Documentation

TwinCAT Safety PLC

PC based Safety Controller

Version: 1.2.0
Date: 2017-06-29

BECKHOFF

Table of contents

1 Foreword

1.1 Notes on the documentation

Target group

This description is aimed specifically at trained qualified persons with a control and automation technology background, who are familiar with the current national and international standards and guidelines.
These persons must be trained in the development, validation and verification of safety-related applications in a high-level language in accordance with the normative software lifecycle, based on the requirements of EN 61508.

The following instructions and explanations must be followed during installation and commissioning of the components.

The qualified personnel must ensure that the application of the described products meets all safety requirements, including all applicable laws, specifications, regulations and standards.

Origin of the document

This documentation was originally written in German. All other languages are derived from the German original.

Currentness

Please check whether you are using the current and valid version of this document. The current version can be downloaded from the Beckhoff homepage at http://www.beckhoff.com/english/download/twinsafe.htm. In case of doubt, please contact Technical Support [▶ 112].

Product features

Only the product features specified in the current user documentation are valid. Further information given on the product pages of the Beckhoff homepage, in emails or in other publications is not authoritative.

Disclaimer

The documentation has been prepared with care. The products described are subject to cyclical revision. For that reason the documentation is not in every case checked for consistency with performance data, standards or other characteristics. We reserve the right to revise and change the documentation at any time and without prior announcement. No claims for the modification of products that have already been supplied may be made on the basis of the data, diagrams and descriptions in this documentation.

Trademarks

Beckhoff®, TwinCAT®, EtherCAT®, Safety over EtherCAT®, TwinSAFE®, XFC® and XTS® are registered trademarks of and licensed by Beckhoff Automation GmbH.
Other designations used in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owners.

Patent Pending

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents: EP1590927, EP1789857, DE102004044764, DE102007017835 with corresponding applications or registrations in various other countries.

The TwinCAT Technology is covered, including but not limited to the following patent applications and patents: EP0851348, US6167425 with corresponding applications or registrations in various other countries.

EtherCAT® is registered trademark and patented technology, licensed by Beckhoff Automation GmbH, Germany

Copyright

Delivery conditions

In addition, the general delivery conditions of the company Beckhoff Automation GmbH & Co. KG apply.

1.2 Safety instructions

1.2.1 Delivery state

All the components are supplied in particular hardware and software configurations appropriate for the application. Modifications to hardware or software configurations other than those described in the documentation are not permitted, and nullify the liability of Beckhoff Automation GmbH & Co. KG.

1.2.2 Operator's obligation to exercise diligence

The operator must ensure that

- the TwinSAFE products are only used as intended (see chapter Product description);
- the TwinSAFE products are only operated in sound condition and in working order.
- the TwinSAFE products are operated only by suitably qualified and authorized personnel.
- the personnel is instructed regularly about relevant occupational safety and environmental protection aspects, and is familiar with the operating instructions and in particular the safety instructions contained herein.
- the operating instructions are in good condition and complete, and always available for reference at the location where the TwinSAFE products are used.
- none of the safety and warning notes attached to the TwinSAFE products are removed, and all notes remain legible.

1.2.3 Description of safety symbols

In these operating instructions the following symbols are used with an accompanying safety instruction or note. The safety instructions must be read carefully and followed without fail!

⚠ **DANGER**	**Serious risk of injury!** **Failure** to follow the safety instructions associated with this symbol directly endangers the life and health of persons.
⚠ **WARNING**	**Risk of injury!** **Failure** to follow the safety instructions associated with this symbol endangers the life and health of persons.
⚠ **CAUTION**	**Personal injuries!** **Failure** to follow the safety instructions associated with this symbol can lead to injuries to persons.
! **Attention**	**Damage to the environment or devices** **Failure** to follow the instructions associated with this symbol can lead to damage to the environment or equipment.
i **Note**	**Tip or pointer** This symbol indicates information that contributes to better understanding.

1.3 Documentation issue status

Version	Comment
1.2.0	• Application development in Safety C updated
1.1.0	• Description of safe helper functions added • General revision of all chapters
1.0.0	• First released version • Certificate added • Diagnostic data updated
0.3	• Target group updated • Operation updated - Configuration of the hardware platform
0.2	• Provisional version for certification
0.0.1	• First draft. For internal use only.

2 System description

2.1 Extension of the Beckhoff I/O system with safety functions

The TwinSAFE products from Beckhoff enable convenient expansion of the Beckhoff I/O system with safety components, and integration of all the cabling for the safety circuit within the existing fieldbus cable. Safe signals can be mixed with standard signals without restriction. The transfer of safety-related TwinSAFE telegrams is handled by the standard controller. Maintenance is simplified significantly thanks to faster diagnosis and simple replacement of components.

The following basic functionalities are included in the TwinSAFE components:
digital inputs (e.g. EL19xx, EP1908), digital outputs (e.g. EL29xx), drive components (e.g. AX5805) and logic units (e.g. EL6900, EL6910, TwinCAT Safety PLC). For a large number of applications, the complete safety sensor and actuator technology can be wired on these components. The necessary logic link between the inputs and outputs is handled by the EL69xx or the TwinCAT Safety PLC. In addition to Boolean operations, the EL6910 also enables analog operations. The TwinCAT Safety PLC enables development of safety-related logic in Safety C.

2.2 TwinCAT Safety PLC

The TwinCAT Safety PLC is used to realize the links between safety-related inputs and outputs via the Safety-over-EtherCAT protocol (FSoE).

The TwinCAT Safety PLC meets the requirements of IEC 61508:2010 SIL 3 and EN ISO 13849-1:2015 (Cat 4, PL e).

The TwinCAT Safety PLC realizes a safety-related runtime environment on a standard Industrial PC. Currently only Beckhoff IPCs can be used. Further information on permitted configurations can be found in the document “List of approved system configurations” on the Beckhoff website.

The safety-related logic can be created in Safety C, in future also via the graphical TwinSAFE Editor.

2.3 Safety concept

TwinSAFE: Safety and I/O technology in one system

- Extension of the familiar Beckhoff I/O system with TwinSAFE components
- Safe and non-safe components can be combined as required
- Logic linking of the I/Os in the TwinCAT Safety PLC
- Suitable for applications up to SIL 3 according to EN 61508:2010 and Cat 4, PL e according to EN ISO 13849-1:2015
- Safety-relevant networking of machines via bus systems
- In the event of an error, all TwinSAFE components always switch to the deenergized and therefore safe state
- No safety requirements for the higher-level standard TwinCAT system

Safety over EtherCAT protocol (FSoE)

- Transfer of safety-relevant data via any media (“genuine black channel”)
- TwinSAFE communication via fieldbus systems such as EtherCAT, Lightbus, PROFIBUS, PROFINET or Ethernet
- IEC 61508:2010 SIL 3 compliant
- FSoE is IEC standard (IEC 61784-3-12) and ETG standard (ETG.5100)

Fail-safe principle (fail stop)

The basic rule for a safety system such as TwinSAFE is that failure of a part, a system component or the overall system must never lead to a dangerous condition. The safe state is always the switched off and wattless state.

⚠ CAUTION	**Safe state** For all TwinSAFE components the safe state is always the switched-off, wattless state.

3 Product description

WARNING

System limits

The TwinCAT Safety PLC is only permitted for hardware platforms that are included in the "List of approved system configurations".
The TwinSAFE Editor for engineering and the TwinCAT Safety PLC runtime must be installed and used on physically different PCs.

Note

Software environment

To have the full functionality of the TwinCAT Safety PLC available, it is necessary to use Visual Studio 2015 Professional or a later version.

3.1 Intended use

WARNING

Caution - Risk of injury!

The TwinCAT Safety PLC must not be used outside the scope of the intended use described below!

The TwinCAT Safety PLC expands the application area of the Beckhoff I/O system with functions that enable it to be used for machine safety applications. The TwinCAT Safety PLC is intended for safety functions of machines and directly related tasks in industrial automation. They are therefore only approved for applications with a defined fail-safe state. This safe state is the wattless state. Fail-safety according to the relevant standards is required.

The software part of the TwinCAT Safety PLC is a software-based safety controller, which may only be used on approved system configurations (consisting of development environment, runtime environment and hardware platform).

CAUTION

Permitted system configurations

The certificate for the TwinCAT Safety PLC covers only system configurations that are included in the “List of approved system configurations”.

Any system configurations that are not included in the “List of approved system configurations” are not covered by the TwinCAT Safety PLC certificate.

For applications with different system configurations, the customer is responsible for demonstrating compliance with the required safety level.

CAUTION

Note the Machinery Directive

The TwinCAT Safety PLC and the TwinSAFE terminals may only be used in machines that are covered by the Machinery Directive.

CAUTION

Ensure traceability

The operator must ensure traceability of the equipment via the serial number.

CAUTION

Industrial PC used

Please note the technical data of Industrial PC used and ensure that it is only used as intended.

CAUTION

Security

The TwinCAT Safety PLC is regarded as a self-contained system. Accordingly, the user is responsible for evaluating and implementing appropriate safety and security measures for the individual components, including the development PC and the runtime environment.

! **Attention**	**User name and password** Users must ensure that their login data are not accessible to unauthorized persons.

3.2 Technical data

Product ID	TwinCAT Safety PLC
Number of inputs	0
Number of outputs	0
Status indicator	depending on the used hardware platform
Minimum/maximum cycle time	approx. 500 µs / depending on the project size
Fault response time	≤ watchdog times
Watchdog time	min. 1 ms, max. 60,000 ms
Input process image	Dynamic, according to the TwinSAFE configuration in TwinCAT 3
Output process image	Dynamic, according to the TwinSAFE configuration in TwinCAT 3
Supply voltage (SELV/PELV)	depending on the used hardware platform (see document “List of approved system configurations”)
Permissible ambient temperature (operation)	0 °C to +55 °C (unless specified otherwise in the technical data for the hardware platform)
Permissible ambient temperature (transport/storage)	-25 °C to +65 °C (unless specified otherwise in the technical data for the hardware platform)
Permissible humidity	5% to 95%, non-condensing (unless specified otherwise in the technical data for the hardware platform)
Permissible air pressure (operation/storage/transport)	750 hPa to 1100 hPa (unless specified otherwise in the technical data for the hardware platform) (This corresponds to an altitude of approx. - 690 m to 2450 m above sea level, based on an international standard atmosphere)
Climate category according to EN 60721-3-3	3K3 (unless specified otherwise in the technical data for the hardware platform)
Permissible level of contamination according to EN 60664-1	Level of contamination 2 (unless specified otherwise in the technical data for the hardware platform)
Inadmissible operating conditions	TwinSAFE components must not be used under the following conditions: • under the influence of ionizing radiation (exceeding the natural background radiation) • in corrosive environments • in an environment that leads to unacceptable contamination of the hardware platform
Vibration / shock resistance	conforms to EN 60068-2-6 / EN 60068-2-27
EMC resistance burst / ESD	conforms to EN 61000-6-2 / EN 61000-6-4
Shocks	depending on the used hardware platform (see document “List of approved system configurations”)
Protection class	IP20
Permitted installation position	depending on the used hardware platform (see document “List of approved system configurations”)
Technical approvals	TÜV SÜD

3.3 Safety parameters

Key data	TwinCAT Safety PLC
Lifetime [a]	not applicable (if a value is required for calculations, 20 can be assumed)
Proof test interval [a]	not required [1]
PFH$_D$	5.5E-10
%SIL3 of PFH$_D$	0.55%
PFD$_{avg}$	5.5E-10
%SIL3 of PFD$_{avg}$	0.000055%
MTTF$_D$	high
DC	> 99%
Performance level	PL e
Category	4
HFT	0
Classification element [2]	Type B

1. No special proof tests are required during the entire service life of the TwinCAT Safety PLC.
2. Classification according to IEC 61508-2:2010 (see chapters 7.4.4.1.2 and 7.4.4.1.3)

The TwinCAT Safety PLC can be used for safety-related applications as defined in
IEC 62061:2005/A2:2015 SIL3, IEC 61508:2010 to SIL3 and EN ISO 13849-1:2015 to PL e (Cat4).

Further information on calculating or estimating the MTTF$_D$ value from the PFH$_D$ value can be found in the TwinSAFE application manual or in EN ISO 13849-1:2015, Table K.1.

In terms of safety-related parameters, the Safety-over-EtherCAT communication is already considered with 1% of SIL3 according to the protocol specification.

DANGER

Occurrence of serious internal errors during processing

The system must no longer be operated if more than one serious error occurs per hour.
If this is the case, the first thing to check are the basic conditions listed under intended use and the technical data of the hardware platform used.
If the problem persists, please contact Beckhoff support.

3.4 Project design limits

The project design limits depend on the licensing. Different licenses are available for different maximum numbers of permitted FSoE connections.

4 Operation

Please ensure that the TwinCAT Safety PLC is only transported, supported and operated under the ambient conditions specified for the respective hardware platform (see technical data for the corresponding hardware platform).

WARNING

Risk of injury!

The TwinSAFE components must not be used under the following conditions.

- under the influence of ionizing radiation (exceeding the natural background radiation)
- in corrosive environments
- in an environment that leads to unacceptable contamination of the hardware platform

Attention

Electromagnetic compatibility

The TwinSAFE components comply with the current standards on electromagnetic compatibility with regard to spurious radiation and immunity to interference in particular. However, in cases where devices such as mobile phones, radio equipment, transmitters or high-frequency systems that exceed the interference emissions limits specified in the standards are operated near TwinSAFE components, the function of the TwinSAFE components may be impaired.

Attention

Using the hardware platform

The hardware platform (see List of approved system configurations), on which the TwinCAT Safety PLC is to be installed and operated, may only be used in machines that are configured and installed in accordance with the requirements of EN 60204-1, Chapter 4.4.2.

Attention

Configuration of the hardware platform

The hardware platform must be configured such that “common mode” interference according to IEC 61000-4-16 is avoided in the frequency range between 0 Hz and 150 kHz.

4.1 Installation

4.1.1 Safety instructions

Before installing and commissioning the TwinCAT Safety PLC, read the safety instructions in the introduction to this documentation and the safety instructions in the corresponding documentation of the hardware platform used.

4.1.2 Specifications for transport and storage

Instructions for transport and storage can be found in the documentation for the respective hardware platform.

4.1.3 Mechanical installation

Instructions for the mechanical installation can be found in the documentation for the respective hardware platform. Note in particular the permitted installation position.

4.1.4 Electrical installation

Instructions for the electrical installation can be found in the documentation for the respective hardware platform.

4.1.5 Software installation

The TwinCAT Safety PLC is always installed together with TwinCAT. The installation of TwinCAT 3.1 build 4022 or higher always includes the latest approved version of the TwinCAT Safety PLC. It is activated via the corresponding license.

4.1.6 TwinSAFE reaction times

The TwinSAFE terminals together with the TwinCAT Safety PLC form a modular system that exchanges safety-related data via the Safety-over-EtherCAT protocol. This chapter is intended to help determine the system response time between the signal change at the sensor and the response at the actuator.

Typical response time

The typical response time is the time required for transferring a piece of information from the sensor to the actuator, when the whole system operates normally, without error.

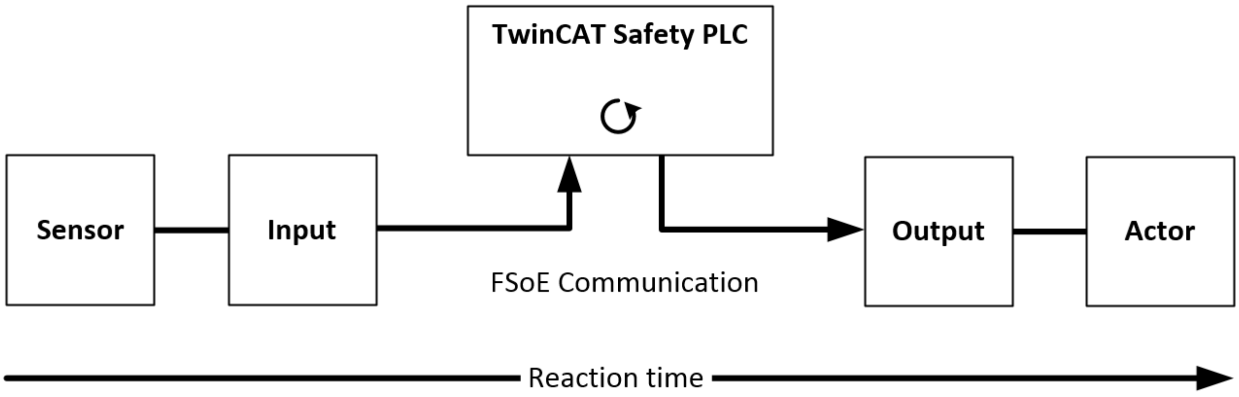


Fig. 1: Typical response time

Definition	Description
RTSensor	Response time of the sensor, until the signal is made available at the interface. Typically provided by the sensor manufacturer.
RTInput	Response time of the safe input, e.g. EL1904 or EP1908. This time can be found in the technical data. For the EL1904 the time is 4 ms, for example.
RTComm	Response time of the communication. This is typically 3 times the EtherCAT cycle time, since a new Safety-over-EtherCAT telegram has to be generated before new data can be sent. These times directly depend on the standard control system (cycle time of the PLC/NC/SafetyTask). Note which task synchronously controls the EtherCAT segment.
RTLogic	Response time of the TwinCAT Safety PLC. This is the cycle time of the task in which the TwinCAT Safety PLC is executed, if no timeout errors occur.
RTOutput	Response time of the output terminal. This is typically between 2 and 3 ms.
RTActor	Response time of the actuator. This information is typically provided by the actuator manufacturer
WDComm	Watchdog time of the communication

The typical response time is based on the following formula:

$$ReactionTime_{typ} = RT_{Sensor} + RT_{Input} + 3 * RT_{Comm} + RT_{Logic} + 3 * RT_{Comm} + RT_{output} + RT_{Actor}$$

with

$$ReactionTime_{typ} = 5ms + 4ms + 3 * 1ms + 1ms + 3 * 1ms + 3ms + 15ms = 34ms$$

Worst case response time

The worst-case response time is the maximum time required for switching off the actuator in the event of an error.

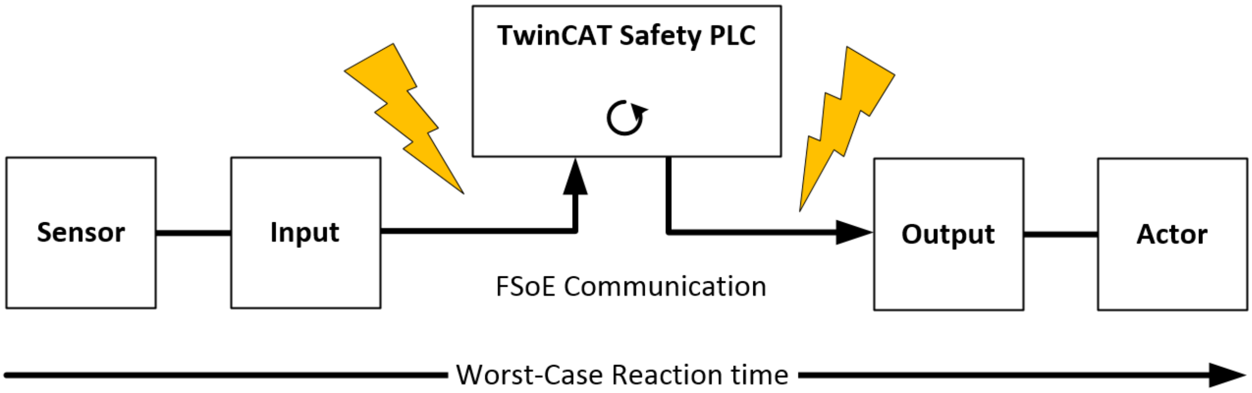


Fig. 2: Worst case response time

It is assumed that a signal change takes place at the sensor, and that this is passed to the input. A communication error occurs just at the moment when the signal is to be passed to the communication interface. This is detected by the logic once the watchdog time of the communication link has elapsed. This information should then be passed on to the output, resulting in a further communication error. This fault is detected at the output once the watchdog time has elapsed, resulting in shutdown.

This results in the following formula for the worst-case response time:

$$ReactionTime_{max} = WD_{Comm} + WD_{Comm} + RT_{Actor}$$

with

$$ReactionTime_{max} = 100ms + 100ms + 15ms = 215ms$$

4.2 Configuration of the TwinCAT Safety PLC in TwinCAT

4.2.1 Configuration requirements

Configuration of the TwinCAT Safety PLC requires TwinCAT automation software version 3.1 build 4022 or higher. The latest version is available for download from the Beckhoff website at www.beckhoff.de .

Note	**TwinCAT support** The TwinCAT Safety PLC cannot be used under TwinCAT 2.

4.2.2 Creating a safety project in TwinCAT 3

A safety project created in Safety C must be developed based on the applicable standards. See also chapter Safety C application development [▶ 50]

DANGER	**Source text of the safety application** The user source text must be developed based on the applicable standards, in particular IEC 61508:2010. See also chapter Verification and validation [▶ 81].

4.2.2.1 Add new item

In TwinCAT 3 a new project can be created via *Add New Item…* in the context menu of the *Safety* node.

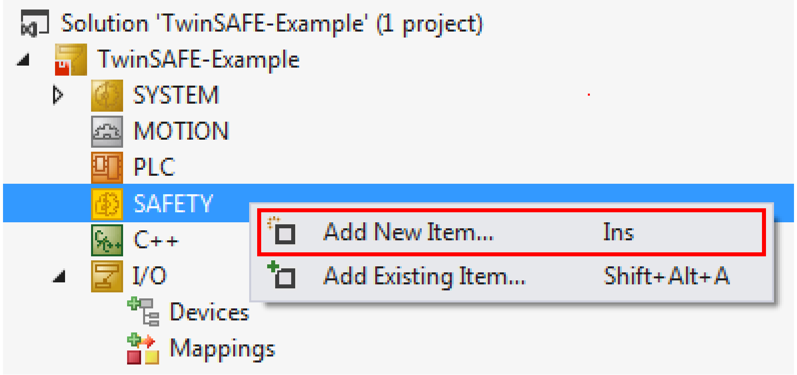
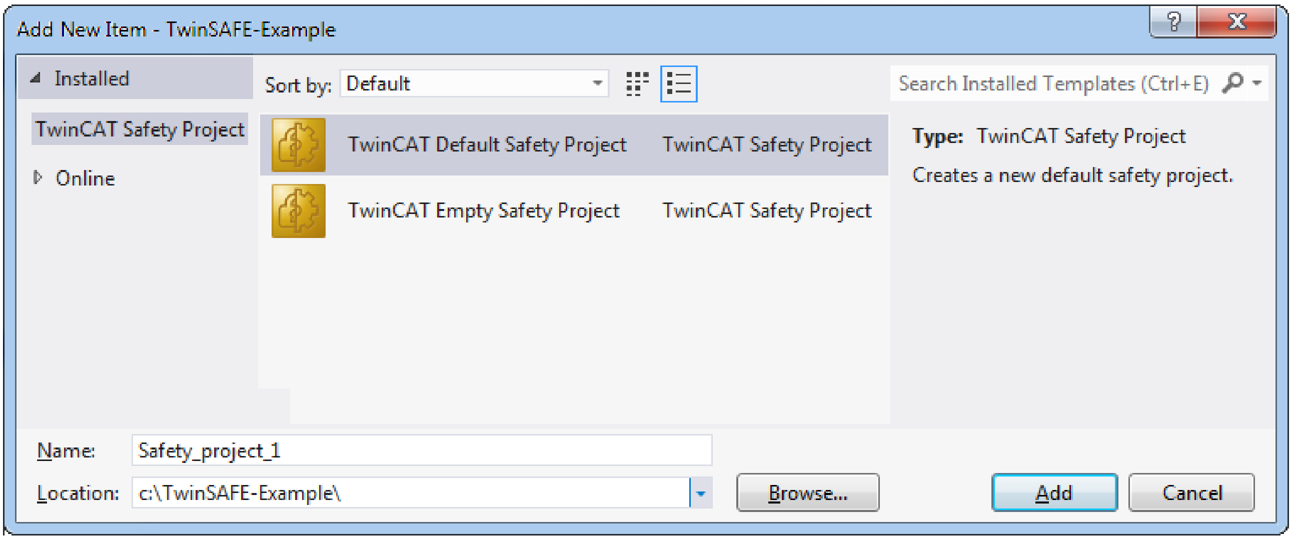


Fig. 3: Creating a safety project - Add New Item

The project name and the directory can be freely selected.



Add New Item - TwinSAFE-Example

Installed

TwinCAT Safety Project

Online

Sort by: Default

TwinCAT Default Safety Project	TwinCAT Safety Project
TwinCAT Empty Safety Project	TwinCAT Safety Project

Type: TwinCAT Safety Project

Creates a new default safety project.

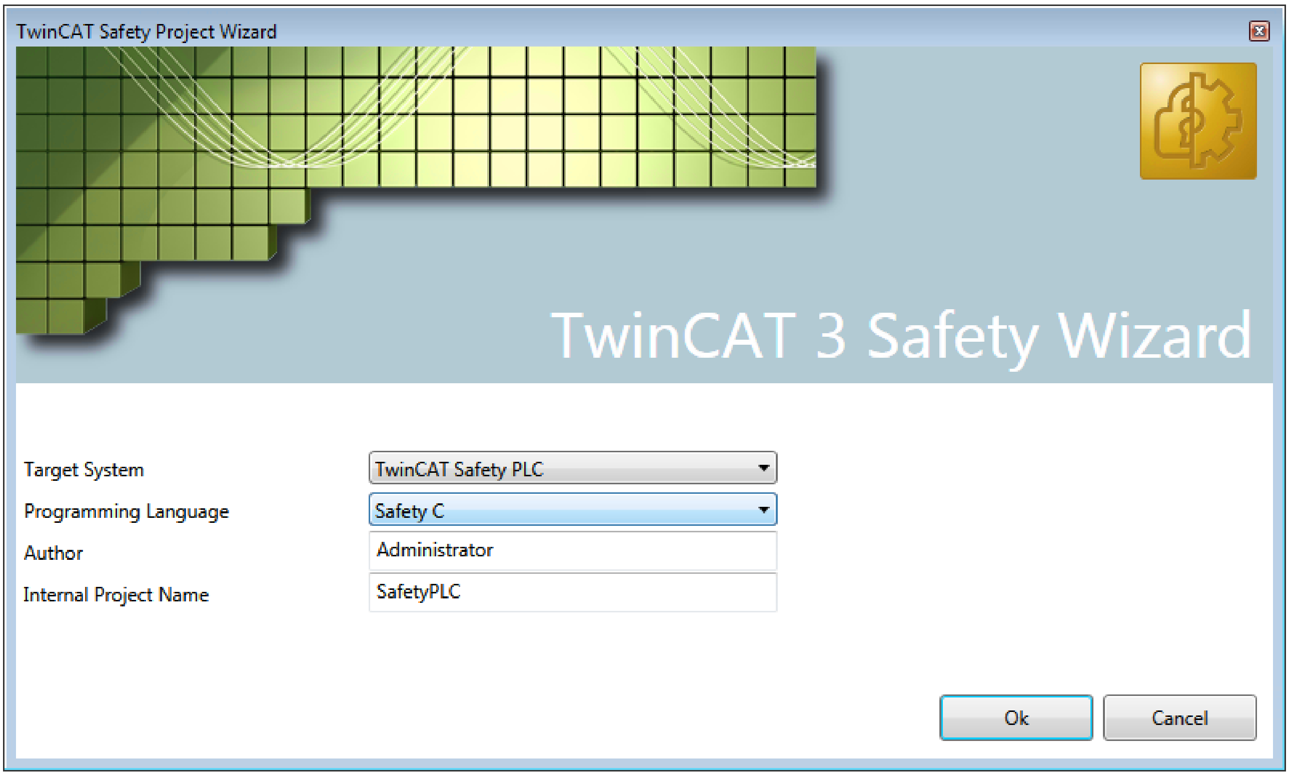
Name: Safety_project_1

Location: c:\TwinSAFE-Example\

Fig. 4: Creating a safety project - project name and directory

4.2.2.2 TwinCAT Safety Project Wizard

In the TwinCAT Safety Project wizard you can then select the target system, the programming language, the author and the internal project name. Select *TwinCAT Safety PLC* as the target system and *Safety C* as the programming language. The author and the internal project name can be freely selected by the user.



TwinCAT Safety Project Wizard

TwinCAT 3 Safety Wizard

Target System: TwinCAT Safety PLC

Programming Language: Safety C

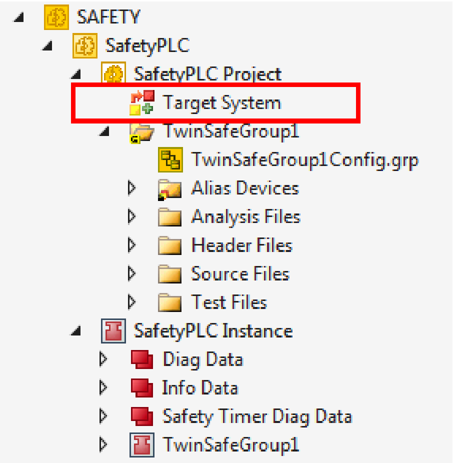
Author: Administrator

Internal Project Name: SafetyPLC

Fig. 5: TwinCAT Safety Project Wizard

4.2.2.3 Target System

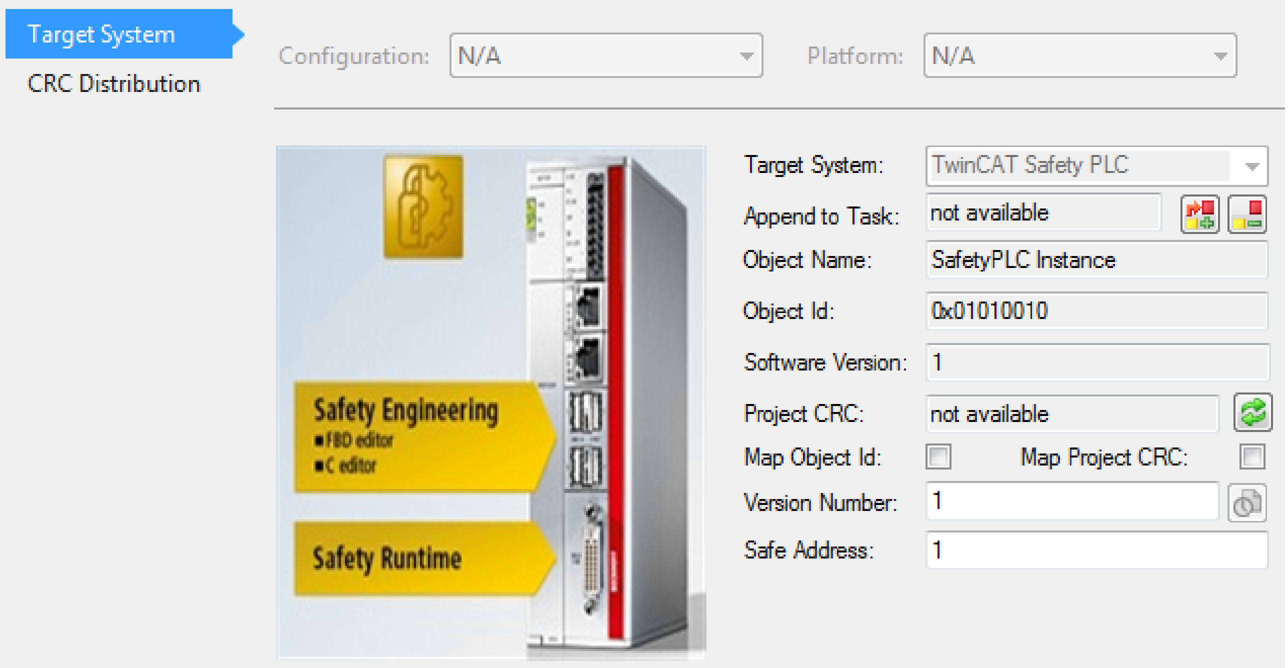
Once the project has been created with the project wizard, the safety project can be assigned to the task for the corresponding safety application by selecting the *Target System* node.



- SAFETY
  - SafetyPLC
    - SafetyPLC Project
      - Target System
      - TwinSafeGroup1
        - TwinSafeGroup1Config.grp
        - Alias Devices
        - Analysis Files
        - Header Files
        - Source Files
        - Test Files
    - SafetyPLC Instance
      - Diag Data
      - Info Data
      - Safety Timer Diag Data
      - TwinSafeGroup1

Fig. 6: Target system in the Solution Explorer

The target system is set to *TwinCAT Safety PLC* in the drop-down list. Use the link button next to *Append to Task* to link the target system to the task, with which the TwinCAT Safety PLC is to be executed.



Configuration: N/A
Platform: N/A

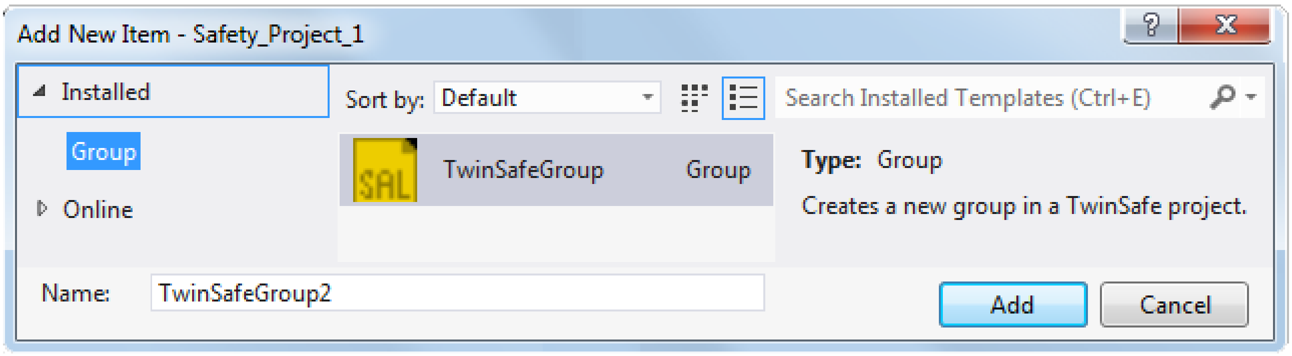
Target System: TwinCAT Safety PLC
Append to Task: not available
Object Name: SafetyPLC Instance
Object Id: 0x01010010
Software Version: 1
Project CRC: not available
Map Object Id: □ Map Project CRC: □
Version Number: 1
Safe Address: 1

Fig. 7: Target System Property Page

4.2.2.4 TwinSAFE groups

Creating TwinSAFE groups makes sense for realizing different safety zones in a machine or in situations where different C++ source files are to be used. A connection error within a group (here: alias device) leads to a ComError for the group, resulting in disabling of all outputs for this group.

A further group can be created by opening the context menu of the safety project and selecting *Add* and *New Item...*.



Add New Item - Safety_Project_1

Sort by: Default

Type: Group

Creates a new group in a TwinSafe project.

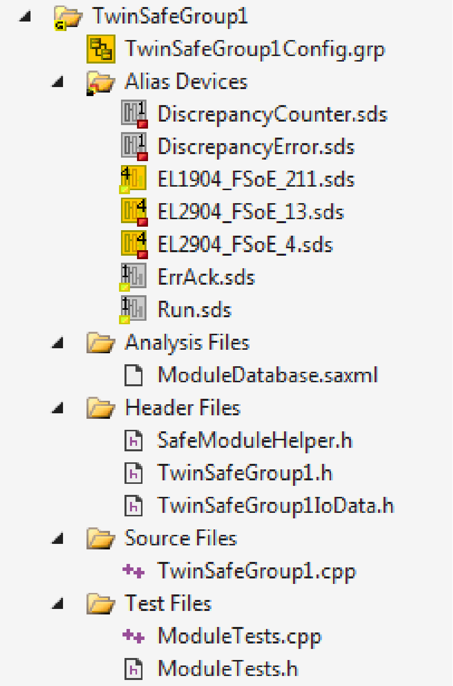
Name: TwinSafeGroup2

Fig. 8: Creating a TwinSAFE group

A group consists of subitems for the group configuration (*.grp), alias devices (*.sds), header files (*.h) and source files (*.cpp). In addition there are subitems for test and for analysis files.

For each group there is one header file and one source file, which the user can use and adapt for the safety application. These are the files *<GroupName>.h* and *<GroupName>.cpp*.

