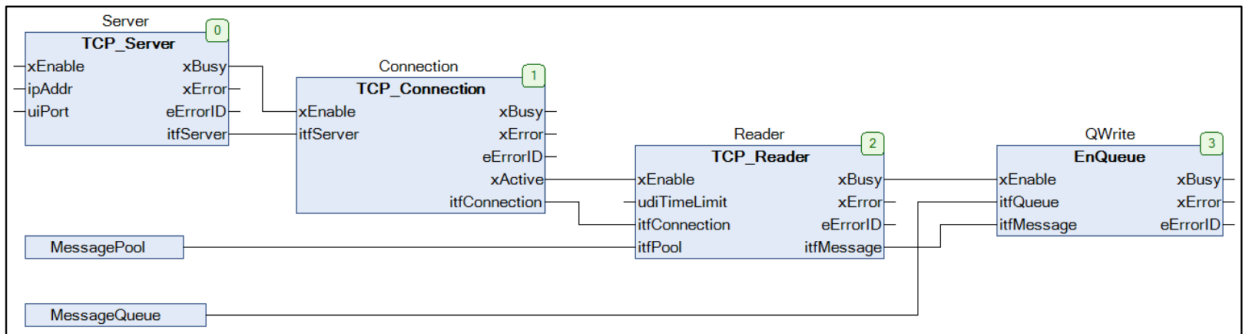
1  REPEAT
2      xAgain := FALSE;
3      CASE eState OF
4          STATE.DORMANT: HandleDormantState(xAgain=>xAgain);
5          STATE.EXECUTING: HandleExecutingState(xAgain=>xAgain);
6          STATE.DONE: HandleDoneState(xAgain=>xAgain);
7          STATE.ERROR: HandleErrorState(xAgain=>xAgain);
8          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. Совместное использование ФБ

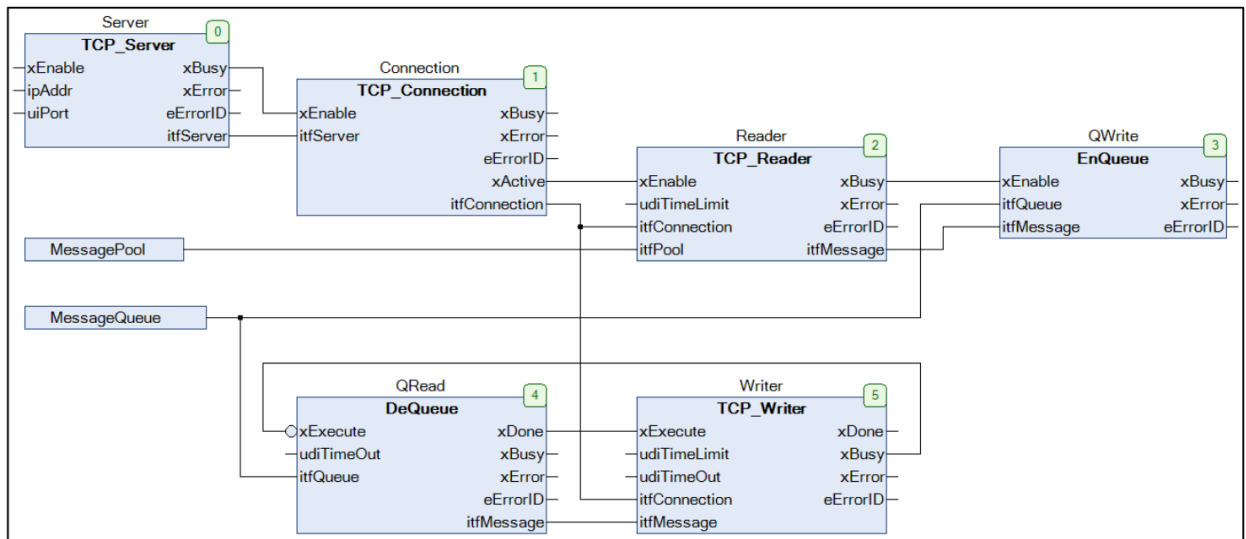
Совместное использование функциональных блоков осуществляется по одной из вышеописанных моделей поведения. Если блоки соответствуют модели поведения **PLCopen**, то установить связи между экземплярами очень легко, в особенности на графических языках программирования. Приведенный ниже пример демонстрирует преимущества модели **PLCopen**.

Пример: чтение данных по TCP



ФБ **TCP_Server** ([LConC](#)) создает слушающий сокет. Сокет представляет собой пару «IP-адрес – порт». После подключения клиента активируется блок **TCP_Connection** ([LConC](#)), который обрабатывает соединение. С помощью ФБ **TCP_Reader** ([LConTIC](#)) полученное от клиента сообщение размещается в структуре **MessagePool**. Для последующей работы с этими сообщениями используется блок **QWrite** ([LCon](#)), который обрабатывает их в порядке очереди.

Расширенный пример: эхо-сервер

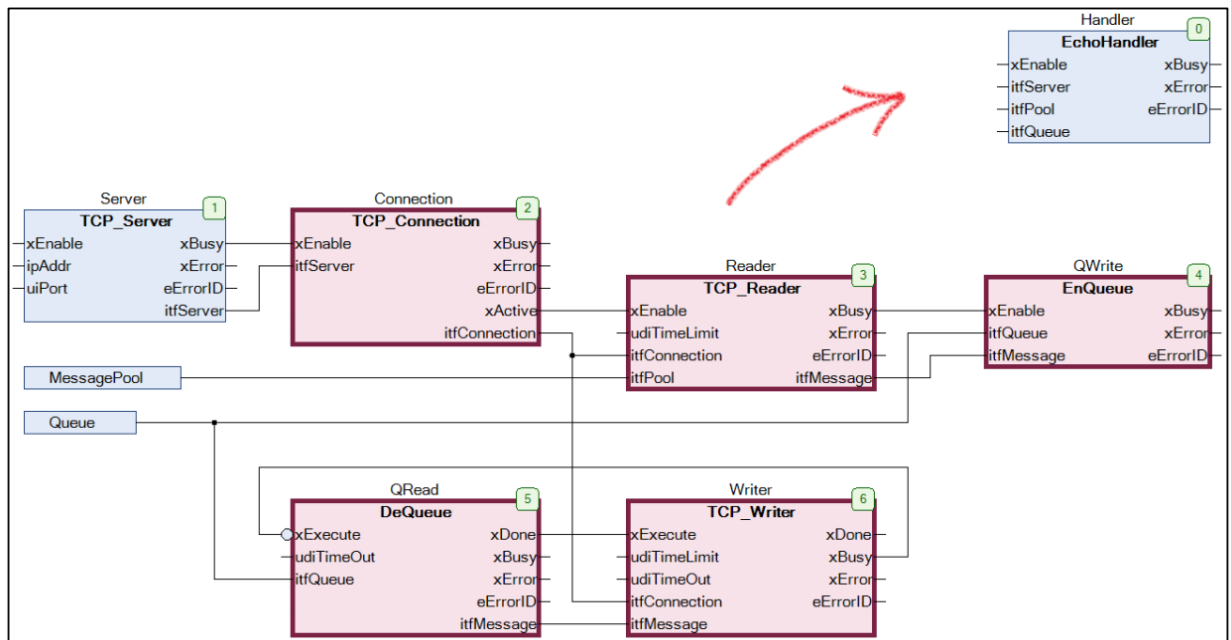


Функциональный блок **TCP_Writer** ([ETrigTITo](#)) в порядке очереди получает сообщения из структуры **MessageQueue** помощью ФБ **QRead** и отправляет их обратно клиенту через установленное с ним соединение (**TCP_Connection**). Таким образом, предыдущий пример был расширен до эхо-сервера.

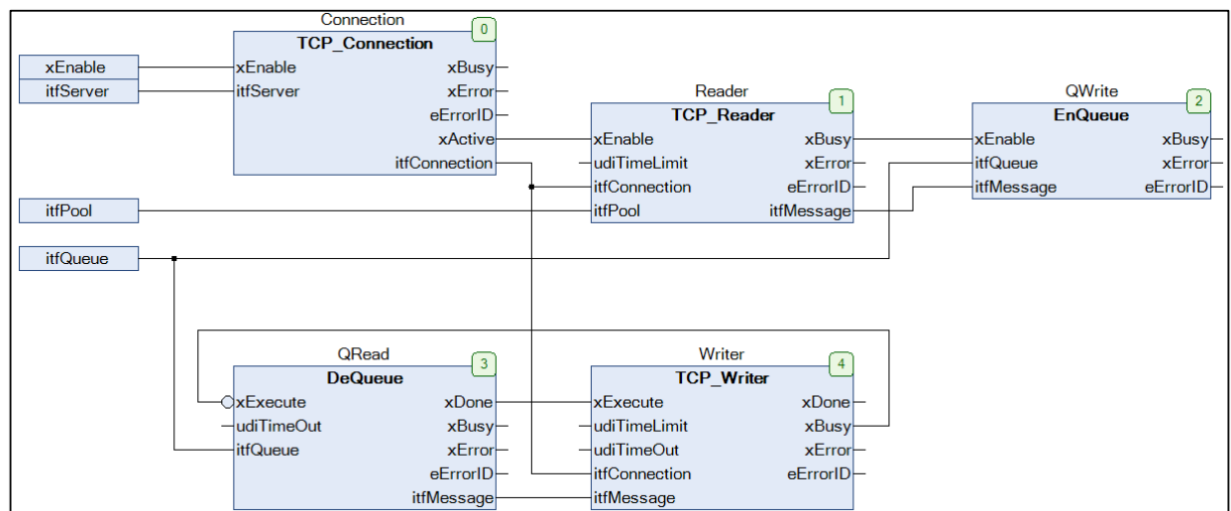
Примечание: функциональный блок **TCP_Reader** должен иметь возможность обновлять сообщение на выходе каждый цикл. Если в текущем цикле нет нового сообщения, то выход должен иметь нулевое значение. Соответственно, для данного ФБ подойдет тип **Enable**. Отправка сообщения с помощью ФБ **TCP_Writer** может занимать несколько циклов, поэтому для данного блока следует использовать тип **Execute**. Для совместной работы ФБ разных типов требуется дополнительный промежуточный блок, который будет создавать очередь сообщений. В данном примере таким блоком является **QRead**. Обратите внимание, что на вход **xExecute** этого блока подается инвертированное значение выхода **xBusy** ФБ **TCP_Reader** – таким образом, не требуется дополнительный цикл на передку данных между блоками и ни одно сообщение не будет пропущено.

Полный пример: многопоточный эхо-сервер

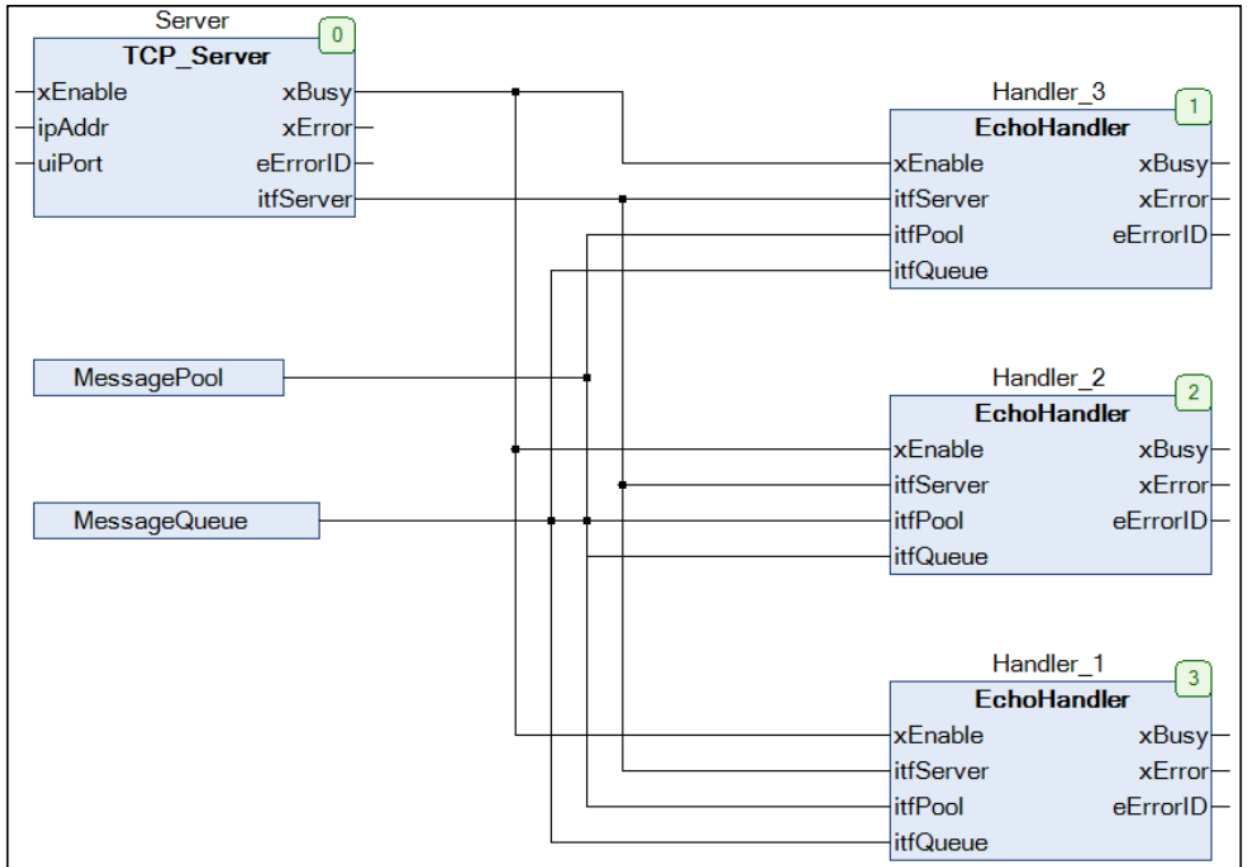
Объедините блоки **TCP_Reader**, **QWrite**, **QRead** и **TCP_Writer** в один функциональный блок с названием **EchoHandler**.



Состав ФБ EchoHandler:



Объявите нужное количество экземпляров ФБ **EchoHandler**:



В результате вы получите многопоточный эхо-сервер.

4. Введение в объектно-ориентированные расширения МЭК 61131-3

В 3-й редакции стандарта **МЭК 61131-3** были добавлены объектно-ориентированные расширения: классы, методы, интерфейсы, наследование и т.д. В данном документе не приводится подробное описание этих расширений; подразумевается, что читатель с ними знаком.

Возможности ООП в спецификациях используются с осторожностью; из всего набора доступных средств применяются только методы с защищенным (PROTECTED) и закрытым (PRIVATE FINAL) интерфейсом – это позволяет оптимизировать внутреннюю структуру ФБ.

Функциональные блоки базируются на машине состояний, которая определяет доступные состояния и переходы между ними, но не содержит операций, которые должны выполняться в том или ином состоянии. Поэтому описанные в документе ФБ не предназначены для использования в программе в чистом виде; пользователю следует создать свой ФБ на базе одного из существующих с помощью наследования. Методы с защищенным интерфейсом должны быть переписаны в соответствии с конкретной задачей.

Тем не менее, использование ООП расширений не является обязательным требованием при создании ФБ, совместимых с моделью поведения **PLCopen**. В рассматриваемых примерах каждый вызов метода может быть заменен на код этого метода. Для создания пользовательского ФБ вместо наследования можно скопировать и отредактировать существующий блок, заменив его методы на нужный для выполнения задачи код.

5. Описание ФБ типа Execute

5.1. Базовый ФБ: ETrig

ФБ **ETrig** является простейшим блоком типа **Execute** и имеет только один вход. Ниже приведено текстовое и графическое представление входов и выходов блока.

Текстовое представление	Графическое представление
<pre> FUNCTION_BLOCK ETrig VAR_INPUT xExecute: BOOL; END_VAR VAR_OUTPUT xDone: BOOL; xBusy: BOOL; xError: BOOL; iErrorID: INT; END_VAR </pre>	

На рис. 4 приведена диаграмма состояний ФБ **ETrig**. Блок имеет 5 возможных состояний: *Dormant*, *Busy*, *Done*, *Reset* и *Error*.

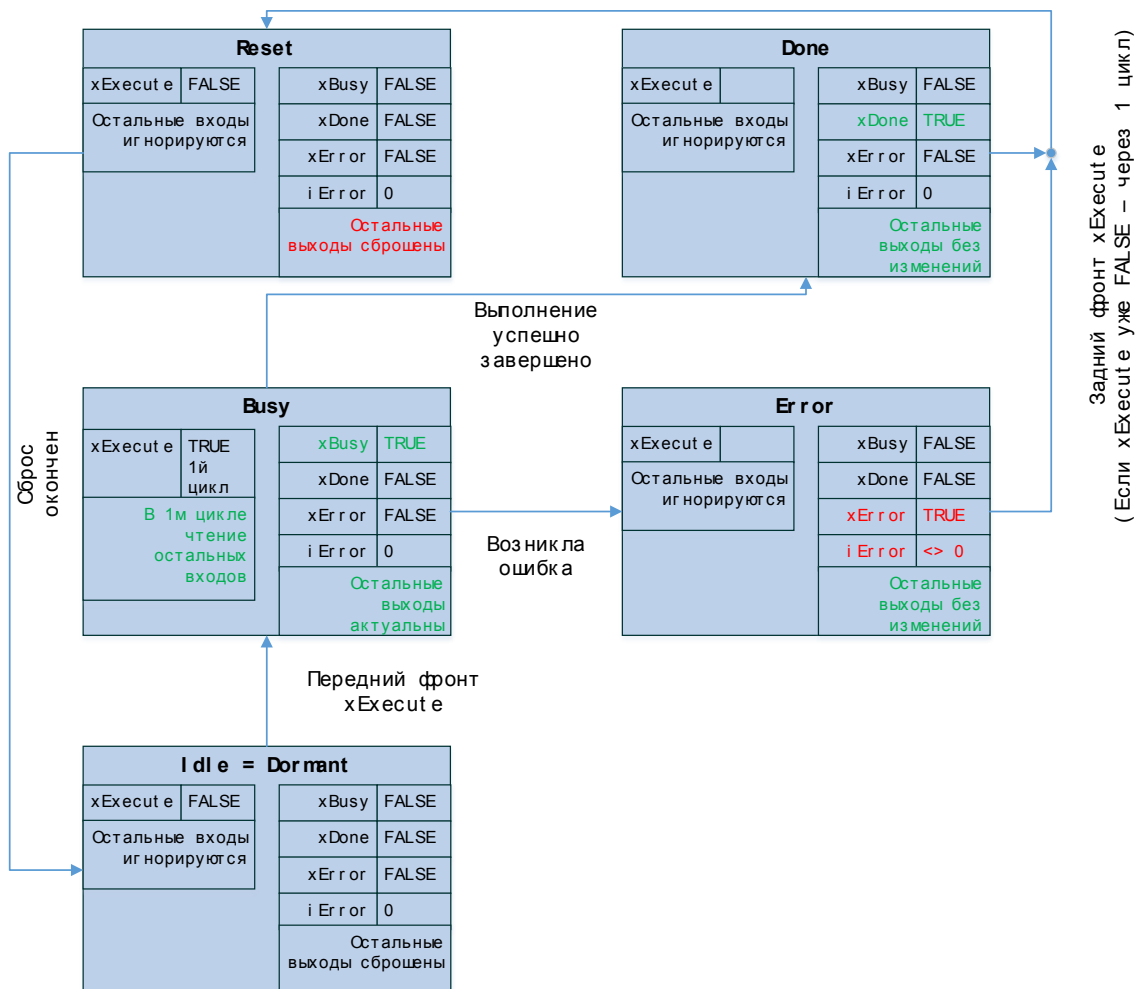


Рис. 4. Диаграмма состояний ФБ **ETrig**

По переднему фронту на входе **xExecute** блок переходит из состояния *Dormant* в состояние *Busy*, при этом значения пользовательских входов⁹ копируются и сохраняются внутри блока. Изменение значений входов в процессе работы блока не повлияет на выполняемую операцию. Выход **xBusy** принимает значение TRUE. Блок начинает выполнять заданную операцию. В процессе работы при выполнении одного из следующих условий произойдет переход из состояния *Busy*, при этом выход **xBusy** примет значение FALSE:

- если операция будет успешно завершена, то произойдет переход в состояние *Done*, при этом выход **xDone** примет значение TRUE;
- если во время выполнения операции возникнет ошибка, то произойдет переход в состояние *Error*, при этом выход **xError** примет значение TRUE, а на выходе **eError** будет отображаться код соответствующей ошибки из перечисления **ERROR**.

В каждый момент времени только один из статусных выходов (**xDone** или **xError**) может иметь значение TRUE. Это значение должно удерживаться как минимум один цикл ПЛК.

Если ФБ находился в состоянии *Done* или *Error*, то после появления значения FALSE на входе **xExecute** блок перейдет в состояние *Reset*. Все выходы инициализируются их начальными значениями. Выполнение всех операций прекращается, и все занятые ресурсы системы освобождаются. Выходы **xDone** и **xError** устанавливаются в значение FALSE. После переинициализации блока происходит переход в состояние *Dormant*.

Пример реализации ФБ ETrig на языке ST с использованием ООП

В приведенном ниже примере используются элементы ООП. Пример реализации данного ФБ без использования ООП приведен в [Приложении 2](#).

Перечисление STATE:

```
TYPE STATE :
(
    DORMANT,      // ожидание запуска
    EXECUTING,    // выполнение операции
    DONE,         // завершение операции
    ERROR,        // ошибка
    RESETTING     // переинициализация
);
END_TYPE
```

Перечисление ERROR:

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

⁹ Речь идет о входах реального ФБ, связанных с конкретной задачей (прим. пер.).