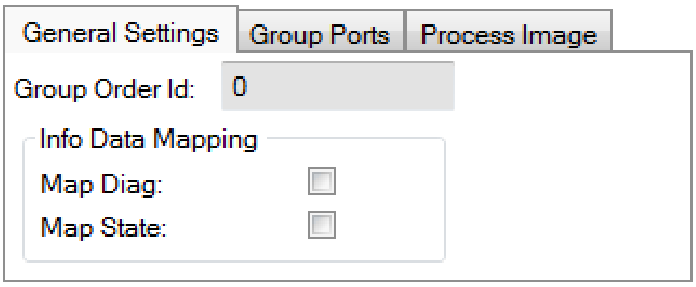
The test files ModuleTests.cpp and ModuleTests.h can be used for debugging the safety application. In these files the safe inputs and outputs can be set and remain set if breakpoints are used, without having to enable the whole configuration. In this state the communication is not safe!



- TwinSafeGroup1
  - TwinSafeGroup1Config.grp
  - Alias Devices
    - DiscrepancyCounter.sds
    - DiscrepancyError.sds
    - EL1904_FSoE_211.sds
    - EL2904_FSoE_13.sds
    - EL2904_FSoE_4.sds
    - ErrAck.sds
    - Run.sds
  - Analysis Files
    - ModuleDatabase.saxml
  - Header Files
    - SafeModuleHelper.h
    - TwinSafeGroup1.h
    - TwinSafeGroup1IoData.h
  - Source Files
    - TwinSafeGroup1.cpp
  - Test Files
    - ModuleTests.cpp
    - ModuleTests.h

Fig. 9: TwinSAFE group

The group configuration is used for the general group settings, including the info data or group ports for error acknowledge and run/stop.



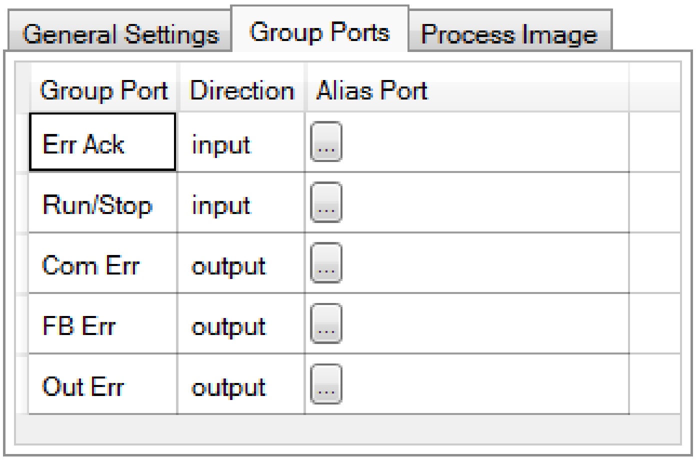
Group Order Id: 0

Info Data Mapping

Map Diag: □

Map State: □

Fig. 10: TwinSAFE Group - General Settings



Group Port	Direction	Alias Port
Err Ack	input	...
Run/Stop	input	...
Com Err	output	...
FB Err	output	...
Out Err	output	...

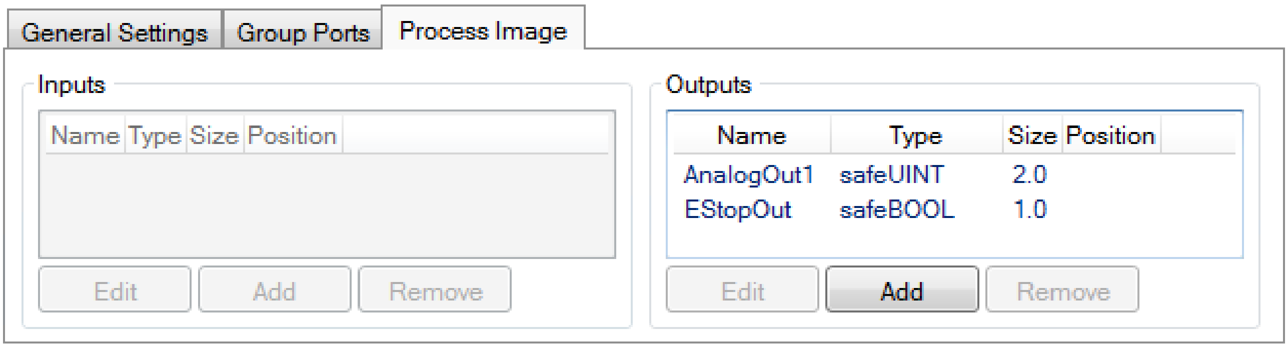
Fig. 11: TwinSAFE Group - Group Ports

In addition, there is an option to create an internal process image for the TwinSAFE group. This process image contains all the signals for use in other TwinSAFE groups. The defined variables are made available to all other groups in a structure called TSGData in the header file <GroupName>IoData.h.

Note

TwinSAFE group outputs

Please ensure that TwinSAFE groups only have outputs in the TSGData structure. These outputs can be read by all other groups. It is not possible to define inputs for a TwinSAFE group.



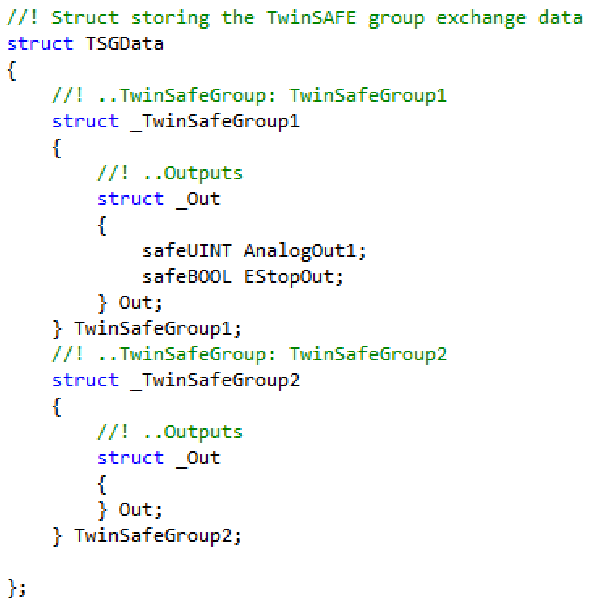
Inputs

Name	Type	Size	Position

Outputs

Name	Type	Size	Position
AnalogOut1	safeUINT	2.0	
EStopOut	safeBOOL	1.0	

Fig. 12: TwinSAFE group process image



```
//! Struct storing the TwinSAFE group exchange data
struct TSGData
{
    //! ..TwinSafeGroup: TwinSafeGroup1
    struct _TwinSafeGroup1
    {
        //! ..Outputs
        struct _Out
        {
            safeUINT AnalogOut1;
            safeBOOL EStopOut;
        } Out;
    } TwinSafeGroup1;
    //! ..TwinSafeGroup: TwinSafeGroup2
    struct _TwinSafeGroup2
    {
        //! ..Outputs
        struct _Out
        {
        } Out;
    } TwinSafeGroup2;

};
```

Fig. 13: TSGData struct

4.2.2.5 Alias devices

The communication between the safety logic and the I/O level is realized via an alias level. At this alias level (subnode *Alias Devices*) corresponding alias devices are created for all safe inputs and outputs, and also for standard signal types. For the safe inputs and outputs, this can be done automatically via the I/O configuration.

The connection- and device-specific parameters are set via the alias devices.

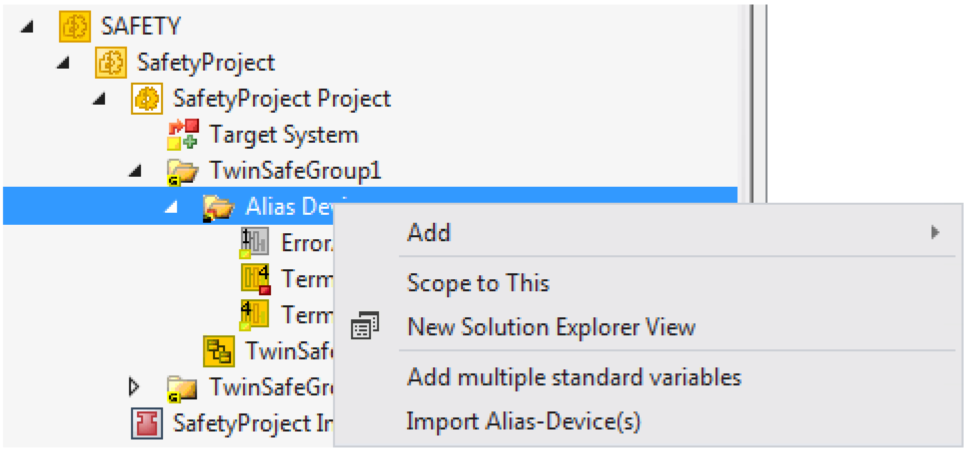
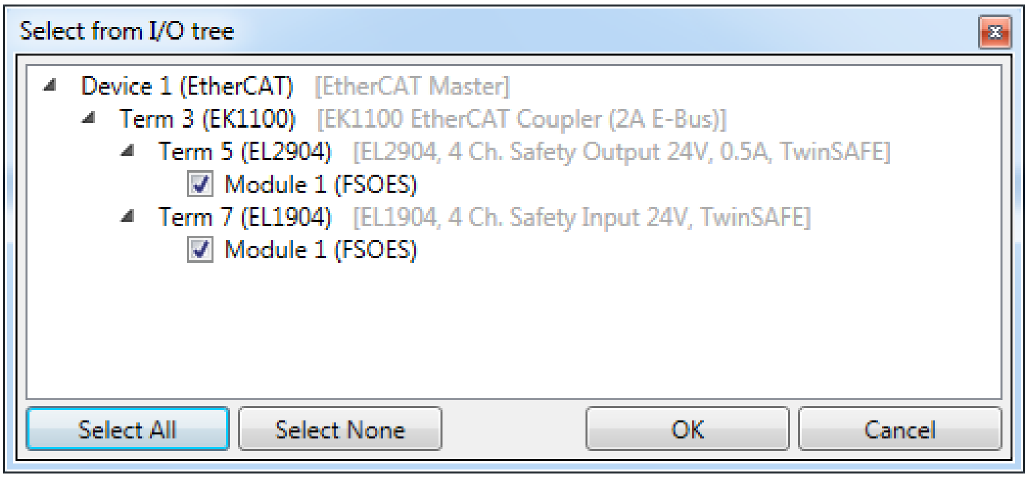


Fig. 14: Starting the automatic import from the I/O configuration

If the automatic import is started from the I/O configuration, a selection dialog opens, in which the individual terminals, which are to be imported automatically, are selected.

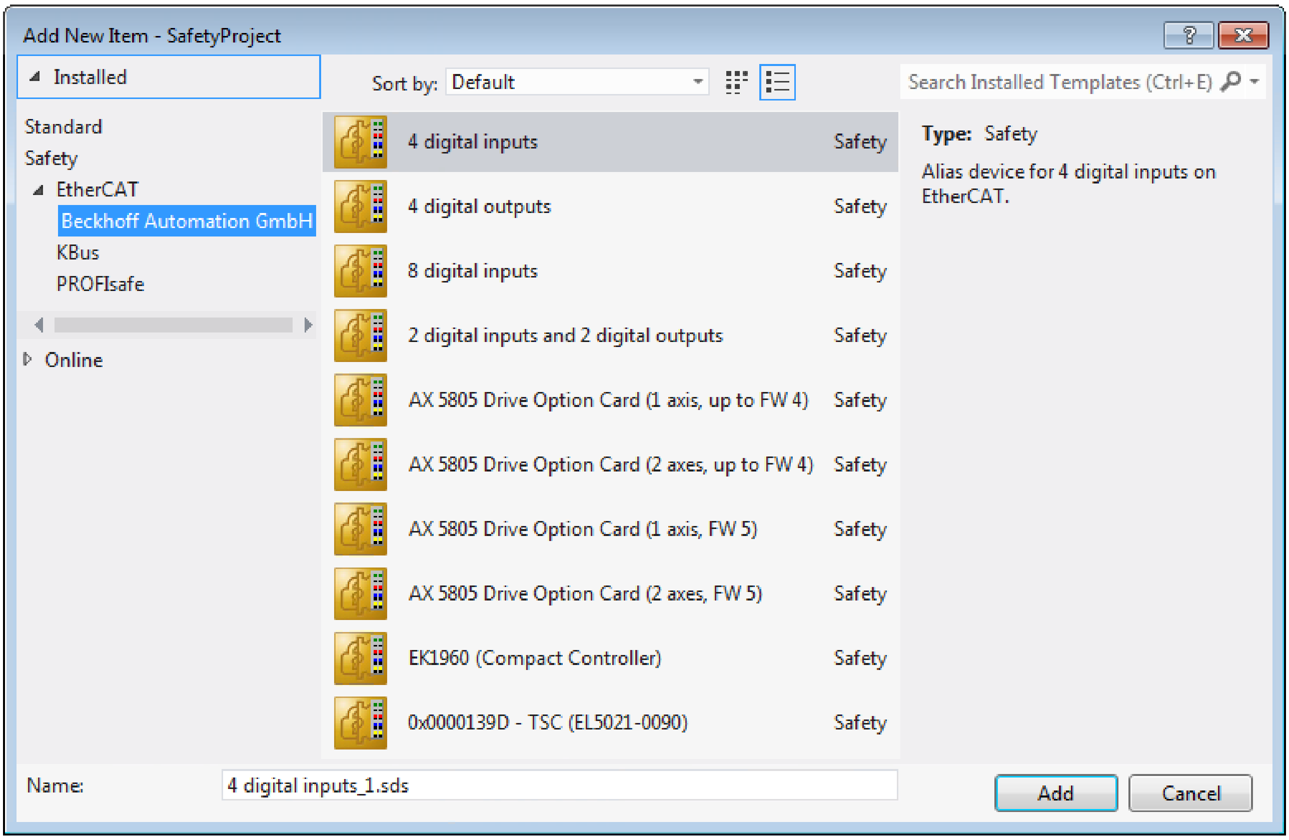


Select from I/O tree
Device 1 (EtherCAT) [EtherCAT Master]
Term 3 (EK1100) [EK1100 EtherCAT Coupler (2A E-Bus)]
Term 5 (EL2904) [EL2904, 4 Ch. Safety Output 24V, 0.5A, TwinSAFE]
☑ Module 1 (FSOES)
Term 7 (EL1904) [EL1904, 4 Ch. Safety Input 24V, TwinSAFE]
☑ Module 1 (FSOES)
Select All
Select None
OK
Cancel

Fig. 15: Selection from the I/O tree

The alias devices are created in the safety project when the dialog is closed via OK.

Alternatively, the user can create the alias devices individually. To this end select *Add* and *New item* from the context menu, followed by the required device.



Add New Item - SafetyProject
Installed
Standard
Safety
EtherCAT
Beckhoff Automation GmbH
KBus
PROFIsafe
Online
Sort by: Default
Search Installed Templates (Ctrl+E)
4 digital inputs Safety
4 digital outputs Safety
8 digital inputs Safety
2 digital inputs and 2 digital outputs Safety
AX 5805 Drive Option Card (1 axis, up to FW 4) Safety
AX 5805 Drive Option Card (2 axes, up to FW 4) Safety
AX 5805 Drive Option Card (1 axis, FW 5) Safety
AX 5805 Drive Option Card (2 axes, FW 5) Safety
EK1960 (Compact Controller) Safety
0x0000139D - TSC (EL5021-0090) Safety
Add
Cancel

Type: Safety

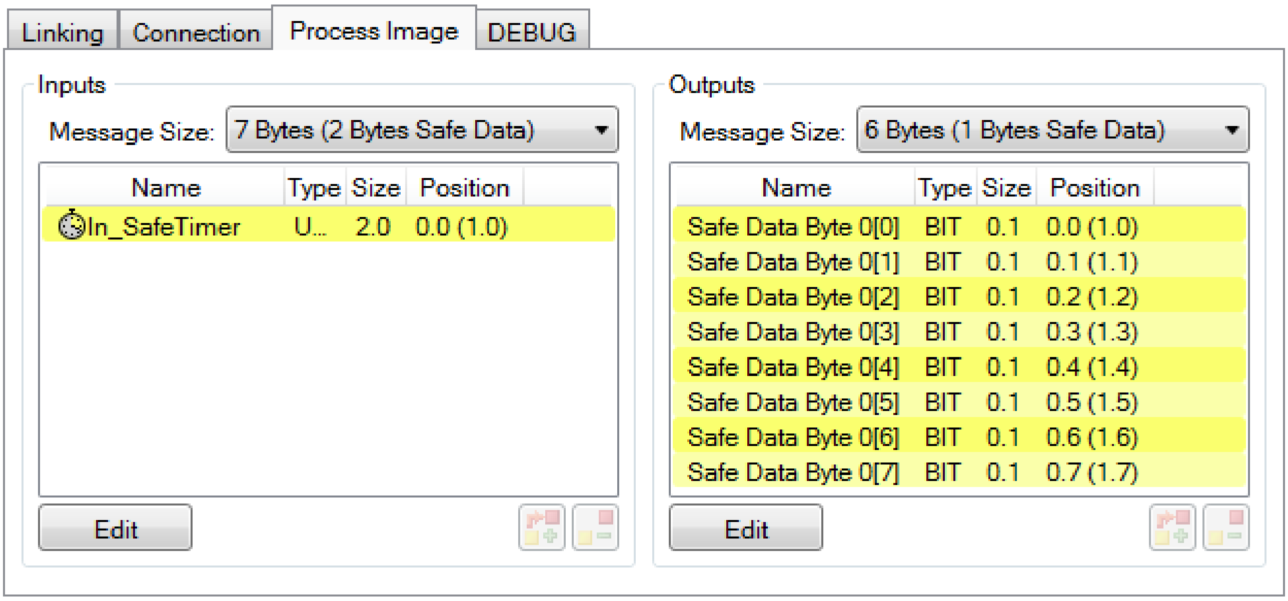
Alias device for 4 digital inputs on EtherCAT.

Name: 4 digital inputs_1.sds

Fig. 16: Creating alias devices by the user

4.2.2.6 Safe time signal

A safety project for the TwinCAT Safety PLC is only valid if a safe external time signal is available for executing the safety project. To this end at least one of the safe communication links must offer functionality for providing a safe time signal via the safe communication link. This may be an EL6910 TwinSAFE component, for example. The EL6910 TwinSAFE component is used to illustrate the process of assigning a safe time value to the input process image via the *Process Image* tab of the alias device.



Inputs

Message Size: 7 Bytes (2 Bytes Safe Data)

Name	Type	Size	Position
In_SafeTimer	U...	2.0	0.0 (1.0)

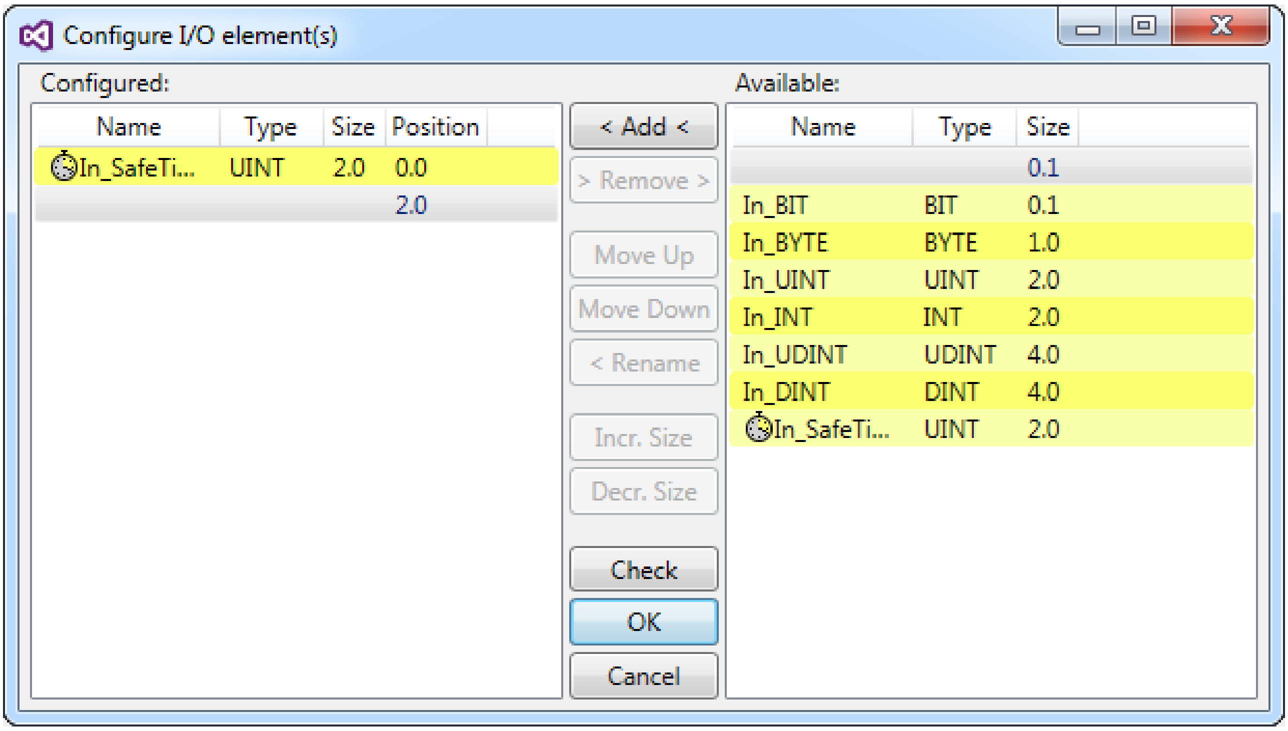
Outputs

Message Size: 6 Bytes (1 Bytes Safe Data)

Name	Type	Size	Position
Safe Data Byte 0[0]	BIT	0.1	0.0 (1.0)
Safe Data Byte 0[1]	BIT	0.1	0.1 (1.1)
Safe Data Byte 0[2]	BIT	0.1	0.2 (1.2)
Safe Data Byte 0[3]	BIT	0.1	0.3 (1.3)
Safe Data Byte 0[4]	BIT	0.1	0.4 (1.4)
Safe Data Byte 0[5]	BIT	0.1	0.5 (1.5)
Safe Data Byte 0[6]	BIT	0.1	0.6 (1.6)
Safe Data Byte 0[7]	BIT	0.1	0.7 (1.7)

Fig. 17: Alias device - Process Image tab

Select *Edit* in this dialog to adapt the process image and add the SafeTimer.



Configured:

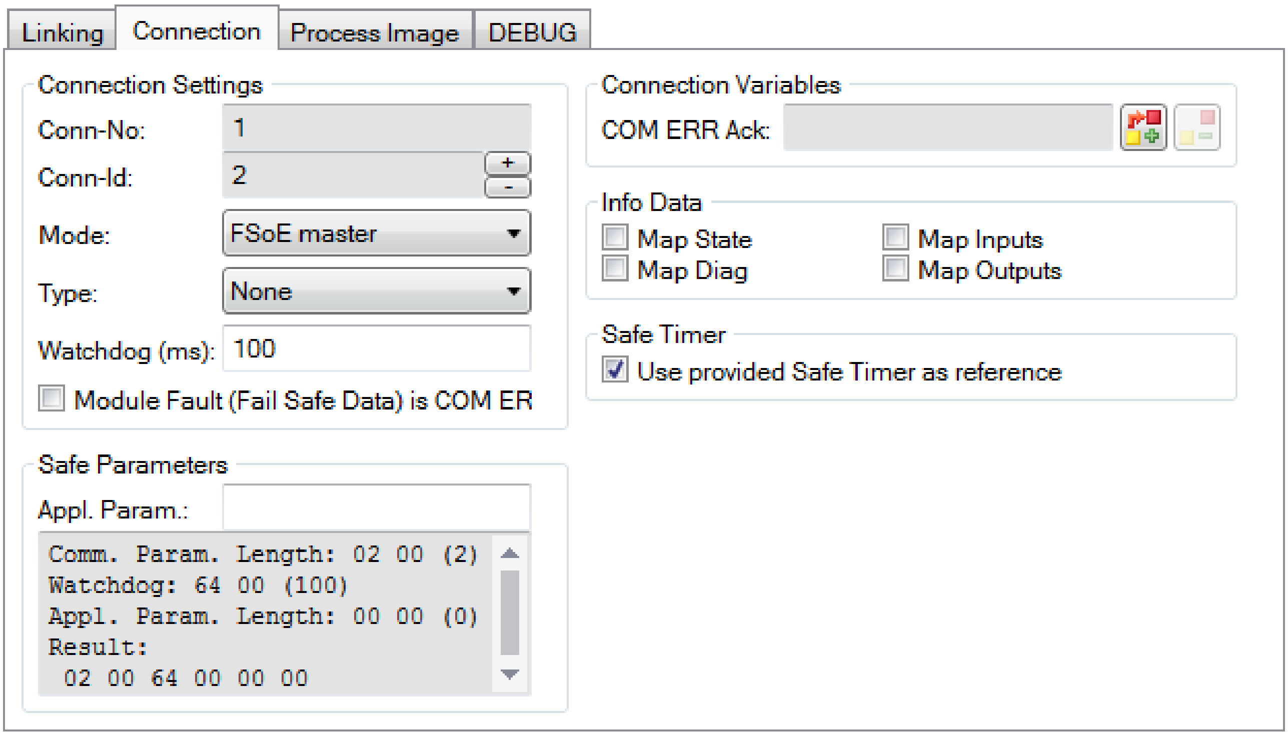
Name	Type	Size	Position
In_SafeTi...	UINT	2.0	0.0
			2.0

Available:

Name	Type	Size
		0.1
In_BIT	BIT	0.1
In_BYTE	BYTE	1.0
In_UINT	UINT	2.0
In_INT	INT	2.0
In_UDINT	UDINT	4.0
In_DINT	DINT	4.0
In_SafeTi...	UINT	2.0

Fig. 18: Configuring the I/O elements

In addition, tick the checkbox for *Use provided Safe Timer as reference* under the *Connection* tab.



Connection Settings

Conn-No: 1

Conn-Id: 2

Mode: FSoE master

Type: None

Watchdog (ms): 100

□ Module Fault (Fail Safe Data) is COM ER

Safe Parameters

Appl. Param.:

```
Comm. Param. Length: 02 00 (2)
Watchdog: 64 00 (100)
Appl. Param. Length: 00 00 (0)
Result:
 02 00 64 00 00 00
```

Connection Variables

COM ERR Ack:

Info Data

□ Map State □ Map Inputs

□ Map Diag □ Map Outputs

Safe Timer

☑ Use provided Safe Timer as reference

Fig. 19: Alias device - Connection tab

For a safety project a specific TwinSAFE component must be selected as provider for a safe time signal, to ensure that a safety project can be loaded and started successfully. The safety project is only executed if the provided safe time signal is available (i.e. the corresponding communication link must be in DATA state).

An error in the context of the safe time signal leads to triggering of the safe state for the TwinCAT Safety PLC.

4.2.2.7 Parameterization of the alias device

The settings can be opened by double-clicking on the alias device in the safety project structure.

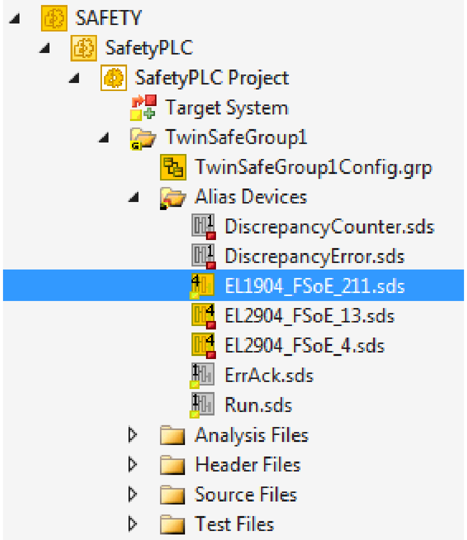
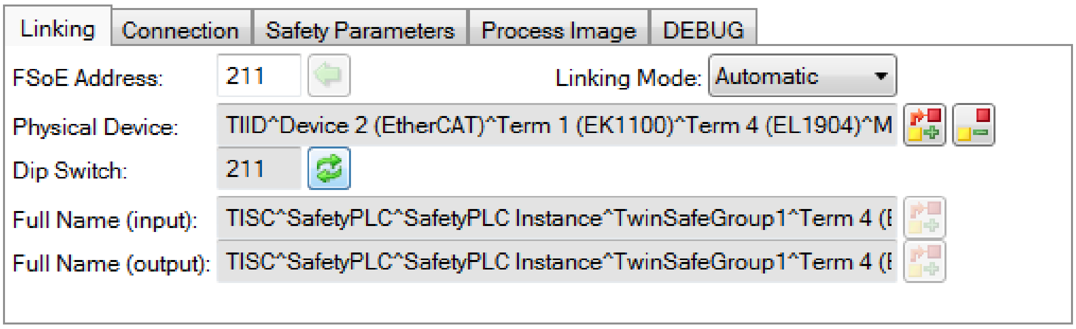


Fig. 20: Alias device in the safety project structure

The *Linking* tab contains the FSoE address, the checkbox for setting as *External Device* and the link to the physical I/O device. If an ADS online connection to the physical I/O device exists, the DIP switch setting is displayed. Re-reading of the setting can be started via the button . The links to the TwinCAT Safety PLC process image are displayed under *Full Name (input)* and *Full Name (output)*.



Linking | Connection | Safety Parameters | Process Image | DEBUG

FSoE Address: 211

Linking Mode: Automatic

Physical Device: TIID^Device 2 (EtherCAT)^Term 1 (EK1100)^Term 4 (EL1904)^M

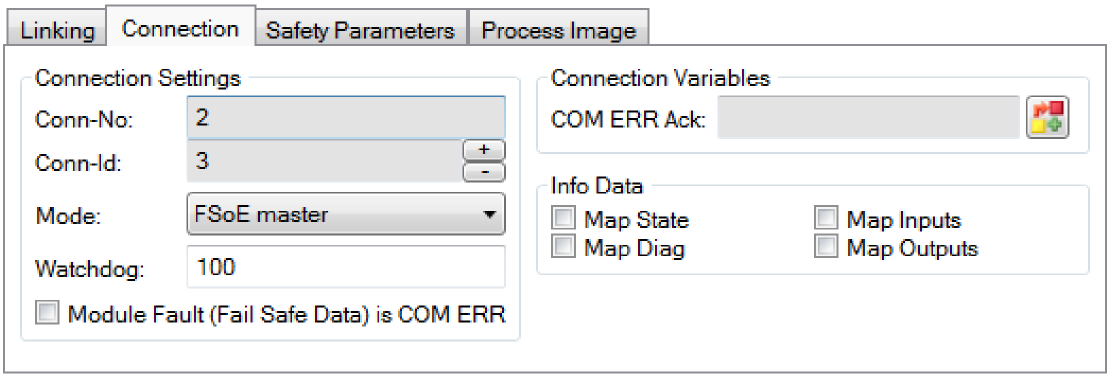
Dip Switch: 211

Full Name (input): TISC^SafetyPLC^SafetyPLC Instance^TwinSafeGroup1^Term 4 (E

Full Name (output): TISC^SafetyPLC^SafetyPLC Instance^TwinSafeGroup1^Term 4 (E

Fig. 21: Links to the TwinCAT Safety PLC process image

The *Connection* tab shows the connection-specific parameters.



Linking | Connection | Safety Parameters | Process Image

Connection Settings

Conn-No: 2

Conn-Id: 3

Mode: FSoE master

Watchdog: 100

□ Module Fault (Fail Safe Data) is COM ERR

Connection Variables

COM ERR Ack:

Info Data

□ Map State □ Map Inputs

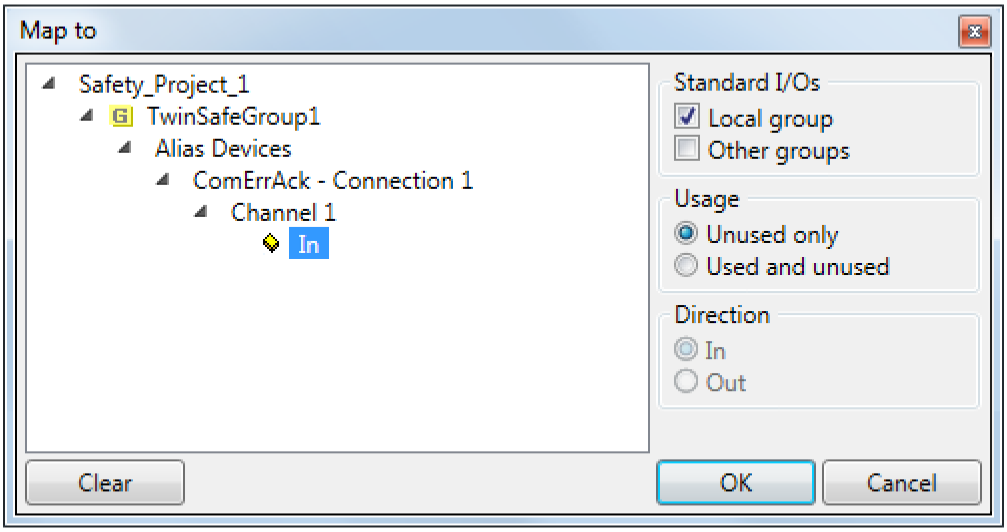
□ Map Diag □ Map Outputs

Fig. 22: Connection-specific parameters

Parameter	Description	User interaction required
Conn-No.	Connection number - automatically assigned by the TwinCAT system	No
Conn-ID	Connection ID: pre-allocated by the system; can be changed by the user. A Conn ID must be unique within a configuration. Duplicate connection IDs result in an error message.	Control
Mode	FSoE master: TwinCAT Safety PLC is FSoE master for this device. FSoE slave: TwinCAT Safety PLC is FSoE slave for this device.	Control
Watchdog	Watchdog time for this connection. A ComError is generated, if the device fails to return a valid telegram to the TwinCAT Safety PLC within the watchdog time.	Yes
Module Fault is ComError	This checkbox is used to specify the behavior in the event of an error. If the checkbox is ticked and a module error occurs on the alias device, this also leads to a connection error and therefore to disabling of the TwinSAFE group, in which this connection is defined.	Yes
ComErrAck	If ComErrAck is linked to a variable, the connection must be reset via this signal in the event of a communication error, before the corresponding group can be reset.	Yes
Info Data	The info data to be shown in the process image of the TwinCAT Safety PLC can be defined via these checkboxes. Further information can be found in the documentation for *TwinCAT function blocks for TwinSAFE logic terminals*.	Yes

The TwinCAT Safety PLC supports activation of a ComErrAck for each connection. If this signal is connected, the respective connection must be reset after a communication error via the signal ComErrAck,

in addition to the ErrAck of the TwinSAFE group. This signal is linked via the link button next to COM ERR Ack. The following dialog can be used for selecting an alias device. The signal can be canceled via the *Clear* button in the link dialog.



- Safety_Project_1
  - TwinSafeGroup1
    - Alias Devices
      - ComErrAck - Connection 1
        - Channel 1
          - In

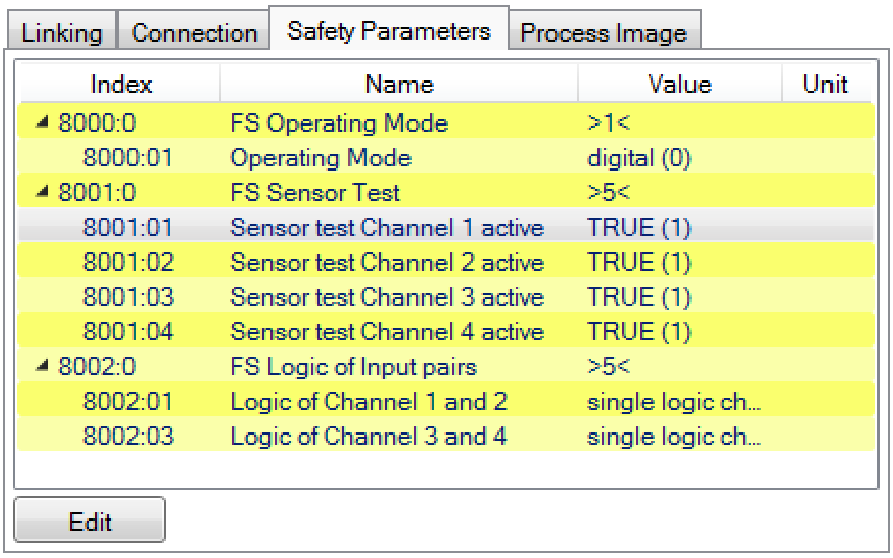
Standard I/Os
☑ Local group
☐ Other groups

Usage
● Unused only
○ Used and unused

Direction
○ In
○ Out

Fig. 23: Selecting an alias device

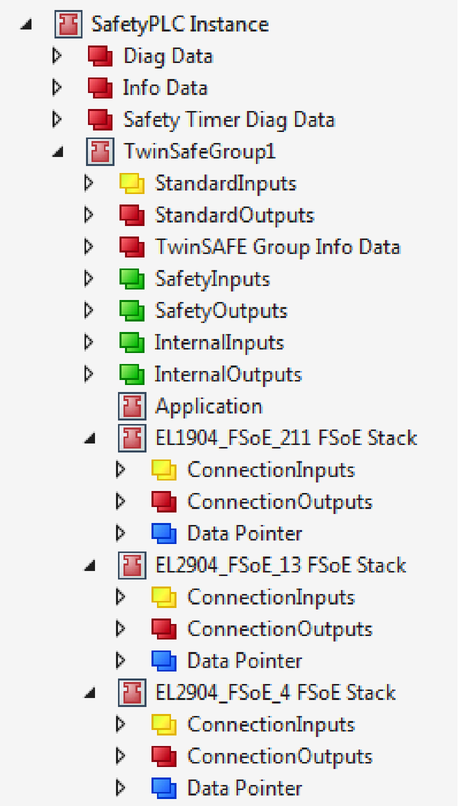
The safety parameters matching the device are displayed under the *Safety Parameters* tab. They have to be set correctly to match the required performance level. Further information can be found in the TwinSAFE application manual.



Index	Name	Value	Unit
8000:0	FS Operating Mode	>1<	
8000:01	Operating Mode	digital (0)	
8001:0	FS Sensor Test	>5<	
8001:01	Sensor test Channel 1 active	TRUE (1)	
8001:02	Sensor test Channel 2 active	TRUE (1)	
8001:03	Sensor test Channel 3 active	TRUE (1)	
8001:04	Sensor test Channel 4 active	TRUE (1)	
8002:0	FS Logic of Input pairs	>5<	
8002:01	Logic of Channel 1 and 2	single logic ch...	
8002:03	Logic of Channel 3 and 4	single logic ch...	

Fig. 24: Safety parameter for the device

For each alias device an entry with the corresponding FSoE stack is created in the safety PLC. It contains links to the safe input and output components and also provides the data pointer for access to the safe inputs and outputs within the safety application.

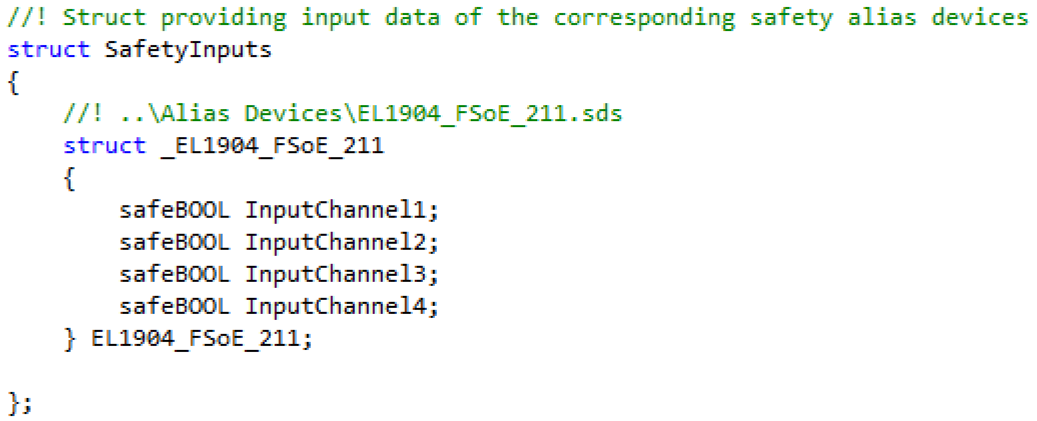


```
SafetyPLC Instance
    Diag Data
    Info Data
    Safety Timer Diag Data
    TwinSafeGroup1
        StandardInputs
        StandardOutputs
        TwinSAFE Group Info Data
        SafetyInputs
        SafetyOutputs
        InternalInputs
        InternalOutputs
        Application
        EL1904_FSoE_211 FSoE Stack
            ConnectionInputs
            ConnectionOutputs
            Data Pointer
        EL2904_FSoE_13 FSoE Stack
            ConnectionInputs
            ConnectionOutputs
            Data Pointer
        EL2904_FSoE_4 FSoE Stack
            ConnectionInputs
            ConnectionOutputs
            Data Pointer
```

Fig. 25: Safety PLC instance - Alias devices

The data for each individual connection are declared as struct data type in <GroupName>IoData.h and instantiated in the header file <GroupName>.h.

The user can access a safe input directly via the instance variable, e.g. *sSafetyInputs.EL1904_FSoE_211.InputChannel1*.



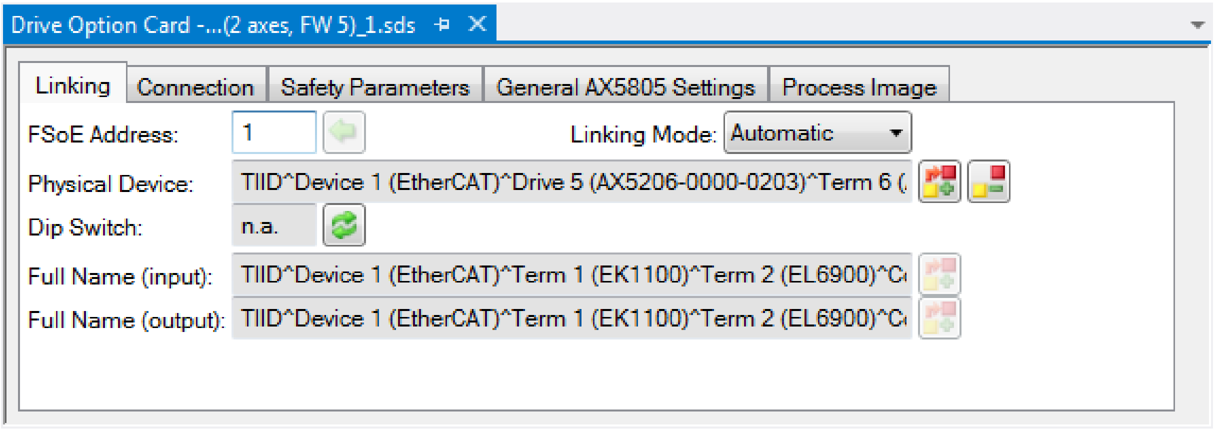
```
//! Struct providing input data of the corresponding safety alias devices
struct SafetyInputs
{
    //! ..\Alias Devices\EL1904_FSoE_211.sds
    struct _EL1904_FSoE_211
    {
        safeBOOL InputChannel1;
        safeBOOL InputChannel2;
        safeBOOL InputChannel3;
        safeBOOL InputChannel4;
    } EL1904_FSoE_211;

};
```

Fig. 26: Structure of the alias device

4.2.2.8 Connection to AX5805/AX5806

There are separate dialogs for linking an AX5805 or AX5806 TwinSAFE Drive option card, which can be used to set the safety functions of the AX5000 safety drive options.

Creating and opening of an alias device for an AX5805 results in five tabs; the *Linking*, *Connection* and *Safety Parameters* tabs are identical to other alias devices.



FSoE Address: 1

Linking Mode: Automatic

Physical Device: TIID^Device 1 (EtherCAT)^Drive 5 (AX5206-0000-0203)^Term 6 (

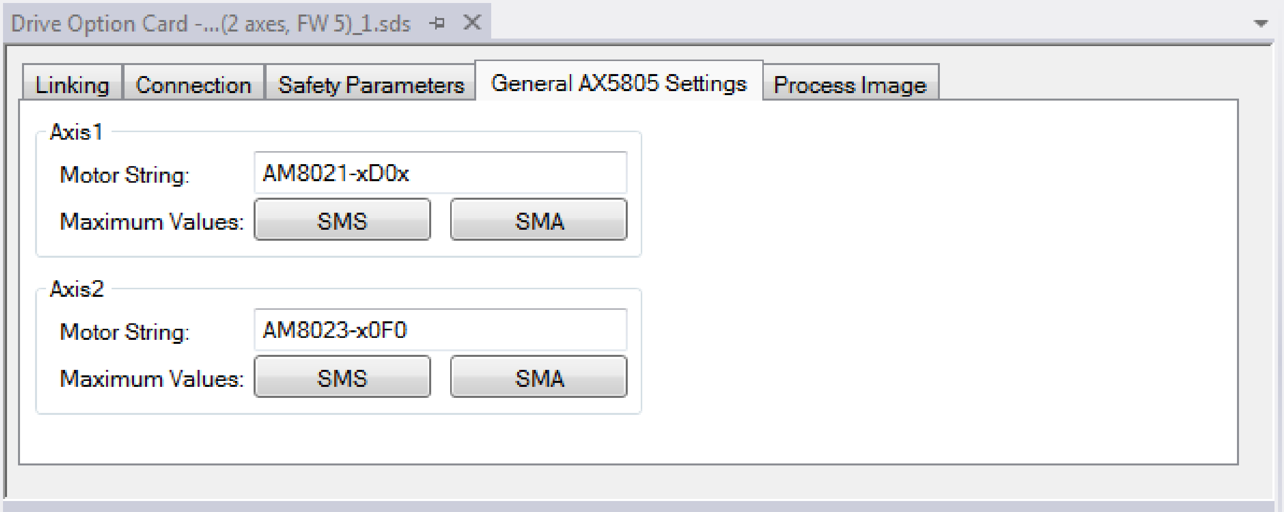
Dip Switch: n.a.

Full Name (input): TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 2 (EL6900)^C

Full Name (output): TIID^Device 1 (EtherCAT)^Term 1 (EK1100)^Term 2 (EL6900)^C

Fig. 27: AX5000 safety drive functions

The *General AX5805 Settings* tab can be used to set the motor string and the SMS and SMA functions for one or two axes, depending on the added alias device.



Axis1

Motor String: AM8021-xD0x

Maximum Values:

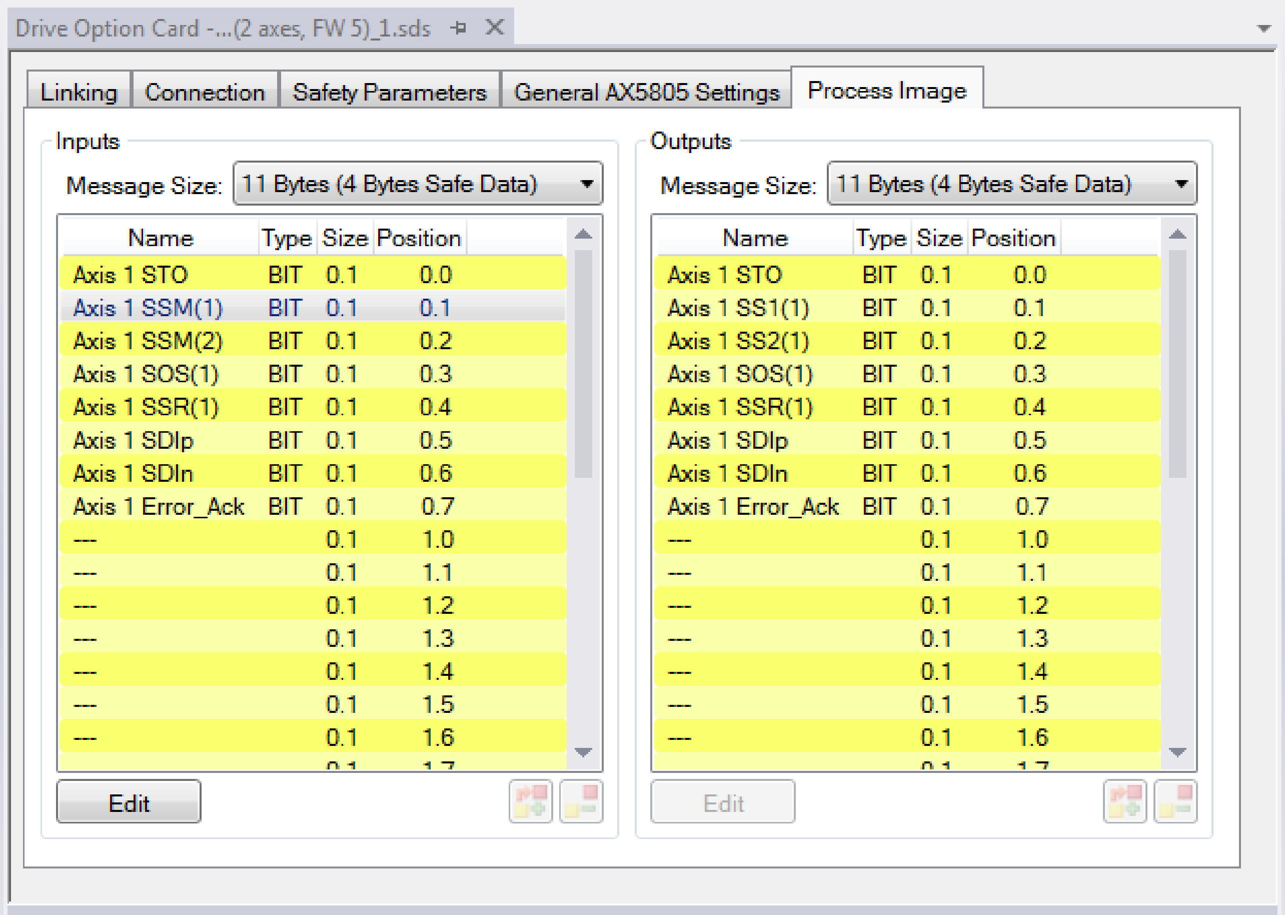
Axis2

Motor String: AM8023-x0F0

Maximum Values:

Fig. 28: AX5000 safety drive options - general AX5805 settings

The Process Image tab can be used to set the different safety functions for the AX5805.



Inputs

Message Size: 11 Bytes (4 Bytes Safe Data)

Name	Type	Size	Position
Axis 1 STO	BIT	0.1	0.0
Axis 1 SSM(1)	BIT	0.1	0.1
Axis 1 SSM(2)	BIT	0.1	0.2
Axis 1 SOS(1)	BIT	0.1	0.3
Axis 1 SSR(1)	BIT	0.1	0.4
Axis 1 SDIp	BIT	0.1	0.5
Axis 1 SDIn	BIT	0.1	0.6
Axis 1 Error_Ack	BIT	0.1	0.7
---		0.1	1.0
---		0.1	1.1
---		0.1	1.2
---		0.1	1.3
---		0.1	1.4
---		0.1	1.5
---		0.1	1.6
		0.1	1.7

Outputs

Message Size: 11 Bytes (4 Bytes Safe Data)

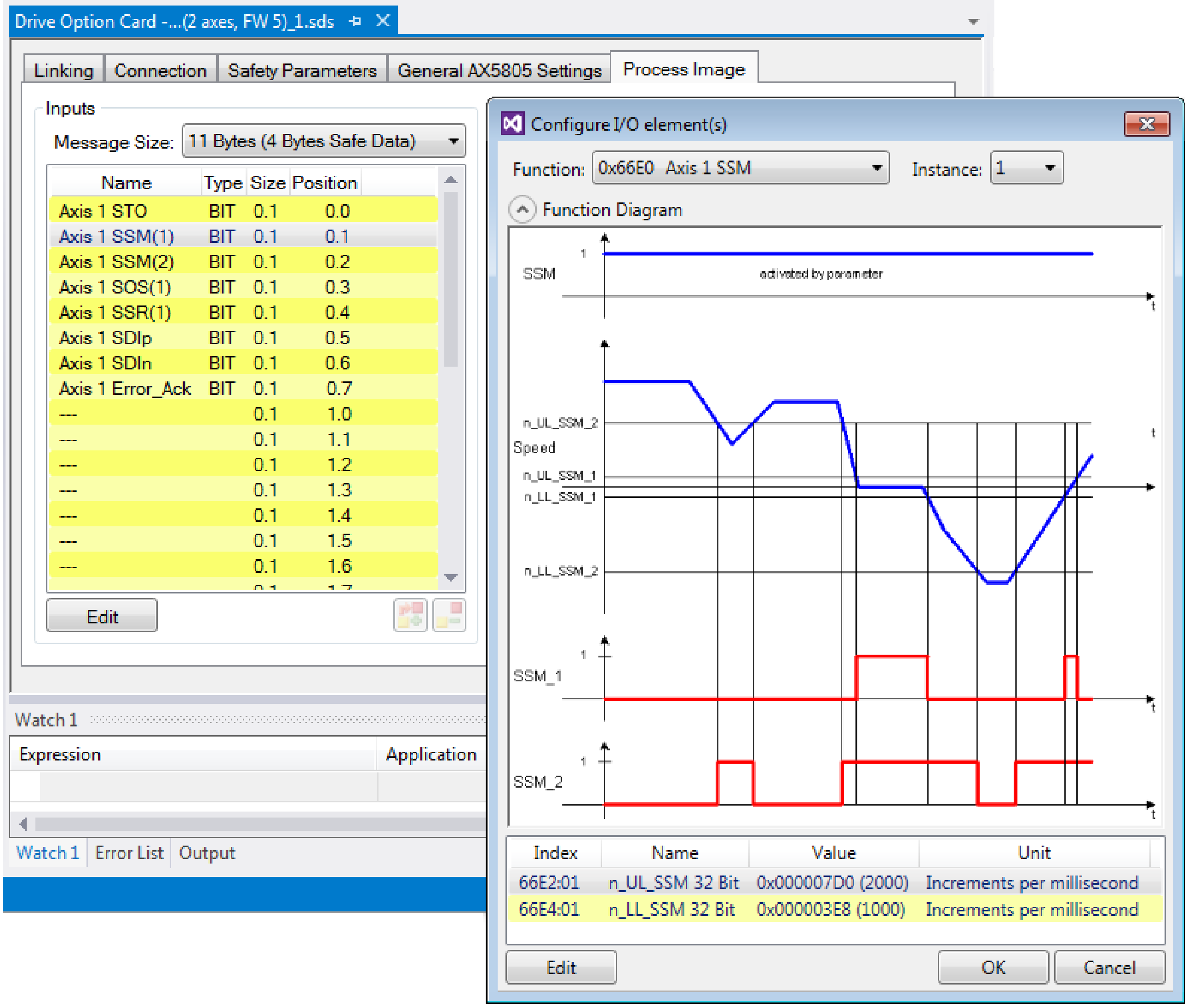
Name	Type	Size	Position
Axis 1 STO	BIT	0.1	0.0
Axis 1 SS1(1)	BIT	0.1	0.1
Axis 1 SS2(1)	BIT	0.1	0.2
Axis 1 SOS(1)	BIT	0.1	0.3
Axis 1 SSR(1)	BIT	0.1	0.4
Axis 1 SDIp	BIT	0.1	0.5
Axis 1 SDIn	BIT	0.1	0.6
Axis 1 Error_Ack	BIT	0.1	0.7
---		0.1	1.0
---		0.1	1.1
---		0.1	1.2
---		0.1	1.3
---		0.1	1.4
---		0.1	1.5
---		0.1	1.6
		0.1	1.7

Fig. 29: AX5000 safety drive options - Process Image

The parameters under the *General AX5805 Settings* and *Process Image* tabs are identical to the parameters under the *Safety Parameters* tab. Offers user-friendly display and editing of the parameters. The parameters under the *Safety Parameters* tab can also be edited.

The parameters for this function can be set by selecting a function in the inputs or outputs and pressing the *Edit* button. New safety functions can be added in the process image by selecting an empty field (---) and pressing *Edit*.

The parameter list corresponding to the safety function can be shown; in addition, an optional diagram of the function can be shown. At present the diagram is still static and does not show the currently selected values.



Message Size: 11 Bytes (4 Bytes Safe Data)

Name	Type	Size	Position
Axis 1 STO	BIT	0.1	0.0
Axis 1 SSM(1)	BIT	0.1	0.1
Axis 1 SSM(2)	BIT	0.1	0.2
Axis 1 SOS(1)	BIT	0.1	0.3
Axis 1 SSR(1)	BIT	0.1	0.4
Axis 1 SDIp	BIT	0.1	0.5
Axis 1 SDIn	BIT	0.1	0.6
Axis 1 Error_Ack	BIT	0.1	0.7
---		0.1	1.0
---		0.1	1.1
---		0.1	1.2
---		0.1	1.3
---		0.1	1.4
---		0.1	1.5
---		0.1	1.6
		0.1	1.7

Function: 0x66E0 Axis 1 SSM

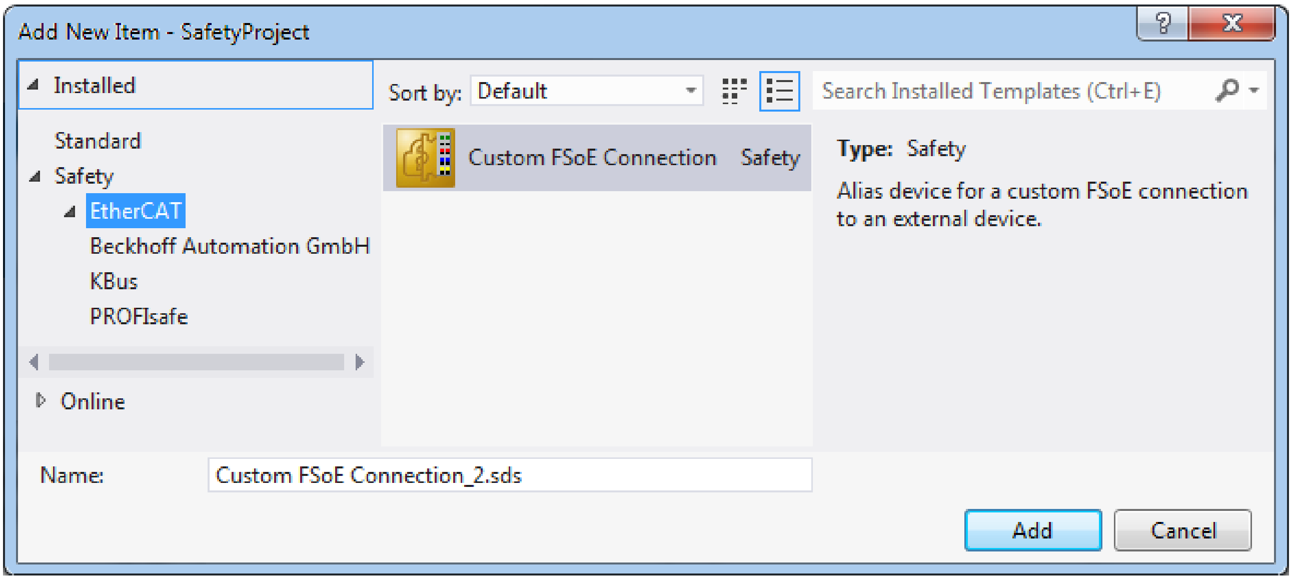
Instance: 1

Index	Name	Value	Unit
66E2:01	n_UL_SSM 32 Bit	0x000007D0 (2000)	Increments per millisecond
66E4:01	n_LL_SSM 32 Bit	0x000003E8 (1000)	Increments per millisecond

Fig. 30: AX5000 safety drive options - Function Diagram

4.2.2.9 External connection

An external *Custom FSoE Connection* can be created for a connection to a further EL69x0, EJ6910, KL6904 or third-party device. If a dedicated ESI file exists for a third-party device, the device is listed as a selectable safety device, and the *Custom FSoE Connection* option is not required.



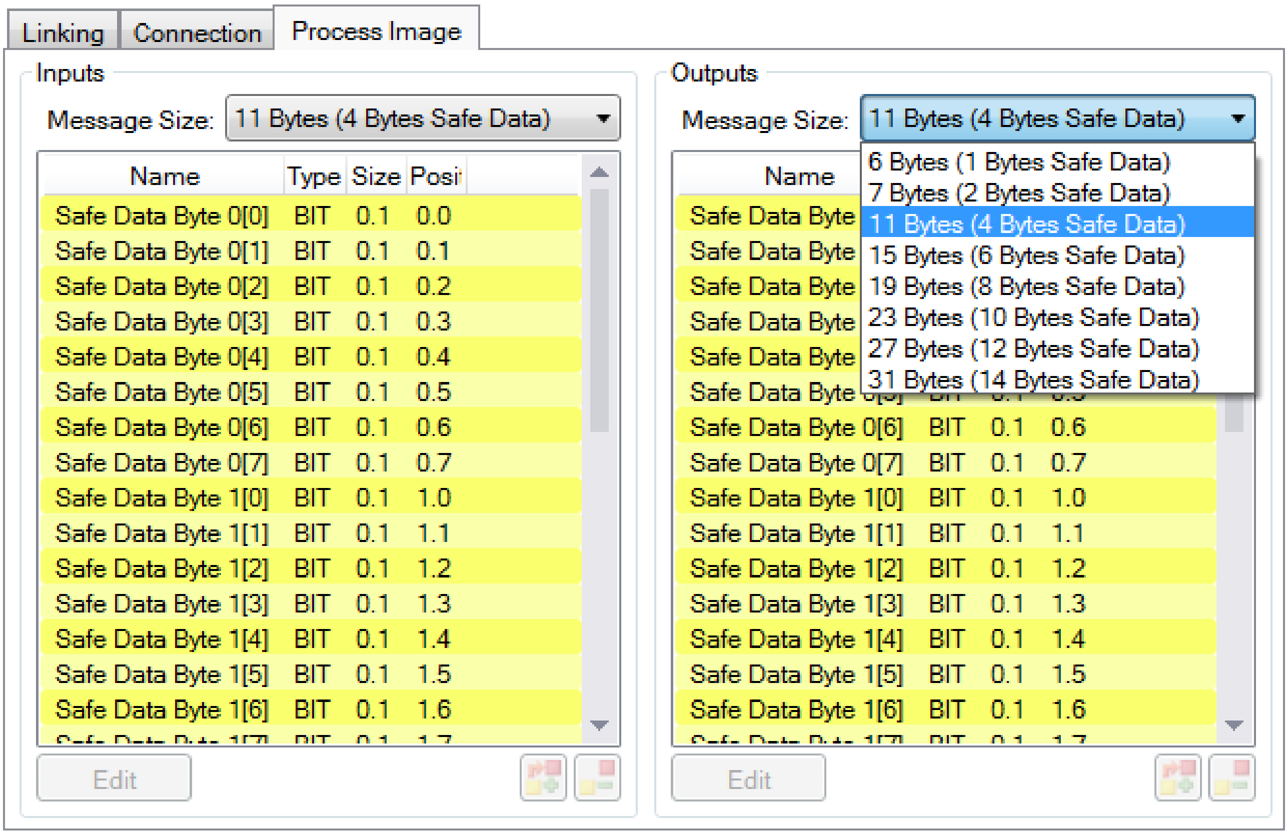
Type: Safety

Alias device for a custom FSoE connection to an external device.

Name: Custom FSoE Connection_2.sds

Fig. 31: Creating an external connection (Custom FSoE Connection)

Before the connection can be used and linked further, the process image size must be parameterized. This can be set under the *Process Image* tab. Suitable data types for different numbers of safety data are provided in the dropdown lists for the input and output parameters.



Inputs

Message Size: 11 Bytes (4 Bytes Safe Data)

Name	Type	Size	Posi
Safe Data Byte 0[0]	BIT	0.1	0.0
Safe Data Byte 0[1]	BIT	0.1	0.1
Safe Data Byte 0[2]	BIT	0.1	0.2
Safe Data Byte 0[3]	BIT	0.1	0.3
Safe Data Byte 0[4]	BIT	0.1	0.4
Safe Data Byte 0[5]	BIT	0.1	0.5
Safe Data Byte 0[6]	BIT	0.1	0.6
Safe Data Byte 0[7]	BIT	0.1	0.7
Safe Data Byte 1[0]	BIT	0.1	1.0
Safe Data Byte 1[1]	BIT	0.1	1.1
Safe Data Byte 1[2]	BIT	0.1	1.2
Safe Data Byte 1[3]	BIT	0.1	1.3
Safe Data Byte 1[4]	BIT	0.1	1.4
Safe Data Byte 1[5]	BIT	0.1	1.5
Safe Data Byte 1[6]	BIT	0.1	1.6
Safe Data Byte 1[7]	BIT	0.1	1.7

Outputs

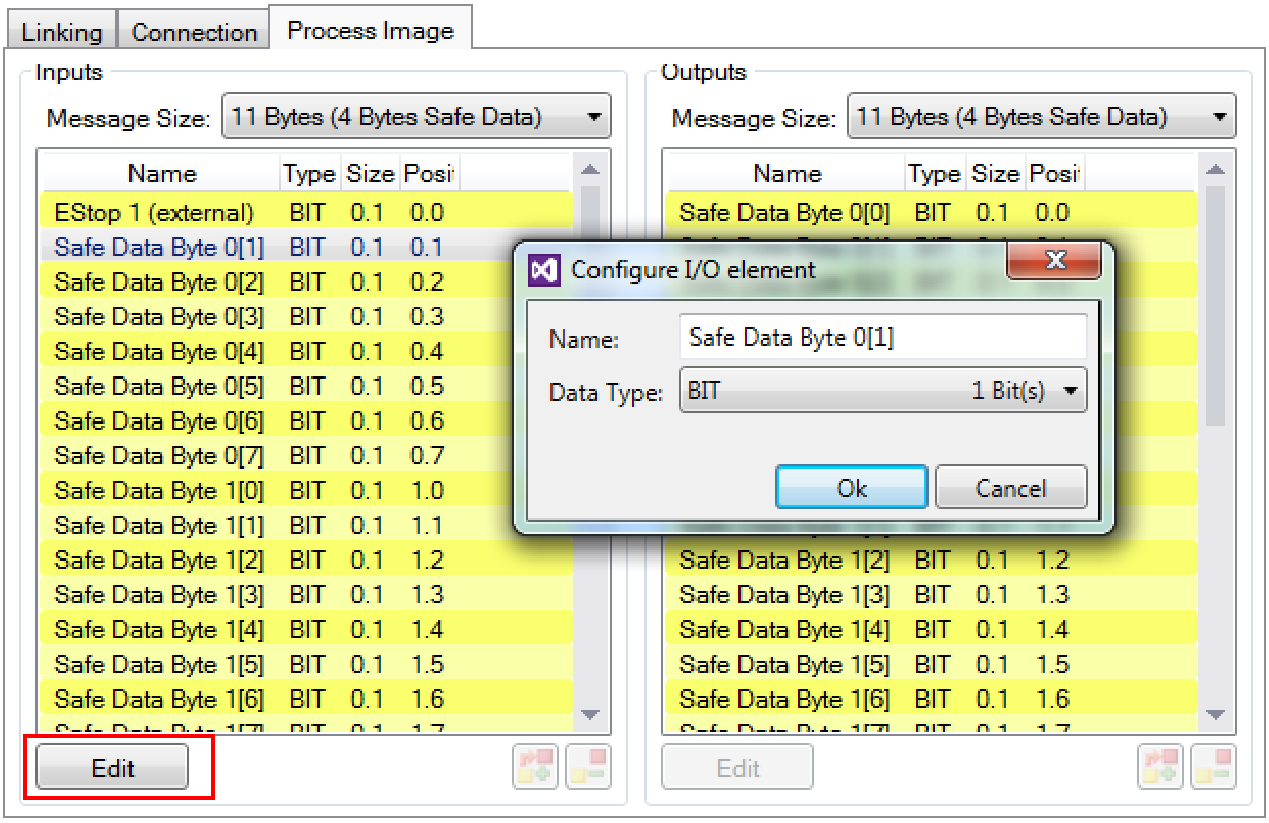
Message Size: 11 Bytes (4 Bytes Safe Data)

- 6 Bytes (1 Bytes Safe Data)
- 7 Bytes (2 Bytes Safe Data)
- 11 Bytes (4 Bytes Safe Data)
- 15 Bytes (6 Bytes Safe Data)
- 19 Bytes (8 Bytes Safe Data)
- 23 Bytes (10 Bytes Safe Data)
- 27 Bytes (12 Bytes Safe Data)
- 31 Bytes (14 Bytes Safe Data)

Name			
Safe Data Byte			
Safe Data Byte			
Safe Data Byte			
Safe Data Byte			
Safe Data Byte			
Safe Data Byte			
Safe Data Byte 0[6]	BIT	0.1	0.6
Safe Data Byte 0[7]	BIT	0.1	0.7
Safe Data Byte 1[0]	BIT	0.1	1.0
Safe Data Byte 1[1]	BIT	0.1	1.1
Safe Data Byte 1[2]	BIT	0.1	1.2
Safe Data Byte 1[3]	BIT	0.1	1.3
Safe Data Byte 1[4]	BIT	0.1	1.4
Safe Data Byte 1[5]	BIT	0.1	1.5
Safe Data Byte 1[6]	BIT	0.1	1.6
Safe Data Byte 1[7]	BIT	0.1	1.7

Fig. 32: Parameterization of the process image size

Once the size is selected, the individual signals within the telegram can be renamed, so that a corresponding plain text is displayed when these signals are used in the logic. If the signals are not renamed, the default name is displayed in the editor (Safe Data Byte 0[0], …).



Inputs

Message Size: 11 Bytes (4 Bytes Safe Data)

Name	Type	Size	Posi
EStop 1 (external)	BIT	0.1	0.0
Safe Data Byte 0[1]	BIT	0.1	0.1
Safe Data Byte 0[2]	BIT	0.1	0.2
Safe Data Byte 0[3]	BIT	0.1	0.3
Safe Data Byte 0[4]	BIT	0.1	0.4
Safe Data Byte 0[5]	BIT	0.1	0.5
Safe Data Byte 0[6]	BIT	0.1	0.6
Safe Data Byte 0[7]	BIT	0.1	0.7
Safe Data Byte 1[0]	BIT	0.1	1.0
Safe Data Byte 1[1]	BIT	0.1	1.1
Safe Data Byte 1[2]	BIT	0.1	1.2
Safe Data Byte 1[3]	BIT	0.1	1.3
Safe Data Byte 1[4]	BIT	0.1	1.4
Safe Data Byte 1[5]	BIT	0.1	1.5
Safe Data Byte 1[6]	BIT	0.1	1.6

Outputs

Message Size: 11 Bytes (4 Bytes Safe Data)

Name	Type	Size	Posi
Safe Data Byte 0[0]	BIT	0.1	0.0
Safe Data Byte 1[2]	BIT	0.1	1.2
Safe Data Byte 1[3]	BIT	0.1	1.3
Safe Data Byte 1[4]	BIT	0.1	1.4
Safe Data Byte 1[5]	BIT	0.1	1.5
Safe Data Byte 1[6]	BIT	0.1	1.6

Configure I/O element

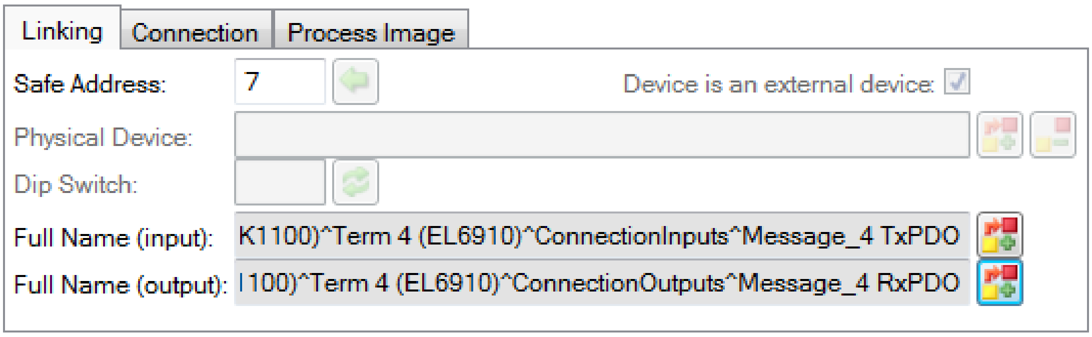
Name: Safe Data Byte 0[1]

Data Type: BIT 1 Bit(s)

Ok Cancel

Fig. 33: Renaming the individual signals within the telegram

The connection is linked under the *Linking* tab. The Link button next to *Full Name (input)* and *Full Name (output)* can be used to select the corresponding variable.