Код ФБ ETrig:

```

FUNCTION_BLOCK ETrig
VAR_INPUT
    // по переднему фронту начинается выполнение заданной операции
    // по заднему фронту после завершения операции блок будет переинициализирован
    xExecute      : BOOL;
END_VAR

VAR_OUTPUT
    // флаг «операция успешно завершена»
    xDone         : BOOL;
    // флаг «операция в процессе выполнения»
    xBusy         : BOOL;
    // флаг «произошла ошибка»
    xError        : BOOL;
    // код ошибки
    eErrorID      : ERROR;
END_VAR

VAR
    eState        : STATE;
    xFirstInvocation : BOOL := TRUE;
    xResetRequest  : BOOL;
END_VAR

VAR_TEMP
    xAgain        : BOOL;
END_TEMP

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;

```

Обработчик для состояния Dormant:

```

METHOD PRIVATE FINAL HandleDormantState
VAR_OUTPUT
    xAgain : BOOL;
END_VAR

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

```

Обработчик для состояния Executing:

```

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

```

Обработчик для состояния Done:

```

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

```

Обработчик для состояния Error:

```

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

```


Обработчик для состояния Resetting:

```

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

```

Пример реализации уникальных методов ФБ

Приведенный ниже код следует использовать только в качестве примера. Программист должен написать свою реализацию для конкретной задачи.

Реализация CyclicAction:

```

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

IF xFirstInvocation THEN
    (* Starting *)
    // при первом вызове сохраняем запоминаем значения входов ФБ
    xFirstInvocation := FALSE;
END_IF
(* Executing *)
// выполняем заданную операцию. Если она успешно завершена, то => xComplete := TRUE
// если произошла ошибка, то xError := TRUE и присваиваем eErrorID код ошибки

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

IF xComplete OR eErrorID <> ERROR.NO_ERROR THEN
    (* Cleaning *)
    // если необходимо, освобождаем ресурсы системы
END_IF

```

Реализация ResetAction:

```

METHOD PROTECTED ResetAction
VAR_OUTPUT
    xComplete : BOOL;
END_VAR
// освобождаем все используемые ресурсы системы
// переинициализируем переменные
xComplete := TRUE;

```

5.2. Добавляем возможность прерывания операции (ETrigA)

Теперь расширим функционал ФБ **ETrig** возможностью прерывания операции. Для этого добавим вход **xAbort**, который будет использоваться для подачи команды на прерывание, и выход **xAborted**, с помощью которого можно будет определить, что работа блока была прервана¹⁰. Новый блок будет называться **ETrigA**. Ниже приведено текстовое и графическое представление его входов и выходов.

Текстовое представление	Графическое представление
<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>	

На рис. 5 приведена диаграмма состояний ФБ **ETrigA**. Блок имеет 7 возможных состояний: *Dormant*, *Executing*, *Aborting*, *Done*, *Error*, *Aborted* и *Resetting*.

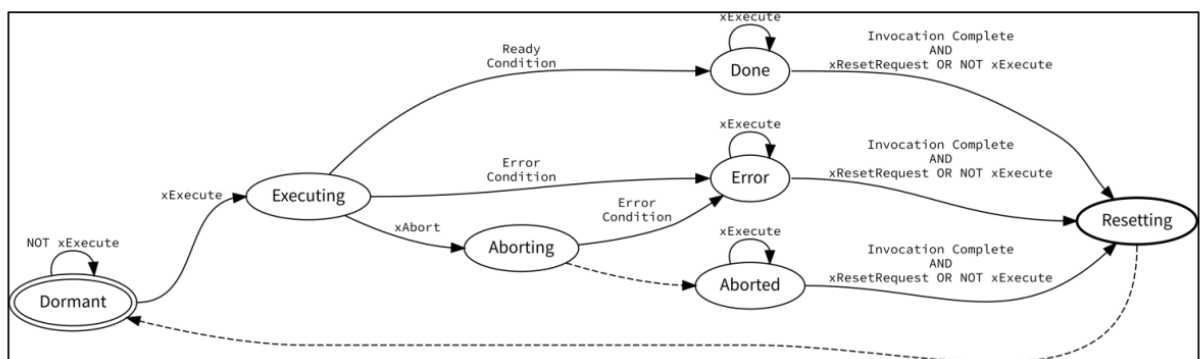
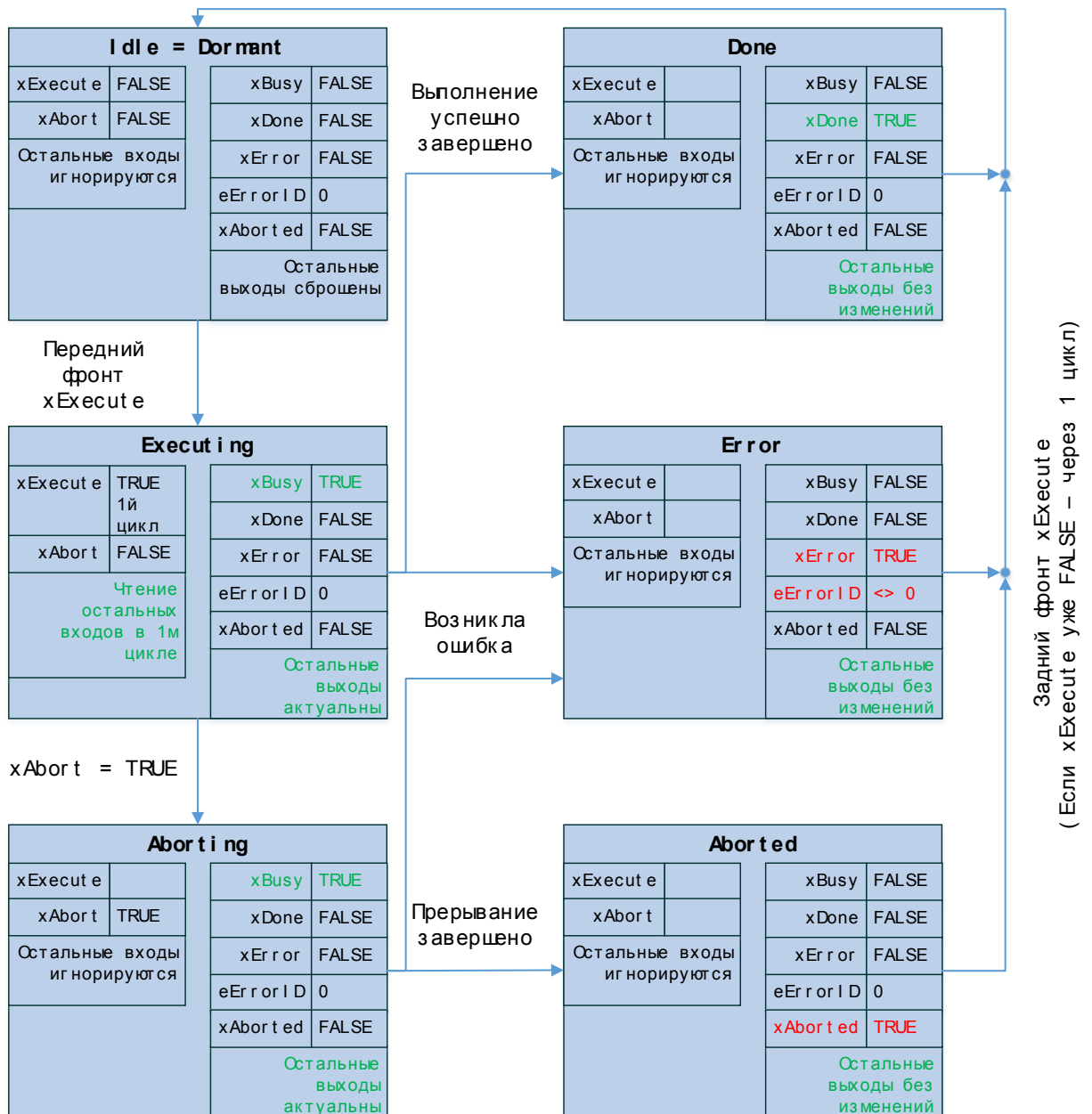


Рис. 5. Диаграмма состояний ФБ **ETrigA**

¹⁰ В спецификации по управлению движением (Motion Control) ФБ не имеют входа **xAbort**. Прерывание работы этих блоков происходит не по команде пользователя, а по заложенным в них алгоритмам.

Рис. 6. Диаграмма состояний ФБ ETriga с описаниями входов/выходов¹¹

По переднему фронту на входе `xExecute` блок переходит из состояния *Dormant* в состояние *Executing*, при этом значения пользовательских входов¹² копируются и сохраняются внутри блока. Изменение значений входов в процессе работы блока не повлияет на выполняемую операцию. Выход `xBusy` принимает значение TRUE. Блок начинает выполнять заданную операцию. В процессе работы при выполнении одного из определенных условий произойдет переход из состояния *Executing*, при этом выход `xBusy` примет значение FALSE:

¹¹ На этом рисунке состояние Resetting объединено с состоянием Dormant.

¹² Речь идет о входах реального ФБ, связанных с конкретной задачей (прим. пер.).

- если операция будет успешно завершена, то произойдет переход в состояние *Done*, при этом выход **xDone** примет значение TRUE;
- если во время выполнения операции возникнет ошибка, то произойдет переход в состояние *Error*, при этом выход **xError** примет значение TRUE, а на выходе **eError** будет отображаться код соответствующей ошибки из перечисления **ERROR**;
- если во время выполнения операции поступит команда на ее прерывание (вход **xAbort** примет значение TRUE), то произойдет переход в состояние *Aborting*. Выполняемая операция будет прервана, и на выходе **xAborted** будет установлено значение TRUE. После этого произойдет переход в состояние *Aborted*.

В каждый момент времени только один из статусных выходов (**xDone**, **xError**, **xAborted**) может иметь значение TRUE. Это значение должно удерживаться как минимум один цикл ПЛК.

Если ФБ находился в состоянии *Done* или *Error*, то после появления значения FALSE на входе **xExecute** блок перейдет в состояние *Reset*. Все выходы инициализируются их начальными значениями. Выполнение всех операций прекращается, и все занятые ресурсы системы освобождаются. Выходы **xDone**, **xError** и **xAborted** устанавливаются в значение FALSE. После переинициализации блока происходит переход в состояние *Dormant*.

Пример реализации ФБ ETrigA на языке SFC

Ниже приведен пример реализации ФБ **ETrigA** на языке SFC в соответствии с диаграммой состояний, изображенной на рис. 5. Каждое состояние представляет собой шаг SFC, содержащий нужные действия. Каждое действие имеет соответствующий классификатор.

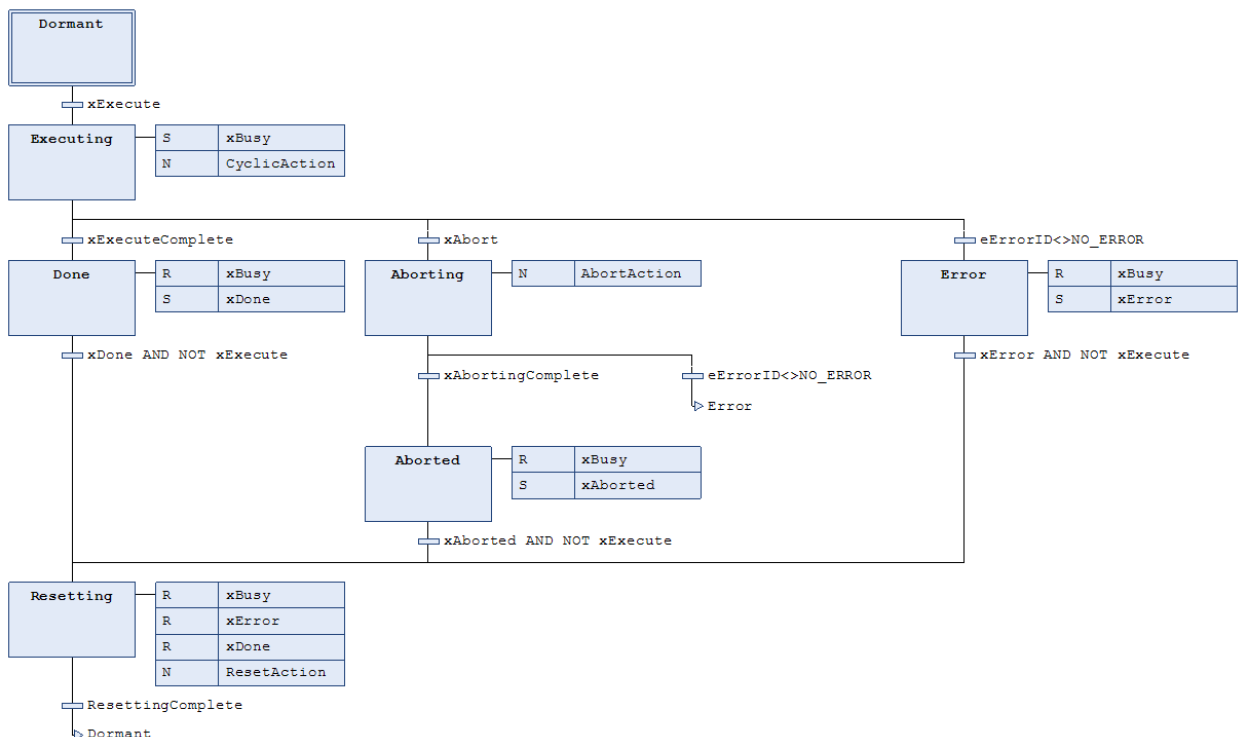


Рис. 7. Пример реализации ФБ **ETrigA** на языке SFC

Описание используемых в примере классификаторов действий:

N	Несохраняемое действие (<i>Non-stored</i>). Данное действие будет выполняться в каждом рабочем цикле, пока активен шаг.
R	Сброс (<i>Reset</i>). Действие деактивируется.
S	Сохраняемое действие (<i>Stored</i>). Действие активируется и остается активным до сброса. Действие продолжит выполняться в каждом цикле даже тогда, когда шаг уже неактивен.

Более детальная информация приведена в стандарте МЭК 61131-3.

Пример реализации ФБ **ETrigA** на языке ST приведен в [Приложении 1.3.5](#).

5.3. Добавляем таймеры

Независимо от наличия входа прерывания **xAbort** можно ограничить время выполнения ФБ с помощью таймеров. Существует три варианта ограничения:

1. **TimeLimit (TI)** – время, затрачиваемое на выполнении операции в пределах одного цикла ПЛК, ограничивается значением входа **udiTimeLimit** (задается в микросекундах). Таким образом, сложные операции будут выполняться несколько циклов, но при этом не будут блокировать работу других задач приложения;
2. **TimeOut (To)** – общее время выполнения операции ограничивается значением входа **udiTimeOut** (задается в микросекундах);
3. Совместное использование вариантов 1 и 2 (**TITo**).

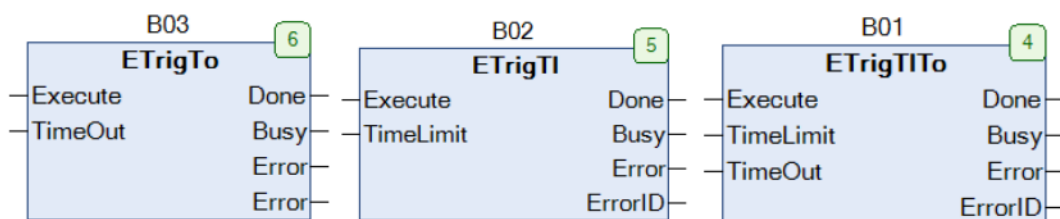
Как работает вход **udiTimeLimit**?

ФБ может выполнять сложную операцию, и это потребует длительного времени. Вход **udiTimeLimit** позволяет определить максимальное время, которое блок может потратить на попытку выполнения операции в пределах одного цикла ПЛК. Таким образом, сложные операции будут выполняться несколько циклов, но при этом не будут блокировать работу других задач приложения.

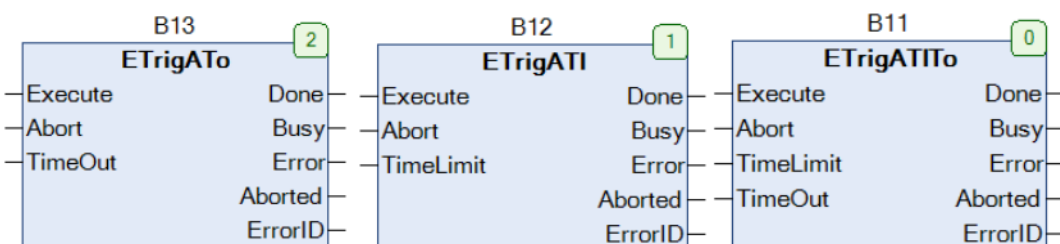
Как работает вход **udiTimeOut**?

При выполнении некоторых операций функциональному блоку могут потребоваться сигналы о возникновении определенных внешних событий (например, получения ответа от другого устройства). Блок ожидает их либо в своем внутреннем цикле (состояние *BUSY WAIT*), либо выполняет проверку их возникновения при каждом своем вызове. Вход **udiTimeOut** позволяет ограничить время этого ожидания. При его превышении блок детектирует ошибку – выход **xError** принимает значение TRUE, а выход **eErrorID** – значение *ERROR.TIME_OUT*.

Примеры ФБ с таймерами и без входа прерывания



Примеры ФБ с таймерами и входом прерывания



Примечание: в данных примерах названия входов/выходов приведены без префиксов.

Ниже приведена диаграмма состояний для наиболее сложного ФБ – **ETrigATiTo**. Выполнение условия **TimeOut** приводит к переходу в состояние **ERROR**. Условие **TimeLimit** проверяется в состоянии *Executing*.

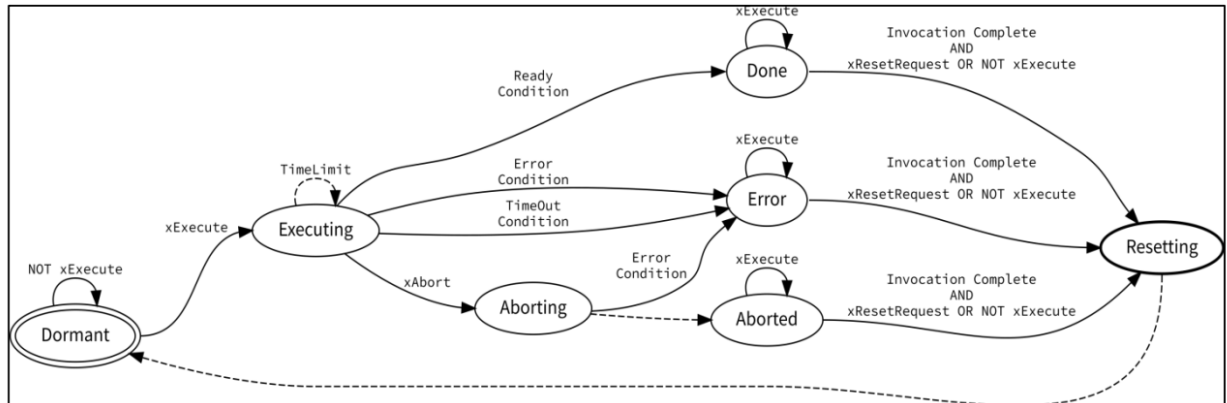


Рис. 9. Диаграмма состояний ФБ **ETrigATiTo**

Примечание: в приведенных выше диаграммах не используются триггеры переднего и заднего фронта (**R_TRIG** и **F_TRIG**). Вместо этого контролируются значения входов. Например, значение **TRUE** переменной **xExecute** определяет переход из состояния *Dormant* в состояние *Executing*.

Детальное описание принципа работы ФБ **ETrigATiTo**

1. Функциональный блок однократно вызывается в каждом цикле ПЛК независимо от каких-либо условий;
2. По переднему фронту на входе **xExecute** блок переходит из состояния *Dormant* в состояние **Executing**;
 - Значения пользовательских входов¹³ копируются и сохраняются внутри блока. Изменение значений входов в процессе работы блока не повлияет на выполняемую операцию;
 - Выход **xBusy** устанавливается в значение **TRUE**. После этого переменной **xExecute** можно присвоить значение **FALSE** (*быстрое подтверждение*);
3. Начинается выполнение заданной операции;
4. Если время выполнения операции превышает значение **udiTimeLimit** (в мкс), то операция останавливается и будет продолжена при следующем вызове ФБ;
5. После окончания операции или выполнении определенных условий блок переходит в состояние *Done*, *Error* или *Aborted*. При этом выход **xBusy** принимает значение **FALSE**, а статусный выход, соответствующий новому состоянию (**xDone**, **xError** или **xAborted**) принимает значение **TRUE** как минимум на один цикл ПЛК. В каждый момент времени только один из статусных выходов может иметь значение **TRUE**. По заднему фронту на выходе **xBusy** инвертированное значение входа **xExecute** копируется внутрь ФБ и будет использоваться как условие сброса ФБ (см. **xResetRequest** в методах *HandleDoneState*, *HandleErrorState* и *HandleAbortedState*). Как следует из диаграммы состояний (см. выше), сброс ФБ производится:
 - при *быстром подтверждении* – сразу же после завершения операции;
 - при *стандартном подтверждении* – по заднему фронту входа **xExecute**.

¹³ Речь идет о входах реального ФБ, связанных с конкретной задачей (прим. пер.).