Safe Address: 7

Device is an external device: ☑

Physical Device:

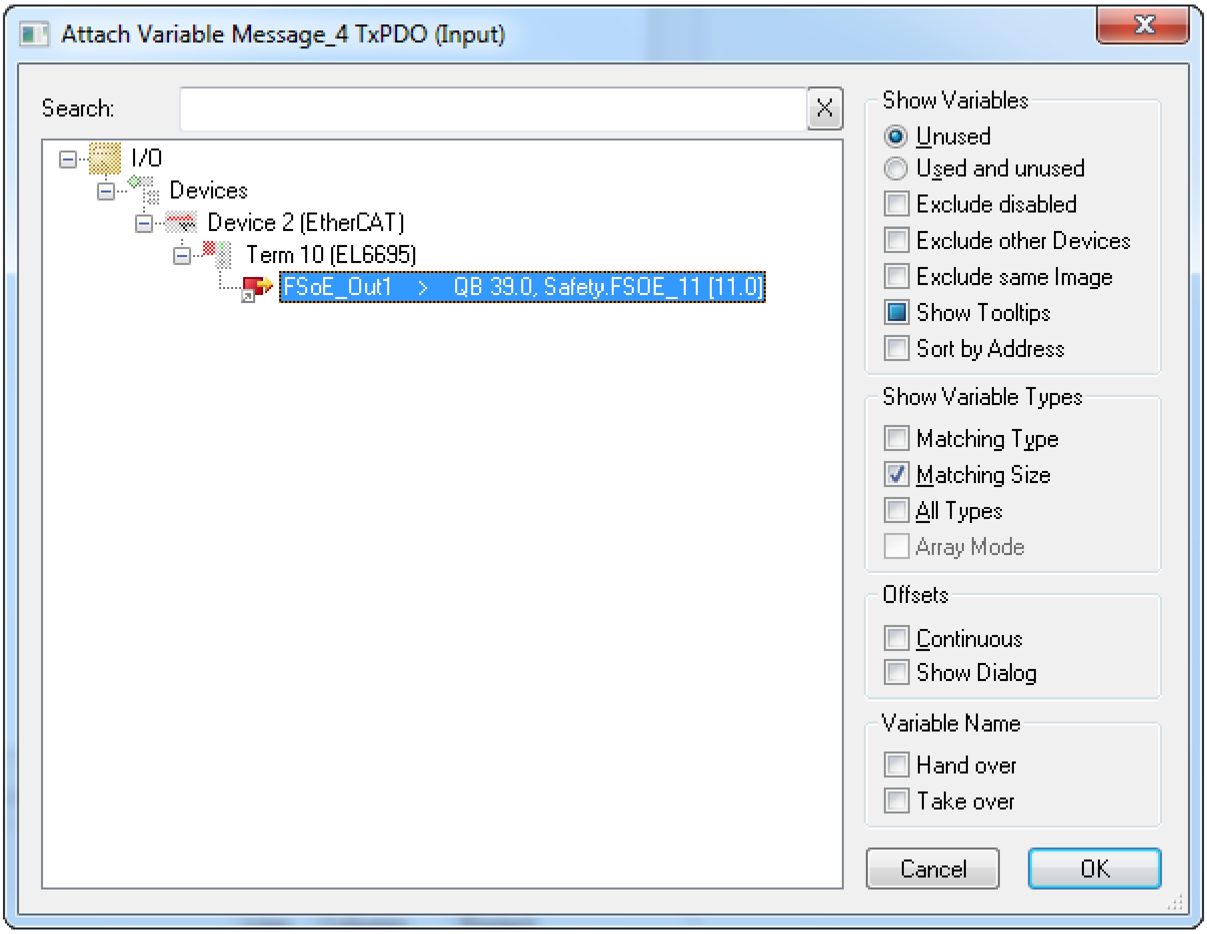
Dip Switch:

Full Name (input): K1100)^Term 4 (EL6910)^ConnectionInputs^Message_4 TxPDO

Full Name (output): I100)^Term 4 (EL6910)^ConnectionOutputs^Message_4 RxPDO

Fig. 34: Selecting the variables

This can be a PLC variable, for example, which is then forwarded to the remote device or can be linked directly with the process image of an EtherCAT Terminal (e.g. EL69x0 or EL6695).



Attach Variable Message_4 TxPDO (Input)

Search:

- I/O
  - Devices
    - Device 2 (EtherCAT)
      - Term 10 (EL6695)
        - FSoE_Out1 > QB 39.0, Safety.FSOE_11 [11.0]

Show Variables
- ● Unused
- ○ Used and unused
- □ Exclude disabled
- □ Exclude other Devices
- □ Exclude same Image
- ■ Show Tooltips
- □ Sort by Address

Show Variable Types
- □ Matching Type
- ☑ Matching Size
- □ All Types
- □ Array Mode

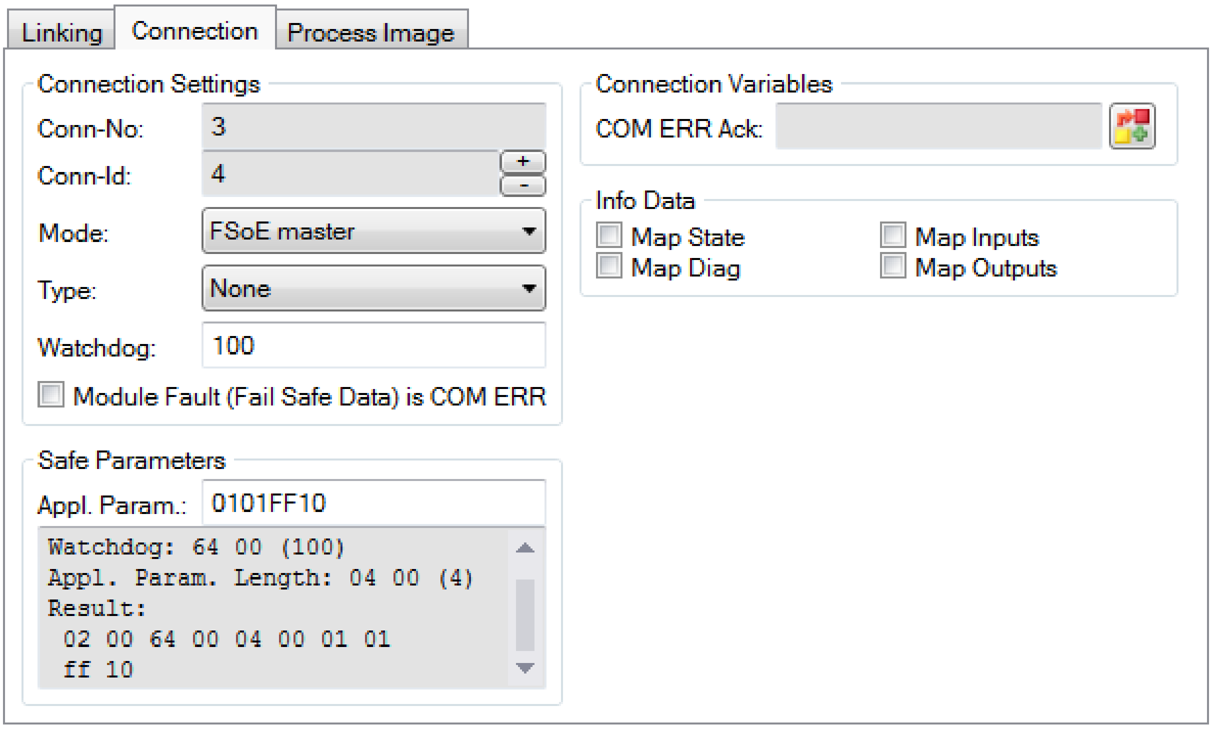
Offsets
- □ Continuous
- □ Show Dialog

Variable Name
- □ Hand over
- □ Take over

Fig. 35: Direct linking with the process image of an EtherCAT Terminal

Further information can be found in the TwinCAT documentation for the variable selection dialog.

The *Connection* tab is used to set the connection-specific parameters.



Connection Settings

Conn-No:	3
Conn-Id:	4
Mode:	FSoE master
Type:	None
Watchdog:	100

□ Module Fault (Fail Safe Data) is COM ERR

Safe Parameters

Appl. Param.: 0101FF10

```
Watchdog: 64 00 (100)
Appl. Param. Length: 04 00 (4)
Result:
 02 00 64 00 04 00 01 01
 ff 10
```

Connection Variables

COM ERR Ack:

Info Data
- □ Map State
- □ Map Diag
- □ Map Inputs
- □ Map Outputs

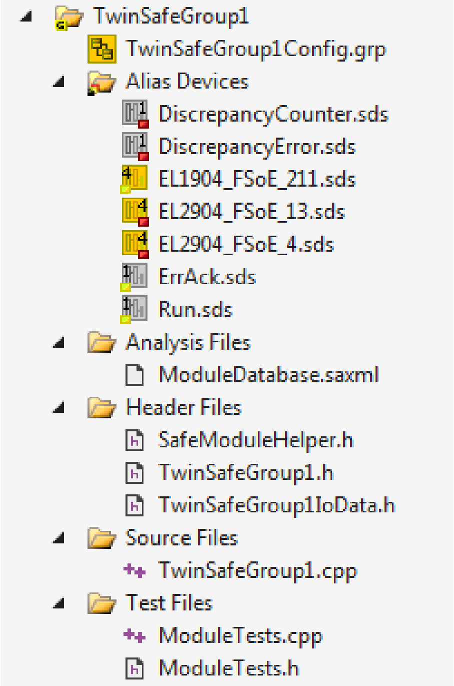
Fig. 36: Connection-specific parameters

Detailed information about the individual settings can be found in the following table.

Parameter	**Description**	**User interaction required**
Conn-No.	Connection number: automatically assigned by the TwinCAT system	No
Conn-ID	Connection ID: pre-allocated by the system; can be changed by the user. A Conn ID must be unique within a configuration. Duplicate connection IDs result in an error message	Control
Mode	FSoE master: TwinCAT Safety PLC is FSoE master for this device. FSoE slave: TwinCAT Safety PLC is FSoE slave for this device. (This option is not supported in the first version of the TwinCAT Safety PLC).	Control
Type	None: Setting for third-party equipment, for which no ESI file is available. KL6904: Setting for KL6904 (safety parameter inactive) EL69XX: Setting for EL6900/EL6930/EL6910/EJ6910 (safety parameter inactive)	Yes
Watchdog	Watchdog time for this connection: A ComError is generated, if the device fails to return a valid telegram to the TwinCAT Safety PLC within the watchdog time.	Yes
Module Fault is ComError	This checkbox is used to specify the behavior in the event of an error. If the checkbox is ticked and a module error occurs on the alias device, this also leads to a connection error and therefore to disabling of the TwinSAFE group, in which this connection is defined.	Yes
Safe Parameters (Appl. Param)	Device-specific parameters: The parameter length is automatically calculated from the number of characters that is entered. This information will typically be provided by the device manufacturer.	Yes
ComErrAck	If ComErrAck is linked to a variable, the connection must be reset via this signal in the event of a communication error.	Yes
Info Data	The info data to be shown in the process image of the TwinCAT Safety PLC can be defined via these checkboxes. Further information can be found in the documentation for *TwinCAT* function blocks for TwinSAFE logic terminals.	Yes

4.2.2.10 TwinSAFE group - Header files

The subfolder *Header Files* of the TwinSAFE group contains a list of all header files assigned to this group.



- TwinSafeGroup1
  - TwinSafeGroup1Config.grp
  - Alias Devices
    - DiscrepancyCounter.sds
    - DiscrepancyError.sds
    - EL1904_FSoE_211.sds
    - EL2904_FSoE_13.sds
    - EL2904_FSoE_4.sds
    - ErrAck.sds
    - Run.sds
  - Analysis Files
    - ModuleDatabase.saxml
  - Header Files
    - SafeModuleHelper.h
    - TwinSafeGroup1.h
    - TwinSafeGroup1IoData.h
  - Source Files
    - TwinSafeGroup1.cpp
  - Test Files
    - ModuleTests.cpp
    - ModuleTests.h

Fig. 37: TwinSAFE group - Header files

The header files *SafeModuleHelper.h* and *<GroupName>IoData.h* are automatically created by the safety editor. The files are not write-protected, i.e. the user could modify them, although they are recreated during the compile process, which means all modifications would be overwritten.

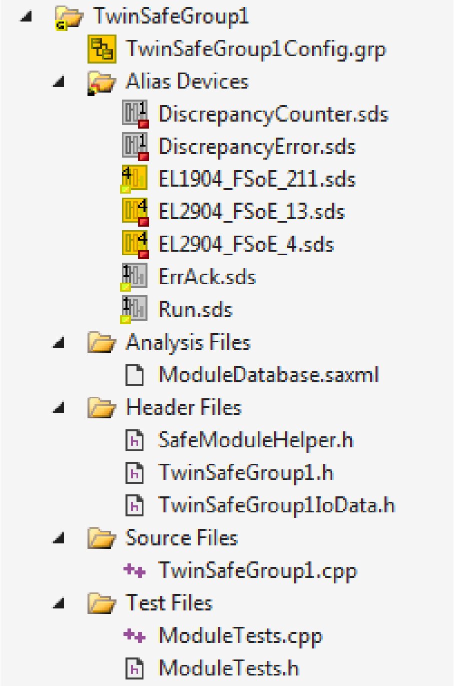
SafeModuleHelper.h contains type definitions, macros and functions created by the safety editor.

<GroupName>IoData.h contains the I/O data structures of the alias devices and the TwinSAFE groups.

The header file *<GroupName>.h* can be used and expanded by the programmer. Here you can create type definitions, variables and functions for the safety application module (<GroupName>.cpp).

4.2.2.11 TwinSAFE group - Source files

The subfolder *Source Files* of the TwinSAFE group contains the C++ source file assigned to this group.



```
TwinSafeGroup1
    TwinSafeGroup1Config.grp
    Alias Devices
        DiscrepancyCounter.sds
        DiscrepancyError.sds
        EL1904_FSoE_211.sds
        EL2904_FSoE_13.sds
        EL2904_FSoE_4.sds
        ErrAck.sds
        Run.sds
    Analysis Files
        ModuleDatabase.saxml
    Header Files
        SafeModuleHelper.h
        TwinSafeGroup1.h
        TwinSafeGroup1IoData.h
    Source Files
        TwinSafeGroup1.cpp
    Test Files
        ModuleTests.cpp
        ModuleTests.h
```

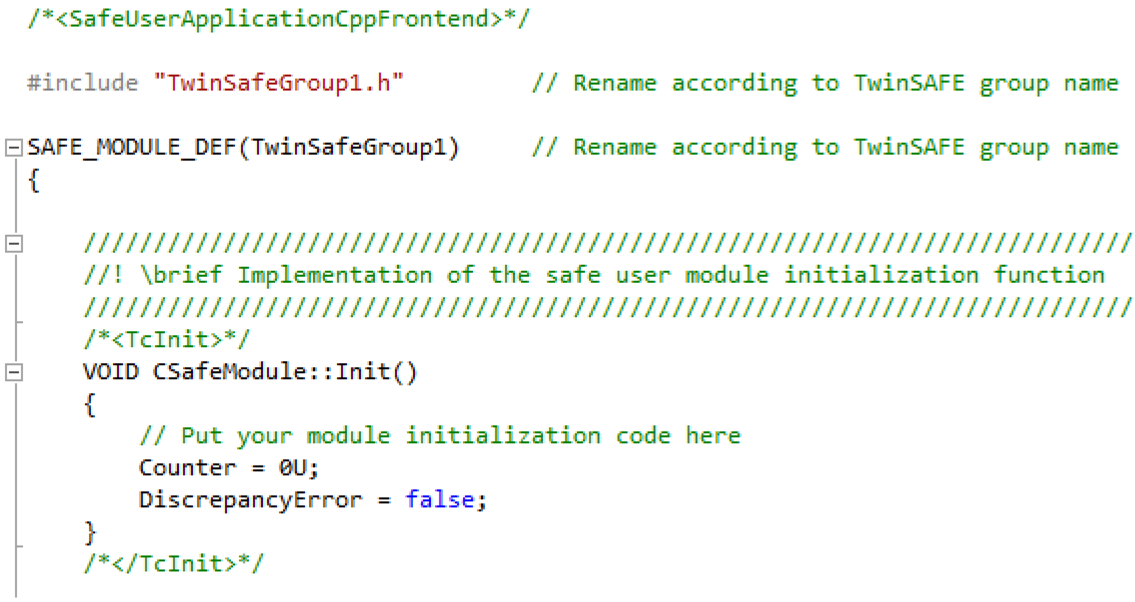
Fig. 38: TwinSAFE group - Source files

The file <GroupName>.cpp is split into four parts. Four module functions are pre-defined and cannot be changed. These include the Init function, which is called when the user application is initialized. The other functions are InputUpdate, CycleUpdate, OutputUpdate, which are used for integrating the user application in the cyclic process. Each of these functions is therefore called in each cyclic process (in the order InputUpdate, CycleUpdate, OutputUpdate). These functions may only be called by the safe runtime environment.

Note

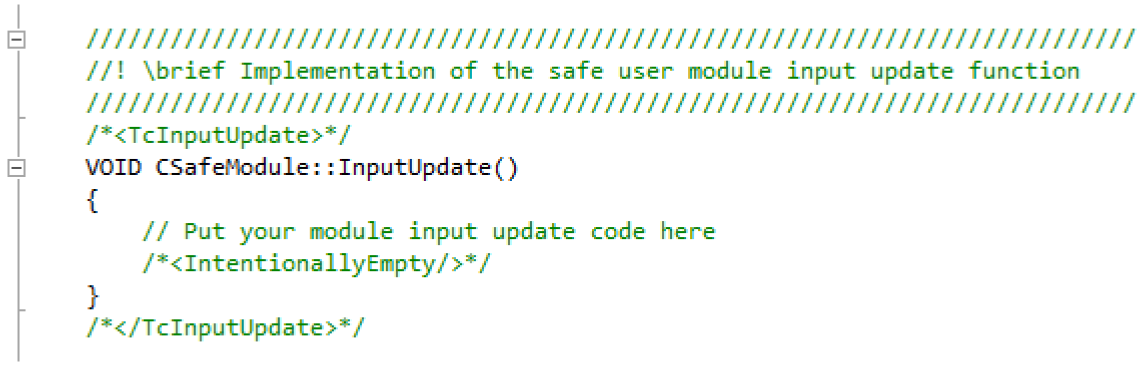
Module variables

All module variables (i.e. variables defined in file <TwinSAFE GroupName>.h) have to be initialized as part of the Init function.



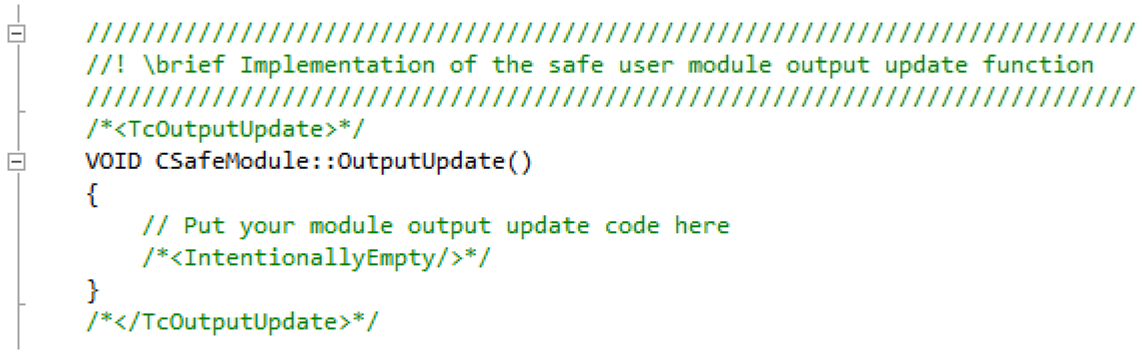
```
/*<SafeUserApplicationCppFrontend>*/

#include "TwinSafeGroup1.h"         // Rename according to TwinSAFE group name

SAFE_MODULE_DEF(TwinSafeGroup1)     // Rename according to TwinSAFE group name
{

    ///////////////////////////////////////////////////////////////////////////
    //! \brief Implementation of the safe user module initialization function
    ///////////////////////////////////////////////////////////////////////////
    /*<TcInit>*/
    VOID CSafeModule::Init()
    {
        // Put your module initialization code here
        Counter = 0U;
        DiscrepancyError = false;
    }
    /*</TcInit>*/
```

Fig. 39: Init function



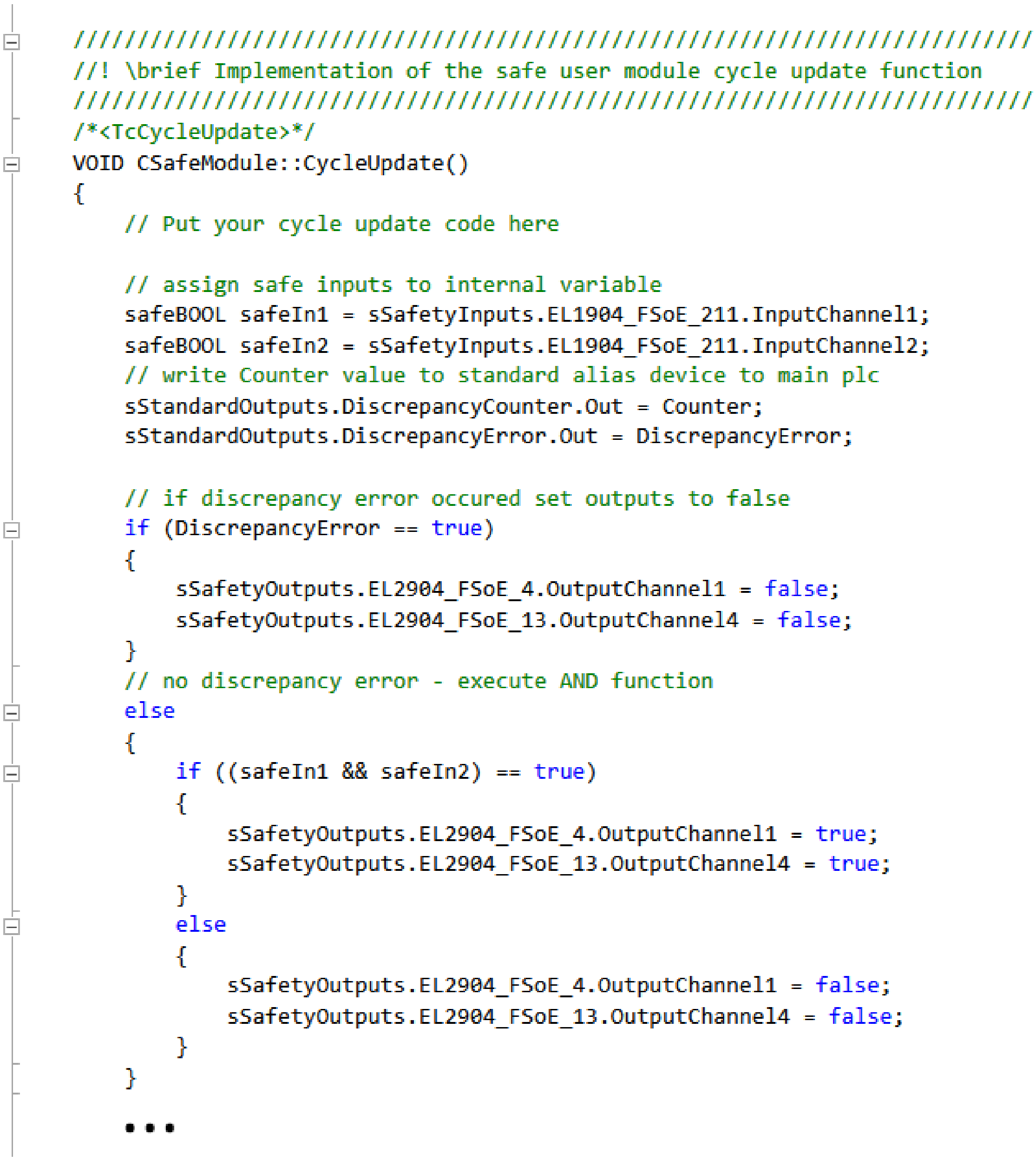
```
///////////////////////////////////////////////////////////////////////////
//! \brief Implementation of the safe user module input update function
///////////////////////////////////////////////////////////////////////////
/*<TcInputUpdate>*/
VOID CSafeModule::InputUpdate()
{
    // Put your module input update code here
    /*<IntentionallyEmpty/>*/
}
/*</TcInputUpdate>*/
```

Fig. 40: InputUpdate function



```
///////////////////////////////////////////////////////////////////////////
//! \brief Implementation of the safe user module output update function
///////////////////////////////////////////////////////////////////////////
/*<TcOutputUpdate>*/
VOID CSafeModule::OutputUpdate()
{
    // Put your module output update code here
    /*<IntentionallyEmpty/>*/
}
/*</TcOutputUpdate>*/
```

Fig. 41: OutputUpdate function



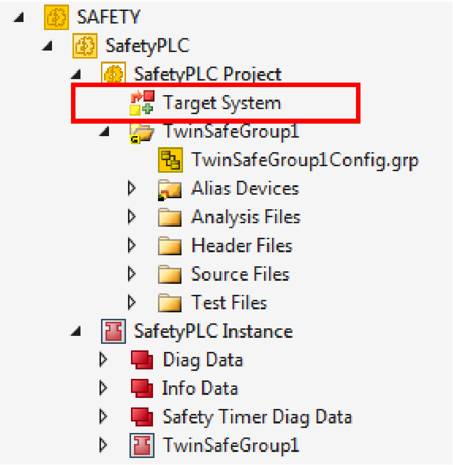
```
////////////////////////////////////////////////////////////////////////////
//! \brief Implementation of the safe user module cycle update function
////////////////////////////////////////////////////////////////////////////
/*<TcCycleUpdate>*/
VOID CSafeModule::CycleUpdate()
{
    // Put your cycle update code here

    // assign safe inputs to internal variable
    safeBOOL safeIn1 = sSafetyInputs.EL1904_FSoE_211.InputChannel1;
    safeBOOL safeIn2 = sSafetyInputs.EL1904_FSoE_211.InputChannel2;
    // write Counter value to standard alias device to main plc
    sStandardOutputs.DiscrepancyCounter.Out = Counter;
    sStandardOutputs.DiscrepancyError.Out = DiscrepancyError;

    // if discrepancy error occured set outputs to false
    if (DiscrepancyError == true)
    {
        sSafetyOutputs.EL2904_FSoE_4.OutputChannel1 = false;
        sSafetyOutputs.EL2904_FSoE_13.OutputChannel4 = false;
    }
    // no discrepancy error - execute AND function
    else
    {
        if ((safeIn1 && safeIn2) == true)
        {
            sSafetyOutputs.EL2904_FSoE_4.OutputChannel1 = true;
            sSafetyOutputs.EL2904_FSoE_13.OutputChannel4 = true;
        }
        else
        {
            sSafetyOutputs.EL2904_FSoE_4.OutputChannel1 = false;
            sSafetyOutputs.EL2904_FSoE_13.OutputChannel4 = false;
        }
    }

    ...
```

Fig. 42: CycleUpdate function

4.2.3 CRC distribution

For automatic startup the TwinCAT Safety PLC has to check whether the current project was enabled for the present system. To this end an internally calculated checksum is distributed to TwinSAFE components, which can be selected by the user, and verified when the TwinCAT Safety PLC starts up. If the comparison fails, the TwinCAT Safety PLC does not start. If the comparison is successful, the safety project is executed on the TwinCAT Safety PLC.

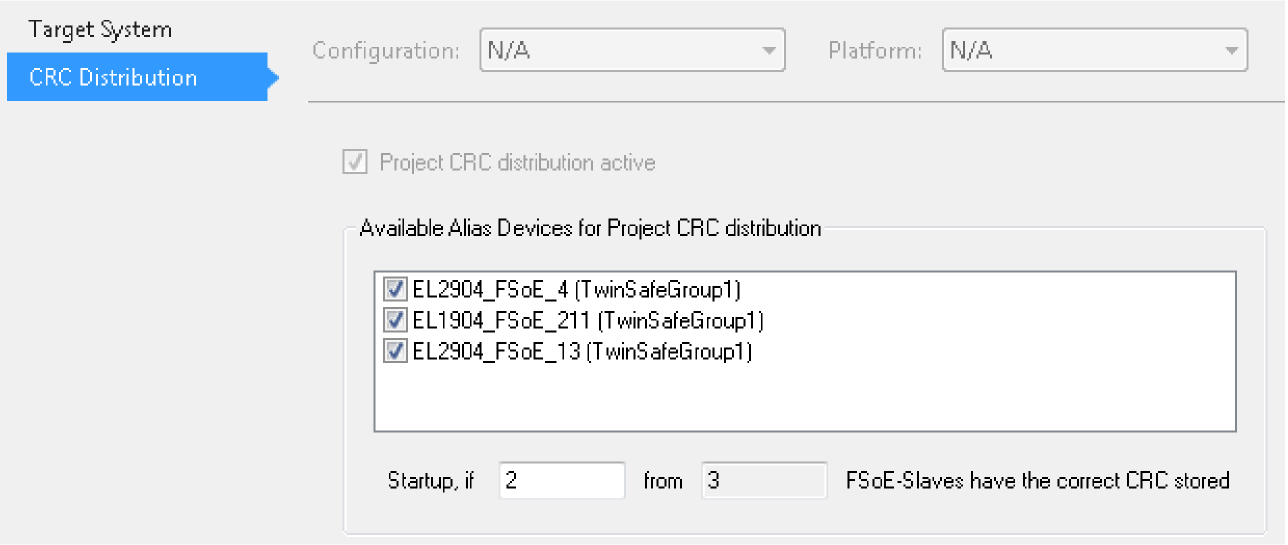
Double-click on Target System to open the Target System dialog. In addition to the target system, the CRC distribution can also be configured here.



- SAFETY
  - SafetyPLC
    - SafetyPLC Project
      - Target System
      - TwinSafeGroup1
        - TwinSafeGroup1Config.grp
        - Alias Devices
        - Analysis Files
        - Header Files
        - Source Files
        - Test Files
    - SafetyPLC Instance
      - Diag Data
      - Info Data
      - Safety Timer Diag Data
      - TwinSafeGroup1

Fig. 43: Target System

The CRC Distribution dialog lists all safe alias devices that can be used for the CRC distribution. The checkbox next to each entry can be used to specify whether the CRC is to be stored on the component. In addition, the user can specify how many of the selected components have to return the correct CRC for the TwinCAT Safety PLC to start. At least one component has to be selected here, in order to enable the safety project to be downloaded and enabled for the TwinCAT Safety PLC.



Configuration: N/A Platform: N/A

☑ Project CRC distribution active

Available Alias Devices for Project CRC distribution

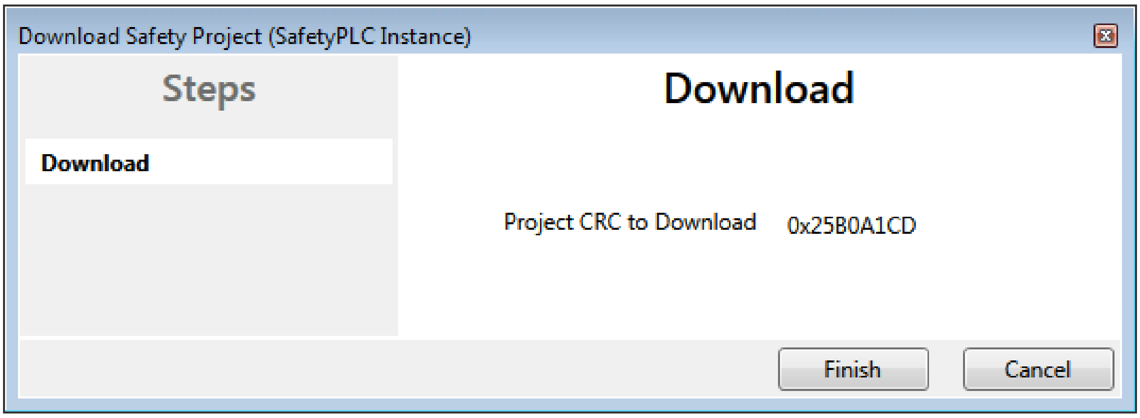
☑ EL2904_FSoE_4 (TwinSafeGroup1)
☑ EL1904_FSoE_211 (TwinSafeGroup1)
☑ EL2904_FSoE_13 (TwinSafeGroup1)

Startup, if 2 from 3 FSoE-Slaves have the correct CRC stored

Fig. 44: Dialog CRC Distribution

4.2.4 Downloading the safety application

The safety configuration is downloaded in two stages: download and activation (unlock). For a download the user first has to establish a connection to the required target system. This is done via the standard mechanisms included in TwinCAT 3 for connecting to a non-local runtime environment (including controlled user authentication). The safety application can then be downloaded to the runtime environment (the runtime environment has to be in Config state). Use the download button in the Safety toolbar, or the menu item Download Safety Project . No user input is required for downloading the safety application. The project CRC of the safety project to be downloaded is displayed in the Download dialog. Use *Finish* to confirm the project CRC and start the actual download.



Download Safety Project (SafetyPLC Instance)

Project CRC to Download 0x25B0A1CD

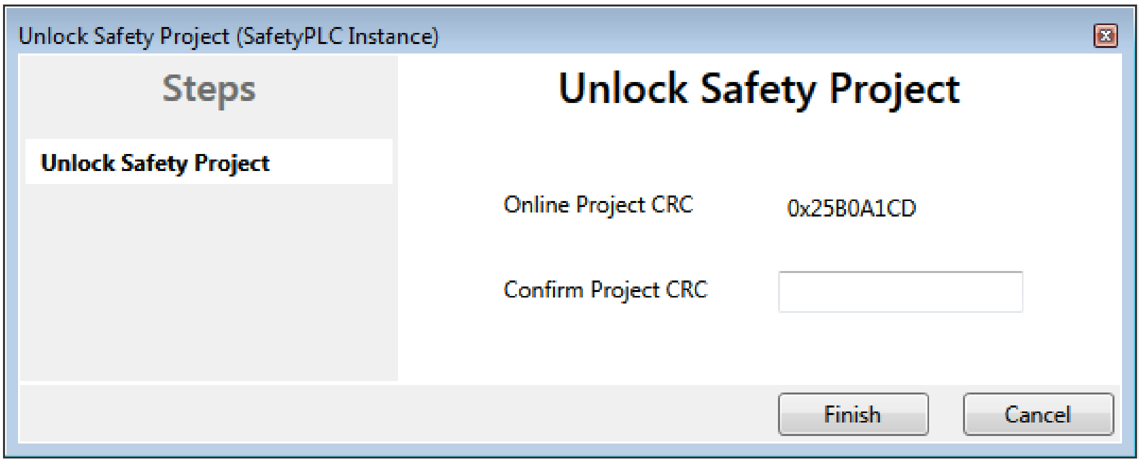
Fig. 45: Downloading the safety application

⚠ CAUTION	**Use only qualified tools** A qualified tool (TwinCAT 3.1) must be used for downloading and activating the project on the TwinCAT Safety PLC.
⚠ DANGER	**Source text of the safety application** The user source text must be developed based on the applicable standards, in particular IEC 61508:2010. See also chapter Verification and validation [▶ 81].
⚠ CAUTION	**Identification of the target system** Before downloading and activating the safety project, the user must ensure that the linked target system is the required target system. An error during this process can only be detected during verification and validation of the safety application.

4.2.5 Activating the safety application

Once the safety application has been downloaded successfully, it has to be activated before it can be executed.

To this end the current configuration has to be activated first, and TwinCAT has to be started in run mode. If the safety project is not activated, a corresponding message appears in the TwinCAT 3 output window. Once the configuration is active, the activation can be started via the unlock button 🔧 or via the menu.



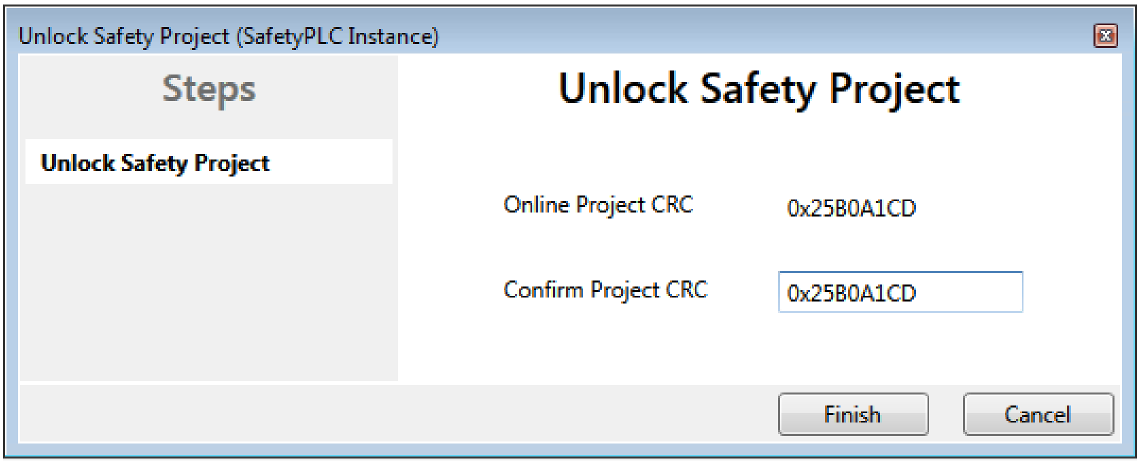
Unlock Safety Project (SafetyPLC Instance)

Online Project CRC 0x25B0A1CD

Confirm Project CRC

Fig. 46: Unlock Safety Project

The user has to confirm the displayed online CRC.