Описание состояний, в которые блок может перейти после выполнения операции, приведено ниже:

a. Done:

Если заданная операция успешно завершена, то блок переходит в состояние *Done* и выход **xDone** принимает значение TRUE;

b. Error:

Если в процессе выполнения операции происходит ошибка, то блок переходит в состояние *Error*. Выход **xError** принимает значение TRUE, а на выход **eErrorID** транслируется код ошибки, определенный в перечислении **ERROR**;

c. Timeout:

Если общее время выполнения операции превышает значение входа **udiTimeOut** (в мкс), то блок переходит в состояние *Error*. Выход **xError** принимает значение TRUE, а на выход **eErrorID** транслируется код ошибки *ERROR.TIME_OUT*;

d. Abort:

Если вход **xAbort** принимает значение TRUE, то блок переходит в состояние *Aborting*. Выполнение заданной операции прерывается, после чего выход **xAborted** устанавливается в значение TRUE и блок переходит в состояние *Aborted*. Если при прерывании операции возникают ошибки в работе блока, то блок переходит в состояние *Error*. Выход **xError** принимает значение TRUE, а на выход **eErrorID** транслируется код ошибки, определенный в перечислении **ERROR**;

6. После появления на одном из статусных выходов (**xDone**, **xError** или **xAborted**) значения TRUE вход **xExecute** может быть опять установлен в значение TRUE для повторного выполнения операции (*быстрое подтверждение*);

7. Значения статусных выходов должны устанавливаться в TRUE как минимум на один цикл ПЛК. В каждый момент времени может быть активен только один из этих выходов;

8. Значения нестатусных выходов ФБ (т.е. всех выходов, кроме **xDone**, **xBusy**, **xError**, **eErrorID** и **xAborted**) являются действительными только в том случае, если выход **xDone** имеет значение TRUE;

9. Если после окончания выполнения операции вход **xExecute** принимает значение FALSE (*стандартное подтверждение*) или выполняется условие сброса ФБ (*быстрое подтверждение*), то блок переходит в состояние *Resetting*;

10. Все выходы инициализируются их начальными значениями. Выполнение всех операций прекращается, и все занятые ресурсы системы освобождаются. Выходы **xDone**, **xError** и **xAborted** устанавливаются в значение FALSE;

11. После переинициализации блока происходит переход в состояние *Dormant*.

5.4. Пример реализации ФБ ETrigATITo на языке ST

Пример реализации ФБ ETrigATITo на языке ST приведен в [Приложении 1.3.8](#).

6. Описание ФБ типа Enable

6.1. Базовый ФБ: LCon

ФБ **LCon** является простейшим блоком типа **Enable** и имеет только один вход. Ниже приведено текстовое и графическое представление входов и выходов блока.

Текстовое представление	Графическое представление
<pre> FUNCTION_BLOCK LCon VAR_INPUT xEnable: BOOL; END_VAR VAR_OUTPUT xDone: BOOL; xBusy: BOOL; xError: BOOL; eErrorID: INT; END_VAR </pre>	

На рис. 9 приведена диаграмма состояний ФБ **LCon**. Блок имеет 6 возможных состояний: *Dormant*, *Executing*, *Done*, *Aborting*, *Error* и *Resetting*.

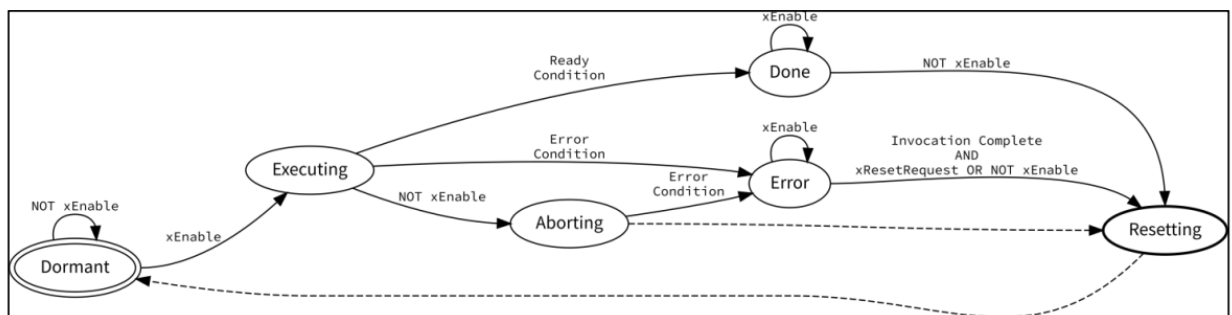
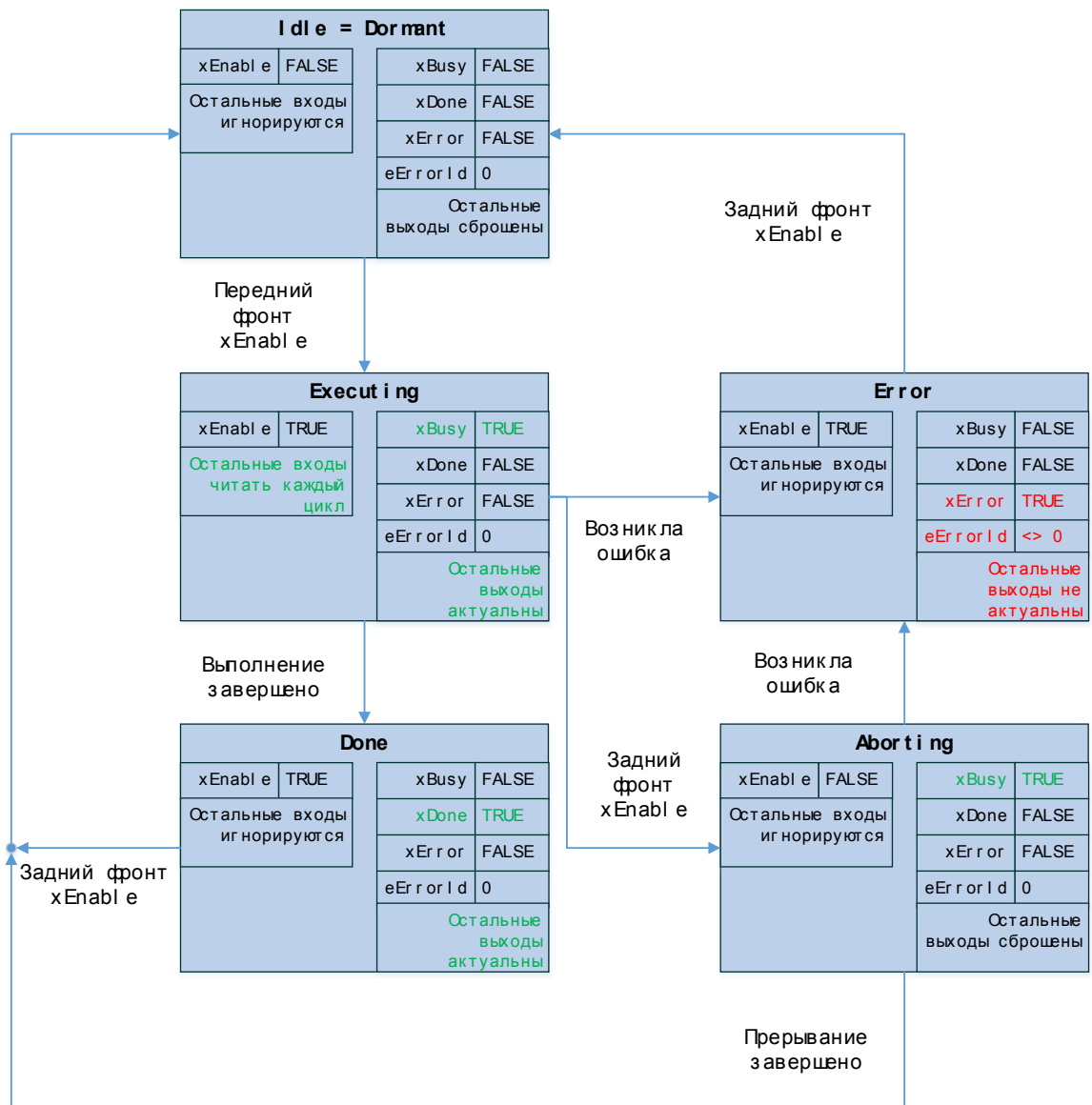


Рис. 9. Диаграмма состояний ФБ **LCon**

Рис. 10. Диаграмма состояний ФБ LCon с описаниями входов/выходов¹⁴

6.2. Пример реализации ФБ LCon на языке ST с использованием ООП

Перечисление STATE:

```

TYPE STATE :
(
    DORMANT,      // ожидание запуска
    EXECUTING,    // выполнение операции
    ABORTING,     // прерывание операции
    DONE,         // завершение операции
    ERROR,        // ошибка
    RESETTING     // переинициализация
);
END_TYPE

```

¹⁴ На этом рисунке состояние Resetting объединено с состоянием Dormant.

Перечисление ERROR:

```

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

```

Код ФБ LCon:

```

FUNCTION_BLOCK LCon
VAR_INPUT
    // пока вход имеет значение TRUE - операция выполняется
    // если вход принимает значение FALSE, то выполнение операции прекращается
    xEnable:  BOOL;
END_VAR
VAR_OUTPUT
    // флаг «операция успешно завершена»
    xDone:    BOOL;
    // флаг «операция в процессе выполнения»
    xBusy:    BOOL;
    // флаг «произошла ошибка»
    xError:   BOOL;
    // код ошибки
    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;

```

Обработчик для состояния Dormant:

```

METHOD PRIVATE FINAL HandleDormantState
VAR_OUTPUT
    xAgain: BOOL;
END_VAR

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

```

Обработчик для состояния Executing:

```

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

```

Обработчик для состояния Aborting:

```

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

```

Обработчик для состояния Done:

```

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

```

Обработчик для состояния Error:

```

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

```

Обработчик для состояния Resetting:

```

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

```

Пример реализации уникальных методов ФБ

Приведенный ниже код следует использовать только в качестве примера. Программист должен написать свою реализацию для конкретной задачи.

Реализация CyclicAction:

```

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

IF xEnable THEN
    (* Executing *)
    // при каждом вызове считываем значения входов
    // выполняем операцию. Если она успешно завершена, то => xComplete := TRUE
    // если произошла ошибка, то xError := TRUE и присваиваем eErrorID код ошибки
    xComplete := TRUE;
    eErrorID := ERROR.NO_ERROR;
END_IF

```

```
IF NOT xEnable OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN
    (* Cleaning *)
    // освобождаем все используемые ресурсы системы
END_IF
```

Реализация AbortAction:

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

// прекращаем выполнение операции
// если произошла ошибка, то xError := TRUE и присваиваем eErrorID код ошибки
xComplete := TRUE;
eErrorID := ERROR.NO_ERROR;
```

Реализация ResetAction:

```
METHOD PROTECTED ResetAction
VAR_OUTPUT
    xComplete : BOOL;
END_VAR
// освобождаем все используемые ресурсы системы
// переинициализируем переменные
xComplete := TRUE;
```

6.3. Добавляем таймеры

Время выполнения ФБ можно ограничить с помощью таймеров. Существует три варианта ограничения:

1. **TimeLimit (TI и TIC¹⁵)** – время, затрачиваемое на выполнении операции в пределах одного цикл ПЛК, ограничивается значением входа **udiTimeLimit** (задается в микросекундах). Таким образом, сложные операции будут выполняться несколько циклов, но при этом не будут блокировать работу других задач приложения;
2. **TimeOut (To)** – общее время выполнения операции ограничивается значением входа **udiTimeOut** (задается в микросекундах);
3. Совместное использование вариантов 1 и 2 (**TITo**).

Более подробная информация приведена в [п. 3.4](#).

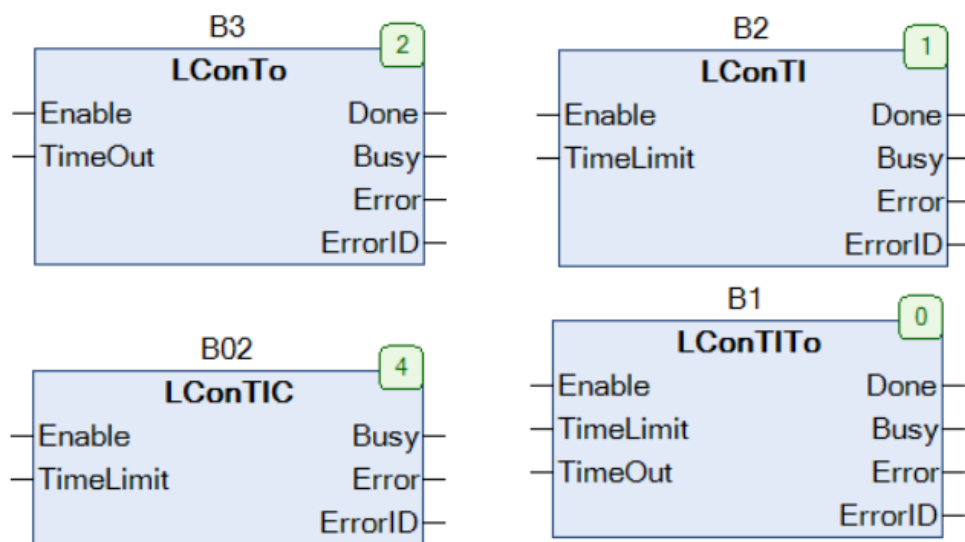


Рис. 11. Примеры ФБ типа **Enable** с таймерами

Примечание: в данных примерах названия входов/выходов приведены без префиксов.

Пример реализации ФБ LConTI на языке SFC

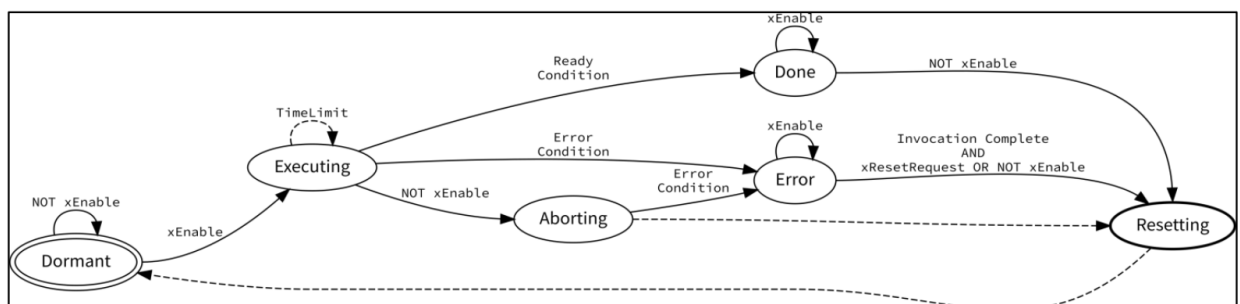
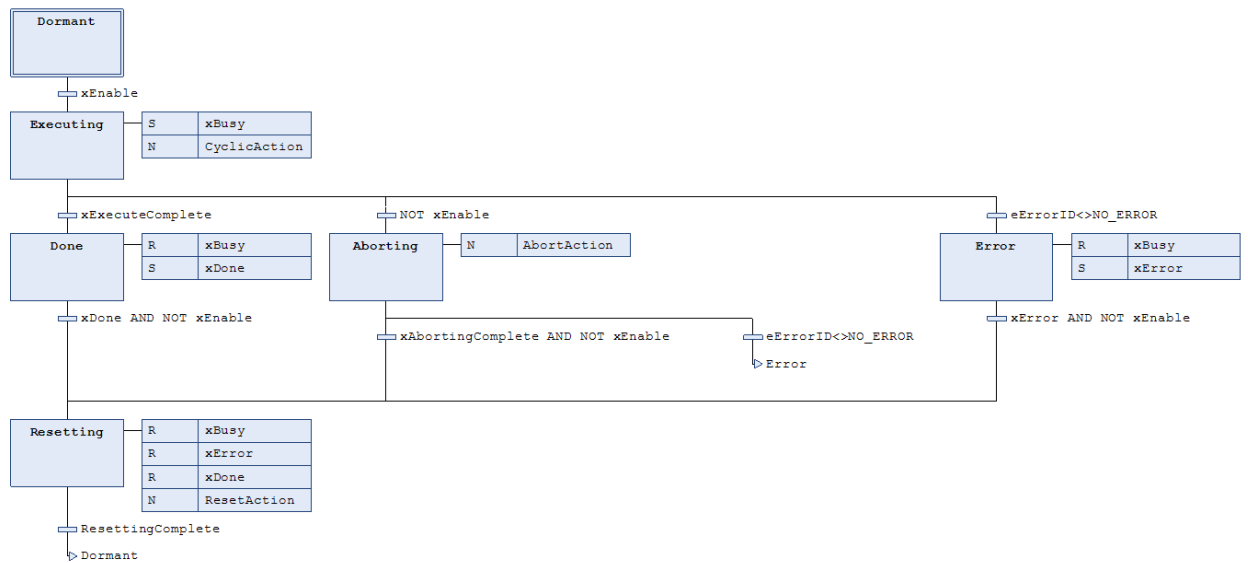


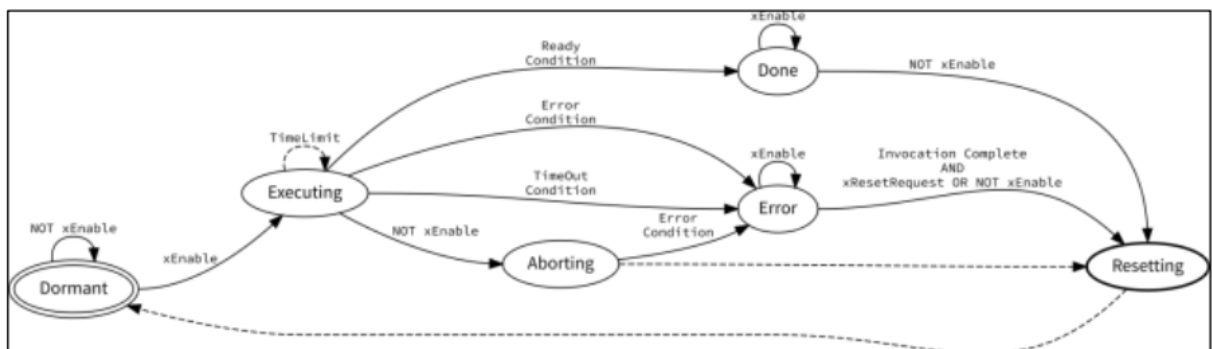
Рис. 12. Диаграмма состояний ФБ **LConTI**

¹⁵ «TIC» означает TimeLimit Continuous – такие ФБ не имеют выхода xDone (т.е. выполняемая ими операция не имеет условия завершения)

Ниже приведен пример реализации ФБ **LConTI** на языке SFC в соответствии с диаграммой состояний, изображенной на рис. 12. Каждое состояние представляет собой шаг SFC, содержащий нужные действия. Каждое действие имеет соответствующий классификатор.

Рис. 13. Пример реализации ФБ **LConTI** на языке SFC

Пример реализации ФБ **LConTIto** на языке SFC

Рис. 14. Диаграмма состояний ФБ **LConTIto**

Детальное описание принципа работы ФБ **LConTIto**

1. Функциональный блок однократно вызывается в каждом цикле ПЛК независимо от каких-либо условий;
2. По переднему фронту на входе **xEnable** блок переходит из состояния *Dormant* в состояние *Executing*;
 - Значения пользовательских входов¹⁶ не сохраняются внутри блока. Изменение значений входов в процессе работы блока будет влиять на выполняемую операцию;
 - Выход **xBusy** устанавливается в значение TRUE;
3. Начинается выполнение заданной операции;
4. Если время выполнения операции превышает значение **udiTimeLimit** (в мкс), то операция останавливается и будет продолжена при следующем вызове ФБ;

¹⁶ Речь идет о входах реального ФБ, связанных с конкретной задачей (прим. пер.).

5. После окончания операции или выполнении определенных условий блок переходит в состояние *Done*, *Error* или *Aborted*. При этом выход **xBusy** принимает значение FALSE, а выход, соответствующий новому состоянию (**xDone**, **xError** или **xAborted**) принимает значение TRUE как минимум на один цикл ПЛК. В каждый момент времени только один из статусных выходов (**xDone**, **xError** или **xAborted**) может иметь значение TRUE. По заднему фронту на выходе **xBusy** инвертированное значение входа **xEnable** копируется внутрь ФБ и будет использоваться как условие сброса ФБ (см. **xResetRequest** в методе, *HandleErrorState*). Как следует из диаграммы состояний (см. выше), сброс ФБ производится:

- при *быстром подтверждении* – сразу же после завершения операции;
- при *стандартном подтверждении* – по заднему фронту входа **xEnable**.

Описание состояний, в которые блок может перейти после выполнения операции, приведено ниже:

a. *Done*¹⁷:

Если заданная операция успешно завершена, то блок переходит в состояние *Done* и выход **xDone** принимает значение TRUE;

b. *Error*:

Если в процессе выполнения операции происходит ошибка, то блок переходит в состояние *Error*. Выход **xError** принимает значение TRUE, а на выход **eErrorID** транслируется код ошибки, определенный в перечислении **ERROR**;

c. *Timeout*:

Если общее время выполнения операции превышает значение входа **udTimeOut** (в мкс), то блок переходит в состояние *Error*. Выход **xError** принимает значение TRUE, а на выход **eErrorID** транслируется код ошибки *ERROR.TIME_OUT*;

d. *Abort*:

Если вход **xEnable** принимает значение FALSE, то блок переходит в состояние *Aborting*. Выполнение заданной операции прерывается, после чего выход **xAborted** устанавливается в значение TRUE и блок переходит в состояние *Resetting*. Если при прерывании операции возникают ошибки в работе блока, то блок переходит в состояние *Error*. Выход **xError** принимает значение TRUE, а на выход **eErrorID** транслируется код ошибки, определенный в перечислении *ERROR*;

6. После появления на выходе **xError** значения TRUE вход **xEnable** может быть опять установлен в значение TRUE (*быстрое подтверждение*);

7. Значения перечисленных выходов (**xDone** и **xError**) должны устанавливаться в TRUE как минимум на один цикл ПЛК. В каждый момент времени может быть активен только один из этих выходов;

8. Если вход **xEnable** принимает значение FALSE (*стандартное подтверждение*) или выполняется условие сброса ФБ (*быстрое подтверждение*), то блок переходит в состояние *Resetting*;

9. Все выходы инициализируются их начальными значениями. Выполнение всех операций прекращается, и все занятые ресурсы системы освобождаются. Выходы **xDone** и **xError** устанавливаются в значение FALSE;

10. После переинициализации блока происходит переход в состояние *Dormant*.

Пример реализации ФБ **LConTITo** на языке SFC идентичен реализации ФБ **LConTI** (см. рис.

13). Пример реализации на языке ST приведен в [Приложении 1.4.4](#).

¹⁷ В некоторых ситуациях может потребоваться ФБ без выхода *Done*. Примерами таких ФБ могут служить *MC_Power* из спецификации управления движением и *TCP_Server*, рассмотренный в п. 3.7. В этом случае в качестве основы для блока следует выбрать ФБ *LConC* или *LConTIC*.

Приложение 1. Спецификации ФБ типов Execute и Enable

В этом разделе представлены примеры реализации ФБ согласно модели **PLCopen**. Исходный код примеров доступен на сайте <http://www.plcopen.org> в виде текстовых файлов. Их содержимое может быть легко скопировано в любую среду разработки с поддержкой языка ST.

Приложение 1.1. Основная информация о примерах

Некоторые фрагменты примеров очень похожи. Методы с модификатором **PRIVATE FINAL** являются критически важной частью реализации и должны использоваться именно в приведенной форме. Методы с модификатором **PROTECTED** являются шаблонами, которые пользователь должен адаптировать под свои задачи.

Необходимо обратить внимание на следующие принципиальные моменты:

- Порядок обработки переменных **xExecute**, **xComplete**, **eErrorID** и **xAbort** определяет поведение блока в тех случаях, когда их значения изменяются одновременно (в одном цикле ПЛК);
- Обработка переменной **xResetRequest** определяет поведение блока при *быстром подтверждении сброса*;
- Обработка встроенного ФБ **TimingController** определяет поведение блока при превышении общего максимального времени выполнения (**xTimeOut**) или времени выполнения, выделенного на один цикл ПЛК (**xTimeLimit**);
- Если блок перешел в состояние *Abort* или *Error*, то он не может перейти в состояние *Done* до следующего перезапуска. В этих состояниях выход **eErrorID** не может иметь значение *ERROR.NO_ERROR*. При возникновении ошибки во время прерывания операции блок может перейти в состояние *Error*, при этом код ошибки **eErrorID** будет перезаписан;
- Ошибка превышения общего максимального времени выполнения ФБ (*ERROR.TIME_OUT*) имеет наименьший приоритет, и ее код не может перезаписать любую другую ошибку (**eErrorID** ≠ *ERROR.TIME_OUT*);
- Выполнение шагов, в названиях которых есть суффикс *-ing* (например, *Executing*), может занимать несколько циклов ПЛК. Они будут выполняться до тех пор, пока не перейдут в одно из своих внутренних состояний (*Done*, *Abort*, *Timeout* или *Error*).

Любая реализация модели поведения **PLCopen** должна учитывать описанные выше особенности и соответствовать спецификациям из данного документа. Чтобы избежать неверных интерпретаций, в дополнение к исходному коду примеров также приведены циклограммы ФБ.

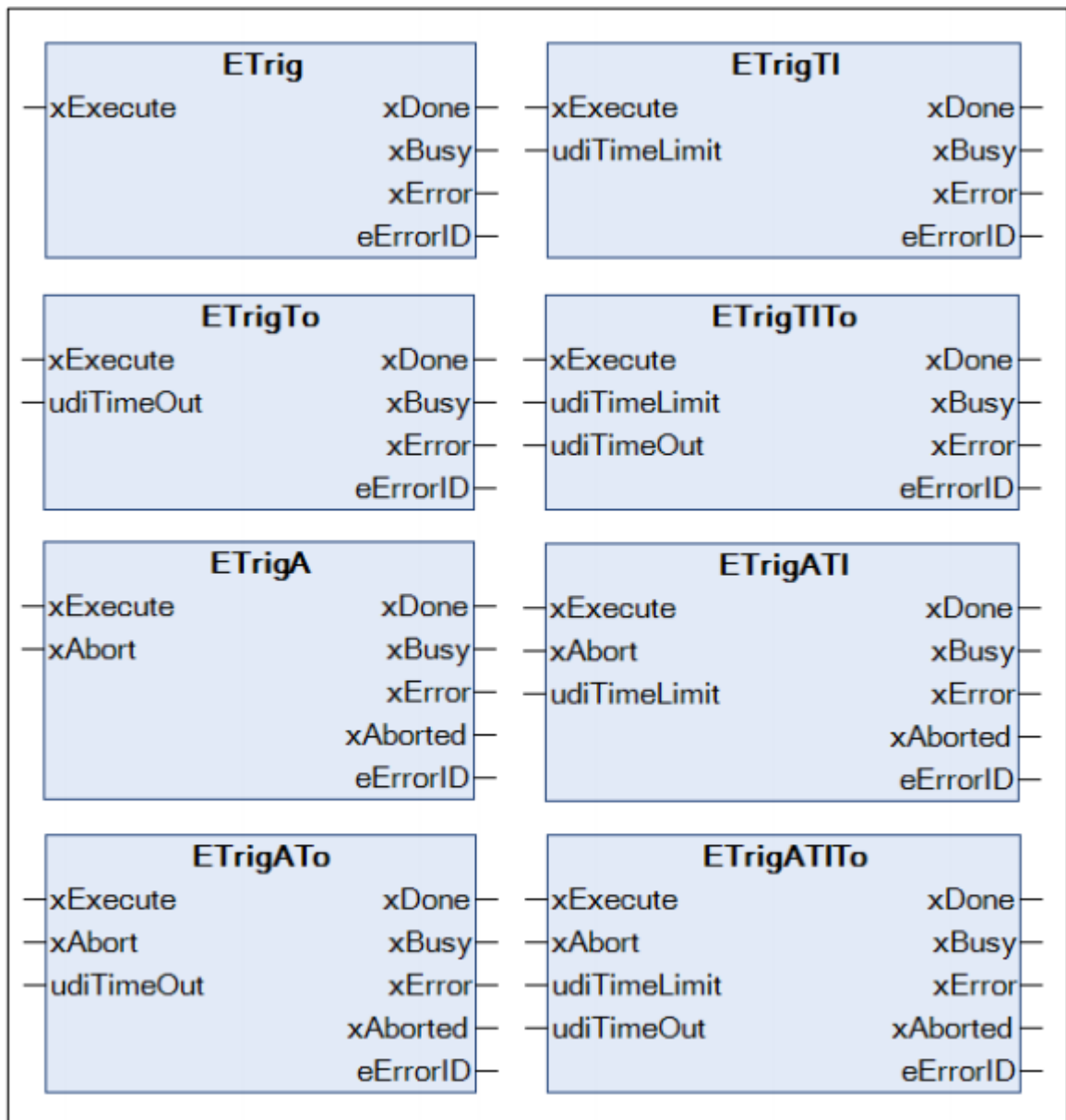
Приложение 1.2. Обзор входов/выходов

ФБ	Входы				Выходы				Описание
	Execute	TimeOut	TimeLimit	Abort	Done	Busy	Error/ErrorID	Aborted	
ETrig	+				+	+	+		Базовый ФБ типа Execute.
ETrigTI	+		+		+	+	+		ФБ типа Execute со входом TimeLimit.
ETrigTo	+	+			+	+	+		ФБ типа Execute со входом TimeOut.
ETrigTITo	+	+	+		+	+	+		ФБ типа Execute со входами TimeLimit и TimeOut.
ETrigA	+			+	+	+	+	+	ФБ типа Execute со входом Abort.
ETrigATI	+		+	+	+	+	+	+	ФБ типа Execute со входами Abort и TimeLimit.
ETrigATo	+	+		+	+	+	+	+	ФБ типа Execute со входами Abort и TimeOut.
ETrigATITo	+	+	+	+	+	+	+	+	ФБ типа Execute со входами Abort, TimeLimit и TimeOut.

ФБ	Входы			Выходы			Описание
	Enable	TimeOut	TimeLimit	Busy	Error/ErrorID	Done	
LCon	+			+	+	+	Базовый ФБ типа Enable.
LConTI	+		+	+	+	+	Базовый ФБ типа Enable со входом TimeLimit.
LConTo	+	+		+	+	+	Базовый ФБ типа Enable со входом TimeOut.
LConTITo	+	+	+	+	+	+	Базовый ФБ типа Enable со входами TimeLimit и TimeOut.
LConC	+			+	+		Базовый ФБ типа Enable Continuous ¹⁸ .
LConTIC	+		+	+	+		Базовый ФБ типа Enable Continuous ¹⁷ со входом TimeLimit.

¹⁸ Такие ФБ не имеют выхода xDone (т.е. выполняемая ими операция не имеет условия завершения).

Приложение 1.3. Обзор ФБ типа Execute

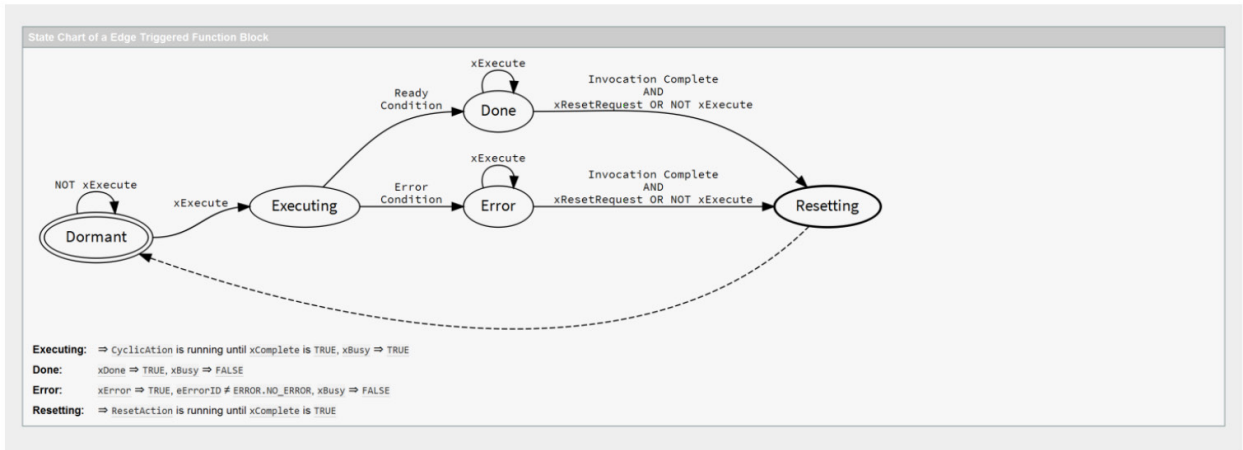


Примечание: на практике перечисления STATE и ERROR являются общими для библиотеки, а не объявляются для каждого ФБ, как в примерах ниже.

Приложение 1.3.1. ФБ ETrig

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

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
        
```

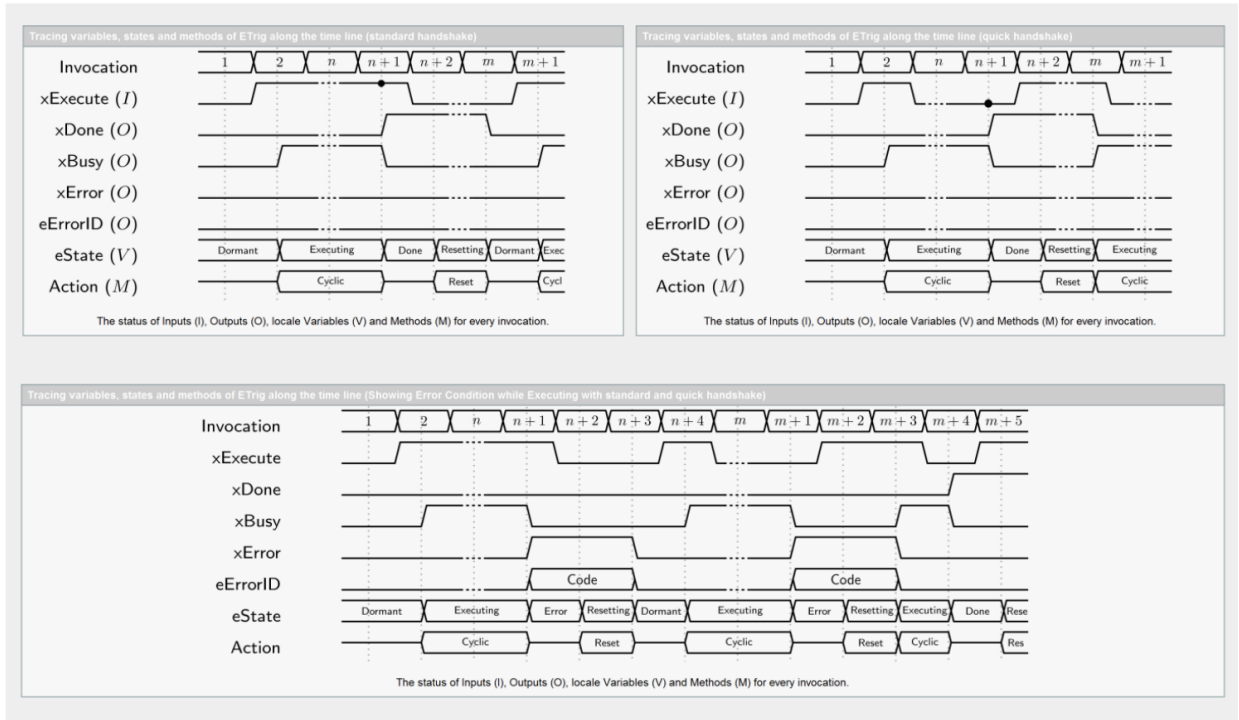
Implementation of the Function Block ETrig

```

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
27 REPEAT
28  xAgain := FALSE;
29  CASE eState OF
30    STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
31    STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
32    STATE.DONE: HandleDoneState(xAgain⇒xAgain);
33    STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
34    STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
35  END_CASE
36 UNTIL NOT xAgain
37 END_REPEAT;
        
```

<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Resetting State</p> <pre> 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 </pre>	<p>The Handler of the Resetting State</p> <pre> 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary implementation)</p> <pre> 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 (* clearing *) 25 // if possible free as much allocated resources 26 // as possible 27 END_IF </pre>	<p>The Implementation of the Reset Action (Exemplary implementation)</p> <pre> 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; </pre>

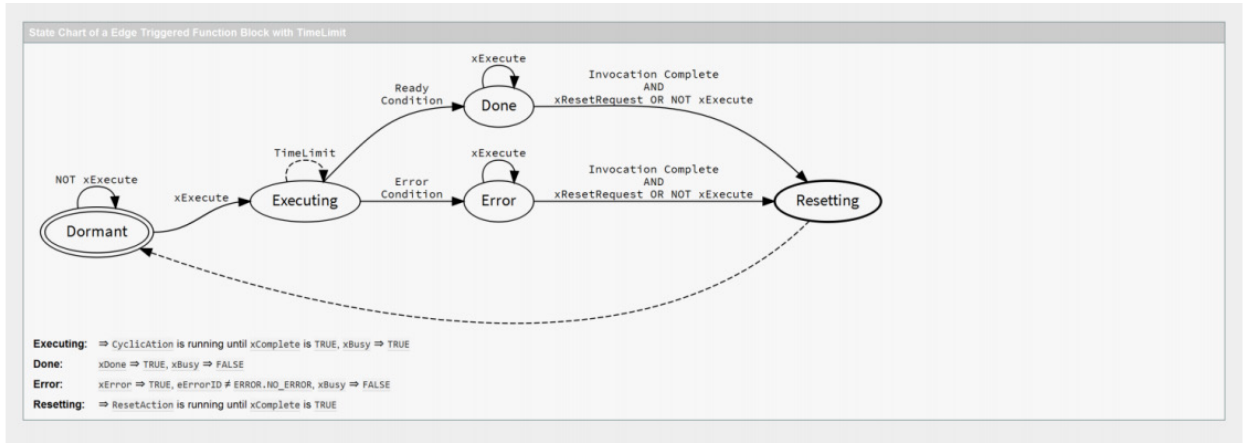
Циклограмма



Приложение 1.3.2. ФБ ETrigTI

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of ETrigTI

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
        
```

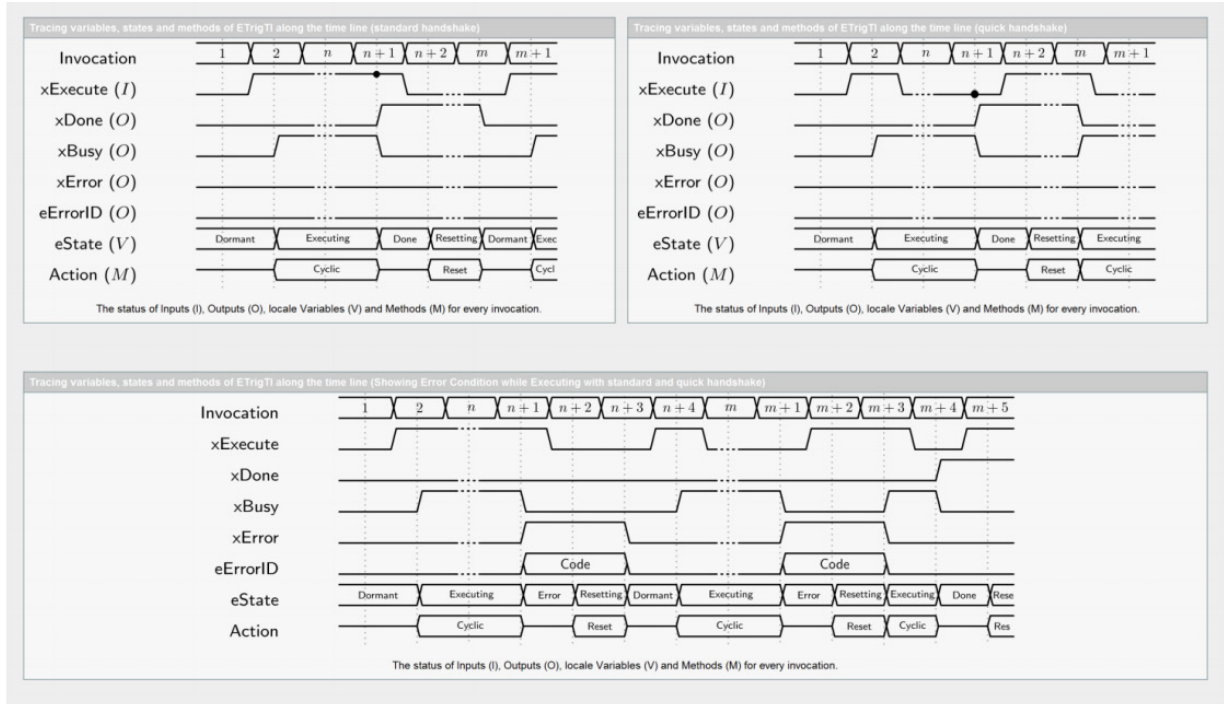
Implementation of the Function Block ETrigTI

```

1  FUNCTION_BLOCK ETrigTI
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 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 REPEAT
31 xAgain := FALSE;
32 CASE eState OF
33 STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
34 STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
35 STATE.DONE: HandleDoneState(xAgain⇒xAgain);
36 STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
37 STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
38 END_CASE
39 UNTIL NOT xAgain
40 END_REPEAT;
        
```


<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 tTimingController.StartInvocationTimer(); 11 12 CyclicAction(13 xComplete=>xComplete, 14 eErrorID=>eErrorID 15); 16 17 IF eErrorID <> ERROR.NO_ERROR THEN 18 eState := STATE.ERROR; 19 xAgain := TRUE; 20 ELSIF xComplete THEN 21 eState := STATE.DONE; 22 xAgain := TRUE; 23 END_IF </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
	<p>The Handler of the Resetting State</p> <pre> 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 xFirstInvocation THEN 11 (* Starting *) 12 // for the first (!) invocation, 13 // sample the input variables 14 tTimingController.TimeLimit := udTimeLimit; 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 tTimingController.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 </pre>	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

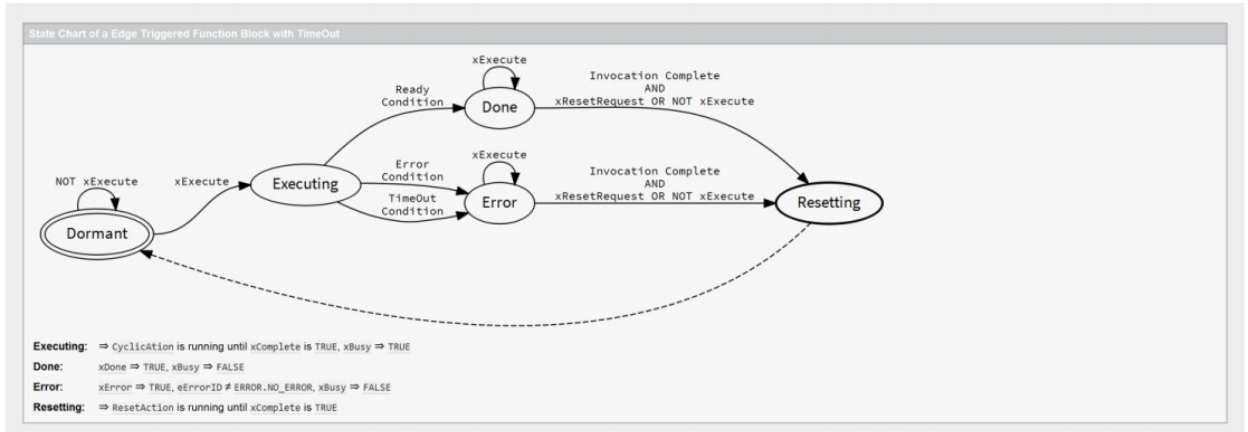
Циклограмма



Приложение 1.3.3. ФБ ETrigTo

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of ETrigTo

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
        
```