Unlock Safety Project (SafetyPLC Instance)

Steps

Unlock Safety Project

Unlock Safety Project

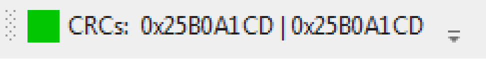
Online Project CRC 0x25B0A1CD

Confirm Project CRC 0x25B0A1CD

Fig. 47: Unlock Safety Project

Confirming the CRC with the *Finish* button enables and starts the safety application. While the safety application starts up, the CRC is distributed to the safe communication devices configured as part of the CRC distribution. When the TwinCAT system is restarted, the safety application starts without having to activate it again.

After the activation the TwinSAFE CRC toolbar shows the same CRC for online and offline.

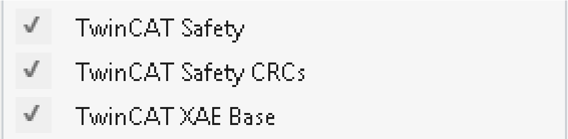


🟩 CRCs: 0x25B0A1CD | 0x25B0A1CD

Fig. 48: Identical CRCs

4.2.6 Safety and CRC toolbars

Right-click in the toolbar area of TwinCAT 3.1 to activate the Safety toolbar and the Safety CRC toolbar.



✓ TwinCAT Safety

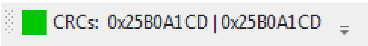
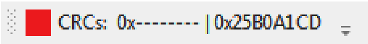
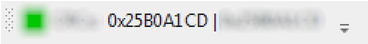
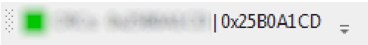
✓ TwinCAT Safety CRCs

✓ TwinCAT XAE Base

Fig. 49: Activating the Safety and CRC toolbars

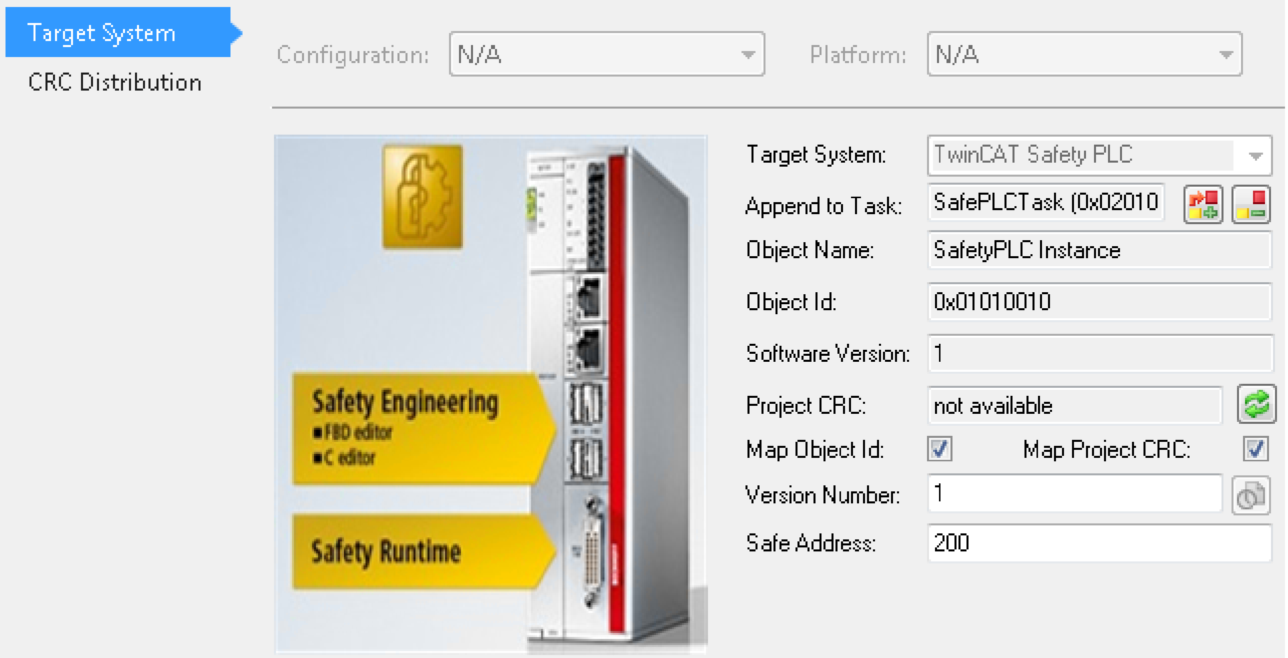
Safety Toolbar	Description
✓	Checking the safety application
✓	Checking the safety application, including hardware level
	Downloading the safety application to the TwinCAT Safety PLC
	Deleting the safety application from the TwinCAT Safety PLC
🔧	Activating the safety application
👓	currently not used

The CRC toolbar shows the online and offline CRC. In addition, an icon indicates whether or not they are identical.

CRC Toolbar	Description
 🟩 CRCs: 0x25B0A1CD | 0x25B0A1CD	Green icon: CRCs are identical
 🟥 CRCs: 0x-------- | 0x25B0A1CD	Red icon: CRCs are different
 🟩 0x25B0A1CD |	Online CRC (32-Bit)
 🟩 | 0x25B0A1CD	Offline CRC (32-Bit)

4.2.7 Info data

Info data for the target system



Target System

CRC Distribution

Configuration: N/A

Platform: N/A

Target System: TwinCAT Safety PLC

Append to Task: SafePLCTask (0x02010

Object Name: SafetyPLC Instance

Object Id: 0x01010010

Software Version: 1

Project CRC: not available

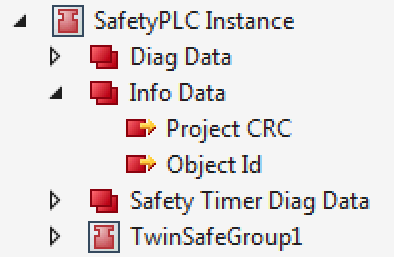
Map Object Id: ☑ Map Project CRC: ☑

Version Number: 1

Safe Address: 200

Fig. 50: Target system - Map Object ID and Map Project CRC

The checkboxes *Map Object ID* and *Map Project CRC* can be used to specify that the object ID and the project CRC should be copied into the process image of the TwinCAT Safety PLC on the target system. From here, the entries *Object Id* and *Project CRC* can be linked to the standard PLC.

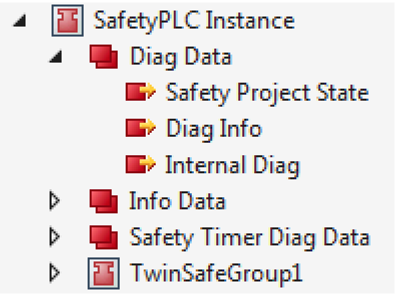


- SafetyPLC Instance
  - Diag Data
  - Info Data
    - Project CRC
    - Object Id
  - Safety Timer Diag Data
  - TwinSafeGroup1

Fig. 51: Target System Info Data

Info Data	Description
Project CRC	32-bit project CRC (online)
Object Id	Unique ID of the TwinCAT Safety PLC instance (unique across the whole TwinCAT 3 project)

Diag Data - TwinCAT Safety PLC



SafetyPLC Instance
Diag Data
Safety Project State
Diag Info
Internal Diag
Info Data
Safety Timer Diag Data
TwinSafeGroup1

Fig. 52: Diag Data - TwinCAT Safety PLC

Diag Data	Description
Safety Project State	• 601 (0x0259) Init The Init state is assumed when the instance starts up, in order to test the internal configuration before the actual startup. • 602 (0x025A) Run The Run state is assumed when no error occurred during startup. In Run state the configured TwinSAFE groups are integrated in the cyclic process. • 603 (0x025B) Error The Error state is assumed if an internal error occurs. This state can only be existed by restarting the whole configuration. In Error state the TwinSAFE groups (and therefore also the subordinate communication links and the subordinate user application) are no longer executed, with the result that the Safe state is assumed. • 604 (0x025C) Checking Download Completion In this state the system checks whether the safety project to be started has been downloaded properly and is valid. An error during this process leads to Error state. • 605 (0x025D) Checking Unlocking Data In this state the system checks whether the safety project to be started has been successfully pre-enabled. If the check is unsuccessful, state 606 is assumed. • 606 (0x025E) Waiting for Activation This state is assumed if the safety project was not enabled in advance. The state can only be exit by triggering activation of the safety project from the corresponding TwinCAT 3 project. • 607 (0x025F) Writing Unlocking Data Once the safety project has been activated successfully, corresponding data are written to indicate the successful activation for the next startup. • 608 (0x0260) Waiting for Project CRC Acknowledge If a safety project is started that was already activated, the activation data are verified with the data previously queried by the safe communication devices previously configured as part of the CRC distribution.
Diag Info	Diagnostic information for the instance
Internal Diag	Internal diagnostic information (not relevant for users)

Safety Timer Diag Data - TwinCAT Safety PLC

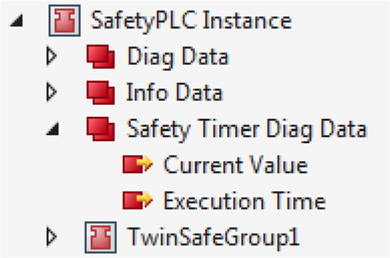
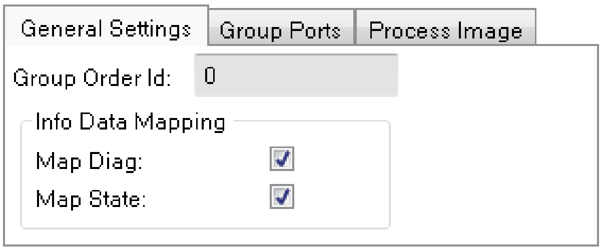


Fig. 53: Safety Timer Diag Data - TwinCAT Safety PLC

Safety Timer Diag Data	Description
Current Value	Current value of the safe time signal
Execution Time	Internally determined execution time from the start of InputUpdate to the end of OutputUpdate for the whole instance

Group info data



Group Order Id: 0

Info Data Mapping

Map Diag: ☑

Map State: ☑

Fig. 54: Group MapDiag MapState

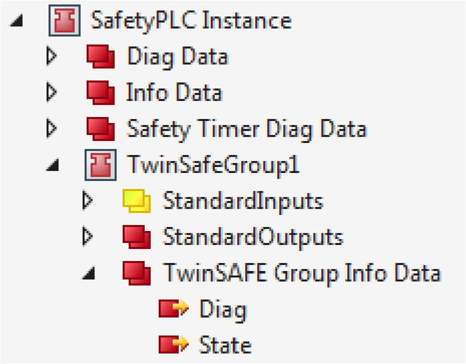
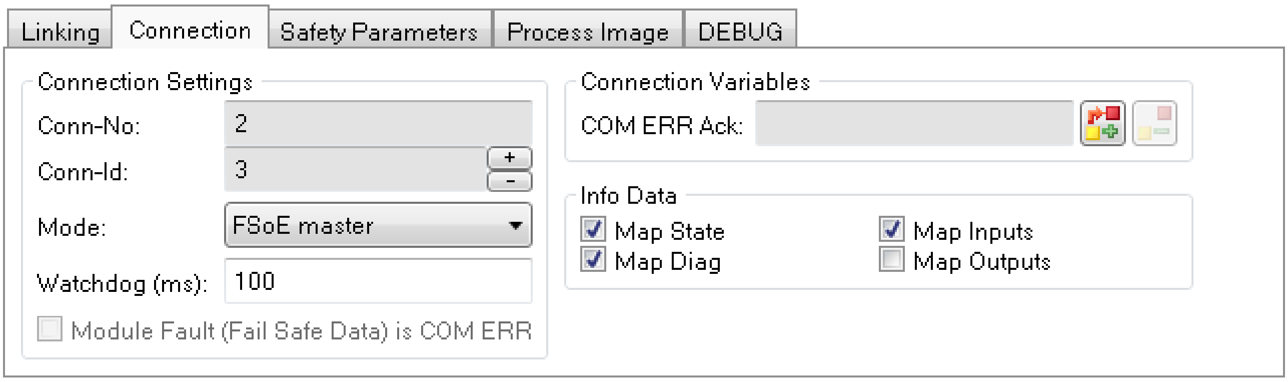


Fig. 55: Group info data

Group Info Data	Description
State	• 701 (0x02BD) Stop The Stop state is assumed while the instance starts up. During operation this state is assumed if the corresponding Run/Stop signal is configured and is not set to TRUE. • 702 (0x02BE) Run The Run state is assumed if none of the safe connections involved is faulty and the Run/Stop input (if configured) is set to TRUE. • 703 (0x02BF) Safe The Safe state is assumed if at least one of the connections is not in Data state. • 704 (0x02C0) Error The Error state is assumed if an application error or a communication error occurs. • 705 (0x02C1) Reset The Reset state is assumed if the ErrorAck signal shows a rising edge in Error state. • 706 (0x02C2) Global Error The Global Error state is assumed if a fatal error occurs during internal processing. This state can only be existed by restarting the current configuration.
Diag	• Diagnostic information

Connection info data



Connection Settings

Conn-No: 2

Conn-Id: 3

Mode: FSoE master

Watchdog (ms): 100

□ Module Fault (Fail Safe Data) is COM ERR

Connection Variables

COM ERR Ack:

Info Data

☑ Map State ☑ Map Inputs

☑ Map Diag □ Map Outputs

Fig. 56: FSoE connection map info data

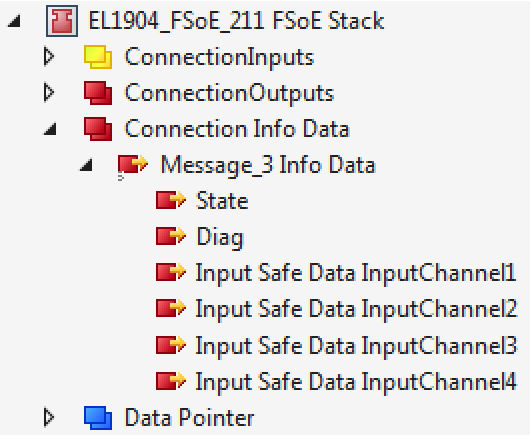

EL1904_FSoE_211 FSoE Stack
- ConnectionInputs
- ConnectionOutputs
- Connection Info Data
  - Message_3 Info Data
    - State
    - Diag
    - Input Safe Data InputChannel1
    - Input Safe Data InputChannel2
    - Input Safe Data InputChannel3
    - Input Safe Data InputChannel4
- Data Pointer

Fig. 57: FSoE connection info data

Connection Info Data	Description
State	• 100 (0x64) Reset The reset state is used to re-initialize the Safety over EtherCAT connection after the power-on or a Safety over EtherCAT communication error. • 101 (0x65) Session During the transition to or in the Session state a session ID is transferred from the Safety over EtherCAT master to the Safety over EtherCAT slave, which responds with its own session ID. • 102 (0x66) Connection In Connection state a connection ID is transferred from the Safety over EtherCAT master to the Safety over EtherCAT slave. • 103 (0x67) Parameter In Parameter state safe communication and device-specific application parameters are transferred. • 104 (0x68) Data In Data state Safety over EtherCAT cycles are transferred until either a communication error occurs or a Safety over EtherCAT node is stopped locally.
Diag	• xxxx 0001 - Invalid command • xxxx 0010 - Unknown command • xxxx 0011 - Invalid connection ID • xxxx 0100 - Invalid CRC • xxxx 0101 - Watchdog expired • xxxx 0110 - Invalid FSoE address • xxxx 0111 - Invalid data • xxxx 1000 - Invalid communication parameter length • xxxx 1001 - Invalid communication parameters • xxxx 1010 - Invalid user parameter length • xxxx 1011 - Invalid user parameters • xxxx 1100 - FSoE master reset • xxxx 1101 - Module error detected on slave, with option "Module error is ComError" activated • xxxx 1110 - Module error detected on EL290x, with option "Error acknowledge active" activated • xxxx 1111 - Slave not yet started, or unexpected error argument • xxx1 xxxx - FSoE slave error detected • xx1x xxxx - FSoE slave reports Failsafe Value active • x1xx xxxx - StartUp • 1xxx xxxx - FSoE master reports Failsafe Value active
Inputs	safe inputs of the connection
Outputs	safe outputs of the connection

4.2.8 Task settings

Right-click on *Tasks* and select *Add New Item…* to create a new task.

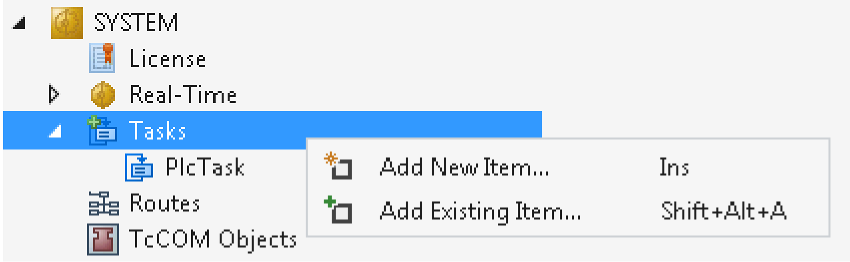
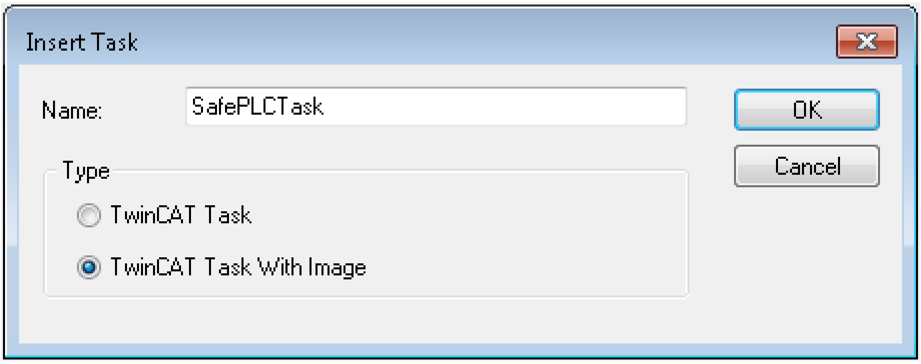


Fig. 58: Adding a new task

The Insert Task dialog is used to enter a task name and to specify whether the task is to be created with or without image. Both options are available for the TwinCAT Safety PLC; in the example With Image is selected.



Name: SafePLCTask

○ TwinCAT Task

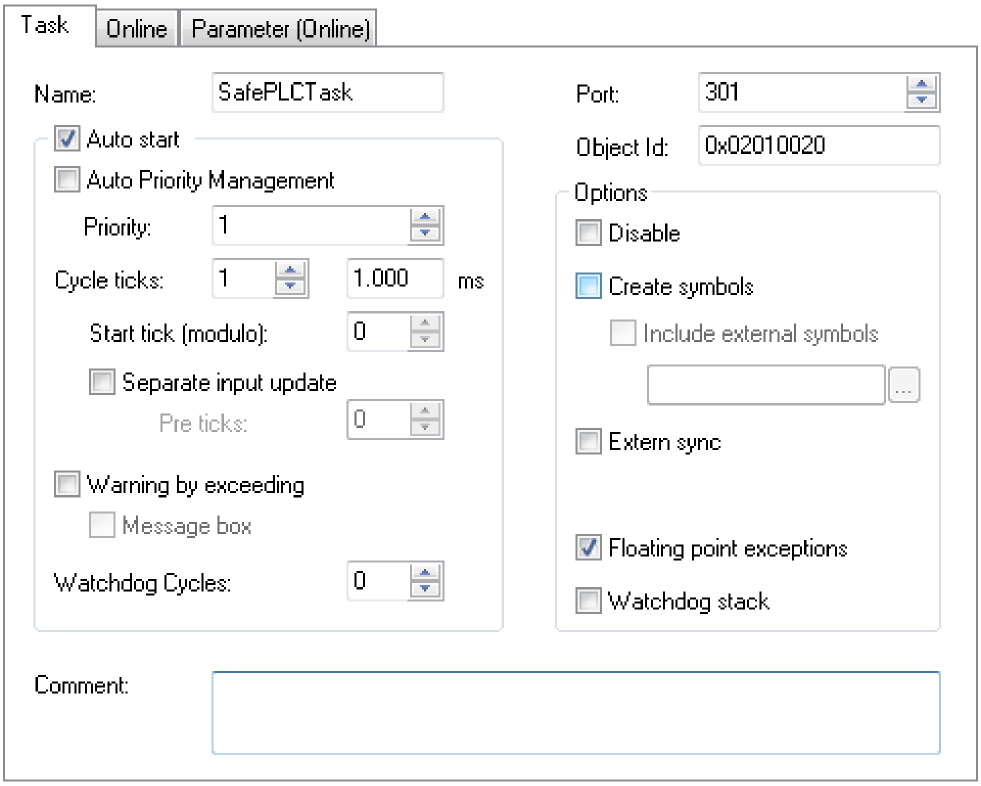
● TwinCAT Task With Image

Fig. 59: Dialog Insert Task

Settings

Double-click on the task to open the task settings. Here you can set the cycle time and the priority.

Note	**Cycle time and priority** The cycle time can be selected freely but should be selected such that no limits are exceeded (ExceedCounter does not increment). The priority should be set as high as possible (low number), in order to minimize disruptions and jitter of the TwinCAT Safety PLC.



Name: SafePLCTask

☑ Auto start

□ Auto Priority Management

Priority: 1

Cycle ticks: 1 1.000 ms

Start tick (modulo): 0

□ Separate input update

Pre ticks: 0

□ Warning by exceeding

□ Message box

Watchdog Cycles: 0

Port: 301

Object Id: 0x02010020

Options

□ Disable

□ Create symbols

□ Include external symbols

□ Extern sync

☑ Floating point exceptions

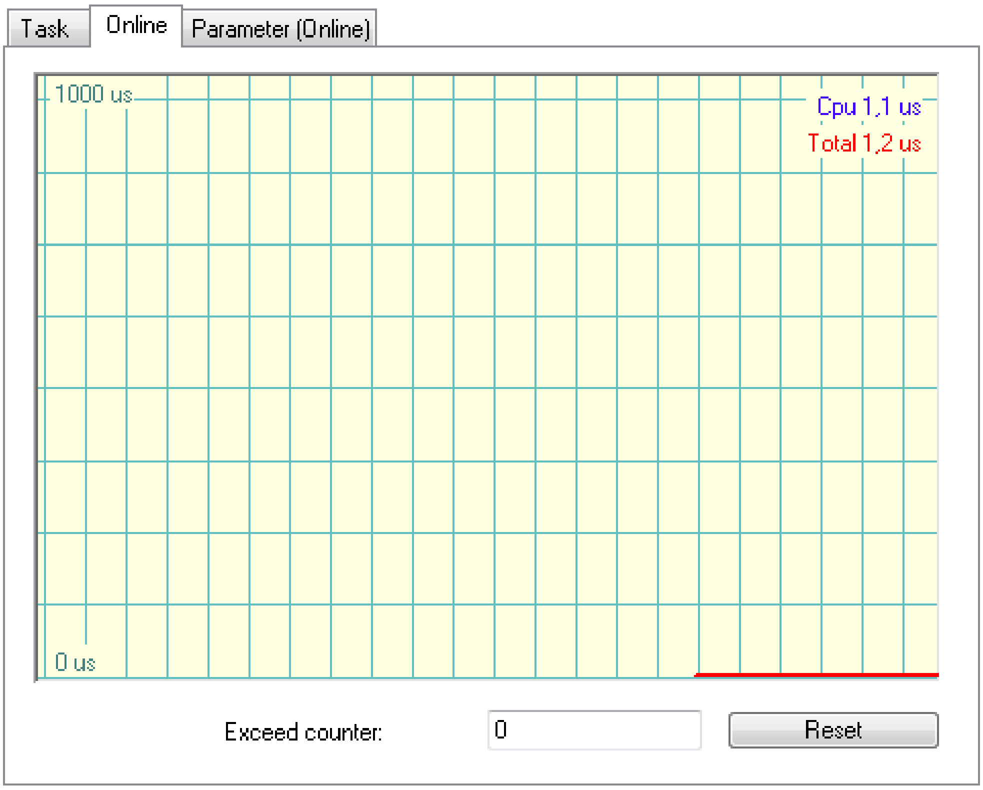
□ Watchdog stack

Comment:

Fig. 60: Task settings

Checking the ExceedCounter

The Online tab for the respective task can be used to check the execution time and the exceed counter.



Exceed counter: 0

Fig. 61: Task execution time and exceed counter

5 Safety C application development

DANGER

Safety C application development

The user source text must be developed based on the applicable standards, in particular IEC 61508:2010.

DANGER

Warnings relating to the advanced configuration process for the safety-related application

The user must evaluate all warnings that may occur during the build process and rectify the causes or comment/document them as appropriate.

5.1 Programming in Safety C

5.1.1 Differentiation between programming in Safety C and C/C++

Safety C is a C++-based high-level language for programming safety applications for the TwinCAT Safety PLC with a safe language scope, which complies with the strongly typed and modular version of the C language, without dynamic memory or pointer (arithmetic). The syntax and semantics of Safety C therefore generally match the corresponding, valid C++ language subset (see ISO standard C++11 N3242, for example), as processed by the corresponding C++ compiler. The Safety C compile process for the TwinCAT Safety PLC requires the Microsoft Visual C++ 2015 compiler or higher.

C++ is essentially an upward compatible extension of C with language support for object orientation and meta-programming, so that C programs can also be processed by C++ compiler (with some restrictions). Programming in Safety C therefore also permits limited utilization of object-oriented C++ extensions for data encapsulation and modularization of program modules by the application developer, although typical C++ concepts such as inheritance, polymorphism and generic template programming are basically not used. From an application developer perspective, programming in Safety C is therefore closer to the development procedure in C.

Compared with function block-based programming, high-level languages such as C/C++ enable programmers much more freedom, although this also creates potential sources of systematic application errors. In addition, the ISO standards for C/C++ deliberately leave scope for implementing efficient compilers, so that a standard-compliant C/C++ program may contain undefined or platform-dependent behavior (e.g. data type widths, division by zero or overflow/underflow of signed integer data types).

The safety standards to be applied for PLC systems therefore require the full scope of C/C++ to be restricted for the development of safety functions in high-level languages (see IEC 61508-3:2010, for example) through the application of language subsets with coding rules, in order to avoid programs with ambiguous semantics and reduce the risk of generating faulty program code or programs with unexpected behavior. Furthermore, this is intended to facilitate tool-based program analysis and verification, as well as manual analyses through code inspections.

The permissible language scope of Safety C largely prevents generation of source code with undefined behavior. At some points alternative helper functions with unambiguous semantics or integrated detection of undefined behavior are offered, so that the application developer has the option to choose between native operations and helper functions.

The main restrictions of Safety C compared with C and C++ can be summarized as follows:

- No support (with a few exceptions) for object orientation, meta-programming or other typical C++ extensions (compared with C)
- Limited data types and strong typing of all data types to avoid implicit type conversion effects
- Limitation to simple statements, operators and expressions to avoid unexpected results and side effects
- Limitation of control flow statements (*if-else*, *for*, *while*, and *switch-case*) for understandable and program sequences that can be analyzed

- No direct or indirect recursion
- No dynamic data structures, no pointer- or address-based data access or function calls
- No global or static functions and variables
- Specification of structured source code templates as a basis for the development of safe program modules (application, function block)
- No user-defined preprocessor or compiler statements for conditional source code compilation or implementation of further source code files
- No implementation of standard libraries or legacy code

5.1.2 Source code templates

When a TwinSAFE group is created, header and source code files for the safer user program are created at the same time. Of particular relevance for programmers are the files <GroupName>.h and <GroupName>.cpp. Furthermore, the user is offered the option to define custom function blocks (not yet supported in V1).

The templates are designed such that programmers can implement modifications and extensions only within predefined ranges. It is not permissible to create further *.h / *.cpp files within the TwinSAFE group.

Preprocessor defines, type definitions (structs, enums) and module variables can be created in the module header file (<group name>.h) (enums are not yet supported in V1). Only declaration of the module class with user-defined module variables and functions is permitted, no implementation.

The module class is implemented in the file <GroupName>.cpp.

Once a group has been created, changes to <group name>.h and <group name>.cpp can only be made by the user. Any changes are immediately secured through checksums (and are thus indirectly indicated to the user through a change in the project CRC). This means that, if the group name is renamed, the source code of <group name>.h and <group name>.cpp must be adapted manually. This applies to the name-specific source code components that are generated at the time of creation (see comments in the source code template).

Additional header files are dynamically created by the safety editor and cannot be modified by the user. Any changes the user may have made in these files are overwritten during the compile process!

5.1.2.1 Application module for a TwinSAFE group

Source code template for declaring a Safety C application module <TwinSAFE GroupName>.h

```
///////////////////////////////////////////////////////////////////////////////
//! \file TwinSafeGroup1.h
//! \brief Header file of the TwinSafeGroup1 application module
//! \ingroup TwinSafeGroup1
//! \defgroup TwinSafeGroup1
//! \brief Put brief description of your application module here
//! \authors Administrator
//! \copyright Put affiliation and copyright notice here
//! \version V1.0
//! \date 2016-09-29
//! \ingroup Empty
///////////////////////////////////////////////////////////////////////////////

//\internal////////////////////////////////////////////////////////////////////
//! XML tags <...> enclosed by C style block comment markups are protected for
//! structural and semantic code analysis. Do NOT remove or reorder any of the
//! mandatory markups within the source code template as safe build process may
//! fail otherwise! For further information on how to write compliant Safety C
//! user code please refer to the provided Safety C coding guidelines document!
///////////////////////////////////////////////////////////////////////////////

/*<SafeUserApplicationHFrontend>*/
#pragma once

/*<UserDefinedIncludes>*/          // Include other safe module headers here
```

```
/*</UserDefinedIncludes>*/

#include "TwinSafeGroup1IoData.h"  // Rename according to TwinSAFE group name

/*<UserDefinedDefines>*/           // Define preprocessor constants here
/*</UserDefinedDefines>*/

NAMESPACE(TwinSafeGroup1)          // Rename according to TwinSAFE group name
{

     /*<UserDefinedTypes>*/        // Define custom data types here
     /*</UserDefinedTypes>*/

///////////////////////////////////////////////////////////////////////////
//! \class TwinSafeGroup1
//! \brief Declaration of the Safety C user application module class
//! \details Put detailed description of your module functionality here
///////////////////////////////////////////////////////////////////////////
     SAFE_MODULE(TwinSafeGroup1)    // Rename according to TwinSAFE group name
{

     // Public module interface
     PUBLIC:
          VOID Init();                      //!< Module initialization function
          VOID InputUpdate();               //!< Module input update function
          VOID OutputUpdate();              //!< Module output update function
          VOID CycleUpdate();               //!< Module cycle update function

          SafetyInputs sSafetyInputs;       //!< Safe input data struct
          SafetyOutputs sSafetyOutputs;     //!< Safe output data struct
          StandardInputs sStandardInputs;   //!< Non-safe input data struct
          StandardOutputs sStandardOutputs; //!< Non-safe output data struct

          safeUINT16 u16SafeTimer           //!< Safe external timer input (in ms)

          TSGData sTSGData;                 //!< TwinSAFE group exchange data struct

     // Module internals
     PRIVATE:

          /*<UserDefinedVariables>*/        // Define internal variables here
          /*</UserDefinedVariables>*/

          /*<UserDefinedFunctions>*/        // Define internal functions here
          /*</UserDefinedFunctions>*/

          SAFE_MODULE_EXPORT();
     };

     //! Reference to project FCS symbol
     extern UINT32 SAFETY_PROJECT_FCS;      // Do NOT read, write or remove!

};
/*</SafeUserApplicationHFrontend>*/
```

Note

Permissible modifications <TwinSAFE GroupName>.h

- NAMESPACE <TwinSAFE GroupName> (if the TwinSAFE group name changes in the project tree, the user must adjust this entry accordingly)
- SAFE_MODULE <TwinSAFE GroupName> (if the TwinSAFE group name changes in the project tree, the user must adjust this entry accordingly)
- User includes only between
  /*<UserDefinedInclude>*/ ... /*</UserDefinedInclude>*/
- User defines only between
  /*<UserDefinedDefines>*/ ... /*</UserDefinedDefines>*/
- User type definitions only between
  /*<UserDefinedTypes>*/ ... /*</UserDefinedTypes>*/
- User variables only between
  /*<UserDefinedVariables>*/ ... /*</UserDefinedVariables>*/
- User functions only between
  /*<UserDefinedFunctions>*/ ... /*</UserDefinedFunctions>*/
- Comments can amended/modified as required
  (except protected comments of the form /*<…>*/)

Source code template for implementing a Safety C application module: <TwinSAFE GroupName>.cpp

```
///////////////////////////////////////////////////////////////////////////////
//! \file TwinSafeGroup1.cpp
//! \brief Source file of the TwinSafeGroup1 application module
//! \ingroup TwinSafeGroup1
//! \authors Administrator
//! \copyright Put affiliation and copyright notice here
//! \version V1.0
//! \date 2016-09-29
//! \details Put detailed description of your module implementation here
///////////////////////////////////////////////////////////////////////////////

///\internal//////////////////////////////////////////////////////////////////
//! XML tags <...> enclosed by C style block comment markups are protected for
//! structural and semantic code analysis. Do NOT remove or reorder any of the
//! mandatory markups within the source code template as safe build process may
//! fail otherwise! For further information on how to write compliant Safety C
//! user code please refer to the provided Safety C coding guidelines document!
///////////////////////////////////////////////////////////////////////////////

/*<SafeUserApplicationCppFrontend>*/

#include "TwinSafeGroup1.h"       // Rename according to TwinSAFE group name

SAFE_MODULE_DEF(TwinSafeGroup1)   // Rename according to TwinSAFE group name
{
     /////////////////////////////////////////////////////////////////////////////
     //! \brief Implementation of the safe user module initialization function
     /////////////////////////////////////////////////////////////////////////////
     /*<TcInit>*/
     VOID CSafeModule::Init()
     {
          // Put your module initialization code here
     }
     /*</TcInit>*/

     /////////////////////////////////////////////////////////////////////////////
     //! \brief Implementation of the safe user module input update function
     /////////////////////////////////////////////////////////////////////////////
     /*<TcInputUpdate>*/
     VOID CSafeModule::InputUpdate()
     {
          // Put your module input update code here
     }
     /*</TcInputUpdate>*/

     /////////////////////////////////////////////////////////////////////////////
     //! \brief Implementation of the safe user module output update function
     /////////////////////////////////////////////////////////////////////////////
     /*<TcOutputUpdate>*/
     VOID CSafeModule::OutputUpdate()
     {
          // Put your module output update code here
     }
     /*</TcOutputUpdate>*/

     /////////////////////////////////////////////////////////////////////////////
     //! \brief Implementation of the safe user module cycle update function
     /////////////////////////////////////////////////////////////////////////////
     /*<TcCycleUpdate>*/
     VOID CSafeModule::CycleUpdate()
     {
          // Put your cycle update code here
     }
     /*</TcCycleUpdate>*/

     /*<UserDefinedFunctionsDef>*/        // Implement internal module functions here
     /*</UserDefinedFunctionsDef>*/

     //! Reference to project FCS symbol
     extern UINT32 SAFETY_PROJECT_FCS;    // Do NOT read, write or remove!
};

// Rename according to TwinSAFE group name
SAFE_MODULE_DEF_EXPORT(TwinSafeGroup1);

/*</SafeUserApplicationCppFrontend>*/
```

Note

Valid modifications <TwinSAFE GroupName>.cpp

- SAFE_MODULE_DEF <TwinSAFE GroupName> (if the TwinSAFE group name changes in the project tree, the user must adjust this entry accordingly)
- SAFE_MODULE_DEF_EXPORT <TwinSAFE GroupName> (if the TwinSAFE group name changes in the project tree, the user must adjust this entry accordingly)
- User program initialization only between /*<TcInit>/ … /*</TcInit>*/
- User program input update only between /*<TcInputUpdate>/ … /*</TcInputUpdate>*/
- User program cycle update only between /*<TcCycleUpdate>/ … /*</TcCycleUpdate>*/
- User program output update only between /*<TcOutputUpdate>/ … /*</TcOutputUpdate>*/
- User functions only in the range between /*<UserDefinedFunctions>*/ … /*<UserDefinedFunctions>*/
- In order to avoid warnings, for unused or empty code blocks a comment should be added of the form:
  /*<IntentionallyEmpty/>*/

5.2 Safe coding rules

5.2.1 Definitions

Term	Explanation
strict/strong typing	Implicit type conversions are not permitted in Safety C. For each operation, assignment or parameter passing, the data type of all operands must match the target data type (with few exceptions that are regarded as safe). Any discrepancy between the data types must be resolved through an explicit conversion. (see chapter Strong typing [▶ 58])
pure functions	The function always provides the same result, when the same arguments are transferred to the function. The result of the function does not depend on internal locals or globals and variables, other internal information or input signals. A pure function is a function without side effect. All helper functions provided are pure functions.
non-pure functions / impure functions	Each function that uses state variables or input/output signals or other internal information should be regarded as potentially "impure" or "non-pure", since it may return different results at different call times. A non-pure / impure function is a function with side effect. User-defined functions are generally regarded as impure, even if they meet the criteria of a pure function.
Operator precedence	Determines the order to be used for the operators of a programming language, whereby a composite expression can be transferred to a syntax tree for further processing by the compiler. In Safety C, the operator precedences are clearly defined through the derivation from the C/C++ standard.
Operator associativity	Determines the order of operators of the same precedence in a composite expression, unless explicitly given by parentheses. In Safety C, the operator associativity is clearly defined through the derivation from the C/C++ standard. The expression (a + b + c) is therefore unambiguously processed through the left associativity of the addition with the following implicit parentheses: ((a + b) + c) This effect is particularly important in cases where the order of the operators to be applied can affect on the overall result. In Safety C the operator associativity must also be taken into account for compliance with the strong typing.
Evaluation sequence	Determines the order to be used when evaluating operands, such as performing function calls or incrementing a variable as part of a composite expression. The evaluation sequence can affect the result of an expression if it contains several mutually dependent side effects. The C/C++ standard does not define a clear evaluation sequence! For the sake of clarity of Safety C Code, restrictions are therefore imposed on expressions, in order to avoid the effects of an ambiguous evaluation sequence.
Recursion	A recursion means that a function or procedure calls itself. A distinction is made between direct and indirect recursion: direct recursion: A() calls A() indirect recursion: A() calls B(), B() calls A(), ...
Short circuit evaluation	Short circuit evaluation or conditional evaluation refers to a behavior, which for Boolean operations results in premature abortion of the evaluation. If a result is unambiguously determined after part of an operation, the following sub-operations are no longer executed. Example: c = a&&b; If a is false, the result c is unambiguously determinable; the operation is therefore aborted, and c is directly set to false, without evaluation of b.