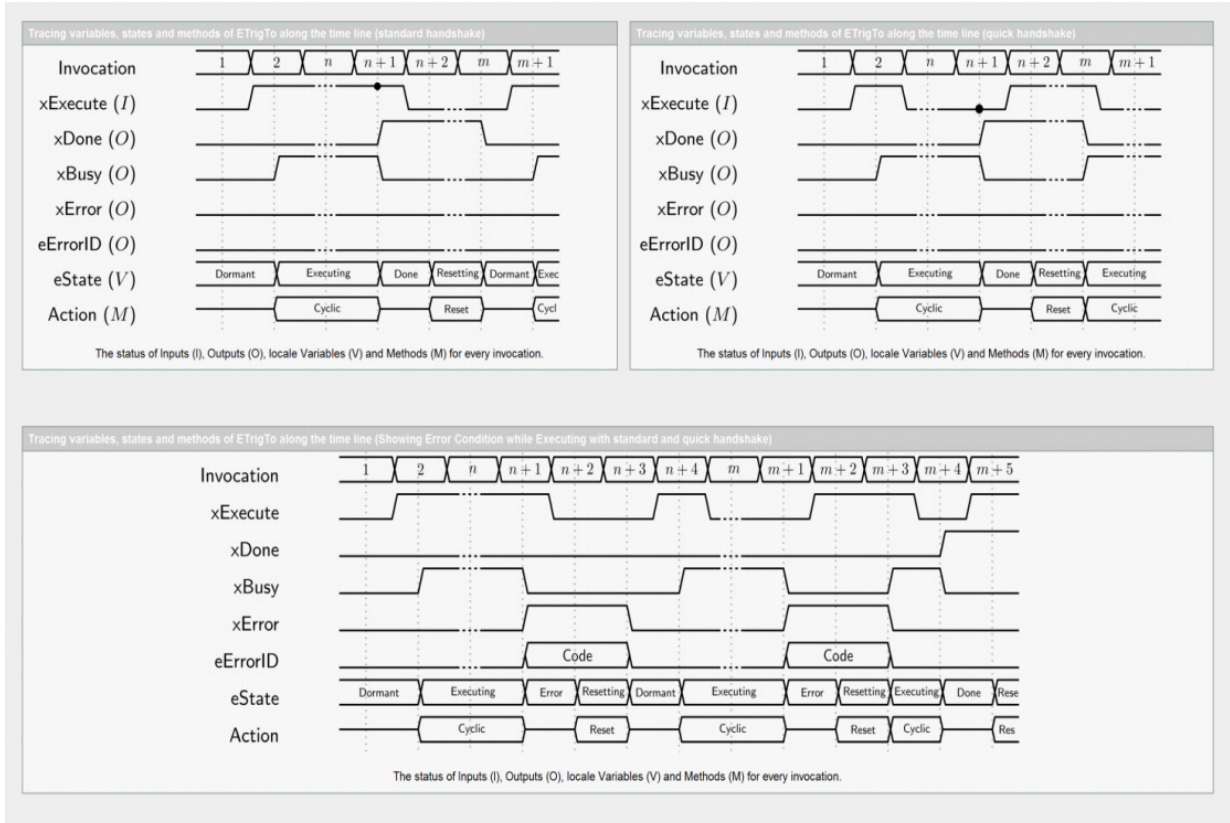
Implementation of the Function Block ETrigTo

```

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   tTimingController: TimingController;
23   eState: STATE;
24   xFirstInvocation: BOOL := TRUE;
25   xResetRequest: BOOL;
26 END_VAR
27 VAR_TEMP
28   xAgain: BOOL;
29 END_VAR
30 REPEAT
31   xAgain := FALSE;
32   CASE eState OF
33     STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
34     STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
35     STATE.DONE: HandleDoneState(xAgain⇒xAgain);
36     STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
37     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
38   END_CASE
39 UNTIL NOT xAgain
40 END_REPEAT;
        
```

<p>The Handler for the Dormant State</p> <pre> 1 METHOD PRIVATE FINAL HandleDormantState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 6 IF xEnable THEN 7 tctTimingController.StartOperationTimer(); 8 xBusy := TRUE; 9 eState := STATE.EXECUTING; 10 xAgain := TRUE; 11 END_IF </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 tctTimingController.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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 tctTimingController.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 tctTimingController.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 </pre>	<p>The Handler of the Resetting State</p> <pre> 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 </pre>
<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>	

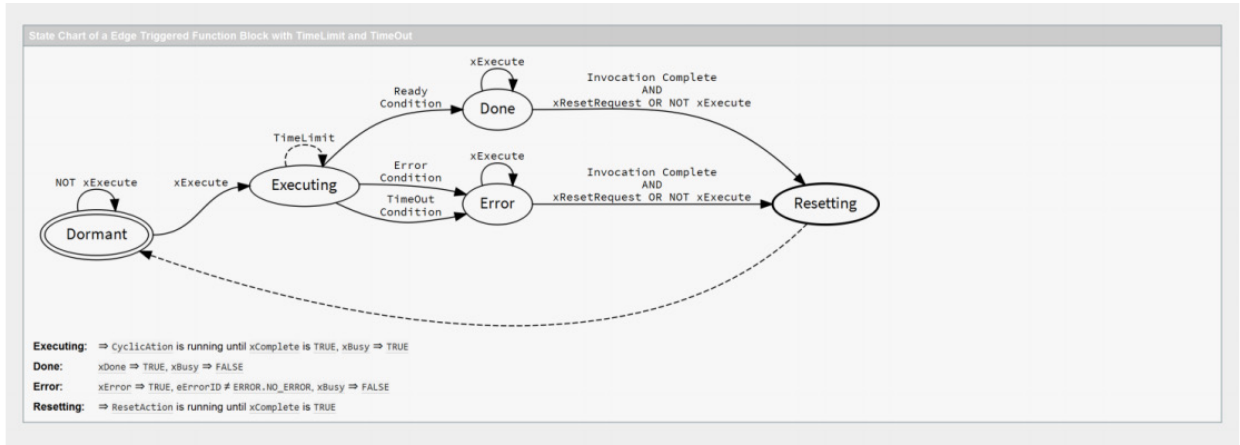
Циклограмма



Приложение 1.3.4. ФБ ETrigTtTo

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of ETrigTtTo

ETrigTtTo (FB)
 BehaviourModel
 CyclicAction
 ResetAction
 StateMachine
 HandleDoneState
 HandleDormantState
 HandleErrorState
 HandleExecutingState
 HandleResettingState

ETrigTtTo
 -xExecute xDone
 -udiTimeLimit xBusy
 -udiTimeOut xError
 eErrorID

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
        
```

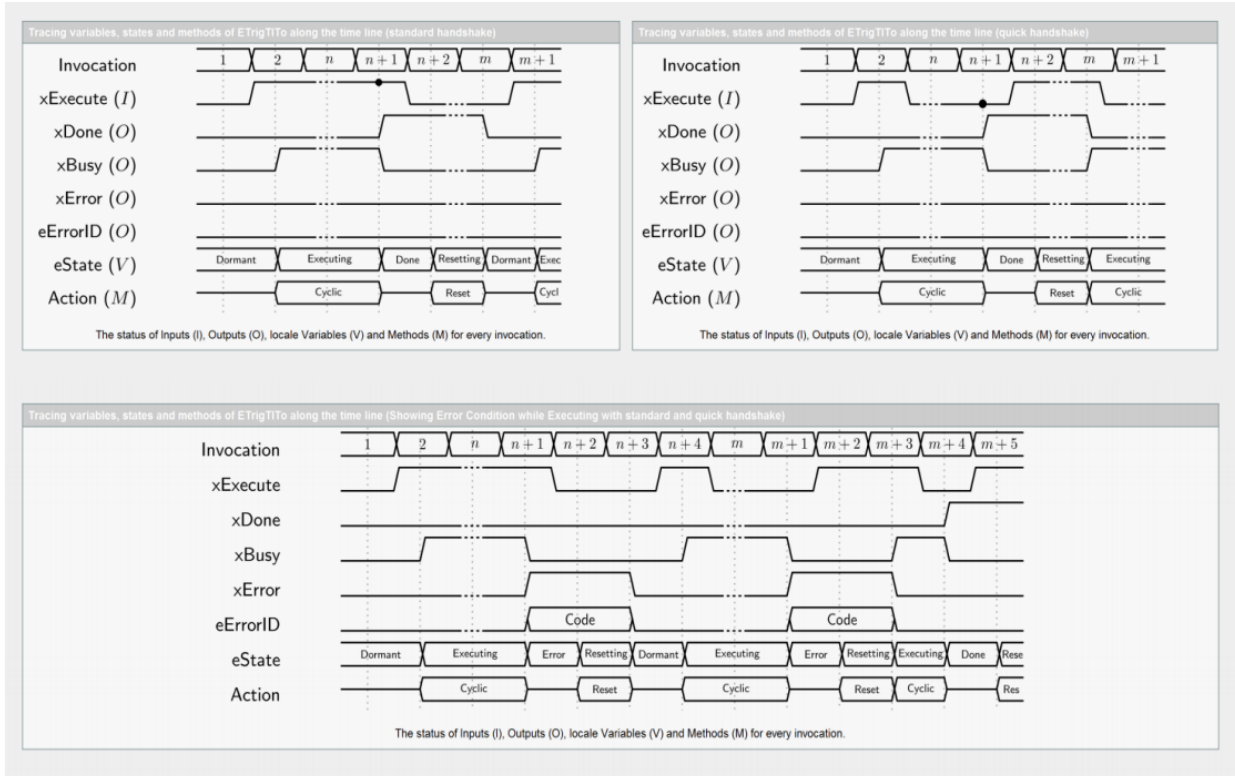
Implementation of the Function Block ETrigTtTo

```

1 FUNCTION_BLOCK ETrigTtTo
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   tTimingController: 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;
        
```

<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 </pre>	<p>The Handler of the Resetting State</p> <pre> 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 </pre>
<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

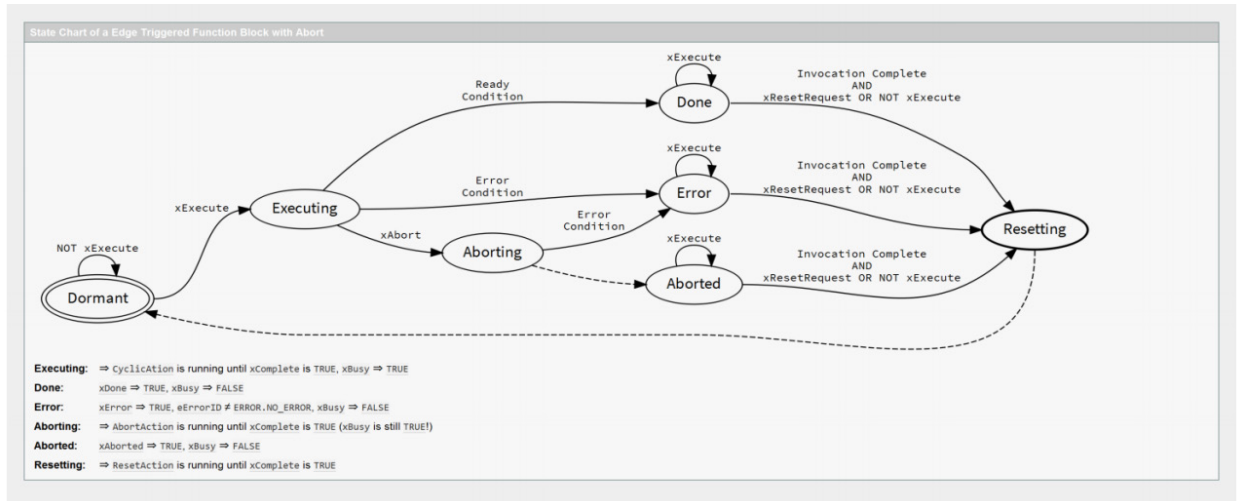
Циклограмма



Приложение 1.3.5. ФБ ETrigA

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of ETrigA

The STATE Enumeration

```

1 TYPE STATE :
2 (
3   DORMANT, // waiting for xExecute
4   EXECUTING, // Cycllication 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
        
```

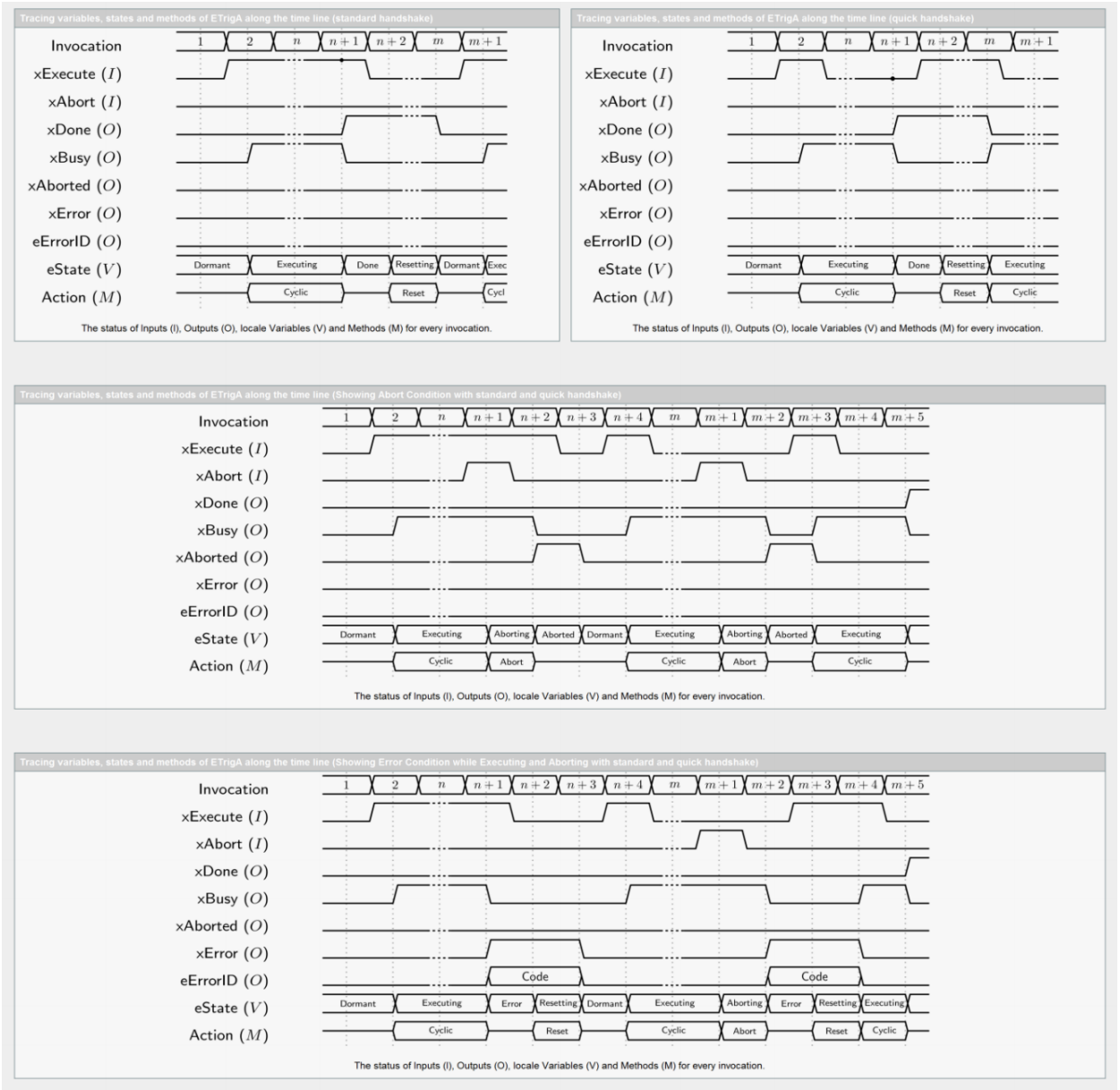
Implementation of the Function Block ETrigA

```

1 FUNCTION_BLOCK ETrigA
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 END_VAR
10 VAR_OUTPUT
11 // ready condition reached
12 xDone : BOOL;
13 // operation is running
14 xBusy : BOOL;
15 // abort condition reached
16 xAborted : BOOL;
17 // error condition reached
18 xError : BOOL;
19 // error code describing error condition
20 eErrorID : ERROR;
21 END_VAR
22 VAR
23   eState : STATE;
24   xFirstInvocation : BOOL := TRUE;
25   xResetRequest : BOOL;
26 END_VAR
27 VAR_TEMP
28   xAgain : BOOL;
29 END_VAR
30 REPEAT
31   xAgain := FALSE;
32   CASE eState OF
33     STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
34     STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
35     STATE.DONE: HandleDoneState(xAgain⇒xAgain);
36     STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
37     STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
38     STATE.ABORTED: HandleAbortedState(xAgain⇒xAgain);
39     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
40   END_CASE
41 UNTIL NOT xAgain
42 END_REPEAT;
        
```

<p>The Handler for the Dormant State</p> <pre> 1 METHOD PRIVATE FINAL HandleDormantState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 6 IF xExecute THEN 7 xBusy := TRUE; 8 eState := xSTATF.FXFRUITING; 9 xAgain := TRUE; 10 END_IF </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 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 </pre>	<p>The Handler for the Aborted State</p> <pre> 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 </pre>
<p>The Handler for the Resetting State</p> <pre> 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 </pre>	<p>The Handler for the Resetting State</p> <pre> 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 </pre>	<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 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; </pre>
<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

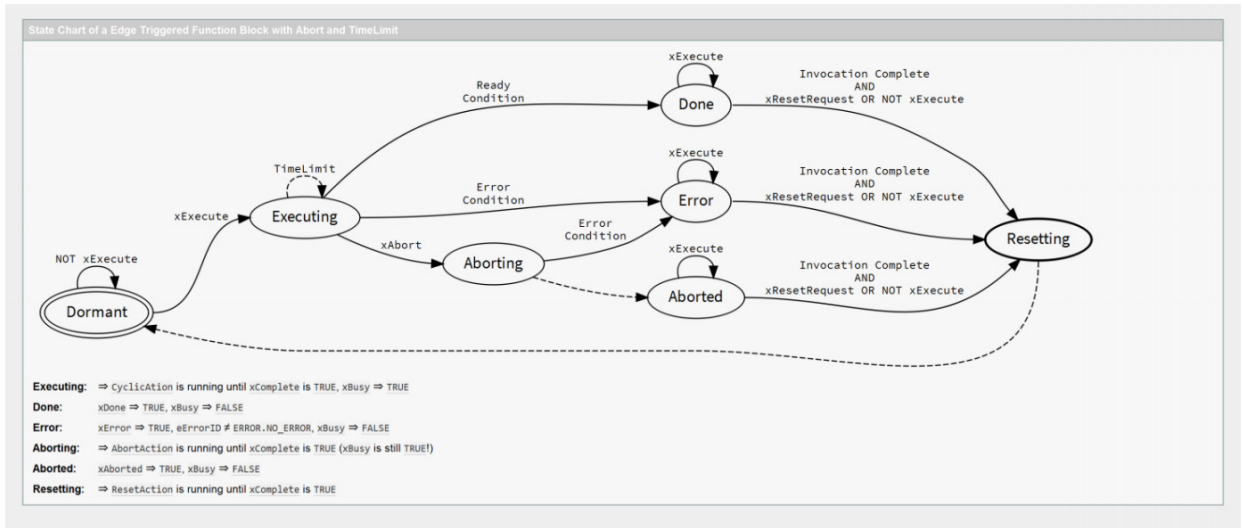
Циклограмма



Приложение 1.3.6. ФБ ETrigATI

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of ETrigATI

- ETrigATI (FB)
 - BehaviourModel
 - AbortAction
 - CyclicAction
 - ResetAction
 - StateMachine
 - HandleAbortedState
 - HandleAbortingState
 - HandleDoneState
 - HandleDormantState
 - HandleErrorState
 - HandleExecutingState
 - HandleResettingState

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
    
```