5.2.2 General

Note	**Data types** In the development of the safety application, the user should ensure that a distinction is made between the data types BOOL and safeBOOL and other safe and non-safe data types. This simplifies the subsequent verification and validation of the realized application. In V1 safe and non-safe data types are identical from the perspective of the type system. The user therefore has to assess intermixing of safe and non-safe input signals based on current standards.

Note	**Control structures** For loops are preferable to while loops, since the termination is easier to realize.

The following general rules apply for context-dependent restriction of operators and function calls in expressions:

- The logic operators &&, || may only be used in conditional expressions of if, for and while statements (and in the conditional expressions of assert statements).
- The conditional expressions of if, for and while statements must return the result of a comparison operation (==, !=, <, >, <=, >=). In addition, complex composite operands of this compare operations should be enclosed in parentheses, e.g.:

```
while(flag==true)        //Statt while(flag)
if (0>(a+b))             //Statt if(0>a+b) oder if(a+b)
```

- Function calls with potential side effects (so-called "non-pure functions") may only be called as simple statements, either without assignment of a return value or (if a return value is used) with direct assignment of the return value to a variable, in order to ensure that the order of evaluation is clearly determined (i.e. they must not be called as part of a condition or switch expression). Permitted (example):
  ```
    INT32 r1 = MyNonPureFunc1();
    INT32 r2 = MyNonPureFunc2();
    INT32 r = r1 + r2;
  ```
  Not permitted (example):
  ```
    INT32 r = MyNonPureFunc1() + MyNonPureFunc2();
  ```
- Functions without side effects ("pure functions") may also be used as part of switch or conditional expressions. Examples of side effect-free functions include helper functions provided by the TwinCAT Safety PLC, such as mathematical functions or conversion functions. They can be combined with "non-pure” functions, as long as there is only one call of a "non-pure” function per expression. Permitted (example):
  ```
  INT32 r1 = MyPureFunc1() + MyNonPureFunc1(MyPureFunc2());
  ```
- It is generally assumed that the predefined module interface functions (Init, InputUpdate, CycleUpdate, OutputUpdate), user-defined functions and function blocks have potential side effects. For this reason they may only be called as simple line statements, with or without assignment (see item 3), e.g.:
  ```
  MyFB1.CycleUpdate();
  int y = MyFunction(42, y);
  y = MyFunction2();
  ```
  Calling the interface functions as entry point for the user application is only permitted from the safe runtime environment. User-defined calls of Init() and CycleUpdate() are only allowed for FB instances (FBs are not supported in V1).
- The operators post-increment (++) and post-decrement (--) may only be used as simple line statements (e.g. i++; i--;) (except the third expression of the statement line in a for head of the loop).
- The assignment operator may only be used as part of simple statements with assignment, with the exception of the first and third expression of the statement line of a for head of the loop (e.g. INT32 i=0;). Multiple assignments are not permitted in a line statement (e.g. a=b=c;), neither are assignments as default value initialization in function signatures.
- In general, all primitive data types must be strongly typed on assignment and for parameter passing and function return, in contrast to standard C/C++. Exceptions are combinations of a smaller source datatype with a larger target datatype with the same signedness (see chapter Strong typing [▶ 58]).

- To avoid unintentional effects, further combinations of data types and operators are restricted (see chapter Strong typing [▶ 58]).
- The use of the explicit type conversion operators requires additional parentheses if the expression to the right of the transformation operator is a non-bracketed, complex expression:
  INT32 i32 = (INT32)u16 * -u16; // Not permitted, since it is not clear which expression is meant as the operand of the type conversion operator
  INT32 i32 = (INT32)(u16) * -u16; // Permitted, since unambiguous

Note

Limitation of complexity

Warnings are issued when a certain complexity is exceeded (known as McCabe index or zyclomatic complexity). An index of < 10 per function is recommended.

- index > 20 per module function
- index > 50 per module (error for index > 1000)

Note

Limitation of the number of module variables

A warning is issued if the number of module variables exceeds 50, an error is issued if the number exceeds 1000.

5.2.3 Strong typing

The following applies for all type identifiers mentioned:

- BOOL is equivalent to safeBOOL.
- INT8 is equivalent to safeINT8, SINT, safeSINT.
- UINT8 is equivalent to safeUINT8, USINT, safeUSINT.
- INT16 is equivalent to safeINT16, INT, safeINT.
- UINT16 is equivalent to safeUINT16, UINT, safeUINT.
- INT32 is equivalent to safeINT32, DINT, safeDINT.
- UINT32 is equivalent to safeUINT32, UDINT, safeUDINT

Explanations for the operator type constraints shown in the following tables:

1. Logic operators (&&, ||, !) may only have variables, literals (*true*, *false*) or expressions of type BOOL as operands.
2. Arithmetic operators (+, -, *, /, %) may only have variables, literals (decimal, hexadecimal, binary) or expressions of integer arithmetic types (U)INT(8/16/32) as operands.
3. Bitwise operators (&, |, ^, ~, <<, >>) may only have variables, literals (decimal, hexadecimal, binary) or expressions of integer unsigned arithmetic types UINT(8/16/32) as operands.
4. The comparison operators (<, >, <=, >=) may not have variables, literals (*true*, *false*) or expressions of type BOOL as operands.
5. Binary operators (comparison, arithmetic, bitwise, logic) with left and right operand (+, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=) may only contain variables, literals (Boolean, decimal, hexadecimal, binary) or expressions of the same type (for other restrictions see above).
6. "Integral promotions" (C++ standard conversions to type INT32) of the result expression of operations with operands of small integer data types (U)INT(8/16) have to be neutralized through an explicit type conversion, if one of the rules under 1. - 5. is violated,
   e.g. in the event of summation of UIN8 expressions in the UINT8 number range:
   1: UINT8 z = …; UINT8 a = …;
   2: z = (UINT8)(a + (UINT8)(a + a)); //instead of a = a + a + a;
   Without type conversion of the INT32 result expression from a+a back to INT8, the type equality rule for the subsequent addition would be violated. The same applies to the subsequent assignment to the UINT8 type. Alternatively, all UINT8 expressions can be previously converted to INT32 expressions, before the summation in the INT32 number range is applied:
   3: UINT8 z = …; UINT8 a = …;
   4: z = ((INT32)a) + ((INT32)a) + ((INT32)a);
   Note, however, that this may lead to a different result. The aim of strong typing is to make the intention of the application developer regarding type conversion effects explicitly visible.
   Permitted (example):
   INT16 i16; UINT8 u8; …
   INT32 i32 = i16; // Not type-equivalent, but permitted
   UINT16 u16 = u8; // Not type-equivalent, but permitted
   Further exceptions apply for the initialization of module variables with constant literals. Permitted (example):
   INT8 i8; // Declaration as module variable
   UNT16 u16; // Declaration as module variable
   …
   i8 = 0; // Permitted, despite the fact that it is a constant literal of type INT32
   u16 = 42U; // Permitted, despite the fact that it is a constant literal of type UINT32
7. The explicit type conversion operator () may only be used for conversion of a variable, literal (decimal, hexadecimal, binary) or expression from one integer arithmetic type (U)INT(8/16/32) to another. The explicit type conversion operator may lead to loss of data or sign. If this is not desired or if it should be detected, the helper functions should be used for the conversion.
8. The same source and target types or function signatures must be used for assignment (=), transfer and return of function parameters and results, or from smaller arithmetic data types to larger data types without conversion between signed and unsigned data type (safe exception of strong typing). For initialization statements of type UINT8 a = 10U, the compiler checks whether the literal matches the declared data type. However, this is only permitted if the declaration and initialization are combined in a simple line statement, or if the initialization applies to a simple module variable.

9. Complex data types (structs) only support assignments for the same type (=), transfer as function parameter and return through functions, as well as the access operator (.) for data members. The application of explicit type conversion to complex data types is generally not forbidden.

Only operator type combinations that have an entry in the following tables are allowed. The type identifier in the cell is the result type of the operation (taking into account C++ standard type conversions). The right-hand operand (RHS) is shown in the header, the left-hand operand (LHS) in the first column.

LHS/RHS: left/right operand of unary or binary operators.

Type rules for arithmetic operators

+,-,*,/,%	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
BOOL							
INT8		INT32					
UINT8			INT32				
INT16				INT32			
UINT16					INT32		
INT32						INT32	
UINT32							UINT32

Unary	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
-x		INT32	INT32	INT32	INT32	INT32	
+x							

Unary	x++	x--
BOOL		
INT8	INT8	INT8
UINT8	UINT8	UINT8
INT16	INT16	INT16
UINT16	UINT16	UINT16
INT32	INT32	INT32
UINT32	UINT32	UINT32

+=, -=, *=, /=, %=	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
BOOL							
INT8		INT8					
UINT8			UINT8				
INT16				INT16			
UINT16					UINT16		
INT32						INT32	
UINT32							UINT32

Type rules for bitwise operators

&,|,^,<<,>>	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
BOOL							
INT8							
UINT8			INT32				
INT16							
UINT16					INT32		
INT32							
UINT32							UINT32

Unary	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
~x			INT32		INT32		UINT32

&=, |=, ^=, <<=, >>=	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
BOOL							
INT8							
UINT8			UINT8				
INT16							
UINT16					UINT16		
INT32							
UINT32							UINT32

Type rules for logic operators

&&, ||	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
BOOL	BOOL						
INT8							
UINT8							
INT16							
UINT16							
INT32							
UINT32							

Unary	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
!x	BOOL						

==, !=	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
BOOL	BOOL						
INT8		BOOL					
UINT8			BOOL				
INT16				BOOL			
UINT16					BOOL		
INT32						BOOL	
UINT32							BOOL

>,<,>=,<=	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
BOOL							
INT8		BOOL					
UINT8			BOOL				
INT16				BOOL			
UINT16					BOOL		
INT32						BOOL	
UINT32							BOOL

Type rules for the explicit cast operator

()	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
BOOL							
INT8		INT8	INT8	INT8	INT8	INT8	INT8
UINT8		UINT8	UINT8	UINT8	UINT8	UINT8	UINT8
INT16		INT16	INT16	INT16	INT16	INT16	INT16
UINT16		UINT16	UINT16	UINT16	UINT16	UINT16	UINT16
INT32		INT32	INT32	INT32	INT32	INT32	INT32
UINT32		UINT32	UINT32	UINT32	UINT32	UINT32	UINT32

()	struct A	struct B	struct C
struct A			
struct B			
struct C			

Type rules for the member access operator

.	struct A	struct B	struct C
BOOL	BOOL	BOOL	BOOL
INT8	INT8	INT8	INT8
UINT8	UINT8	UINT8	UINT8
INT16	INT16	INT16	INT16
UINT16	UINT16	UINT16	UINT16
INT32	INT32	INT32	INT32
UINT32	UINT32	UINT32	UINT32
struct A		struct B	struct C
struct B	struct A		struct C
struct C	struct A	struct B	

Type rules for assignments, and transfer of function parameters and return of return values

=	BOOL	INT8	UINT8	INT16	UINT16	INT32	UINT32
BOOL	BOOL						
INT8		INT8					
UINT8			UINT8				
INT16		INT16		INT16			
UINT16			UINT16		UINT16		
INT32		INT32		INT32		INT32	
UINT32			UINT32		UINT32		UINT32

=	struct A	struct B	struct C
struct A	struct A		
struct B		struct B	
struct C			struct C

5.2.3.1 Examples for the strong type system

This chapter provides examples to illustrate the strong type system used for the TwinCAT Safety PLC.

Consideration of the expressions as trees

The C/C++ standard stipulates that complex expressions are processed according to a tree structure based on the implicit operator precedences and operator associativity.

The tree structure is determined by the C/C++ compiler based on the standard and may include implicit type conversions that are not allowed in Safety C. Apart from a few safe exceptions, in Safety C the permitted operator type combinations or function signatures have to be considered exactly (strong typing, see matrix tables)

The implicit operator precedences and the operator associativity can be influenced and the tree structure can be changed through explicit parentheses.

Trees consist of the following components:

- **Leaf nodes**

(these are literals, data references and parameterless function calls)

- **Inner nodes**

(these are operators or parameterized function calls, which each process one or several sub-expressions)

- **Tree edges**

(these are intermediate results of evaluated partial expressions, each with a data type which can be statically determined at the time of compilation through processing of the nodes according to the tree structure from the leaf node (at the bottom of the tree structure) to the root node (at the top of the tree structure))

The evaluation sequence of the leaf nodes is not clearly defined by the C/C++ standard and is specified by the compiler. In Safety C, expressions with possible side effects have to be restricted, due to the undefined evaluation sequence (see also Expressions and operators [▶ 73])

Summation as example

Initial situation: The user wants to program a summation of integer expressions of type UINT8 in Safety C. The expressions can be literals, variables or function returns, in this case: a, b, c.

Intuitive approach in standard C++:

Case 1:

If the sum can exceed the UINT8 number range:

```
INT32 z = a + b + c;
```

Case 2:

If it is clear that the sum cannot exceed the UINT8 number range, or if implicit truncation (modulo arithmetic) is desired:

```
UINT8 z = a + b + c;
```

5.2.3.1.1 Case 1

Sample 1

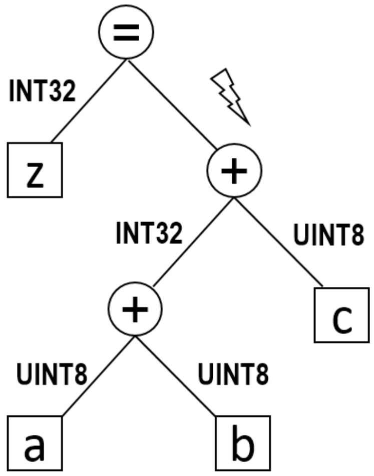


Fig. 62: Strong type system case 1 - example 1

```
UINT8 a; UINT8 b; UINT8 c;
…
INT32 z = a + b + c; // Type system error T8006 : binary ‚+ operator‘ is restricted to operands with
equal type
```

Sample 2

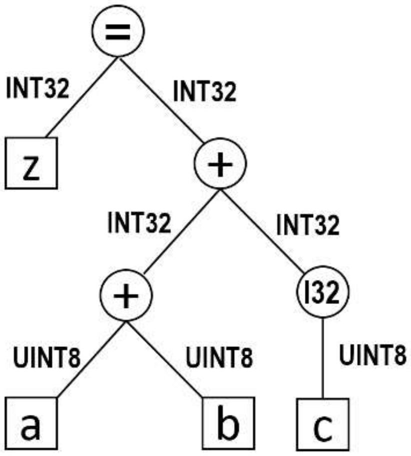


Fig. 63: Strong type system case 1 - example 2

```
UINT8 a; UINT8 b; UINT8 c;
…
INT32 z = a + b + (INT32)c; // OK but intention might be unclear
```

Sample 3

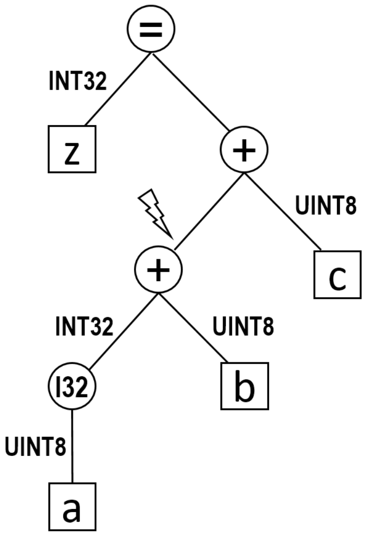


Fig. 64: Strong type system case 1 - example 3

```
UINT8 a; UINT8 b; UINT8 c;
…
INT32 z = ((INT32)a) + b + c; // Again, type system error T8006 as C/C++ binary addition operator is 
left-associative
```

Sample 4

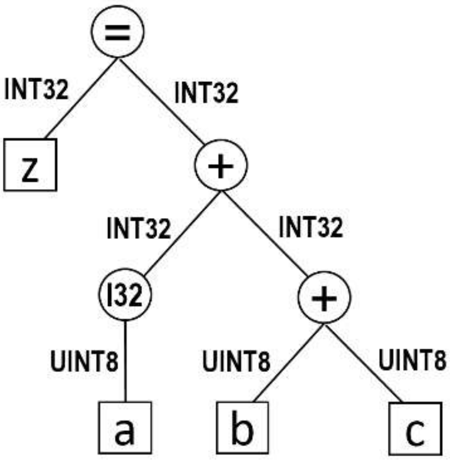


Fig. 65: Strong type system case 1 - example 4

```
UINT8 a; UINT8 b; UINT8 c;
…
INT32 z = ((INT32)a) + (b + c); // OK but intention might be unclear
```

Sample 5

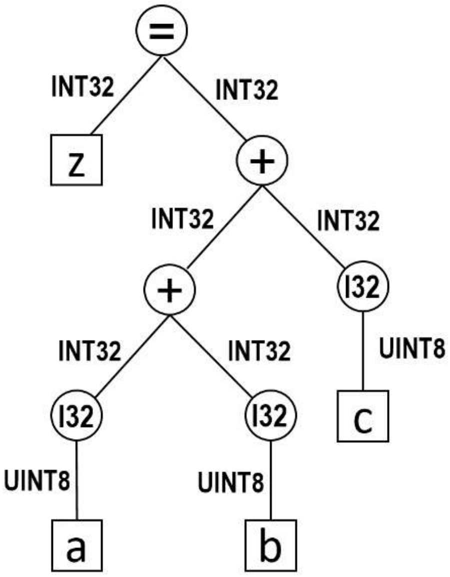


Fig. 66: Strong type system case 1 - example 5

```
UINT8 a; UINT8 b; UINT8 c;
…
INT32 z = ((INT32)a) + ((INT32)b) + ((INT32)c); // OK with clear intention
// Use this when lines get too long:
INT32 z = (INT32)a;
z += (INT32)b;
z += (INT32)c; // OK with clear intention and short lines
// Use this when overflow might accidentally occur and should be trapped:
INT32 z = (INT32)a;
… // Maybe lots of accumulations to z
z = ADDI32(z, (INT32)b);
z = ADDI32(z, (INT32)c);
```

5.2.3.1.2 Case 2

Sample 1

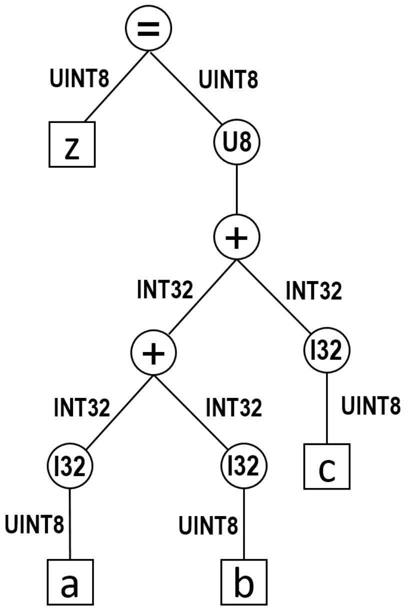


Fig. 67: Strong type system case 2 - example 1

```
UINT8 a; UINT8 b; UINT8 c;
…
UINT8 z = (UINT8)(((INT32)a) + ((INT32)b) + ((INT32)c)); // OK but lots of code to write
```

Sample 2

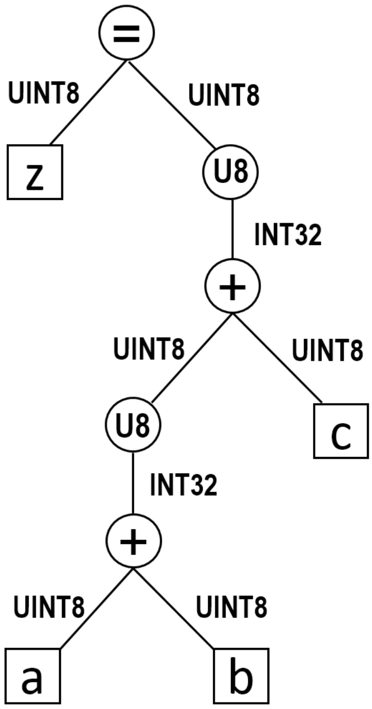


Fig. 68: Strong type system case 2 - example 2

```
UINT8 a; UINT8 b; UINT8 c;
…
UINT8 z = (UINT8)((UINT8)(a + b) + c); // OK but can get hard to read

// Better use this:
UINT8 z = a;
z = (UINT8)(z + b);
z = (UINT8)(z + c); // OK and with clear intention

// Or even that:
UINT8 z = a;
z += b;
z += c; // OK, compact and with clear intention
```

5.3 Permissible language scope

5.3.1 Simple data types

The following table shows the permissible simple data types and alternative type identifiers with value ranges.

Attention

Overflows/underflows and division by zero

Overflows or underflows of the permitted range of values or data type INT32 are not defined in the C/C++ standard and may lead to data inconsistencies. In the TwinCAT Safety PLC such inconsistencies only lead to a safe runtime error if they affect an output telegram. If this case cannot be safely ruled out through the application or by the user, the safe helper functions should be used instead of the native C/C++ operators.
On the other hand, overflows or underflows of the permitted range of values of data type UINT32 are defined in the C/C++ standard and lead to a passage (modulo arithmetic).

Division by zero is not defined in the C/C++ standard and can lead to data inconsistencies or to abortion of the safety application, which in turn leads to a safe runtime error in the TwinCAT Safety PLC. If this case cannot be safely ruled out through the application or by the user, the safe helper functions should be used instead of the native C/C++ operators. Users must select the correct data types for the values expected from their operations.

Data type Standard	Data type Safe	Range of values
BOOL	safeBOOL	FALSE / TRUE
UINT8	safeUINT8	0 .. 255
USINT	safeUSINT	0 .. 255
INT8	safeINT8	-128 .. 127
SINT	safeSINT	-128 .. 127
UINT16	safeUINT16	0 .. 65535
UINT	safeUINT	0 .. 65535
INT16	safeINT16	-32768 .. 32767
INT	safeINT	-32768 .. 32767
UINT32	safeUINT32	0 .. 4.294.967.295
UDINT	safeUDINT	0 .. 4.294.967.295
INT32	safeINT32	-2.147.483.648 .. 2.147.483.647
DINT	safeDINT	-2.147.483.648 .. 2.147.483.647

Note

Data types

INT and UINT are 16-bit types in the TwinCAT Safety PLC, as defined in IEC 61131-3.
In TwinCAT C++, INT and UINT are 32-bit types.

Note

Difference between safe and standard

From a safety perspective there is no difference between processing safe and standard data types (safe<TYPE> / <TYPE>), although we recommended using the prefix "safe" in the type declaration to identify safe signals, in order to facilitate validation and verification of the application. Users must assess any linking of safe and non-safe data within the application based on the applicable standards.

Attention

Invalid types

Any C/C++ type identifiers that are not listed, such as unsigned int, char, etc., are not permitted. Pointer types and reference types are also not permitted. All value transfers must be based on call-by-value.

Note

Specifications

- Local variables must be initialized before they are used and must be used
- Module variables must be initialized in the Init function and read in the code
- Input variables may not be written
- Output variables may not be read

5.3.2 Enumeration types

No provision is made for enumerations (user-defined enum types) in V1. Alternatively, constant definitions can be used, see Literals and constants [▶ 74].

5.3.3 Data structures

5.3.3.1 Structs

Struct types encapsulate the I/O data (safe/non-safe input/output data and cross-group data). These struct types are generated as part of <TwinSAFEGroupName>IoData.h from the alias device configuration.

Default variables of this struct types are created in the TwinSAFE group template as module variables with the name prefix "s".

Furthermore, user-defined struct types can be created in the module header file under <UserDefinedTypes>…</UserDefinedTypes>; this definition can have an optional leading "typedef". A global instance variable for a struct type is NOT permitted.

They may be nested hierarchically. Inner struct types of nested struct type definitions MUST have an inner instance variable for access. Inner struct types are anonymous types and must not be instantiated independent of their hierarchical parent struct type.

Examples

```
typedef struct MyData
{
     INT32   a;
     UINT8   b;
     BOOL    c;
};
```

```
struct MyFunctionInterface
{
     struct
     {
          BOOL  a;
          BOOL  b;
     } In;
     struct
     {
          BOOL  z;
     } Out;
}
```

Variables of struct instances are accessed via the "." access operator.

User-defined struct types can be used as bidirectional interface for transfer of I/O data for function calls. The transfer is still based on value transfer ("call-by-value").

Access examples:

```
MyFunctionInterface callFunc;
callFunc.In.a = true;
callFunc.In.b = false;
callFunc      = MyFunction(callFunc);
BOOL result   = callFunc.Out.z;
```

Struct instances must be initialized before they can be used, just like simple variables. Default initialization within the type definitions are not permitted.

5.3.3.2 Arrays

Not yet supported in V1.

5.3.4 Simple statements

Simple line statement end with a semicolon.

The following types are supported as part of a function body or the body of a control structure block:

Type-0

<expression>;
e.g. function call without return.

Type-1

<type> <identifier>;
e.g. declaration of variables without initialization.

Type-2

<identifier> = <expression>;
e.g. initialization of variables
e.g. operations/function call with result assignment

Type-3

<type> <identifier> = <expression>;

e.g. declaration of variables with initialization.

Special cases

Statements for post-increment/decrement (e.g. i++;) are regarded as type 2 statements.

The same applies to operations with assignment (e.g. a += b).

Restrictions

Combined variable declarations are not permitted:
e.g. INT32 i, j;

Multiple assignments are not permitted:

e.g. a = b = c;

NOTE: Component 1 and 3 of a for control flow statement (for (<1>; <2>; <3>)) are regarded as simple instructions, despite the fact that they appear to belong to a line statement.

Special statements

return <expression>;

Is only permitted at the end of a function with return value.

Break;

Is only permitted at the end of a "case:" / 'default:" block.

Any further C/C++ control flow statements such as goto, continue, throw, etc. are not permitted

5.3.5 Control structures

5.3.5.1 If-Else

Basic form

```
if (<COND>)
{
     <BLOCK>
}
else
{
     <BLOCK>
}
```

Guidelines

- The else branch is compulsory
  (if it is empty, the special comment /*<IntentionallyEmpty/>*/ must be added to avoid a warning)
- No else-if branches are permitted
- Restrictions for the <COND> expression:
  - Must always be the result of a comparison operation (<,>,<=,>=,==,!=); the left and right sub-expressions of comparison may have to be enclosed in parentheses (if they do not consist of a simple literal or identifier)
  - No function calls with potential side effects
  - No assignments, no post-/pre-increment/decrement

[CODE SAMPLE]

```
if ((safeIn1 && safeIn2) == true)
{
     sSafeOutputs.EL2904_FSoE_4.OutputChannel1  = true;
     sSafeOutputs.EL2904_FSoE_13.OutputChannel4 = true;
}
else
{
     sSafeOutputs.EL2904_FSoE_4.OutputChannel1  = false;
     sSafeOutputs.EL2904_FSoE_13.OutputChannel4 = false;
}
```

5.3.5.2 While

Basic form

```
while (<COND>)
{
     <BLOCK>
}
```

Guidelines

- No break statement allowed
- No continue statement allowed
- Restrictions for the <COND> expression:
  - Must always be the result of a comparison operation (<,>,<=,>=,==,!=); the left and right sub-expressions of comparison may have to be enclosed in parentheses (if they do not consist of a simple literal or identifier)
  - No function calls with potential side effects
  - No assignments, no post-/pre-increment/decrement
- Special comment /*<LoopBound max="N"/>*/ must be added at the start of <Block> in order to avoid a warning; N is the number of expected passes (N>1).

[CODE SAMPLE]

```
while (safeCounter < 10)
{
     /*<LoopBound max=”10”/>*/
     safeCounter++;
}
```

5.3.5.3 For

Basic form

```
for (<STMT1>; <COND>; <STMT2>)
{
     <BLOCK>
}
```

- No break statement allowed
- No continue statement allowed
- Restriction for <STMT1> expression
    - Type 3 (see simple statements)
- Restrictions for the <COND> expression
    - Must always be the result of a comparison operation (<,>,<=,>=,==,!=); the left and right sub-expressions of comparison may have to be enclosed in parentheses (if they do not consist of a simple literal or identifier)
    - No function calls with potential side effects
    - No assignments, no post-/pre-increment/decrement
- Restriction for <STMT2> expression
    - Type 2 (see simple statements)
    - Post-increment/decrement instruction
- The following applies if the for loop is not in a basic form, e.g. for (int i=0; i<10; i++):
  Special comment /*<LoopBound max="N"/>*/ must be added at the start of <Block> in order to avoid a warning; N is the number of expected passes (N>1).

[CODE SAMPLE]

```
for (INT32 i=N; i >= 0; i-=2)
{
     /*<LoopBound max=”42”/>*/
     DoSomeComputations();
}
```

5.3.5.4 Switch case

Basic form

```
switch (<EXPR1>)
{
     case <EXPR2>:
       <BLOCK>
       break;
     …
     default:
       <BLOCK>
       break; }
```

Guidelines

- At least one case block is required
- A default block is mandatory
- The case/default block must end with a break statement
- Restriction for <EXPR1> expression
    - No function calls with potential side effects
    - Not a logical expression (no expression of type BOOL)

  - No assignments, no post-/pre-increment/decrement
- Restriction for <EXPR2> expression
  - Constant expression (no variables, no function calls)
  - Not a logical expression (no expression of type BOOL)

5.3.6 Expressions and operators

Note

Expressions and operators

All operations follow the C++ semantics for the corresponding simple data types. All operations use the type extensions defined in the C++ standard (promotion rules), so that the result expression of an operation may not match that of the operand. This may have to be taken into account through an explicit type conversion, due to the strong typing of simple data types in Safety C (no implicit type conversions are permitted, apart from a few exceptions).

Permissible operators

Assignment operators	
binary	a=b

Restrictions:

- Not permitted as part of conditional expressions, no multiple assignments.
- Operands a and b must be of the same type, or signed/signed or unsigned/unsigned and with bit width a greater than b.

Arithmetic operators	
unary	-a
binary	a+b, a-b, a*b, a/b, a%b
with assignment	a+=b, a-=b, a*=b, a/=b, a%=b
Post-increment/decrement	a++, a--

Restrictions:

- Only simple, arithmetic data types are permitted (no BOOL, no structs).
- The operands a, b must be of the same type.
- With assignment only permitted as part of type 2 statements.
- Increment/decrement not permitted as part of expressions (only as simple line statement).
- Overflows/underflows may generate undefined behavior. Carry out appropriate checks or use safe helper functions!

Bitwise operators	
unary	~a
binary	a&b, a|b, a^b, a<<b, a>>b
with assignment	a&=b, a|=b, a^=b, a<<=b, a>>=b

Restrictions:

- Only simple, unsigned arithmetic data types are permitted (UINT8, UINT16, UINT32).
- The operands a, b must be of the same type.
- With assignment only permitted as part of type 2 statements.
- Shift operations can lead to undefined behavior. Carry out appropriate checks or use safe helper functions!

Logic operators	
unary	!a
binary	a&&b, a||b, a!=a

Restrictions:

- Only type BOOL permitted.
- Short-circuit operators &&, || are only permitted as part of conditional expressions.
  As a substitute, see safe helper functions with complete evaluation: AND(a,b), AND3(a,b,c), AND4(a,b,c,d) and OR(a,b), OR3(a,b,c), OR4(a,b,c,d).

Comparison operators	
binary	a==b, a!=b, a<b, a>b, a<=b, a>=b

Restrictions:

- Only simple data types permitted (no structs).
- The operands a, b must be of the same type.
- Comparison of BOOL is only permitted with == and !=.

Explicit type cast	
binary	(<type>)<expression>

Restrictions:

- Only simple data types permitted (no structs).
- No explicit type conversion from/to BOOL is permitted.
  As a substitute for type conversion from BOOL to arithmetic, see safe helper functions with unambiguous definition.
- Explicit conversions can lead to sign and data loss. Carry out appropriate checks or use safe helper functions!

Struct access	
binary	a.b

5.3.7 Literals and constants

Literals can be specified in Boolean, decimal, hexadecimal and binary form. The C/C++ promotion rules apply.

Integer literals

Range of values 0 .. 2^{31}-1 is specified as an expression of type (safe)INT32.

Value Range 2^{31} .. 2^{32}-1 is specified as an expression of type (safe)UINT32.

The suffix "U" defines literals of type UINT32, even if they can be represented as INT32.

Note	**Sign** Literals are specified unsigned, i.e. a minus sign is regarded as an operation. A plus sign is not permitted, since literals are implicitly regarded as positive.