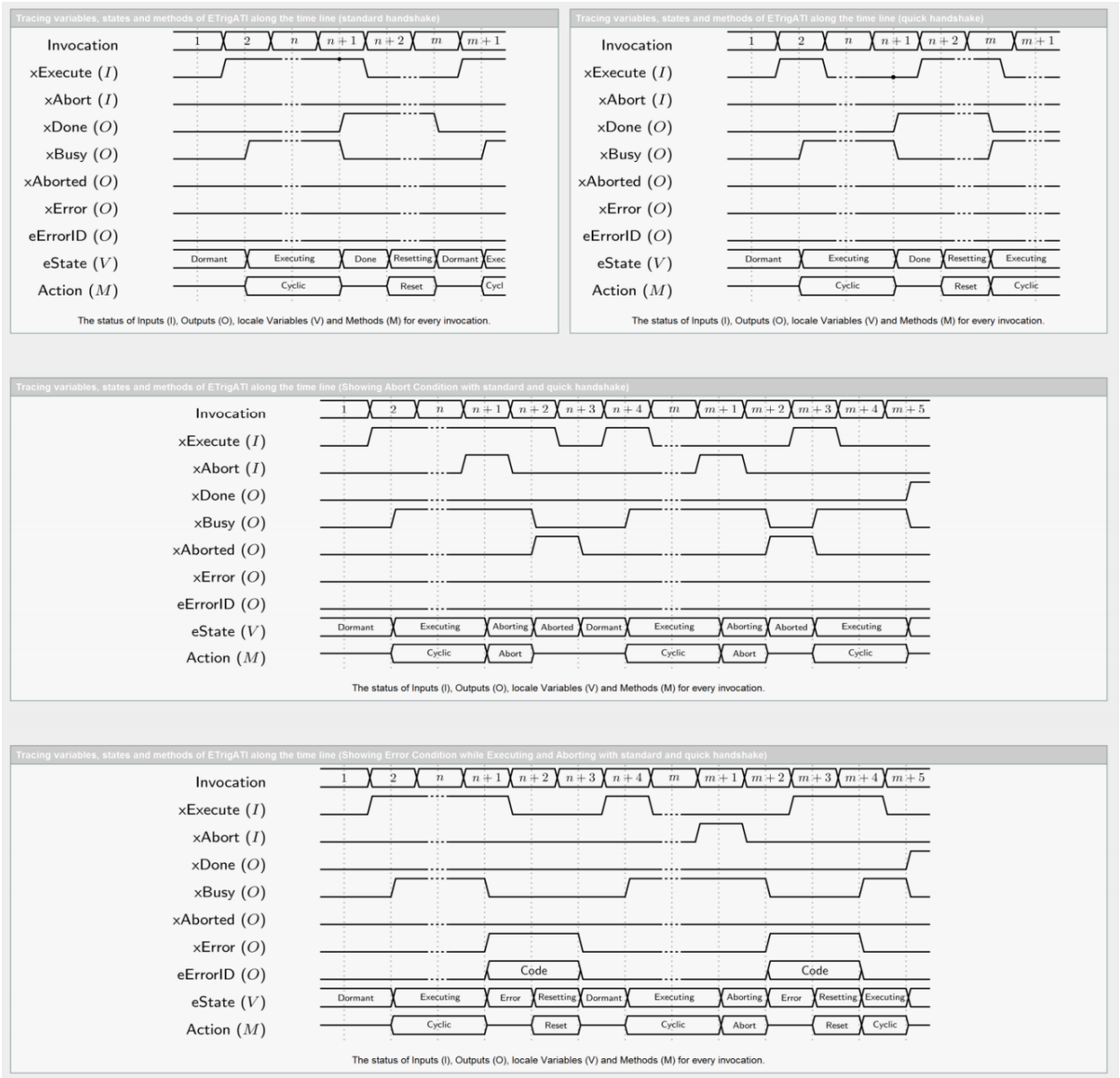
Implementation of the Function Block ETrigATI

```

1  FUNCTION_BLOCK ETrigATI
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 per invocation
10     // [µs], 0 ⇒ no operating time limit
11     udiTimeLimit: 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     tTimingController : TimingController;
27     eState: STATE;
28     xFirstInvocation: BOOL := TRUE;
29     xResetRequest: BOOL;
30 END_VAR
31 VAR_TEMP
32     xAgain: BOOL;
33 END_VAR
34
35 REPEAT
36     xAgain := FALSE;
37     CASE eState OF
38         STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
39         STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
40         STATE.DONE: HandleDoneState(xAgain⇒xAgain);
41         STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
42         STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
43         STATE.ABORTED: HandleAbortedState(xAgain⇒xAgain);
44         STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
45     END_CASE
46 UNTIL NOT xAgain
47 END_REPEAT;
    
```

<p>The Handler for the Dormant State</p> <pre> 1 METHOD PRIVATE FINAL HandleDormantState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 VAR 6 IF xExecute THEN 7 xBusy := TRUE; 8 eState := STATE.EXECUTING; 9 xAgain := TRUE; 10 END_IF </pre>	<p>The Handler for the Done State</p> <pre> 1 METHOD PRIVATE FINAL HandleDoneState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 VAR 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 </pre>
<p>The Handler for the Executing State</p> <pre> 1 METHOD PRIVATE FINAL HandleExecutingState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 VAR 6 xComplete : BOOL; 7 END_VAR 8 IF NOT xAbort THEN 9 tcTimingController.StartInvocationTimer(); 10 CyclicAction(11 xComplete=>xComplete, 12 eErrorID=>eErrorID 13); 14 END_IF 15 IF eErrorID <> ERROR.NO_ERROR THEN 16 eState := STATE.ERROR; 17 xAgain := TRUE; 18 ELSEIF xAbort THEN 19 eState := STATE.ABORTING; 20 xAgain := TRUE; 21 ELSEIF xComplete THEN 22 eState := STATE.DONE; 23 xAgain := TRUE; 24 END_IF </pre>	<p>The Handler for the Error State</p> <pre> 1 METHOD PRIVATE FINAL HandleErrorState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 VAR 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 1 METHOD PRIVATE FINAL HandleAbortingState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 VAR 6 xComplete : BOOL; 7 END_VAR 8 AbortAction(9 xComplete=>xComplete, 10 eErrorID=>eErrorID 11); 12 IF eErrorID <> ERROR.NO_ERROR THEN 13 eState := STATE.ERROR; 14 xAgain := TRUE; 15 ELSEIF xComplete THEN 16 eState := STATE.ABORTED; 17 xAgain := TRUE; 18 END_IF </pre>	<p>The Handler for the Aborted State</p> <pre> 1 METHOD PRIVATE FINAL HandleAbortedState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 VAR 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 IF NOT xAbort THEN 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 REPEAT 18 (* Executing *) 19 // working to reach the ready condition 20 // => xComplete := TRUE 21 // if the maximum invocation time is reached 22 // => xTimeLimit := TRUE 23 // if an error condition is reached 24 // => set eErrorID to a value other than ERROR.NO_ERROR 25 tcTimingController.CheckTiming(26 xTimeLimit=>xTimeLimit 27); 28 xComplete := TRUE; 29 eErrorID := ERROR.NO_ERROR; 30 UNTIL xAbort OR xComplete OR xTimeLimit OR 31 eErrorID <> ERROR.NO_ERROR 32 END_REPEAT 33 END_IF 34 IF xAbort OR xComplete OR eErrorID <> ERROR.NO_ERROR THEN 35 (* Cleaning *) 36 // if possible free as much allocated resources 37 // as possible 38 END_IF </pre>	<p>The Handler of the Resetting State</p> <pre> 1 METHOD PRIVATE FINAL HandleResettingState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 VAR 6 xComplete : BOOL; 7 END_VAR 8 ResetAction(xComplete=>xComplete); 9 IF xComplete THEN 10 xBusy := FALSE; 11 xDone := FALSE; 12 xError := FALSE; 13 xAborted := FALSE; 14 eErrorID := ERROR.NO_ERROR; 15 eState := STATE.DORMANT; 16 xFirstInvocation := TRUE; 17 xAgain := xResetRequest; (* !!! *) 18 xResetRequest := FALSE; 19 END_IF </pre>
<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 1 METHOD PROTECTED AbortAction 2 VAR_OUTPUT 3 xComplete : BOOL; 4 eErrorID : ERROR; 5 END_VAR 6 // abort all running operations 7 // if an error condition is reached set 8 // eErrorID to a value other than ERROR.NO_ERROR 9 xComplete := TRUE; 10 eErrorID := ERROR.NO_ERROR; </pre>	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 1 METHOD PROTECTED ResetAction 2 VAR_OUTPUT 3 xComplete : BOOL; 4 END_VAR 5 // free all allocated resources 6 // reinitialize instance variables 7 xComplete := TRUE; </pre>

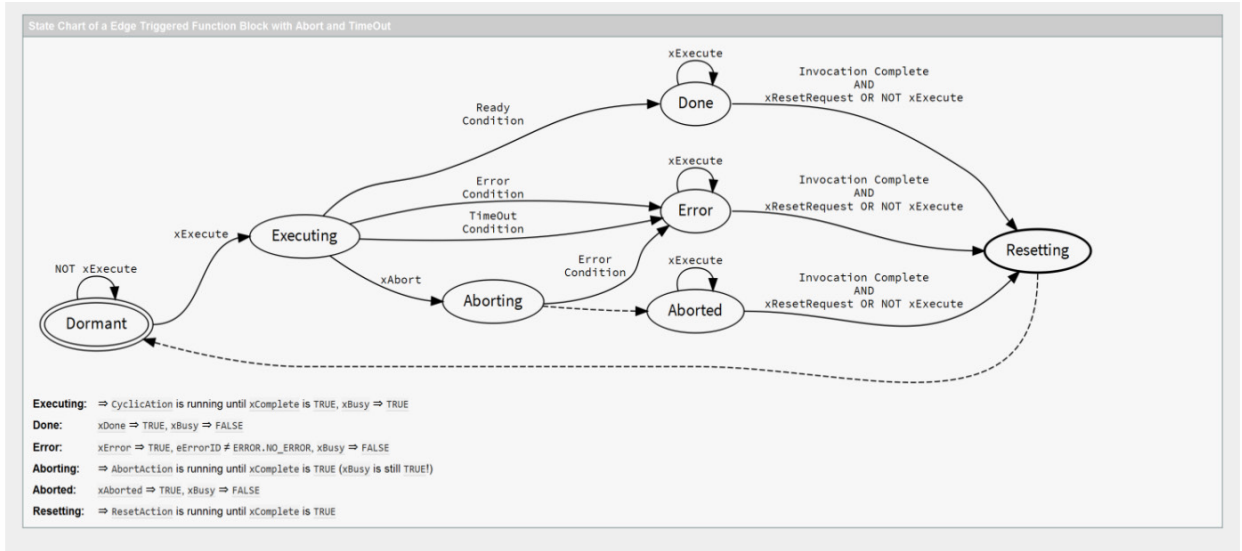
Циклограмма



Приложение 1.3.7. ФБ ETrigATo

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of ETrigATo

- ETrigATo (FB)
 - BehaviourModel
 - AbortAction
 - CyclicAction
 - ResetAction
 - StateMachine
 - HandleAbortedState
 - HandleAbortingState
 - HandleDoneState
 - HandleDormantState
 - HandleErrorState
 - HandleExecutingState
 - HandleResettingState

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
    
```

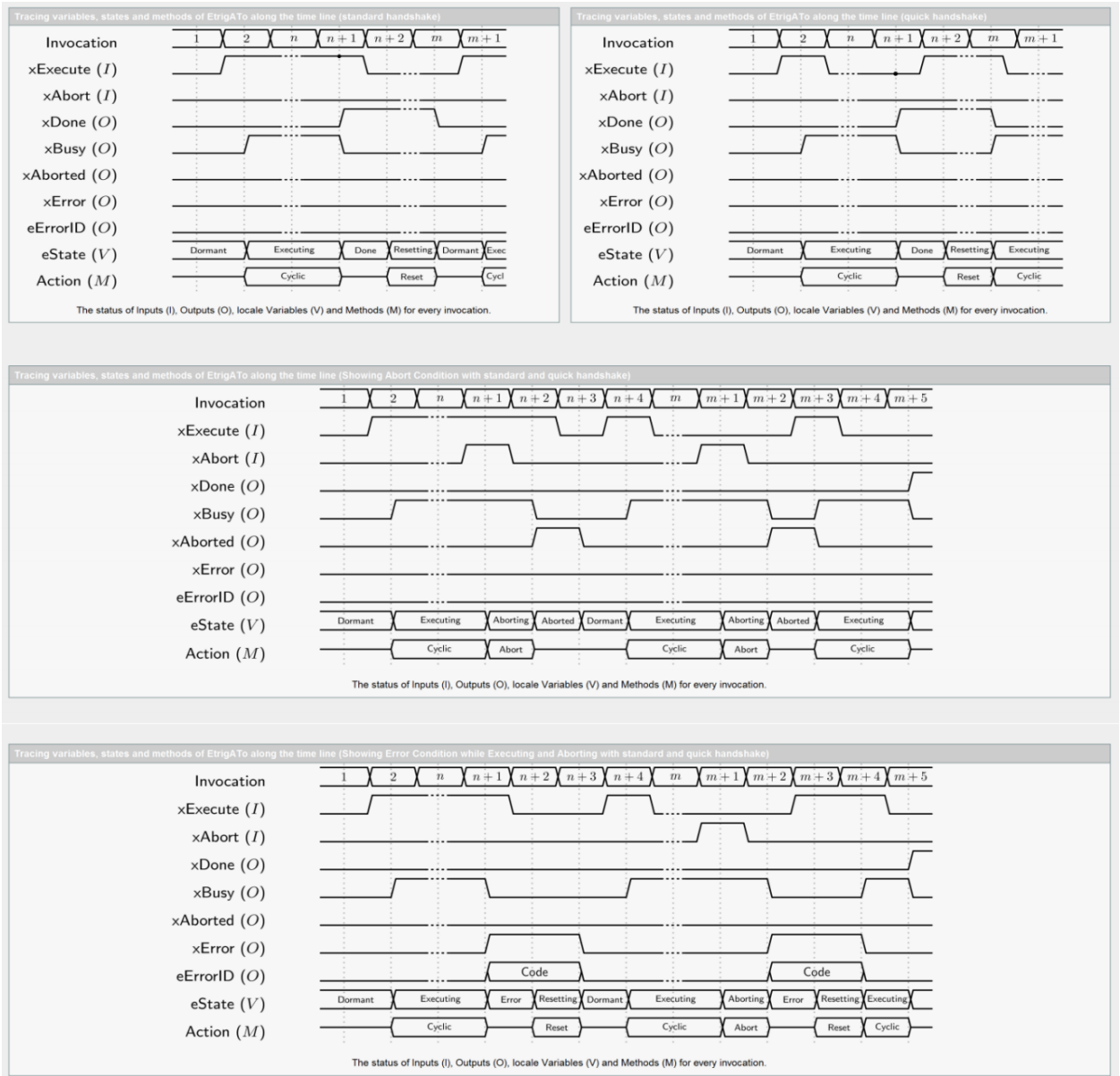
Implementation of the Function Block ETrigATo

```

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 // [ms], 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 tTimingController: TimingController;
27 eState: STATE;
28 xFirstInvocation: BOOL := TRUE;
29 xResetRequest: BOOL;
30 END_VAR
31 VAR_TEMP
32 xAgain: BOOL;
33 END_VAR
34
35 REPEAT
36 xAgain := FALSE;
37 CASE eState OF
38 STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
39 STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
40 STATE.DONE: HandleDoneState(xAgain⇒xAgain);
41 STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
42 STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
43 STATE.ABORTED: HandleAbortedState(xAgain⇒xAgain);
44 STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
45 END_CASE
46 UNTIL NOT xAgain
47 END_REPEAT;
    
```

<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 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 </pre>	<p>The Handler for the Aborted State</p> <pre> 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 </pre>
<p>The Handler for the Resetting State</p> <pre> 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.IDLE; 18 xFirstInvocation := TRUE; 19 xAgain := xResetRequest; (* !!! *) 20 xResetRequest := FALSE; 21 END_IF </pre>	<p>The Handler for the Resetting State</p> <pre> 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.IDLE; 18 xFirstInvocation := TRUE; 19 xAgain := xResetRequest; (* !!! *) 20 xResetRequest := FALSE; 21 END_IF </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 </pre>	<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 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; </pre>
<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

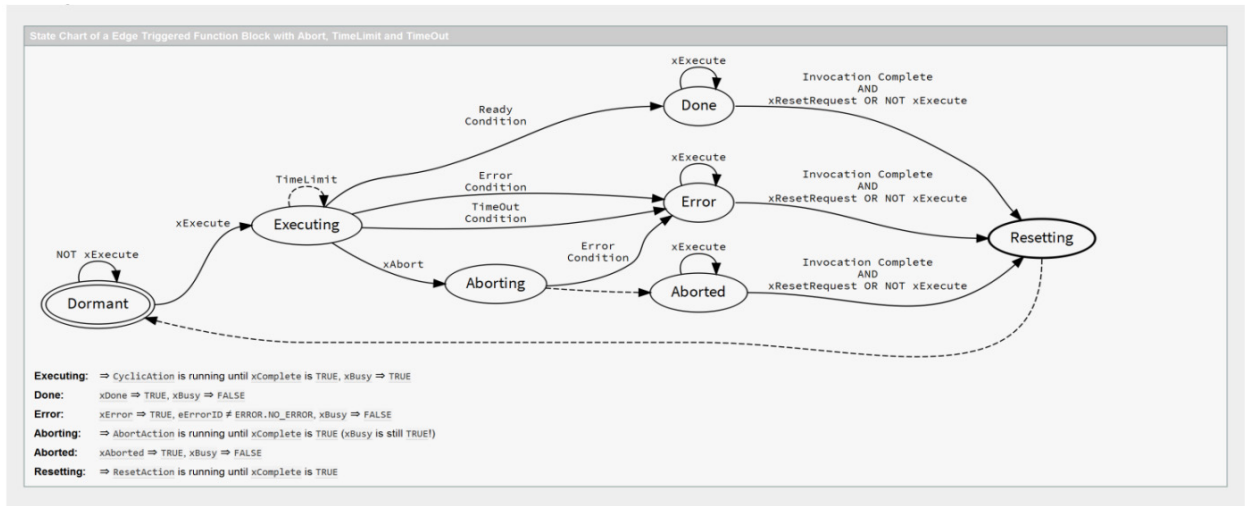
Циклограмма



Приложение 1.3.8. ФБ ETrigATITo

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of ETrigATITo

- ETrigATITo (FB)
 - BehaviourModel
 - AbortAction
 - CyclicAction
 - ResetAction
 - StateMachine
 - HandleAbortedState
 - HandleAbortingState
 - HandleDoneState
 - HandleDormantState
 - HandleErrorState
 - HandleExecutingState
 - HandleResettingState

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
    
```

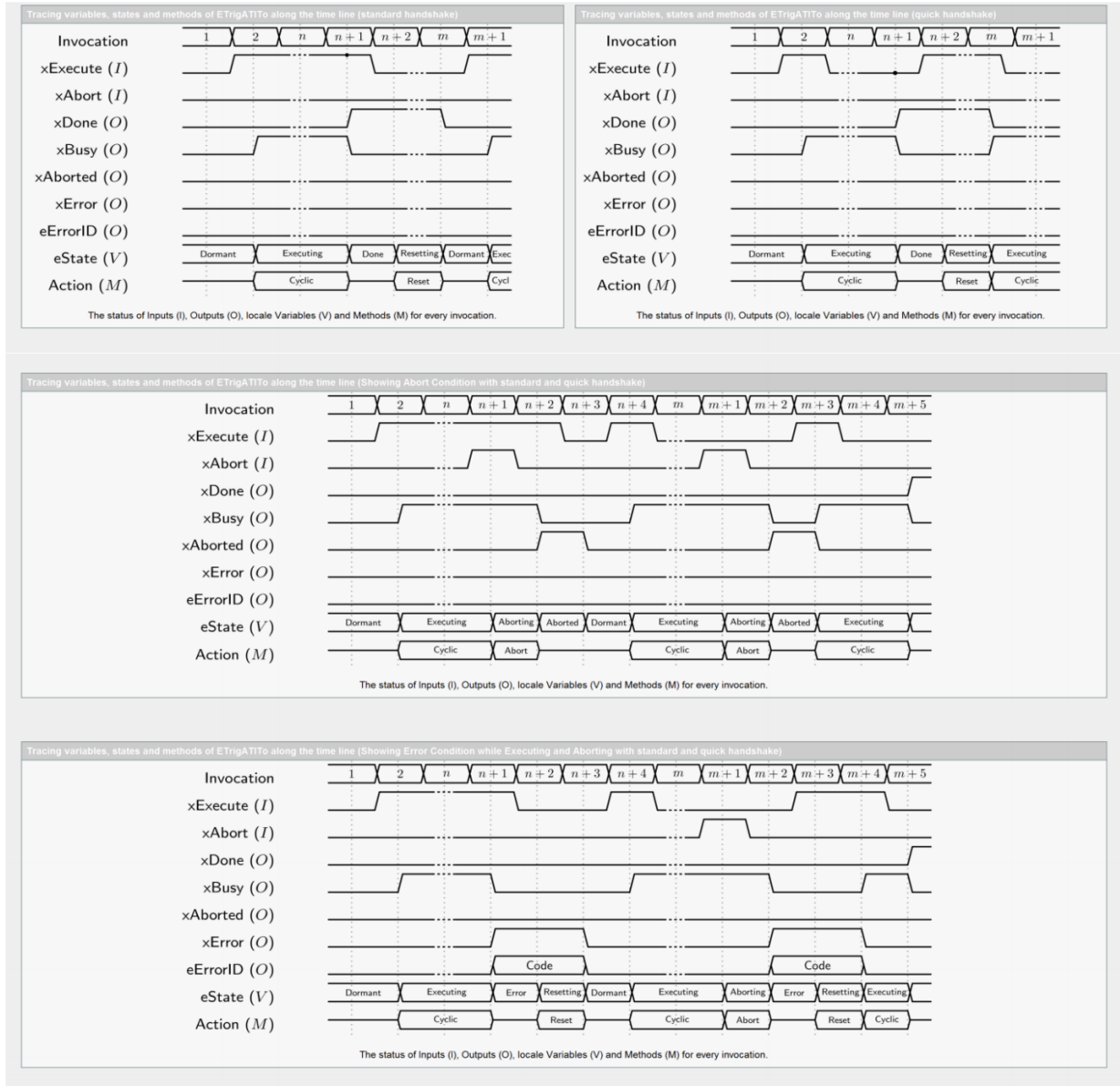
Implementation of the Function Block ETrigATITo

```

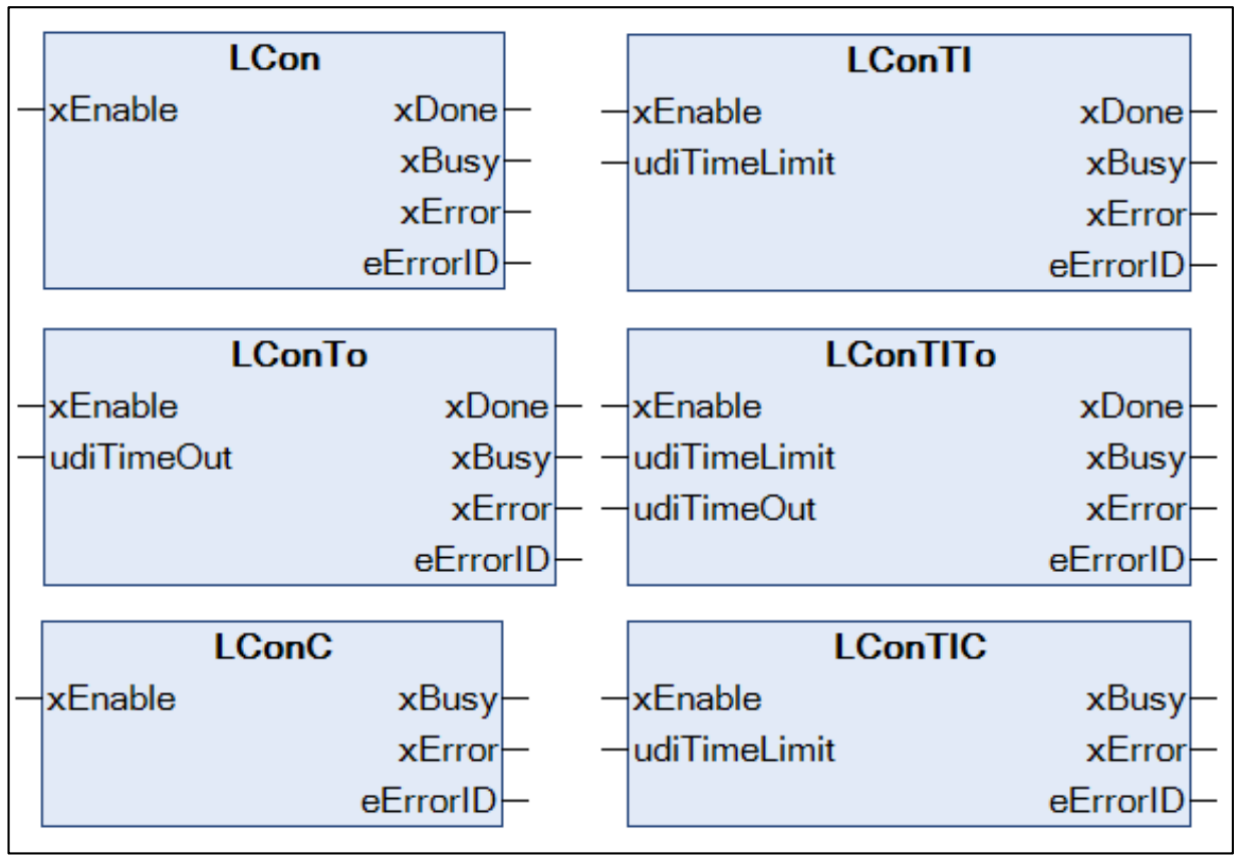
1 FUNCTION_BLOCK ETrigATITo
2 VAR_INPUT
3   // Rising edge starts defined operation
4   // FALSE ⇒ resets the defined operation
5   // after ready condition was reached
6   xExecute: BOOL;
7   // command for abort the operation
8   xAbort: BOOL;
9   // max operating time per invocation
10  // [µs], 0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

<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 tcTimingController.CheckTiming(xTimeout=>xTimeout); 18 END_IF 19 20 IF xTimeout AND eErrorID = ERROR.NO_ERROR THEN 21 eErrorID := ERROR.TIME_OUT; 22 END_IF 23 24 IF eErrorID <> ERROR.NO_ERROR THEN 25 eState := STATE.ERROR; 26 xAgain := TRUE; 27 ELSEIF xAbort THEN 28 eState := STATE.ABORTING; 29 xAgain := TRUE; 30 ELSEIF xComplete THEN 31 eState := STATE.DONE; 32 xAgain := TRUE; 33 END_IF 34 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 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 </pre>	<p>The Handler for the Aborted State</p> <pre> 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 </pre>
<p>The Handler for the Resetting State</p> <pre> 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 </pre>	<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 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; </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 </pre>	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

Циклограмма



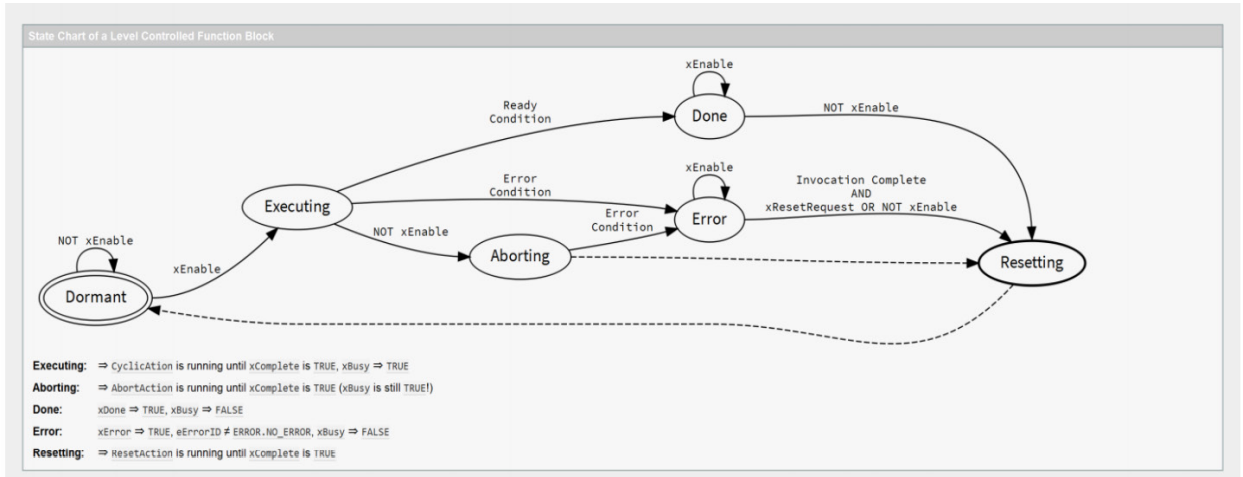
Приложение 1.4. Обзор ФБ типа Enable



Приложение 1.4.1. ФБ LCon

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of LCon

- LCon (FB)
 - BehaviourModel
 - AbortAction
 - CyclicAction
 - ResetAction
 - StateMachine
 - HandleAbortingState
 - HandleDoneState
 - HandleDormantState
 - HandleErrorState
 - HandleExecutingState
 - HandleResettingState

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 LCon

```

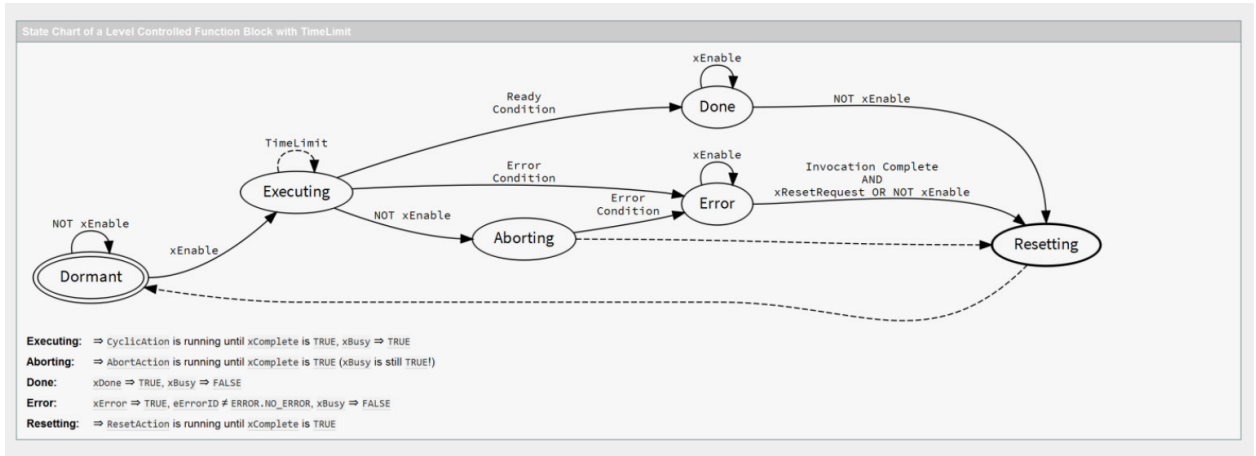
1 FUNCTION_BLOCK LCon
2 VAR_INPUT
3     // TRUE ⇒ activates the defined operation
4     // FALSE ⇒ aborts/resets the defined operation
5     xEnable: BOOL;
6 END_VAR
7 VAR_OUTPUT
8     // ready condition reached
9     xDone: BOOL;
10    // operation is running
11    xBusy: BOOL;
12    // error condition 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
25 REPEAT
26     xAgain := FALSE;
27     CASE eState OF
28         STATE.DORMANT: HandleDormantState(xAgain⇒xAgain);
29         STATE.EXECUTING: HandleExecutingState(xAgain⇒xAgain);
30         STATE.DONE: HandleDoneState(xAgain⇒xAgain);
31         STATE.ERROR: HandleErrorState(xAgain⇒xAgain);
32         STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
33         STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
34     END_CASE
35 UNTIL NOT xAgain
36 END_REPEAT;
    
```

<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 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 </pre>	<p>The Handler for the Resetting State</p> <pre> 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 </pre>	<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 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; </pre>
	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

Приложение 1.4.2. ФБ LConTI

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of LConTI

LConTI (FB)

- BehaviourModel
 - AbortAction
 - CyclicAction
 - ResetAction
- StateMachine
 - HandleAbortingState
 - HandleDoneState
 - HandleDormantState
 - HandleErrorState
 - HandleExecutingState
 - HandleResettingState

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 LConTI

```

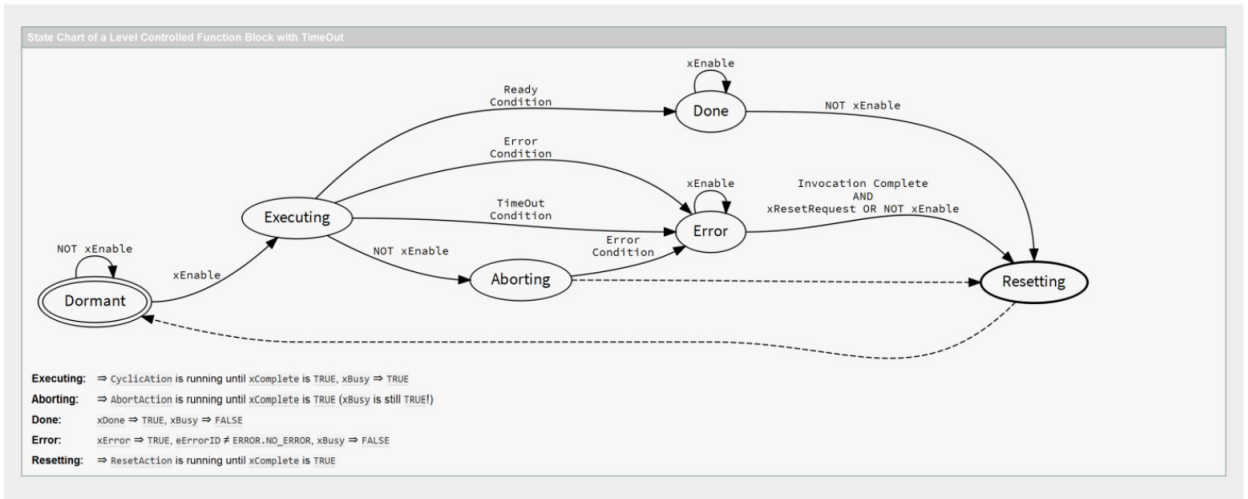
1  FUNCTION_BLOCK LConTI
2  VAR_INPUT
3  // TRUE ⇒ activates the defined operation
4  // FALSE ⇒ aborts/resets the defined operation
5  xEnable: BOOL;
6  // max operating time per invocation
7  // [µs], 0 ⇒ no operating time limit
8  udiTimeLimit: UDINT;
9  END_VAR
10 VAR_OUTPUT
11 // ready condition 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_VAR
28
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;
    
```


<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 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 </pre>	<p>The Handler for the Resetting State</p> <pre> 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 := xTimeLimit; 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 </pre>	<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 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; </pre>
	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

Приложение 1.4.3. ФБ LConTo

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of LConTo

- LConTo (FB)
 - BehaviourModel
 - AbortAction
 - CyclicAction
 - ResetAction
 - StateMachine
 - HandleAbortingState
 - HandleDoneState
 - HandleErrorState
 - HandleExecutingState
 - HandleResettingState

LConTo

Inputs: xEnable, udiTimeOut
Outputs: xDone, xBusy, xError, eErrorID

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   tctTimingController: 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;

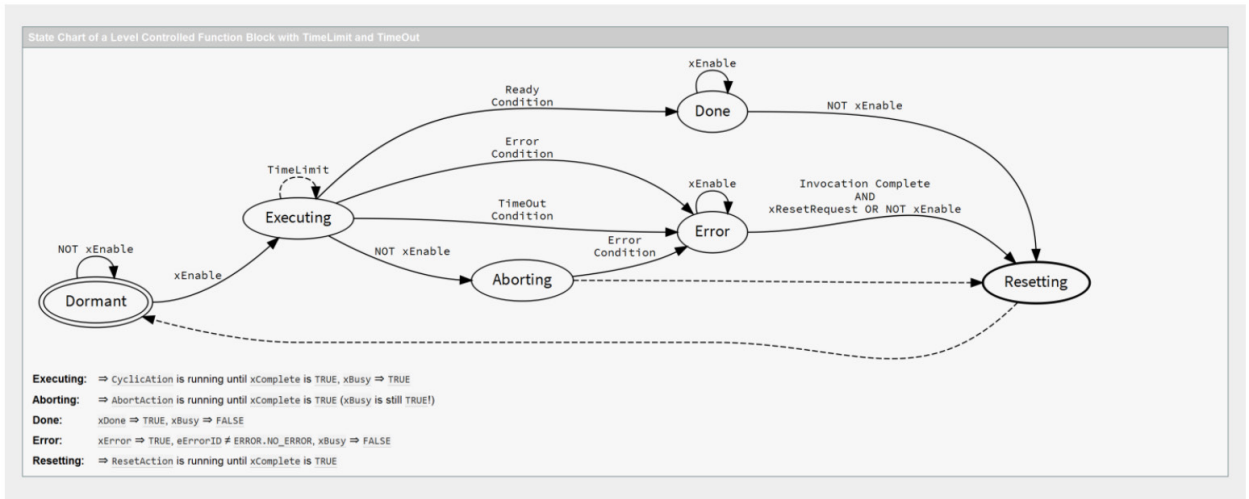
```

<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 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 </pre>	<p>The Handler for the Resetting State</p> <pre> 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 </pre>	<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 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; </pre>
	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

Приложение 1.4.4. ФБ LConTtTo

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of LConTtTo

- BehaviourModel
 - AbortAction
 - CyclicAction
 - ResetAction
- StateMachine
 - HandleAbortingState
 - HandleDoneState
 - HandleDormantState
 - HandleErrorState
 - HandleExecutingState
 - HandleResettingState

LConTtTo

xEnable	xDone
udiTimeLimit	xBusy
udiTimeOut	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 LConTtTo

```

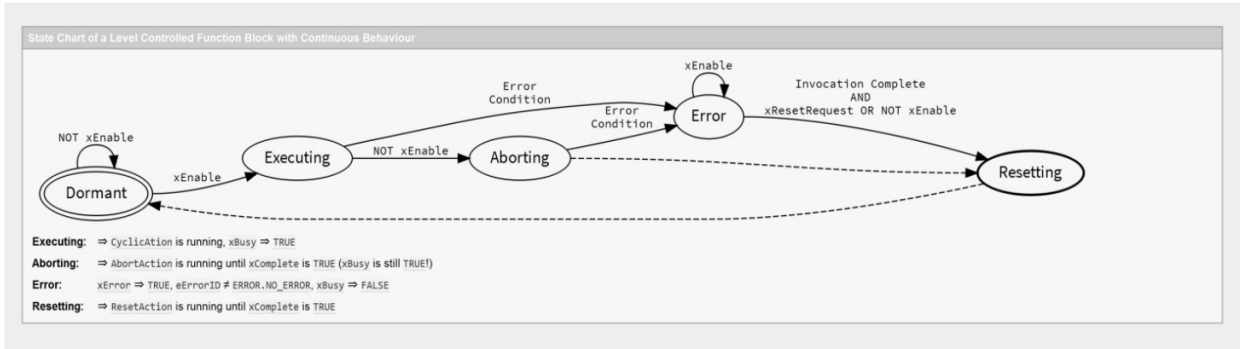
1  FUNCTION_BLOCK LconTtTo
2  VAR_INPUT
3    // TRUE ⇒ activates the defined operation
4    // FALSE ⇒ aborts/resets the defined operation
5    xEnable: BOOL;
6    // max operating time per invocation
7    // [µs], 0 ⇒ no operating time limit
8    udiTimeLimit: UDINT;
9    // max operating time for executing
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_VAR
31
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: HandleAbortingState(xAgain⇒xAgain);
40     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
41   END_CASE
42 UNTIL NOT xAgain
43 END_REPEAT;
    
```

<p>The Handler for the Dormant State</p> <pre> 1 METHOD PRIVATE FINAL HandleDormantState 2 VAR_OUTPUT 3 xAgain : BOOL; 4 END_VAR 5 6 IF xEnable THEN 7 tTimingController.StartOperationTimer(); 8 xBusy := TRUE; 9 eState := STATE.EXECUTING; 10 xAgain := TRUE; 11 END_IF </pre>	<p>The Handler for the Done State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 tTimingController.StartInvocationTimer(); 12 13 CyclicAction(14 xComplete=>xComplete, 15 eErrorID=>eErrorID 16); 17 18 tTimingController.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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 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 </pre>	<p>The Handler for the Resetting State</p> <pre> 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 </pre>
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 tTimingController.TimeLimit := udiTimeLimit; 16 tTimingController.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 tTimingController.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 </pre>	<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 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; </pre>
<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>	

Приложение 1.4.5. ФБ LConC

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

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of LConC

- LConC (FB)
 - BehaviourModel
 - AbortAction
 - CyclicAction
 - ResetAction
 - StateMachine
 - HandleAbortingState
 - HandleDormantState
 - HandleErrorState
 - HandleExecutingState
 - HandleResettingState

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  RESETING // 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 LConC

```

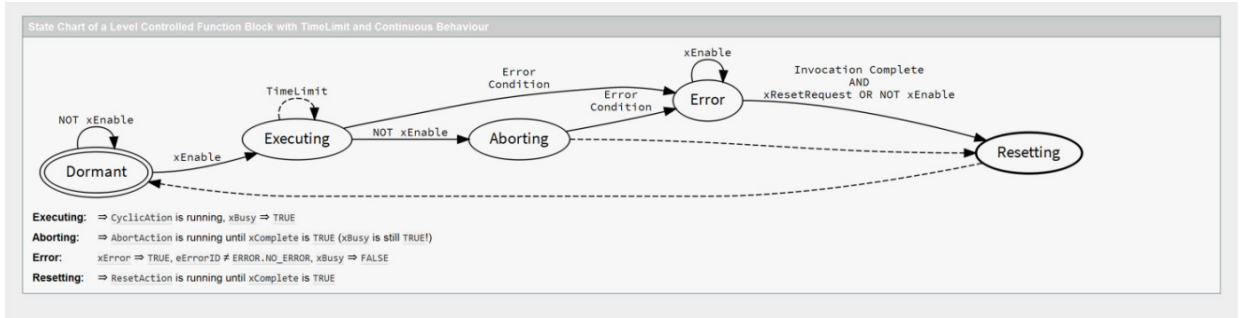
1  FUNCTION_BLOCK LConC
2  VAR_INPUT
3  // TRUE ⇒ activates the defined operation
4  // FALSE ⇒ aborts/resets the defined operation
5  xEnable: BOOL;
6  END_VAR
7  VAR_OUTPUT
8  // operation is running
9  xBusy: BOOL;
10 // error condition reached
11 xError: BOOL;
12 // error code describing error condition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


<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 ELSIF NOT xEnable THEN 16 eState := STATE.ABORTING; 17 xAgain := TRUE; 18 END_IF </pre>	<p>The Handler for the Resetting State</p> <pre> 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 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 ELSIF xComplete THEN 18 eState := STATE.RESETTING; 19 xAgain := TRUE; 20 END_IF </pre>	
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 </pre>	<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 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; </pre>
	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

Приложение 1.4.6. ФБ LConTIC

LConTIC (Level Controlled | Time Limited | Continuous Behaviour)

Диаграмма состояний



Реализация (продолжение на след. стр.)

Overview - The Content of LConTIC

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
        
```

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   // [us], 0 ⇒ no operating time limit
8   udiTimeLimit: UDINT;
9 END_VAR
10 VAR_OUTPUT
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   tcTimingController : TimingController;
20   eState : STATE;
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.ERROR: HandleErrorState(xAgain⇒xAgain);
32     STATE.ABORTING: HandleAbortingState(xAgain⇒xAgain);
33     STATE.RESETTING: HandleResettingState(xAgain⇒xAgain);
34   END_CASE
35 UNTIL NOT xAgain
36 END_REPEAT;
        
```


<p>The Handler for the Dormant State</p> <pre> 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 </pre>	<p>The Handler for the Error State</p> <pre> 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 </pre>
<p>The Handler for the Executing State</p> <pre> 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 </pre>	<p>The Handler for the Resetting State</p> <pre> 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 </pre>
<p>The Handler for the Aborting State</p> <pre> 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; </pre>	
<p>The Implementation of the Cyclic Action (Exemplary Implementation)</p> <pre> 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 24 eErrorID := ERROR.NO_ERROR; 25 26 UNTIL NOT xEnable OR 27 xTimeLimit OR udiTimeLimit = 0 OR 28 eErrorID <> ERROR.NO_ERROR 29 END_REPEAT 30 END_IF 31 32 IF NOT xEnable OR eErrorID <> ERROR.NO_ERROR THEN 33 (* Cleaning *) 34 // if possible free as much allocated resources 35 // as possible 36 END_IF </pre>	<p>The Implementation of the Abort Action (Exemplary Implementation)</p> <pre> 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; </pre>
	<p>The Implementation of the Reset Action (Exemplary Implementation)</p> <pre> 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; </pre>

Приложение 2. Пример реализации ФБ модели PLCopen без ООП

Использование ООП не является обязательным требованием для разработки ФБ, совместимых с моделью поведения **PLCopen**. Ниже приведен пример реализации ФБ **ETrigATlTo** на языке ST без использования ООП (согласно 2-й редакции стандарта МЭК 61131-3).