Boolean literals

The literals false, true are specified as expression of type (safe)BOOL

Decimal format

0-9 with optional suffix U

Hexadecimal format

0-9 and a-f or A-F with prefix 0x or 0X and optional suffix U

Binary format

0-1 with prefix 0b or 0B and optional suffix U

Examples for invalid literals

- true

- false
- 0U
- 987654321
- 0xFF
- 0x0
- 0XFEDCBA98U
- 0b11010100
- 0B0U

Note

Type cast

The type of the captured expression must also be taken into account for the assignment of literals. For example,
INT8 x;
x = 0;
leads to a type error (in this case a type cast through (INT8) is required)
An exception is a declaration statement with combined initialization, provided the literal matches the target type and the sign suffix is selected correctly, e.g.:
UINT16 x = 65535U;
A further exception is the initialization of simple module variables, which can be assigned apart from their declaration in the module header through a type 2 statement with suitable literal, e.g.:
i8ModuleVar = 42;

Constant / preprocessor "defines"

The keyword "const" is not permitted. Constants have to be defined via a preprocessor "define" (see dedicated section in the module header file).
A preprocessor directive for defining a constant must have the following form (with optional parenthesis):

#define <IDENTIFIER> [<TYPECAST>] <-><LITERAL>

Predefined constant

SafeModuleHelper.h defines constants for minimum and maximum values of the permitted data types. Note that the maximum negative value of INT32 (-2147483648) cannot be used directly as literal:

```
#define I8_MIN ((INT8) -128 )
#define I8_MAX ((INT8) 127 )
#define I16_MIN ((INT16) -32768 )
#define I16_MAX ((INT16) 32767 )
#define I32_MIN ( -2147483647-1 )
#define I32_MAX 2147483647
#define U8_MAX ((UINT8) 255U )
#define U16_MAX ((UINT16) 65535U )
#define U32_MAX 4294967295U
```

5.3.8 Function calls and user-defined functions

A TwinSafeGroup module provides the interfaces functions void Init(), void InputUpdate(), void OutputUpdate() and void CycleUpdate(). These may only be called by the TwinCAT Safety PLC runtime.

In addition, the user may declare and define module functions with or without return value. Dedicated sections are available in the .cpp/.h files for this purpose. Functions with return value must have a final ”return” statement.

User-defined functions can be called from the four interface functions and from the specially defined functions, provided this does not lead to direct/indirect recursion. In addition, auxiliary functions are available for application via SafeModuleHelper.h.

Restrictions for function calls

- A distinction is made between “pure functions” (without side effects) and “impure functions” (with possible side effects).
- All user-implemented functions are essentially regarded as “impure functions”, since the module variables and outputs can potentially change.
- Initially, only the functions integrated by SafeModuleHelper.h are regarded as "pure functions".
- Impure functions can only be called within an instruction of type 2 or type 3. An instruction may contain no more than one call of an “impure function”.
- Pure functions may be called anywhere (statements and conditional expressions)
- The return value of a function with return must always be used, either through assignment or as parameter for a further function call, or as operand for an operation.

5.3.9 Asserts and traces

5.3.9.1 Asserts

FAILSAFE_ASSERT(<id>, <cond>)

The instruction FAILSAFE_ASSERT() can be used as tool for defensive programming for dealing with undefined application states at runtime without fallback strategy, e.g. in situations with invalid yet possible inputs. If a response at application level is possible (e.g. through setting of safe default values), a case distinction with if-else/switch control structures should be used instead.

It should be able to trigger a FAILSAFE_ASSERT() through a fault test (negative test) at module test level. If this is not the case, the instruction is presumably unnecessary, since detection of the incorrect execution of the user code is already ensured through the safe runtime environment. For c=a+b;, for example, it can therefore be assumed that the statement is executed correctly or any inconsistency is detected. The same applies to control structures, function calls, etc.

Parameter	Description
<id>	A short, concise C++ identifier (output as plain text either in the module test output or via ADS message in the TwinCAT error list window)
<cond>	Boolean conditional expression, for which the same restrictions apply as for if(), while(), etc. Is triggered if <cond> is FALSE, i.e. <cond> must hold in a valid case!

Triggering in a non-safe module test results in termination of a test case with text output. In the event of a fault test case, the test case is regarded as passed if the termination with given <ID> and in a given test step was expected. Otherwise the test case or fault test case is regarded as failed.

Note

FAILSAFE_ASSERT()

The instruction FAILSAFE_ASSERT() sets the safe state of the TwinSAFE group in the safe runtime environment, if the condition <cond> returns FALSE.

DEBUG_ASSERT(<id>, <cond>)

The instruction DEBUG_ASSERT() can be used for documenting and checking internal assumptions relating to user-defined program code (preconditions, postconditions, invariants) during the test phase.
Example: Testing of return values or parameters and operands BEFORE a function call or an operation.

Parameter	Description
<id>	A short, concise C++ identifier (output as plain text either in the module test output or via ADS message in the TwinCAT error list window)
<cond>	Boolean conditional expression, for which the same restrictions apply as for if(), while(), etc. Is triggered if <cond> is FALSE, i.e. <cond> must hold in a valid case!

Triggering in a non-safe module test results in termination of a test case with text output and evaluation as failed.

ℹ Note	**DEBUG_ASSERT()** In a safe runtime environment the instruction DEBUG_ASSERT() leads to an error or Log-Window message, if the condition <cond> returns FALSE. Execution of the safety-related application is being continued!

TEST_ASSERT(<cond>)

The instruction TEST_ASSERT() is used to check the assumptions relating to the outputs from a program module to be tested as part of a module test. Concrete results can be evaluated and general assumptions can be defined, e.g. to check the relationship between inputs and outputs and internal module variables. TEST_ASSERT() is the test counterpart to DEBUG_ASSERT() and should therefore ideally be defined by an independent person.

Parameter	Description
<cond>	Boolean conditional expression, for which the same restrictions apply as for if(), while(), etc. Is triggered if <cond> is FALSE, i.e. <cond> must hold in a valid case!

ℹ Note	**Using TEST_ASSERT() and DEBUG_ASSERT()** The instruction TEST_ASSERT() is only permitted in the code for the module test bench. It leads to termination of a test case (and its evaluation as failed), if the condition <cond> returns FALSE. The derivation of TEST_ASSERT () and DEBUG_ASSERT () statements in combination with module test and test cover measurement is an effective means for detecting implementation errors or specification errors at the design stage. In addition, generically formulated assertions (pre-conditions, post-conditions, invariants) enable the test coverage to be increased through additional, automatically generated test cases (e.g. through randomized test data).

5.3.9.2 Traces

BRANCH_TRACE()

BRANCH_TRACE() is required for branch coverage measurement in the module test environment, if no external tool is used for this purpose. A branch ID is automatically numbered according to the document sequence. The output is generated via ADS by selecting a corresponding log level when the branch is reached. The output only takes place within the safe runtime environment, not in the module test output.

A BRANCH_TRACE(), if used, must be positioned at the end of a branch, or, if return/break statements are used, directly before the statement.

A warning is generated if they are set redundantly or are incomplete. The test takes place when a BRANCH_TRACE() is used only.

Coverage of the branches is indicated in the output at the end of the module test execution. The output IDs of branches that could not be reached can be assigned to source text lines via the information in ModuleDatabase.saxml in the TwinSAFE group folder “Analysis Files”. A module test branch coverage of 100% is generally regarded as a minimum criterion for safety-related applications! The special comment /*<DefensiveBranch/>*/ can be used to exclude branches from the test coverage measurement. This should only be used in cases where unreachable code is to remain in the source text for a justified reason. This should not include branches for trapping invalid inputs, since these can be covered by a negative test.

DEBUG_TRACE(<expr>)

The instruction DEBUG_TRACE() can be used for test outputs of local variables and intermediate results for simple data types, which cannot be output via the process image.

The log output takes place via ADS in TwinCAT if the safe runtime environment is used. Within a module test a simple text output is used.

Parameter	Description
<expr>	may be a side effect-free expression of a simple data type, i.e. structs are not permitted

5.4 Performance optimizations

The implementation of necessary safety measures results in additional execution effort at runtime. In order to minimize this, the following code optimizations should be considered.

Conditional expressions in control flow statements

Complex calculations in control flow statements, particularly function calls and real-valued mathematical functions (not yet supported in V1), should initially be performed in a line statement, involving assignment to a local variable. The intermediate result stored in the variable can be incorporated in the condition, e.g. in an if-else instruction:

not optimized	optimized

if (SINF32(x) >= 0.0f)
{
    ...
}
else { … }…
	FLOAT y = SINF32(x);
	if (y >= 0.0f)
	{
	    …
	}
	else { … }

For loop conditions, constant sub-expressions that may contain complex sub-calculations should also be assigned to a variable as provisional result in line statements:

not optimized	optimized

#define _K_ 13U
…
while (n < factorial(_K_ - 1U))
{
    …
    n++;
}
	#define _K_ 13U
	…
	UINT32 upper_limit = factorial(_K_ - 1U);
	while (n < upper_limit) {
	    …
	    n++;
	}

If switch case constructs with many cases are used, the switch expression should also be handled externally. The following example illustrates in which cases an optimization should be considered (even for a purely integer based expression):

not optimized	optimized

UINT32 w1;
UINT32 w2;
UINT32 w3;
…
switch((((w1>>8U) & (w2>>16U)) |
            (w3<<24U)) % 0xffU)
{
    case 0x0U:
      … break;
    case 0x1U:
      … break;
    case 0x2U:
      … break;
    …
    case 0xfeU:
      … break;
   default:
      … break;
}
	UINT32 w1;
	UINT32 w2;
	UINT32 w3;
	…
	UINT32 select = ((w1>>8U) & (w2>>16U)) |
	                         (w3<<24U)) % 0xffU;
	switch(select)
	{
	    case 0x0U:
	      … break;
	   case 0x1U:
	      … break;
	    case 0x2U:
	      … break;
	    …
	    case 0xfeU:
	      … break;
	    default:
	      … break;
	}

5.5 Interfacing with the I/O level

Interface to standard inputs and outputs

Standard inputs

```
//! Struct providing input data of the corresponding standard alias devices
struct StandardInputs
{
     //! ..\Alias Devices\ErrAck.sds
     struct _ErrAck
     {
          BOOL In;
     } ErrAck;
     //! ..\Alias Devices\Run.sds
     struct _Run
     {
          BOOL In;
     } Run;
};
```

Standard outputs

```
//! Struct storing output data for the corresponding standard alias devices
struct StandardOutputs
{
     //! ..\Alias Devices\DiscrepancyError.sds
     struct _DiscrepancyError
     {
          BOOL Out;
     } DiscrepancyError;
     //! ..\Alias Devices\DiscrepancyCounter.sds
     struct _DiscrepancyCounter
     {
          UINT32 Out;
     } DiscrepancyCounter;
};
```

Interface to safe inputs and outputs

Safe inputs

```
//! Struct providing input data of the corresponding safety alias devices

struct SafetyInputs
{
     //! ..\Alias Devices\EL1904_FSoE_211.sds
 struct _EL1904_FSoE_211
 {
          safeBOOL InputChannel1;
          safeBOOL InputChannel2;
          safeBOOL InputChannel3;
          safeBOOL InputChannel4;
     } EL1904_FSoE_211;
     //! ..\Alias Devices\2 safe in 2 safe out.sds
     struct __2_safe_in_2_safe_out
     {
          safeBOOL InputChannel1;
          safeBOOL InputChannel2;
     } _2_safe_in_2_safe_out;
     //! ..\Alias Devices\AX 5805 Drive Option.sds
     struct _AX_5805_Drive_Option
     {
          safeBOOL Axis_1_STO;
          safeBOOL Axis_1_SSM1;
          safeBOOL Axis_1_SSM2;
          safeBOOL Axis_1_SOS1;
          safeBOOL Axis_1_SSR1;
          safeBOOL Axis_1_SDIp;
          safeBOOL Axis_1_SDIn;
          safeBOOL Axis_1_Error_Ack;
     } AX_5805_Drive_Option;

};
```

Safe outputs

```
//! Struct storing output data for the corresponding safety alias devices

struct SafetyOutputs
{
     //! ..\Alias Devices\EL2904_FSoE_13.sds
     struct _EL2904_FSoE_13
     {
          safeBOOL OutputChannel1;
          safeBOOL OutputChannel2;
          safeBOOL OutputChannel3;
          safeBOOL OutputChannel4;
     } EL2904_FSoE_13;
     //! ..\Alias Devices\EL2904_FSoE_4.sds
     struct _EL2904_FSoE_4
     {
          safeBOOL OutputChannel1;
          safeBOOL OutputChannel2;
          safeBOOL OutputChannel3;
          safeBOOL OutputChannel4;
     } EL2904_FSoE_4;
     //! ..\Alias Devices\2 safe in 2 safe out.sds
     struct __2_safe_in_2_safe_out
     {
          safeBOOL OutputChannel1;
          safeBOOL OutputChannel2;
     } _2_safe_in_2_safe_out;
     //! ..\Alias Devices\AX 5805 Drive Option.sds
     struct _AX_5805_Drive_Option
     {
          safeBOOL Axis_1_STO;
          safeBOOL Axis_1_SS11;
          safeBOOL Axis_1_SS21;
          safeBOOL Axis_1_SOS1;
          safeBOOL Axis_1_SSR1;
          safeBOOL Axis_1_SDIp;
          safeBOOL Axis_1_SDIn;
          safeBOOL Axis_1_Error_Ack;
     } AX_5805_Drive_Option;

};
```

Interface between TwinSAFE groups

```
    //! Struct storing the TwinSAFE group exchange data
    struct TSGData
    {
        //! ..TwinSafeGroup: TwinSafeGroup1
        struct _TwinSafeGroup1
        {
            //! ..Outputs
            struct _Out
            {
                safeUINT AnalogOut1;
                safeBOOL EStopOut;
            } Out;
        } TwinSafeGroup1;
    };
```

Interface to safe time signal

```
    safeUINT16 u16SafeTimer            //!< Safe external timer input (in ms)
```

The module variables u16SafeTimer can be used to access the safe time signal. This is a 16-bit timer value, which can be used for time-dependent functionalities within the safety application. This time value may not be used in the Init function (since it is not yet available at this time). The timer is only suitable for time measurements across application cycles, since the timer variable remains constant within an application cycle. Write access to the timer variable is not permitted.

5.6 Verification and validation

DANGER

Safety C application development

The user source text should be developed based on the applicable standards, in particular IEC 61508:2010. If the standard to be used is IEC 61508:2010, the explanations for the terms verification and validation can be taken from Part 4 of this standard.

Verification

Confirmation that the requirements are met, based on an examination and provision of evidence.

NOTE

In the context of this standard, verification refers to the process of demonstrating in each phase of the relevant safety lifecycle (total, E/E/PE system and software) that the results for the special inputs meet the aims and requirements specified for the phase in every respect, through analysis, mathematical inference and/or testing.

SAMPLE

Verification activities include:

- checking of the results (documents from all phases of the safety lifecycle), in order to ensure compliance with the aims and requirements of the phase, taking into account the relevant inputs of the phase;
- design reviews;
- executed tests on developed products, to ensure that they operate according to their specification;
- execution of integration tests, in which different parts of a system are assembled step by step, accompanied by testing under ambient conditions, to ensure that all parts work together as specified.

Validation

Confirmation that the special requirements for a special application are met, based on an examination and provision of objective evidence.

NOTE 1

This standard covers three validation phases:

- validation of overall safety (see IEC 61508-1, Figure 2);
- validation of the E/E/PE system (see IEC 61508-1, Figure 3);
- validation of the software (see IEC 61508-1, Figure 4).

NOTE 2

Validation refers to the process of demonstrating that the safety-relevant system meets the specific safety specification in every respect, before and after the installation. Validation of the software, for example, therefore includes examination and provision of evidence that the software meets the safety specification and requirements.

5.7 Online diagnostics

Module tests

The developer can test the created safety application via ModuleTests.cpp in standard C++ mode. During the test the safety application is not compiled and executed in the safety context, but directly within a standard C++-environment. Also, there is no assignment to the task with which the TwinCAT Safety PLC is executed in release mode.

The module test can be accessed via MODULE_TEST_BENCH_DEF(<id>), in which the test coverage measurement is controlled and the test groups are specified via the instruction MODULE_TEST_GROUP(<id>). Each test group requires a unique ID. A test group consolidates defined test cases of a group via the instruction MODULE_TEST_CASE(<id>), which was previously defined via MODULE_TEST_CASE_DEF(<id>). A test case can be subdivided into further logical test steps via the instruction MODULE_TEST_STEP(<id>). A test case defined with MODULE_TEST_CASE_DEF(<id>) can be integrated in a test group as fault test/negative test case by calling it via the instruction MODULE_FIT_CASE(<id>, <step-id>, <asset-id>). It is expected that the test case fails in a given test step with given assertion ID, in order to be deemed to have passed. This mechanism is used to test FAILSAFE_ASSERT instructions, e.g. by using invalid input values.

During creation of a TwinSAFE group, a module test bench template with a general test case is created, which can be used directly for debugging. The interface functions of a TwinSAFE group module are called periodically.

The module to be tested (TwinSAFE group) is already created and available as a test instance within the module test via the variable DUT (Device Under Test). All module variables and module functions of a TwinSAFE group module (including those declared as *private*) can be accessed via DUT.<variable name/function name>.

In the example test case DUT.init(); is called once, followed by calls of DUT.Input-, DUT.Cycle- and DUT.OutputUpdate(); in a For loop.

Before the call the application developer or tester can set the internal variables and analyze the calculation result after the call via a TEST_ASSERT instruction.

The module test is compiled via the context menu under Test Files (Build) and started in Debug mode. In future, the test results can be displayed in the safety editor via Run/Analyze (not yet supported in V1).

In addition, the user can open, extend and execute the corresponding Visual Studio project ModuleTests.vcxproj directly in the TwinSAFE group folder "Test Files". For the source code of the module tests there are no constraints in terms of coding rules or the implementation of standard libraries.

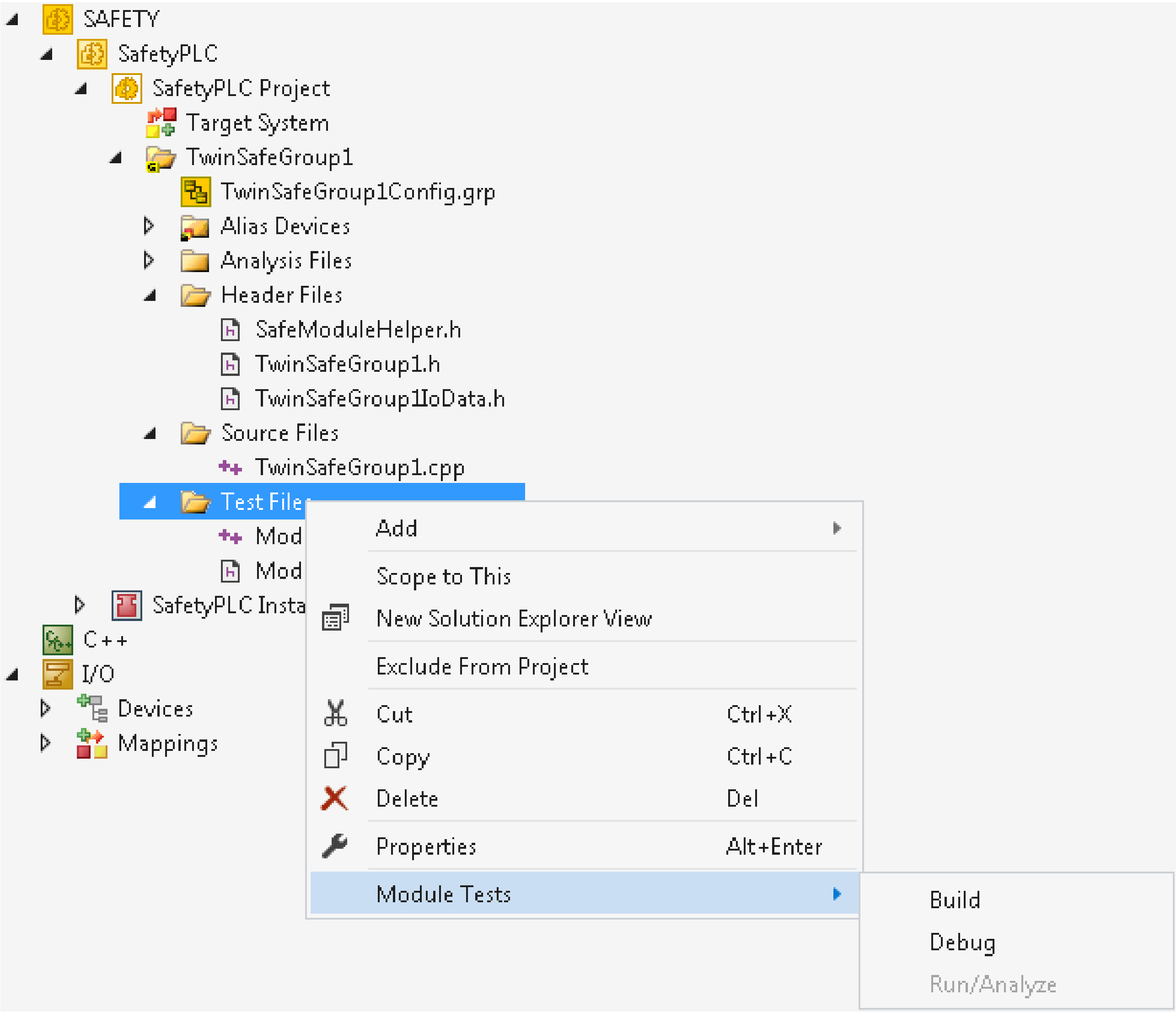


Fig. 69: Module Tests context menu

In Debug mode the usual Visual Studio mechanisms such as breakpoints, step into, step over, etc. can be used from the TwinCAT Safety Editor.

The variable values can be monitored online via the Locals window or via data tips in Visual Studio. The output is displayed in the Output window (Debug).

Example

Extract from TwinSafeGroup1.h

```
      // Module internals
      PRIVATE:

      /*<UserDefinedVariables>*/      // Define internal variables here
            INT32 a;
            INT32 b;
            INT32 z;
            BOOL neg;
      /*</UserDefinedVariables>*/
```

Extract from TwinSafeGroup1.cpp

```
      ///////////////////////////////////////////////////////////////////////////
      //! \brief Implementation of the safe user module initialization function
      ///////////////////////////////////////////////////////////////////////////
      /*<TcInit>*/
      VOID CSafeModule::Init()
      {
            // Put your module initialization code here
            a = 0;
            b = 1;
            z = 0;
            neg = false;
            BRANCH_TRACE();
      }
      /*</TcInit>*/

      ///////////////////////////////////////////////////////////////////////////
      //! \brief Implementation of the safe user module input update function
      ///////////////////////////////////////////////////////////////////////////
      /*<TcInputUpdate>*/
      VOID CSafeModule::InputUpdate()
      {
            // Put your module input update code here
            BRANCH_TRACE();
      }
      /*</TcInputUpdate>*/

      ///////////////////////////////////////////////////////////////////////////
      //! \brief Implementation of the safe user module output update function
      ///////////////////////////////////////////////////////////////////////////
      /*<TcOutputUpdate>*/
      VOID CSafeModule::OutputUpdate()
      {
            // Put your module output update code here
BRANCH_TRACE();
      }
      /*</TcOutputUpdate>*/

      ///////////////////////////////////////////////////////////////////////////
      //! \brief Implementation of the safe user module cycle update function
      ///////////////////////////////////////////////////////////////////////////
      /*<TcCycleUpdate>*/
      VOID CSafeModule::CycleUpdate()
      {
            // Put your cycle update code here
            FAILSAFE_ASSERT(DIV_BY_ZERO, b != 0);
            z = a / b;
            if (z >= 0)
            {
                  neg = false;
                  BRANCH_TRACE();
            }
            else
            {
                  neg = true;
                  BRANCH_TRACE();
            }
            BRANCH_TRACE();
      }
      /*</TcCycleUpdate>*/
```

ModuleTests.cpp

```
///////////////////////////////////////////////////////////////////////////////
//! \file       ModuleTests.cpp
//! \brief      Source file with module test definitions for TwinSafeGroup1
//! \authors    User01
//! \copyright  Put affiliation and copyright notice here
//! \version    V1.0
//! \date       2016-10-20
//! \details    Put detailed description of your module tests here
///////////////////////////////////////////////////////////////////////////////

//! Define name of the safe module under test
#define MODULE_NAME TwinSafeGroup1::CSafeModule

#include "TwinSafeGroup1.h"
#include "ModuleTests.h"
```

```
//! Definition of test case IDs
#define TC_ID_0     0
#define TC_ID_1     1

///////////////////////////////////////////////////////////////////////////
//! \brief Test bench definition containing testsets triggered by TwinCAT3
///////////////////////////////////////////////////////////////////////////
MODULE_TEST_BENCH_DEF()
{

     // Reset branch counters for coverage measurement
     START_COVERAGE_MEASUREMENT();

     // Run test group TG_ID_0
     MODULE_TEST_GROUP(TG_ID_0);

     // Compute branch coverage and identify uncovered branches
     STOP_COVERAGE_MEASUREMENT();

}

///////////////////////////////////////////////////////////////////////////
//! \brief   TC_ID_0 (put a reference to your test specification here)
//! \test    Generic module test sequence calling init and 1000 task cycles
///////////////////////////////////////////////////////////////////////////
MODULE_TEST_CASE_DEF(TC_ID_0)
{

    // Test case starts with an initial test step to prepare preconditions
    MODULE_TEST_STEP(0);
    DUT.Init(); // e.g., call Init() to set state variables to default values

    // Perform post initialization checks on module state variables here,
    // e.g., using TEST_ASSERT(<condition>) statements

    for (int nCycle = 1; nCycle <= 1000; nCycle++)
    { // Execute a test sequence consisting of 1000 module execution cycles

        MODULE_TEST_STEP(nCycle);
        // Apply test step stimuli to safe and non-safe module inputs here
        DUT.u16SafeTimer = (nCycle * 5) % 65536U; // e.g. 5ms task period

        DUT.a = 2*cycle;
        DUT.b = cycle;

        // Perform a single cycle of a periodic task execution
        DUT.InputUpdate();
        DUT.CycleUpdate();
        DUT.OutputUpdate();

        // Perform invariant checks on safe and non-safe module outputs and
        // also state variables here, e.g., using TEST_ASSERT(<condition>)

        // Perform checks on test step response w.r.t. test step stimuli
        // here, e.g., using TEST_ASSERT(<condition>) statements
        TEST_ASSERT(DUT.z == 2); // As (2*cycle)/cycle is always 2

    }

    // Perform checks on the final module state w.r.t. test specification
    // here, e.g., using TEST_ASSERT(<condition>) statements
}
```

```
////////////////////////////////////////////////////////////////////////////
//! \brief   TC_ID_1 (put a reference to your test specification here)
//! \test    Negative test case for invalid input b=0 (division by zero)
////////////////////////////////////////////////////////////////////////////
MODULE_TEST_CASE_DEF(TC_ID_1)
{

    // Test case starts with an initial test step to prepare preconditions
    MODULE_TEST_STEP(0);
    DUT.Init(); // e.g., call Init() to set state variables to default values

    // Perform post initialization checks on module state variables here,
    // e.g., using TEST_ASSERT(<condition>) statements

    // Apply test step stimuli to safe and non-safe module inputs here
    MODULE_TEST_STEP(1);
    DUT.a = 1;
    DUT.b = 0;

    // Perform a single execution cycle
    DUT.InputUpdate();
    DUT.CycleUpdate();
    DUT.OutputUpdate();

    // Should not reach here as CycleUpdate is expected to trigger fail safe!
    TEST_ASSERT(false);

}

////////////////////////////////////////////////////////////////////////////
//! \brief Test group TG_ID_0 definition containing a set of test cases
////////////////////////////////////////////////////////////////////////////
MODULE_TEST_GROUP_DEF(TG_ID_0)
{

    // Run positive example test case TC_ID_0
    MODULE_TEST_CASE_RUN(TC_ID_0);
    // Run negative example test case TC_ID_1 expecting fail-safe
    // assertion DIV_BY_ZERO being triggered at test step 1!
    MODULE_FIT_CASE_RUN(TC_ID_1, 1 /* Step */, DIV_BY_ZERO /* ID */);
}
```

5.8 Safe Helper Functions

The Safe Helper Functions offer users safe extensions for the limited Safety C language scope, which corresponds to a subset of the native C/C++ scope and therefore does not permit inclusion of non-safe standard libraries.

Helper functions are free from side effects, so that they can be used without restriction in all Safety C expressions and statements, i.e. the return value of a helper function only depends on the user-defined function parameters. Furthermore, helper functions do not directly change the module data of a safe application module or other global application data.

In the event of undefined inputs, helper functions respond by assuming the safe state for the TwinSAFE group in which they are integrated in the application execution. Undefined inputs are inputs for which a helper function is unable to generate valid output (referred to as *undef.* below). However, since function signatures enable such inputs to be transferred to helper functions as parameters, they are intercepted internally with a **FAILSAFE_ASSERT** instruction, thereby programmatically triggering the safe error state of the TwinSAFE group. The user is notified of this case via a corresponding log message.

5.8.1 Safe logic functions

The safe logic functions evaluate all Boolean operands, i.e. no "short-circuit evaluation" is used, in contrast to the native C/C++ operators && and ||.

5.8.1.1 AND

Executes a safe logical AND for two Boolean expressions.

Safety C function interface

```
BOOL      AND(BOOL xa,      BOOL xb)
safeBOOL  AND(safeBOOL xa,  safeBOOL xb)
```

Functional specification

`y = AND(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{aligned}
&x_a \in && \{false, true\} \\
&x_b \in && \{false, true\} \\
&y \in && \{false, true\} \\
&f(x_a, x_b) = && x_a \wedge x_b
\end{aligned}$$

5.8.1.2 AND3

Executes a safe logical AND for three Boolean expressions.

Safety C function interface

```
BOOL      AND3(BOOL xa,      BOOL xb,      BOOL xc)
safeBOOL  AND3(safeBOOL xa,  safeBOOL xb,  safeBOOL xc)
```

Functional specification

`y = AND3(xa, xb, xc):` $f(x_a, x_b, x_c) \rightarrow y$

$$\begin{aligned}
&x_a \in && \{false, true\} \\
&x_b \in && \{false, true\} \\
&x_c \in && \{false, true\} \\
&y \in && \{false, true\} \\
&f(x_a, x_b, x_c) = && x_a \wedge x_b \wedge x_c
\end{aligned}$$

5.8.1.3 AND4

Executes a safe logical AND for four Boolean expressions.

Safety C function interface

```
BOOL      AND4(BOOL xa,      BOOL xb,      BOOL xc,      BOOL xd)
safeBOOL  AND4(safeBOOL xa,  safeBOOL xb,  safeBOOL xc,  safeBOOL xd)
```

Functional specification

`y = AND4(xa, xb, xc, xd):` $f(x_a, x_b, x_c, x_d) \rightarrow y$

$$\begin{aligned}
&x_a \in && \{false, true\} \\
&x_b \in && \{false, true\} \\
&x_c \in && \{false, true\} \\
&x_d \in && \{false, true\} \\
&y \in && \{false, true\} \\
&f(x_a, x_b, x_c, x_d) = && x_a \wedge x_b \wedge x_c \wedge x_d
\end{aligned}$$

5.8.1.4 OR

Executes a safe logical OR for two Boolean expressions.

Safety C function interface

```
BOOL       OR(BOOL xa,       BOOL xb)
safeBOOL   OR(safeBOOL xa,   safeBOOL xb)
```

Functional specification

`y = OR(xa, xb):` $f(x_a, x_b) \to y$

$$\begin{array}{ll}
x_a \in & \{false, true\} \\
x_b \in & \{false, true\} \\
y \in & \{false, true\} \\
f(x_a, x_b) = & x_a \vee x_b
\end{array}$$

5.8.1.5 OR3

Executes a safe logical OR for three Boolean expressions.

Safety C function interface

```
BOOL       OR3(BOOL xa,       BOOL xb,       BOOL xc)
safeBOOL   OR3(safeBOOL xa,   safeBOOL xb,   safeBOOL xc)
```

Functional specification

`y = OR3(xa, xb, xc):` $f(x_a, x_b, x_c) \to y$

$$\begin{array}{ll}
x_a \in & \{false, true\} \\
x_b \in & \{false, true\} \\
x_c \in & \{false, true\} \\
y \in & \{false, true\} \\
f(x_a, x_b, x_c) = & x_a \vee x_b \vee x_c
\end{array}$$

5.8.1.6 OR4

Executes a safe logical OR for four Boolean expressions.

Safety C function interface

```
BOOL       OR4(BOOL xa,       BOOL xb,       BOOL xc,       BOOL xd)
safeBOOL   OR4(safeBOOL xa,   safeBOOL xb,   safeBOOL xc,   safeBOOL xd)
```

Functional specification

`y = OR4(xa, xb, xc, xd):` $f(x_a, x_b, x_c, x_d) \to y$

$$\begin{array}{ll}
x_a \in & \{false, true\} \\
x_b \in & \{false, true\} \\
x_c \in & \{false, true\} \\
x_d \in & \{false, true\} \\
y \in & \{false, true\} \\
f(x_a, x_b, x_c, x_d) = & x_a \vee x_b \vee x_c \vee x_d
\end{array}$$

5.8.2 Safe integer arithmetic functions

The safe arithmetic functions for integer data types detect undefined behavior, which can occur with certain C/C++ operators and standard functions due to overflows of the signed 32-bit integer type and for modulo/division by zero. If invalid values are entered, the safe group state is assumed.
NOTE: A UINT16 multiplication with the C/C++ operator "*" can lead to an undefined overflow in the resulting INT32 expression with a result greater than 2^31-1. No helper function is provided for this in V1.

5.8.2.1 ADDI32

Performs a secure addition for the signed 32-bit integer type.

Safety C function interface

```
INT32      ADDI32(INT32 xa,      INT32 xb)
DINT       ADDI32(DINT xa,       DINT xb)
safeINT32  ADDI32(safeINT32 xa,  safeINT32 xb)
safeDINT   ADDI32(safeDINT xa,   safeDINT xb)
```

Functional specification

`y = ADDI32(xa, xb):` $f(x_a, x_b) \to y$

$x_a \in \mathbb{Z}, \quad -2^{31} \leq x_a \leq 2^{31} - 1$
$x_b \in \mathbb{Z}, \quad -2^{31} \leq x_b \leq 2^{31} - 1$
$y \in \mathbb{Z}, \quad -2^{31} \leq y \leq 2^{31} - 1$

$$f(x_a, x_b) = \begin{cases} x_a + x_b, & -2^{31} \leq x_a + x_b \leq 2^{31} - 1 \\ undef., & (x_a + x_b < -2^{31}) \vee (2^{31} \leq x_a + x_b) \end{cases}$$

5.8.2.2 SUBI32

Performs a secure subtraction for the signed 32-bit integer type.

Safety C function interface

```
INT32      SUBI32(INT32 xa,      INT32 xb)
DINT       SUBI32(DINT xa,       DINT xb)
safeINT32  SUBI32(safeINT32 xa,  safeINT32 xb)
safeDINT   SUBI32(safeDINT xa,   safeDINT xb)
```

Functional specification

`y = SUBI32(xa, xb):` $f(x_a, x_b) \to y$

$x_a \in \mathbb{Z}, \quad -2^{31} \leq x_a \leq 2^{31} - 1$
$x_b \in \mathbb{Z}, \quad -2^{31} \leq x_b \leq 2^{31} - 1$
$y \in \mathbb{Z}, \quad -2^{31} \leq y \leq 2^{31} - 1$

$$f(x_a, x_b) = \begin{cases} x_a - x_b, & -2^{31} \leq x_a - x_b \leq 2^{31} - 1 \\ undef., & (x_a - x_b < -2^{31}) \vee (2^{31} \leq x_a - x_b) \end{cases}$$

5.8.2.3 MULI32

Performs a secure multiplication for the signed 32-bit integer type.

Safety C function interface

```
INT32       MULI32(INT32 xa,      INT32 xb)
DINT        MULI32(DINT xa,       DINT xb)
safeINT32   MULI32(safeINT32 xa,  safeINT32 xb)
safeDINT    MULI32(safeDINT xa,   safeDINT xb)
```

Functional specification

`y = MULI32(xa, xb):` $f(x_a, x_b) \to y$

$$\begin{aligned} &x_a \in \mathbb{Z}, && -2^{31} \leq x_a \leq 2^{31} - 1 \\ &x_b \in \mathbb{Z}, && -2^{31} \leq x_b \leq 2^{31} - 1 \\ &y \in \mathbb{Z}, && -2^{31} \leq y \leq 2^{31} - 1 \end{aligned}$$

$$f(x_a, x_b) = \begin{cases} x_a \cdot x_b, & -2^{31} \leq x_a \cdot x_b \leq 2^{31} - 1 \\ undef., & (x_a \cdot x_b < -2^{31}) \vee (2^{31} \leq x_a \cdot x_b) \end{cases}$$

5.8.2.4 DIVI32

Performs a secure division for the signed 32-bit integer type.