```

FUNCTION_BLOCK ETrigATlTo
VAR_INPUT
    // по переднему фронту начинается выполнение заданной операции
    // по заднему фронту после завершения операции блок будет переинициализирован
    xExecute: BOOL;
    // сигнал прерывания операции
    xAbort: BOOL;
    // максимальное время выполнения операции в пределах одного цикла ПЛК (в мкс)
    // 0 - без ограничений
    udiTimeLimit: UDINT;
    // общее максимально допустимое время выполнения операции (в мкс)
    // 0 - ограничений нет
    udiTimeOut: UDINT;
END_VAR

VAR_OUTPUT
    // флаг «операция успешно завершена»
    xDone: BOOL;
    // флаг «операция в процессе выполнения»
    xBusy: BOOL;
    // флаг «произошла ошибка»
    xError: BOOL;
    // флаг «операция была прервана»
    xAborted : BOOL;
    // код ошибки
    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
          // запоминаем значения входов блока
          tcTimingController.uditimeLimit := udiTimeLimit;
          tcTimingController.uditimeOut := udiTimeOut;
          xFirstInvocation := FALSE;
        END_IF

        // после сохранения входов xComplete := TRUE
        // если произошла ошибка, то присваиваем ее код
        // переменной eLocalErrorID

        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 *)
          // после выполнения заданной операции...
          // xComplete := TRUE
          // если превышено максимальное время вызова в цикле...
          // xTimeLimit := TRUE
          // если превышено общее максимальное время вызова...
          // xTimeOut := TRUE
          // если произошла ошибка, то присваиваем ее код...
          // переменной eLocalErrorID

          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
    // прерываем все выполняемые операции
    // если произошла ошибка, то присваиваем ее код...
    // переменной eLocalErrorID
    xLocalAbort := xAbortProposed;
END_IF

// освобождаем все ресурсы системы
// после этого xComplete := TRUE
// если произошла ошибка, то присваиваем ее код...
// переменной eLocalErrorID

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 *)
  // освобождаем все ресурсы системы
  // переинициализируем переменные
  // после этого 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;

```

Приложение 3. Пример промежуточного интерфейса

В рамках данного документа соблюдается определенный стандарт наименования переменных. На практике таких стандартов может быть несколько:

- стандарт наименований для низкоуровневых библиотек;
- стандарт наименований для высокоуровневых библиотек;
- стандарт наименований для пользовательского приложения.

Эти стандарты связаны с разными уровнями абстракции; для взаимодействия этих уровней можно инкапсулировать их друг в друга или использовать псевдонимы (**ALIAS**). Таким образом, для соблюдения всех нужных стандартов создается промежуточный интерфейс между библиотекой и пользовательским приложением.

Рассмотрим пример создания промежуточного интерфейса. На рисунке ниже слева приведен PUBLIC FINAL ФБ **MC_MoveVelocity**, который основан на INTERNAL блоке **MotionCore**, расположенном справа. В свою очередь, блок **MotionCore** наследуется от ФБ **EtrigA**.

В сущности, **MC_MoveVelocity** представляет собой обертку над блоком **MotionCore** с измененными именами переменных (которые соответствуют пользовательскому стандарту наименования). Ключевое слово FINAL запрещает наследовать другие блоки от **MC_MoveVelocity**. Это позволяет менять реализацию блока (например, добавлять локальные переменные). Ключевое слово INTERNAL запрещает использовать ФБ **MotionCore** за пределами его библиотеки. Таким образом, уровни абстракции разделены между собой.

The image shows two side-by-side code editors. The left editor displays the code for the **MC_MoveVelocity** block, which is a **FINAL** block. It defines input and output variables, including `Axis`, `Execute`, `Velocity`, `Acceleration`, `Deceleration`, `Jerk`, `Direction`, `InVelocity`, `Busy`, `CommandAborted`, `Error`, and `ErrorID`. It also includes a `_core` block that maps these variables to the internal `MotionCore` block's variables.

The right editor displays the code for the **MotionCore** block, which is an **INTERNAL** block that **EXTENDS** the `CBM.EtrigA` block. It defines similar input and output variables, including `Axis`, `rVelocity`, `rAcceleration`, `rDeceleration`, `rJerk`, `eDirection`, `eErrorID`, and `SMC_ERROR`. The code is partially obscured by a comment: `(* пользовательская реализация *)`.

Приложение 4. Модель поведения ФБ спецификации PLCopen Motion Control

<p>Обработка входных параметров</p>	<p>Execute: новые значения параметров применяются по переднему фронту на входе Execute.</p> <p>Execute u Continuous Update: новые значения параметров применяются по переднему фронту на входе Execute. Если во время выполнения действия значения параметров меняются, и при этом вход Continuous Update имеет значение TRUE, то новые значения применяются сразу (в том же цикле ПЛК).</p> <p>Enable: новые значения параметров применяются сразу (в том же цикле ПЛК).</p>
<p>Ограничение допустимых значений</p>	<p>Значения параметров блока должны находится в определенном диапазоне (это касается скорости движения, его продолжительности т.д.). Если значения параметра превышает допустимые границы, то ФБ должен генерировать сообщение об ошибке. Пользовательское приложение должно обрабатывать эти ошибки по специфическим для конкретной задачи алгоритмам.</p>
<p>Пропущенные значения параметров</p>	<p>Стандарт МЭК 61131-3 позволяет не присваивать значения всем входам ФБ при его вызове, оставляя некоторые из них «пустыми». В этом случае при первом вызове блока используется значение по умолчанию, а при всех последующих вызовах – значение из предыдущего вызова.</p>
<p>Acceleration, Deceleration и Jerk</p>	<p>Если входы Acceleration, Deceleration или Jerk имеют значение 0, то результат зависит от конкретной реализации. Например, возможны следующие варианты:</p> <ol style="list-style-type: none"> 1. ФБ перейдет в состояние <i>Error</i>; 2. ФБ продолжит работу, но сгенерирует предупреждение; 3. Среда программирования может запрещать пользователю оставить значения данных входов нулевыми; 4. ФБ будет использовать значения по умолчанию из AXIS_REF или настроек устройства; 5. ФБ будет использовать максимально возможное значение. <p>Даже если ваше ПО позволяет оставить значения данных входов нулевыми, используйте эту возможность крайне осторожно (особенно, если приложение должно быть платформо-независимым).</p>
<p>Статусные выходы</p>	<p>Execute: в каждый момент времени только один из статусных выходов (Done, Busy, Error или CommandAborted) может иметь значение TRUE. Если вход Execute имеет значение TRUE, то один из статусных выходов обязательно имеет значение TRUE. У ФБ MC_Stop выходы Active и Done могут быть активные одновременно.</p> <p>Enable: в каждый момент времени только один из статусных выходов (Valid или Error) может иметь значение TRUE.</p>
<p>Сброс выходов</p>	<p>Execute: выходы Done, Error, ErrorID и CommandAborted сбрасываются в FALSE по заднему фронту на входе Execute, если операция, выполняемая блоком, к этому моменту была завершена. Если блок получает передний фронт на входе Execute до окончания выполнения текущей операции, то статусные выходы не принимают никаких значений.</p> <p>Enable: выходы Valid, Enabled, Busy, Error и ErrorID сбрасываются в FALSE по заднему фронту на входе Enable.</p>
<p>Выход Done</p>	<p>Выход Done принимает значение TRUE после успешного завершения операции.</p> <p>Если движение по одной из осей управляется с помощью нескольких последовательно вызываемых ФБ, то в случае прерывания работы одного из блоков другим выход Done не устанавливается в TRUE.</p>

<p>Выход Busy</p>	<p><i>Execute</i>: каждый ФБ имеет выход Busy. Если он имеет значение TRUE, то заданная операция находится в процессе выполнения и значения на выходах блока еще не являются актуальными. Выход Busy принимает значение TRUE по переднему фронту на входе Execute и сбрасывается в FALSE по переднему фронту на любом из других статусных выходов (Done, CommandAborted, или Error).</p> <p><i>Enable</i>: ФБ может иметь выход Busy. Если он имеет значение TRUE, то заданная операция находится в процессе выполнения и значения на выходах блока еще не являются актуальными. Выход Busy принимает значение TRUE по переднему фронту на входе Enable и сбрасывается в FALSE после окончания выполнения операции.</p> <p>Рекомендуется вызывать ФБ в цикле программы до тех пор, пока выход Busy не примет значение FALSE, т.к. до этого момента значения других выходов блока могут быть некорректными.</p>
<p>Выходы InVelocity, InGear, InTorque, InSync</p>	<p>Поведение выходов InVelocity, InGear, InTorque, InSync (далее мы будем называть их «InXxx») отличается от поведения выхода Done. Они имеют значение TRUE, пока соответствующее им входное значение совпадает с действительным (например, InVelocity имеет значение TRUE, пока действительная скорость движения совпадает с уставкой на входе блока). Если установленное и действительное значение не совпадают, то выход InXxx имеет значение FALSE. Выходы InXxx обновляются в независимости от значения на входе Execute (пока выходы Active и Busy имеют значение TRUE). Если после сигнала переднего фронта на входе Execute заданное значение уже совпадает с действительным, то поведение выходов InXxx зависит от конкретной реализации. Выходы InXxx не связаны с реальными значениями; под «действительными» значениями в данном случае понимаются значения внутренних параметров ФБ, которые определяют его работу.</p>
<p>Выход Active</p>	<p>Выход Active требуется для ФБ, имеющих буфер. Выход принимает значение TRUE, когда блок начинает управлять движением по заданной оси. Для ФБ, не имеющих буфера, поведение выходов Busy и Active совпадает.</p> <p>Когда движением по оси управляют несколько ФБ, то у некоторых из них может быть активен выход Busy, но только у одного – выход Active. Исключением являются ФБ, работающие параллельно – например, MC_MoveSuperimposed и MC_Phasing.</p>
<p>Выход CommandAborted</p>	<p>Выход CommandAborted принимает значение TRUE, когда текущая операция движения прерывается другой. Одновременно с этим сбрасываются выходы InXxx.</p>
<p>Модель Enable/Valid</p>	<p>При наличии входа Enable ФБ всегда имеет выход Valid. Пока вход Enable имеет значение TRUE, блок находится в работе. Выход Valid имеет значение TRUE, если другие выходы блока имеют актуальные значения. Значения выходов могут изменяться, пока вход Enable имеет значение TRUE. Если в процессе работы блока происходит ошибка, то выход Valid принимает значение FALSE. После устранения ошибки он возвращается в значение TRUE.</p>
<p>Position и Distance</p>	<p>Переменная Position определяет точку в системе координат. Переменная Distance определяет расстояние между двумя точками.</p>
<p>Знаки</p>	<p>Переменные Acceleration, Deceleration и Jerk могут иметь только положительные значения. Переменные Velocity, Position и Distance могут иметь как положительные, так и отрицательные значения.</p>
<p>Обработка ошибок</p>	<p>Все ФБ имеют два выхода, характеризующие наличие ошибки:</p> <p>Error – флаг ошибки (принимает значение TRUE при возникновении ошибки); ErrorID – код ошибки.</p> <p>Выходы Done и InXxx характеризуют успешное выполнение операции, поэтому они не могут устанавливаться в TRUE при наличии ошибки.</p> <p>Типы ошибок:</p> <ul style="list-style-type: none"> • Ошибки ФБ (например, значения параметров выходят за допустимые пределы); • Ошибки обмена; • Ошибки привода.

	<p>Ошибка в конкретном ФБ не всегда приводит к остановке движения (<i>ErrorStop</i>).</p> <p><i>Execute</i>: выходы ошибок сбрасываются по переднему фронту на входе Execute.</p> <p><i>Enable</i>: выходы ошибок могут быть сброшены в процессе работы (без заднего фронта на входе Enable).</p>
Имена ФБ	<p>При использовании нескольких библиотек в пределах одной среды программирования (например, для поддержки разных приводов) имена ФБ могут быть дополнены уникальным идентификатором: «MC_FBname_SupplierID». Это позволит избежать дублирования имен.</p>
Имена перечислений	<p>В именах перечислений спецификации PLCopen Motion Control используется префикс «mc» (mcPositive, mcNegative).</p>

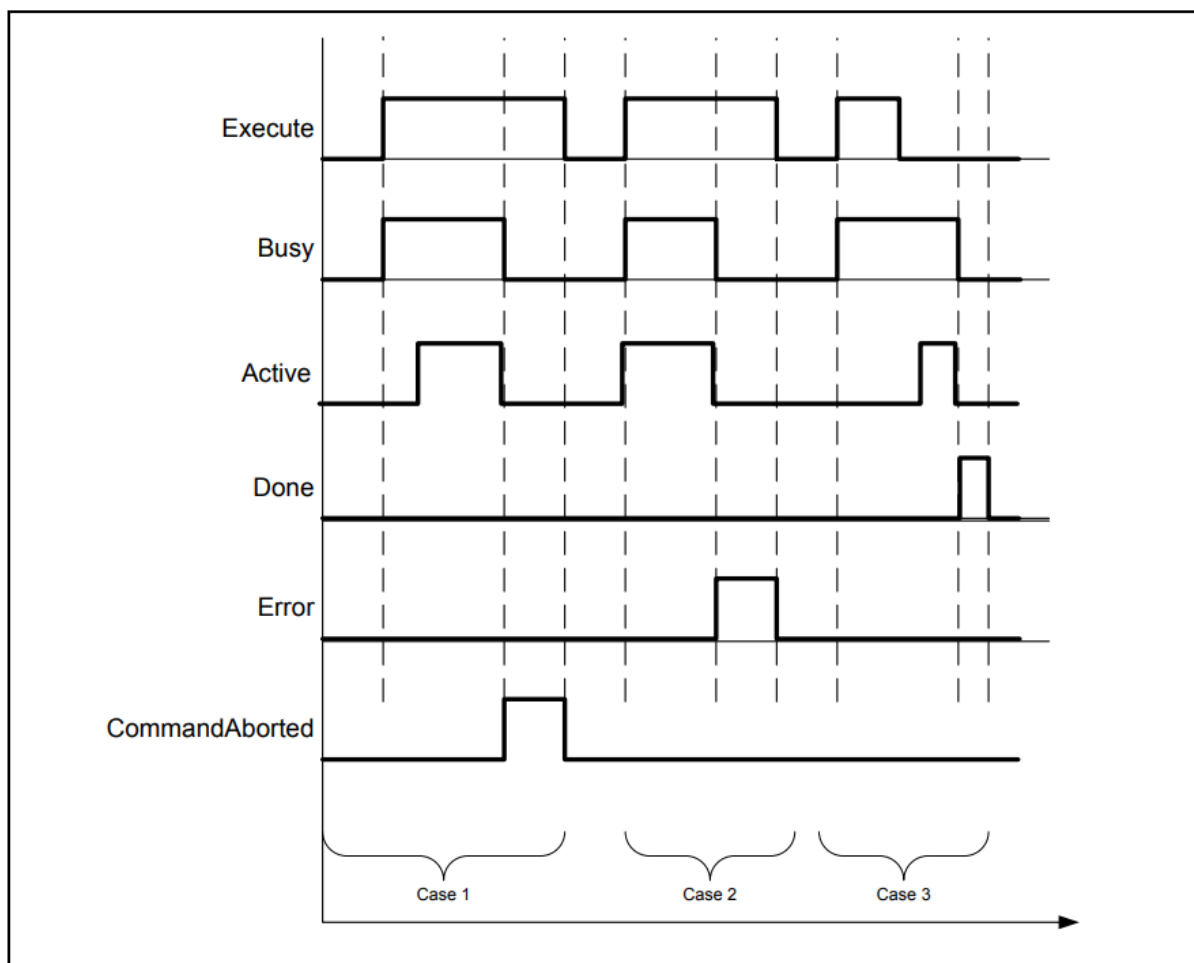


Рис. 16. Циклограмма ФБ модели Execute/Done

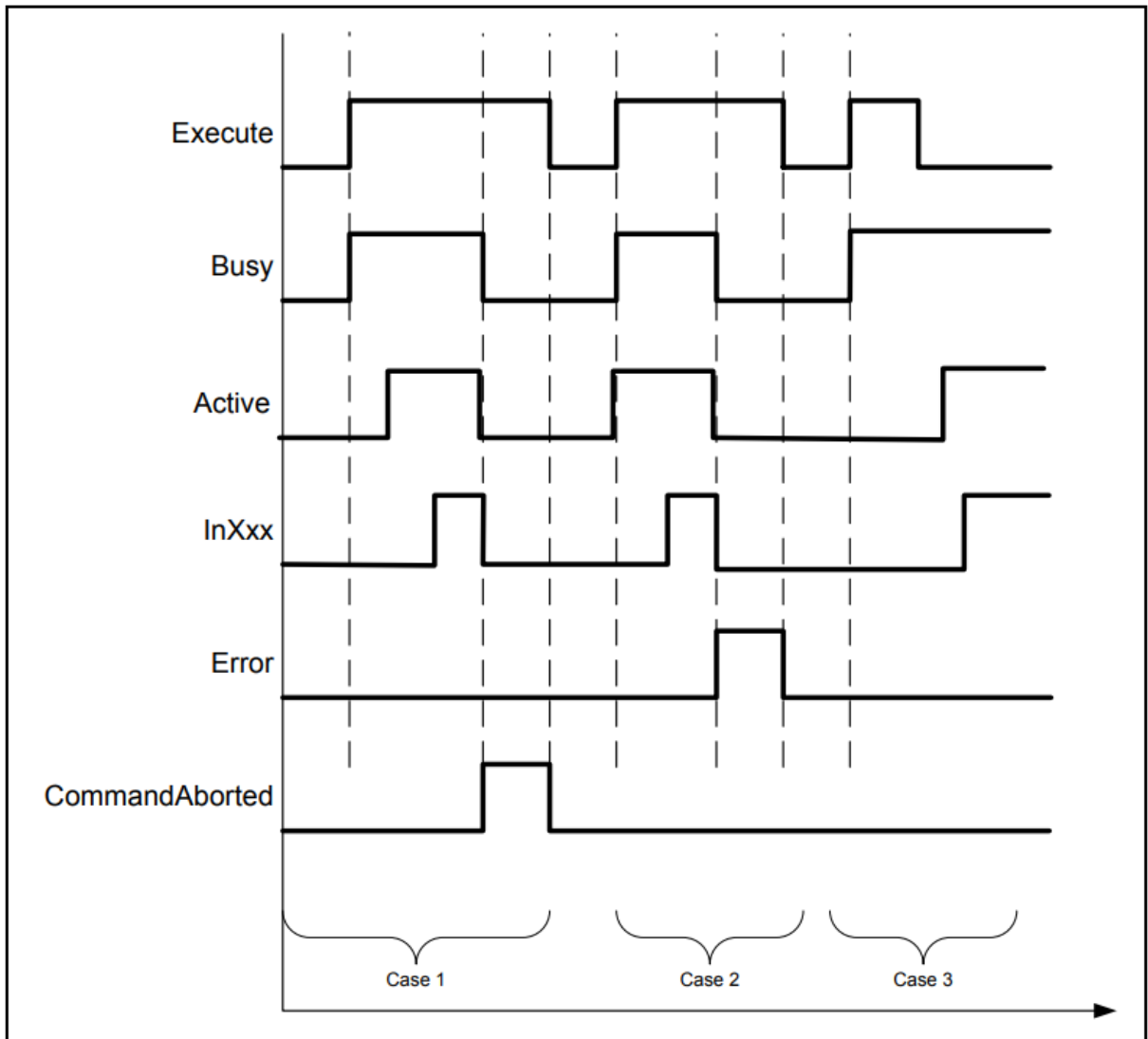


Рис. 17. Циклограмма ФБ модели Execute/InXxx