Safety C function interface

```
INT32       DIVI32(INT32 xa,      INT32 xb)
DINT        DIVI32(DINT xa,       DINT xb)
safeINT32   DIVI32(safeINT32 xa,  safeINT32 xb)
safeDINT    DIVI32(safeDINT xa,   safeDINT xb)
```

Functional specification

`y = DIVI32(xa, xb):` $f(x_a, x_b) \to y$

$$\begin{aligned} &x_a \in \mathbb{Z}, && -2^{31} \leq x_a \leq 2^{31} - 1 \\ &x_b \in \mathbb{Z}, && -2^{31} \leq x_b \leq 2^{31} - 1 \\ &y \in \mathbb{Z}, && -2^{31} \leq y \leq 2^{31} - 1 \end{aligned}$$

$$f(x_a, x_b) = \begin{cases} \lfloor \frac{x_a}{x_b} \rfloor, & (-2^{31} < x_b < 0) \wedge (0 < x_b) \\ undef., & (x_b = -2^{31}) \vee (x_b = 0) \end{cases}$$

5.8.2.5 DIVU32

Performs a secure division for the unsigned 32-bit integer type.

Safety C function interface

```
UINT32      DIVU32(UINT32 xa,      UINT32 xb)
UDINT       DIVU32(UDINT xa,       UDINT xb)
safeUINT32  DIVU32(safeUINT32 xa,  safeUINT32 xb)
safeUDINT   DIVU32(safeUDINT xa,   safeUDINT xb)
```

Functional specification

`y = DIVU32(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{aligned} &x_a \in \mathbb{Z}, && 0 \leq x_a \leq 2^{32} - 1 \\ &x_b \in \mathbb{Z}, && 0 \leq x_b \leq 2^{32} - 1 \\ &y \in \mathbb{Z}, && 0 \leq y \leq 2^{32} - 1 \end{aligned}$$

$$f(x_a, x_b) = \begin{cases} \lfloor \frac{x_a}{x_b} \rfloor, & 0 < x_b \\ undef., & x_b = 0 \end{cases}$$

5.8.2.6 MODI32

Performs a secure residual value calculation for the signed 32-bit integer type.

Safety C function interface

```
INT32       MODI32(INT32 xa,       INT32 xb)
DINT        MODI32(DINT xa,        DINT xb)
safeINT32   MODI32(safeINT32 xa,   safeINT32 xb)
safeDINT    MODI32(safeDINT xa,    safeDINT xb)
```

Functional specification

`y = MODI32(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{aligned} &x_a \in \mathbb{Z}, && -2^{31} \leq x_a \leq 2^{31} - 1 \\ &x_b \in \mathbb{Z}, && -2^{31} \leq x_b \leq 2^{31} - 1 \\ &y \in \mathbb{Z}, && 0 \leq y \leq 2^{31} - 2 \end{aligned}$$

$$f(x_a, x_b) = \begin{cases} x_a - \lfloor \frac{x_a}{x_b} \rfloor \cdot x_b, & (0 \leq x_a) \wedge (0 < x_b) \\ undef., & (x_a < 0) \vee (x_b \leq 0) \end{cases}$$

5.8.2.7 MODU32

Performs a secure residual value calculation for the unsigned 32-bit integer type.

Safety C function interface

```
UINT32       MODU32(UINT32 xa,       UINT32 xb)
UDINT        MODU32(UDINT xa,        UDINT xb)
safeUINT32   MODU32(safeUINT32 xa,   safeUINT32 xb)
safeUDINT    MODU32(safeUDINT xa,    safeUDINT xb)
```

Functional specification

`y = MODU32(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{aligned} &x_a \in \mathbb{Z}, && 0 \leq x_a \leq 2^{32} - 1 \\ &x_b \in \mathbb{Z}, && 0 \leq x_b \leq 2^{32} - 1 \\ &y \in \mathbb{Z}, && 0 \leq y \leq 2^{32} - 2 \end{aligned}$$

$$f(x_a, x_b) = \begin{cases} x_a - \lfloor \frac{x_a}{x_b} \rfloor \cdot x_b, & 0 < x_b \\ undef., & x_b = 0 \end{cases}$$

5.8.2.8 DIVI16

Performs a secure division for the signed 16-bit integer type.

Safety C function interface

```
INT16      DIVI16(INT16 xa,      INT16 xb)
INT        DIVI16(INT xa,        INT xb)
safeINT16  DIVI16(safeINT16 xa,  safeINT16 xb)
safeINT    DIVI16(safeINT xa,    safeINT xb)
```

Functional specification

`y = DIVI16(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{aligned} &x_a \in \mathbb{Z}, && -2^{15} \leq x_a \leq 2^{15} - 1 \\ &x_b \in \mathbb{Z}, && -2^{15} \leq x_b \leq 2^{15} - 1 \\ &y \in \mathbb{Z}, && -2^{15} \leq y \leq 2^{15} - 1 \end{aligned}$$

$$f(x_a, x_b) = \begin{cases} \lfloor \frac{x_a}{x_b} \rfloor, & (-2^{15} < x_b < 0) \wedge (0 < x_b) \\ undef., & (x_b = -2^{15}) \vee (x_b = 0) \end{cases}$$

5.8.2.9 DIVU16

Performs a secure division for the unsigned 16-bit integer type.

Safety C function interface

```
UINT16      DIVU16(UINT16 xa,      UINT16 xb)
UINT        DIVU16(UINT xa,        UINT xb)
safeUINT16  DIVU16(safeUINT16 xa,  safeUINT16 xb)
safeUINT    DIVU16(safeUINT xa,    safeUINT xb)
```

Functional specification

`y = DIVU16(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{aligned} &x_a \in \mathbb{Z}, && 0 \leq x_a \leq 2^{16} - 1 \\ &x_b \in \mathbb{Z}, && 0 \leq x_b \leq 2^{16} - 1 \\ &y \in \mathbb{Z}, && 0 \leq y \leq 2^{16} - 1 \end{aligned}$$

$$f(x_a, x_b) = \begin{cases} \lfloor \frac{x_a}{x_b} \rfloor, & 0 < x_b \\ undef., & x_b = 0 \end{cases}$$

5.8.2.10 MODI16

Performs a secure residual value calculation for the signed 16-bit integer type.

Safety C function interface

```
INT16      MODI16(INT16 xa,      INT16 xb)
INT        MODI16(INT xa,        INT xb)
safeINT16  MODI16(safeINT16 xa,  safeINT16 xb)
safeINT    MODI16(safeINT xa,    safeINT xb)
```

Functional specification

`y = MODI16(xa, xb):` $f(x_a, x_b) \to y$

$$\begin{aligned}
&x_a \in \mathbb{Z}, && -2^{15} \leq x_a \leq 2^{15} - 1 \\
&x_b \in \mathbb{Z}, && -2^{15} \leq x_b \leq 2^{15} - 1 \\
&y \in \mathbb{Z}, && 0 \leq y \leq 2^{15} - 2
\end{aligned}$$

$$f(x_a, x_b) = \begin{cases} x_a - \lfloor \frac{x_a}{x_b} \rfloor \cdot x_b, & (0 \leq x_a) \wedge (0 < x_b) \\ undef., & (x_a < 0) \vee (x_b \leq 0) \end{cases}$$

5.8.2.11 MODU16

Performs a secure residual value calculation for the unsigned 16-bit integer type.

Safety C function interface

```
UINT16       MODU16(UINT16 xa,       UINT16 xb)
UINT         MODU16(UINT xa,         UINT xb)
safeUINT16   MODU16(safeUINT16 xa,   safeUINT16 xb)
safeUINT     MODU16(safeUINT xa,     safeUINT xb)
```

Functional specification

`y = MODU16(xa, xb):` $f(x_a, x_b) \to y$

$$\begin{aligned}
&x_a \in \mathbb{Z}, && 0 \leq x_a \leq 2^{16} - 1 \\
&x_b \in \mathbb{Z}, && 0 \leq x_b \leq 2^{16} - 1 \\
&y \in \mathbb{Z}, && 0 \leq y \leq 2^{16} - 2
\end{aligned}$$

$$f(x_a, x_b) = \begin{cases} x_a - \lfloor \frac{x_a}{x_b} \rfloor \cdot x_b, & 0 < x_b \\ undef., & x_b = 0 \end{cases}$$

5.8.2.12 DIVI8

Performs a secure division for the signed 8-bit integer type.

Safety C function interface

```
INT8       DIVI8(INT8 xa,       INT8 xb)
SINT       DIVI8(SINT xa,       SINT xb)
safeINT8   DIVI8(safeINT8 xa,   safeINT8 xb)
safeSINT   DIVI8(safeSINT xa,   safeSINT xb)
```

Functional specification

`y = DIVI8(xa, xb):` $f(x_a, x_b) \to y$

$$\begin{aligned}
&x_a \in \mathbb{Z}, && -2^{7} \leq x_a \leq 2^{7} - 1 \\
&x_b \in \mathbb{Z}, && -2^{7} \leq x_b \leq 2^{7} - 1 \\
&y \in \mathbb{Z}, && -2^{7} \leq y \leq 2^{7} - 1
\end{aligned}$$

$$f(x_a, x_b) = \begin{cases} \lfloor \frac{x_a}{x_b} \rfloor, & (-2^7 < x_b < 0) \wedge (0 < x_b) \\ undef., & (x_b = -2^7) \vee (x_b = 0) \end{cases}$$

5.8.2.13 DIVU8

Performs a secure division for the unsigned 8-bit integer type.

Safety C function interface

```
UINT8      DIVU8(UINT8 xa,      UINT8 xb)
USINT      DIVU8(USINT xa,      USINT xb)
safeUINT8  DIVU8(safeUINT8 xa,  safeUINT8 xb)
safeUSINT  DIVU8(safeUSINT xa,  safeUSINT xb)
```

Functional specification

`y = DIVU8(xa, xb):` $f(x_a, x_b) \to y$

$$\begin{array}{ll} x_a \in \mathbb{Z}, & 0 \le x_a \le 2^8 - 1 \\ x_b \in \mathbb{Z}, & 0 \le x_b \le 2^8 - 1 \\ y \in \mathbb{Z}, & 0 \le y \le 2^8 - 1 \end{array}$$

$$f(x_a, x_b) = \begin{cases} \lfloor \frac{x_a}{x_b} \rfloor, & 0 < x_b \\ undef., & x_b = 0 \end{cases}$$

5.8.2.14 MODI8

Performs a secure residual value calculation for the signed 8-bit integer type.

Safety C function interface

```
INT8      MODI8(INT8 xa,      INT8 xb)
SINT      MODI8(SINT xa,      SINT xb)
safeINT8  MODI8(safeINT8 xa,  safeINT8 xb)
safeSINT  MODI8(safeSINT xa,  safeSINT xb)
```

Functional specification

`y = MODI8(xa, xb):` $f(x_a, x_b) \to y$

$$\begin{array}{ll} x_a \in \mathbb{Z}, & -2^7 \le x_a \le 2^7 - 1 \\ x_b \in \mathbb{Z}, & -2^7 \le x_b \le 2^7 - 1 \\ y \in \mathbb{Z}, & 0 \le y \le 2^7 - 2 \end{array}$$

$$f(x_a, x_b) = \begin{cases} x_a - \lfloor \frac{x_a}{x_b} \rfloor \cdot x_b, & (0 \le x_a) \wedge (0 < x_b) \\ undef., & (x_a < 0) \vee (x_b \le 0) \end{cases}$$

5.8.2.15 MODU8

Performs a secure residual value calculation for the unsigned 8-bit integer type.

Safety C function interface

```
UINT8      MODU8(UINT8 xa,      UINT8 xb)
USINT      MODU8(USINT xa,      USINT xb)
safeUINT8  MODU8(safeUINT8 xa,  safeUINT8 xb)
safeUSINT  MODU8(safeUSINT xa,  safeUSINT xb)
```

Functional specification

`y = MODU8(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{array}{ll} x_a \in \mathbb{Z}, & 0 \leq x_a \leq 2^8 - 1 \\ x_b \in \mathbb{Z}, & 0 \leq x_b \leq 2^8 - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^8 - 2 \end{array}$$

$$f(x_a, x_b) = \begin{cases} x_a - \lfloor \frac{x_a}{x_b} \rfloor \cdot x_b, & 0 < x_b \\ undef., & x_b = 0 \end{cases}$$

5.8.2.16 NEGI32

Performs a safe arithmetic negation for a signed 32-bit integer type.

Safety C function interface

```
INT32       NEGI32(INT32 x)
DINT        NEGI32(DINT x)
safeINT32   NEGI32(safeINT32 x)
safeDINT    NEGI32(safeDINT x)
```

Functional specification

`y = NEGI32(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^{31} \leq x \leq 2^{31} - 1 \\ y \in \mathbb{Z}, & -2^{31} + 1 \leq y \leq 2^{31} - 1 \end{array}$$

$$f(x) = \begin{cases} -x, & -2^{31} + 1 \leq x \leq 2^{31} - 1 \\ undef., & x = -2^{31} \end{cases}$$

5.8.2.17 NEGI16

Performs a safe arithmetic negation for a signed 16-bit integer type.

Safety C function interface

```
INT16       NEGI16(INT16 x)
INT         NEGI16(INT x)
safeINT16   NEGI16(safeINT16 x)
safeINT     NEGI16(safeINT x)
```

Functional specification

`y = NEGI16(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^{15} \leq x \leq 2^{15} - 1 \\ y \in \mathbb{Z}, & -2^{15} + 1 \leq y \leq 2^{15} - 1 \end{array}$$

$$f(x) = \begin{cases} -x, & -2^{15} + 1 \leq x \leq 2^{15} - 1 \\ undef., & x = -2^{15} \end{cases}$$

5.8.2.18 NEGI8

Performs a safe arithmetic negation for a signed 8-bit integer type.

Safety C function interface

```
INT8      NEGI8(INT8 x)
SINT      NEGI8(SINT x)
safeINT8  NEGI8(safeINT8 x)
safeSINT  NEGI8(safeSINT x)
```

Functional specification

`y = NEGI8(x):` $f(x) \to y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^7 \leq x \leq 2^7 - 1 \\ y \in \mathbb{Z}, & -2^7 + 1 \leq y \leq 2^7 - 1 \end{array}$$

$$f(x) = \begin{cases} -x, & -2^7 + 1 \leq x \leq 2^7 - 1 \\ undef., & x = -2^7 \end{cases}$$

5.8.2.19 ABSI32

Executes a safe absolute value calculation for a signed 32-bit integer type.

Safety C function interface

```
INT32      ABSI32(INT32 x)
DINT       ABSI32(DINT x)
safeINT32  ABSI32(safeINT32 x)
safeDINT   ABSI32(safeDINT x)
```

Functional specification

`y = ABSI32(x):` $f(x) \to y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^{31} \leq x \leq 2^{31} - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^{31} - 1 \end{array}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{31} - 1 \\ -x, & -2^{31} + 1 \leq x < 0 \\ undef., & x = -2^{31} \end{cases}$$

5.8.2.20 ABSI16

Executes a safe absolute value calculation for a signed 16-bit integer type.

Safety C function interface

```
INT16      ABSI16(INT8 x)
INT        ABSI16(INT x)
safeINT16  ABSI16(safeINT16 x)
safeINT    ABSI16(safeINT x)
```

Functional specification

`y = ABSI16(x):` $f(x) \to y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^{15} \leq x \leq 2^{15} - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^{15} - 1 \end{array}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{15} - 1 \\ -x, & -2^{15} + 1 \leq x < 0 \\ undef., & x = -2^{15} \end{cases}$$

5.8.2.21 ABSI8

Executes a safe absolute value calculation for a signed 8-bit integer type.

Safety C function interface

```
INT8      ABSI8(INT8 x)
SINT      ABSI8(SINT x)
safeINT8  ABSI8(safeINT8 x)
safeSINT  ABSI8(safeSINT x)
```

Functional specification

`y = ABSI8(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^7 \leq x \leq 2^7 - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^7 - 1 \end{array}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^7 - 1 \\ -x, & -2^7 + 1 \leq x < 0 \\ undef., & x = -2^7 \end{cases}$$

5.8.3 Safe bit shift functions

The safe bit shift functions prevent undefined C/C++ behavior of the native operators >> and <<. To this end the function signature is used to ensure that no signed operands can be used (this also eliminates the possibility of a signed arithmetic shift), and any shift by the right operand (shift operand), that is greater than or equal to the word width of the left operand, is intercepted.

5.8.3.1 SHLU32

Shifts the bits of an unsigned 32-bit value to the left by up to 31 bits.

Safety C function interface

```
UINT32      SHLU32(UINT32 xa,      UINT32 xb)
UDINT       SHLU32(UDINT xa,       UDINT xb)
safeUINT32  SHLU32(safeUINT32 xa,  safeUINT32 xb)
safeUDINT   SHLU32(safeUDINT xa,   safeUDINT xb)
```

Functional specification

`y = SHLU32(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{array}{ll} x_a \in \mathbb{Z}, & 0 \leq x_a \leq 2^{32} - 1 \\ x_b \in \mathbb{Z}, & 0 \leq x_b \leq 2^{32} - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^{32} - 1 \end{array}$$

$$f(x_a, x_b) = \begin{cases} (x_a \cdot 2^{x_b}) \bmod 2^{32}, & 0 \leq x_b \leq 31 \\ undef., & 32 \leq x_b \leq 2^{32} - 1 \end{cases}$$

5.8.3.2 SHLU16

Shifts the bits of an unsigned 16-bit value to the left by up to 15 bits.

Safety C function interface

```
UINT16       SHLU16(UINT16 xa,       UINT32 xb)
UINT         SHLU16(UINT xa,         UDINT xb)
safeUINT16   SHLU16(safeUINT16 xa,   safeUINT32 xb)
safeUINT     SHLU16(safeUINT xa,     safeUDINT xb)
```

Functional specification

`y = SHLU16(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{array}{ll} x_a \in \mathbb{Z}, & 0 \leq x_a \leq 2^{16} - 1 \\ x_b \in \mathbb{Z}, & 0 \leq x_b \leq 2^{32} - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^{16} - 1 \end{array}$$

$$f(x_a, x_b) = \begin{cases} (x_a \cdot 2^{x_b}) \bmod 2^{16}, & 0 \leq x_b \leq 15 \\ undef., & 16 \leq x_b \leq 2^{32} - 1 \end{cases}$$

5.8.3.3 SHLU8

Shifts the bits of an unsigned 8-bit value to the left by up to 7 bits.

Safety C function interface

```
UINT8       SHLU8(UINT8 xa,       UINT32 xb)
USINT       SHLU8(USINT xa,       UDINT xb)
safeUINT8   SHLU8(safeUINT8 xa,   safeUINT32 xb)
safeUSINT   SHLU8(safeUSINT xa,   safeUDINT xb)
```

Functional specification

`y = SHLU8(xa, xb):` $f(x_a, x_b) \rightarrow y$

$$\begin{array}{ll} x_a \in \mathbb{Z}, & 0 \leq x_a \leq 2^{8} - 1 \\ x_b \in \mathbb{Z}, & 0 \leq x_b \leq 2^{32} - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^{8} - 1 \end{array}$$

$$f(x_a, x_b) = \begin{cases} (x_a \cdot 2^{x_b}) \bmod 2^{8}, & 0 \leq x_b \leq 7 \\ undef., & 8 \leq x_b \leq 2^{32} - 1 \end{cases}$$

5.8.3.4 SHRU32

Shifts the bits of an unsigned 32-bit value to the right by up to 31 bits.

Safety C function interface

```
UINT32       SHRU32(UINT32 xa,       UINT32 xb)
UDINT        SHRU32(UDINT xa,        UDINT xb)
safeUINT32   SHRU32(safeUINT32 xa,   safeUINT32 xb)
safeUDINT    SHRU32(safeUDINT xa,    safeUDINT xb)
```

Functional specification

`y = SHRU32(xa, xb):` $f(x_a, x_b) \to y$

$$
\begin{array}{ll}
x_a \in \mathbb{Z}, & 0 \le x_a \le 2^{32} - 1 \\
x_b \in \mathbb{Z}, & 0 \le x_b \le 2^{32} - 1 \\
y \in \mathbb{Z}, & 0 \le y \le 2^{32} - 1
\end{array}
$$

$$
f(x_a, x_b) = \begin{cases} \lfloor x_a \cdot 2^{-x_b} \rfloor, & 0 \le x_b \le 31 \\ undef., & 32 \le x_b \le 2^{32} - 11 \end{cases}
$$

5.8.3.5 SHRU16

Shifts the bits of an unsigned 16-bit value to the right by up to 15 bits.

Safety C function interface

```
UINT16       SHRU16(UINT16 xa,       UINT32 xb)
UINT         SHRU16(UINT xa,         UDINT xb)
safeUINT16   SHRU16(safeUINT16 xa,   safeUINT32 xb)
safeUINT     SHRU16(safeUINT xa,     safeUDINT xb)
```

Functional specification:

`y = SHRU16(xa, xb):` $f(x_a, x_b) \to y$

$$
\begin{array}{ll}
x_a \in \mathbb{Z}, & 0 \le x_a \le 2^{16} - 1 \\
x_b \in \mathbb{Z}, & 0 \le x_b \le 2^{32} - 1 \\
y \in \mathbb{Z}, & 0 \le y \le 2^{16} - 1
\end{array}
$$

$$
f(x_a, x_b) = \begin{cases} \lfloor x_a \cdot 2^{-x_b} \rfloor, & 0 \le x_b \le 15 \\ undef., & 16 \le x_b \le 2^{32} - 1 \end{cases}
$$

5.8.3.6 SHRU8

Shifts the bits of an unsigned 8-bit value to the right by up to 7 bits.

Safety C function interface

```
UINT8       SHRU8(UINT8 xa,       UINT32 xb)
USINT       SHRU8(USINT xa,       UDINT xb)
safeUINT8   SHRU8(safeUINT8 xa,   safeUINT32 xb)
safeUSINT   SHRU8(safeUSINT xa,   safeUDINT xb)
```

Functional specification

`y = SHRU8(xa, xb):` $f(x_a, x_b) \to y$

$$
\begin{array}{ll}
x_a \in \mathbb{Z}, & 0 \le x_a \le 2^{8} - 1 \\
x_b \in \mathbb{Z}, & 0 \le x_b \le 2^{32} - 1 \\
y \in \mathbb{Z}, & 0 \le y \le 2^{8} - 1
\end{array}
$$

$$
f(x_a, x_b) = \begin{cases} \lfloor x_a \cdot 2^{-x_b} \rfloor, & 0 \le x_b \le 7 \\ undef., & 8 \le x_b \le 2^{32} - 1 \end{cases}
$$

5.8.4 Safe conversion functions (Boolean to integer)

Safe conversions of Boolean expressions to integer data types map the Boolean truth value FALSE to arithmetic 0 and the Boolean truth value TRUE to arithmetic 1 in the respective target data type. This avoids any ambiguity of the explicit type conversion with the cast operator, which may otherwise lead to unexpected behavior for the application developer.

5.8.4.1 BTOI32

Executes a safe conversion of a Boolean expression to a signed 32-bit integer type.

Safety C function interface

```
INT32       BTOI32(BOOL x)
DINT        BTOI32(BOOL x)
safeINT32   BTOI32(safeBOOL x)
safeDINT    BTOI32(safeBOOL x)
```

Functional specification

$$\texttt{y = BTOI32(x):} \quad f(x) \rightarrow y$$

$$\begin{array}{ll} x \in & \{false, true\} \\ y \in \mathbb{Z}, & 0 \leq y \leq 1 \end{array}$$

$$f(x) = \begin{cases} 1, & x \\ 0, & \neg x \end{cases}$$

5.8.4.2 BTOI16

Executes a safe conversion of a Boolean expression to a signed 16-bit integer type.

Safety C function interface

```
INT16       BTOI16(BOOL x)
INT         BTOI16(BOOL x)
safeINT16   BTOI16(safeBOOL x)
safeINT     BTOI16(safeBOOL x)
```

Functional specification

$$\texttt{y = BTOI16(x):} \quad f(x) \rightarrow y$$

$$\begin{array}{ll} x \in & \{false, true\} \\ y \in \mathbb{Z}, & 0 \leq y \leq 1 \end{array}$$

$$f(x) = \begin{cases} 1, & x \\ 0, & \neg x \end{cases}$$

5.8.4.3 BTOI8

Executes a safe conversion of a Boolean expression to a signed 8-bit integer type.

Safety C function interface

```
INT8       BTOI8(BOOL x)
SINT       BTOI8(BOOL x)
safeINT8   BTOI8(safeBOOL x)
safeSINT   BTOI8(safeBOOL x)
```

Functional specification

`y = BTOI8(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in & \{false, true\} \\ y \in \mathbb{Z}, & 0 \leq y \leq 1 \end{array}$$

$$f(x) = \begin{cases} 1, & x \\ 0, & \neg x \end{cases}$$

5.8.4.4 BTOU32

Executes a safe conversion of a Boolean expression to an unsigned 32-bit integer type.

Safety C function interface

```
UINT32       BTOU32(BOOL x)
UDINT        BTOU32(BOOL x)
safeUINT32   BTOU32(safeBOOL x)
safeUDINT    BTOU32(safeBOOL x)
```

Functional specification

`y = BTOU32(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in & \{false, true\} \\ y \in \mathbb{Z}, & 0 \leq y \leq 1 \end{array}$$

$$f(x) = \begin{cases} 1, & x \\ 0, & \neg x \end{cases}$$

5.8.4.5 BTOU16

Executes a safe conversion of a Boolean expression to an unsigned 16-bit integer type.

Safety C function interface

```
UINT16       BTOU16(BOOL x)
UINT         BTOU16(BOOL x)
safeUINT16   BTOU16(safeBOOL x)
safeUINT     BTOU16(safeBOOL x)
```

Functional specification

`y = BTOU16(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in & \{false, true\} \\ y \in \mathbb{Z}, & 0 \leq y \leq 1 \end{array}$$

$$f(x) = \begin{cases} 1, & x \\ 0, & \neg x \end{cases}$$

5.8.4.6 BTOU8

Executes a safe conversion of a Boolean expression to an unsigned 8-bit integer type.

Safety C function interface

```
UINT8       BTOU8(BOOL x)
USINT       BTOU8(BOOL x)
safeUINT8   BTOU8(safeBOOL x)
safeUSINT   BTOU8(safeBOOL x)
```

Functional specification

`y = BTOU8(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in & \{false, true\} \\ y \in \mathbb{Z}, & 0 \leq y \leq 1 \end{array}$$

$$f(x) = \begin{cases} 1, & x \\ 0, & \neg x \end{cases}$$

5.8.5 Safe conversion functions (integer to integer)

Safe conversion functions between integer types are value-preserving and sign-preserving. The conversion functions for all potentially lossy combinations of source and target types therefore intercept cases for which it would not be possible to represent the value of the source type within the range of the target type. The native C/C++ type conversion operator must be used for risk-free type conversion or possibly intended loss of value or sign.

5.8.5.1 I8TOU8

Executes a safe conversion of the signed 8-bit integer type to the unsigned 8-bit integer type.

Safety C function interface

```
UINT8       I8TOU8(INT8 x)
USINT       I8TOU8(SINT x)
safeUINT8   I8TOU8(safeINT8 x)
safeUSINT   I8TOU8(safeSINT x)
```

Functional specification

`y = I8TOU8(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^7 \leq x \leq 2^7 - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^7 - 1 \end{array}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^7 - 1 \\ undef., & -2^7 \leq x < 0 \end{cases}$$

5.8.5.2 I8TOU16

Executes a safe conversion of the signed 8-bit integer type to the unsigned 16-bit integer type.

Safety C function interface

```
UINT16      I8TOU16(INT8 x)
UINT        I8TOU16(SINT x)
safeUINT16  I8TOU16(safeINT8 x)
safeUINT    I8TOU16(safeSINT x)
```

Functional specification

`y = I8TOU16(x):` $f(x) \rightarrow y$

$$x \in \mathbb{Z}, \quad -2^7 \leq x \leq 2^7 - 1$$
$$y \in \mathbb{Z}, \quad 0 \leq y \leq 2^7 - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^7 - 1 \\ undef., & -2^7 \leq x < 0 \end{cases}$$

5.8.5.3 I8TOU32

Executes a safe conversion of the signed 8-bit integer type to the unsigned 32-bit integer type.

Safety C function interface

```
UINT32      I8TOU32(INT8 x)
UDINT       I8TOU32(SINT x)
safeUINT32  I8TOU32(safeINT8 x)
safeUDINT   I8TOU32(safeSINT x)
```

Functional specification

`y = I8TOU32(x):` $f(x) \rightarrow y$

$$x \in \mathbb{Z}, \quad -2^7 \leq x \leq 2^7 - 1$$
$$y \in \mathbb{Z}, \quad 0 \leq y \leq 2^7 - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^7 - 1 \\ undef., & -2^7 \leq x < 0 \end{cases}$$

5.8.5.4 U8TOI8

Executes a safe conversion of the signed 8-bit integer type to the unsigned 8-bit integer type.

Safety C function interface

```
INT8      U8TOI8(UINT8 x)
SINT      U8TOI8(USINT x)
safeINT8  U8TOI8(safeUINT8 x)
safeSINT  U8TOI8(safeUSINT x)
```

Functional specification

`y = U8TOI8(x):` $f(x) \rightarrow y$

$$x \in \mathbb{Z}, \quad 0 \leq x \leq 2^8 - 1$$
$$y \in \mathbb{Z}, \quad 0 \leq y \leq 2^7 - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^7 - 1 \\ undef., & 2^7 \leq x \leq 2^8 - 1 \end{cases}$$

5.8.5.5 I16TOI8

Executes a safe conversion of the signed 16-bit integer type to the signed 8-bit integer type.

Safety C function interface

```
INT8      I16TOI8(INT16 x)
SINT      I16TOI8(INT x)
safeINT8  I16TOI8(safeINT16 x)
safeSINT  I16TOI8(safeINT x)
```

Functional specification

`y = I16TOI8(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^{15} \leq x \leq 2^{15} - 1 \\ y \in \mathbb{Z}, & -2^{7} \leq y \leq 2^{7} - 1 \end{array}$$

$$f(x) = \begin{cases} x, & -2^{7} \leq x \leq 2^{7} - 1 \\ \mathit{undef.}, & (-2^{15} \leq x < -2^{7}) \vee (2^{7} \leq x \leq 2^{15} - 1) \end{cases}$$

5.8.5.6 I16TOU8

Executes a safe conversion of the signed 16-bit integer type to the unsigned 8-bit integer type.

Safety C function interface

```
UINT8      I16TOU8(INT16 x)
USINT      I16TOU8(INT x)
safeUINT8  I16TOU8(safeINT16 x)
safeUSINT  I16TOU8(safeINT x)
```

Functional specification

`y = I16TOU8(x):` $f(x) \rightarrow y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^{15} \leq x \leq 2^{15} - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^{8} - 1 \end{array}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{8} - 1 \\ \mathit{undef.}, & (-2^{15} \leq x < 0) \vee (2^{8} \leq x \leq 2^{15} - 1) \end{cases}$$

5.8.5.7 I16TOU16

Executes a safe conversion of the signed 16-bit integer type to the unsigned 16-bit integer type.

Safety C function interface

```
UINT16      I16TOU16(INT16 x)
UINT        I16TOU16(INT x)
safeUINT16  I16TOU16(safeINT16 x)
safeUINT    I16TOU16(safeINT x)
```

Functional specification

`y = I16TOU16(x):` $f(x) \to y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^{15} \leq x \leq 2^{15} - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^{15} - 1 \end{array}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{15} - 1 \\ undef., & -2^{15} \leq x < 0 \end{cases}$$

5.8.5.8 I16TOU32

Executes a safe conversion of the signed 16-bit integer type to the unsigned 32-bit integer type.

Safety C function interface

```
UINT32      I16TOU32(INT16 x)
UDINT       I16TOU32(INT x)
safeUINT32  I16TOU32(safeINT16 x)
safeUDINT   I16TOU32(safeINT x)
```

Functional specification

`y = I16TOU32(x):` $f(x) \to y$

$$\begin{array}{ll} x \in \mathbb{Z}, & -2^{15} \leq x \leq 2^{15} - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^{15} - 1 \end{array}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{15} - 1 \\ undef., & -2^{15} \leq x < 0 \end{cases}$$

5.8.5.9 U16TOI8

Executes a safe conversion of the signed 16-bit integer type to the unsigned 8-bit integer type.

Safety C function interface

```
INT8      U16TOI8(UINT16 x)
SINT      U16TOI8(UINT x)
safeINT8  U16TOI8(safeUINT16 x)
safeSINT  U16TOI8(safeUINT x)
```

Functional specification

`y = U16TOI8(x):` $f(x) \to y$

$$\begin{array}{ll} x \in \mathbb{Z}, & 0 \leq x \leq 2^{16} - 1 \\ y \in \mathbb{Z}, & 0 \leq y \leq 2^{7} - 1 \end{array}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{7} - 1 \\ undef., & 2^{7} \leq x \leq 2^{16} - 1 \end{cases}$$

5.8.5.10 U16TOU8

Executes a safe conversion of the unsigned 16-bit integer type to the unsigned 8-bit integer type.

Safety C function interface

```
UINT8       U16TOU8(UINT16 x)
USINT       U16TOU8(UINT x)
safeUINT8   U16TOU8(safeUINT16 x)
safeUSINT   U16TOU8(safeUINT x)
```

Functional specification

`y = U16TOU8(x):` $f(x) \rightarrow y$

$$\begin{aligned} &x \in \mathbb{Z}, && 0 \leq x \leq 2^{16} - 1 \\ &y \in \mathbb{Z}, && 0 \leq y \leq 2^{8} - 1 \end{aligned}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{8} - 1 \\ undef., & 2^{8} \leq x \leq 2^{16} - 1 \end{cases}$$

5.8.5.11 U16TOI16

Executes a safe conversion of the signed 16-bit integer type to the unsigned 16-bit integer type.

Safety C function interface

```
INT16       U16TOI16(UINT16 x)
INT         U16TOI16(UINT x)
safeINT16   U16TOI16(safeUINT16 x)
safeINT     U16TOI16(safeUINT x)
```

Functional specification

`y = U16TOI16(x):` $f(x) \rightarrow y$

$$\begin{aligned} &x \in \mathbb{Z}, && 0 \leq x \leq 2^{16} - 1 \\ &y \in \mathbb{Z}, && 0 \leq y \leq 2^{15} - 1 \end{aligned}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{15} - 1 \\ undef., & 2^{15} \leq x \leq 2^{16} - 1 \end{cases}$$

5.8.5.12 I32TOI8

Executes a safe conversion of the signed 32-bit integer type to the signed 8-bit integer type.

Safety C function interface

```
INT8       I32TOI8(INT32 x)
SINT       I32TOI8(DINT x)
safeINT8   I32TOI8(safeINT32 x)
safeSINT   I32TOI8(safeDINT x)
```

Functional specification

`y = I32TOI8(x):` $f(x) \rightarrow y$

$$\begin{aligned} &x \in \mathbb{Z}, && -2^{31} \leq x \leq 2^{31} - 1 \\ &y \in \mathbb{Z}, && -2^{7} \leq y \leq 2^{7} - 1 \end{aligned}$$

$$f(x) = \begin{cases} x, & -2^{7} \leq x \leq 2^{7} - 1 \\ undef., & (-2^{31} \leq x < -2^{7}) \vee (2^{7} \leq x \leq 2^{31} - 1) \end{cases}$$

5.8.5.13 I32TOU8

Executes a safe conversion of the signed 32-bit integer type to the unsigned 8-bit integer type.

Safety C function interface

```
UINT8       I32TOU8(INT32 x)
USINT       I32TOU8(DINT x)
safeUINT8   I32TOU8(safeINT32 x)
safeUSINT   I32TOU8(safeDINT x)
```

Functional specification

`y = I32TOU8(x):` $f(x) \rightarrow y$

$$\begin{aligned} x &\in \mathbb{Z}, & -2^{31} \leq x \leq 2^{31}-1 \\ y &\in \mathbb{Z}, & 0 \leq y \leq 2^{8}-1 \end{aligned}$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{8}-1 \\ undef., & (-2^{31} \leq x < 0) \vee (2^{8} \leq x \leq 2^{31}-1) \end{cases}$$

5.8.5.14 I32TOI16

Executes a safe conversion of the signed 32-bit integer type to the signed 16-bit integer type.

Safety C function interface

```
INT8      I32TOI16(INT32 x)
SINT      I32TOI16(DINT x)
safeINT8  I32TOI16(safeINT32 x)
safeSINT  I32TOI16(safeDINT x)
```

Functional specification

`y = I32TOI16(x):` $f(x) \rightarrow y$

$$\begin{aligned} x &\in \mathbb{Z}, & -2^{31} \leq x \leq 2^{31}-1 \\ y &\in \mathbb{Z}, & -2^{15} \leq y \leq 2^{15}-1 \end{aligned}$$

$$f(x) = \begin{cases} x, & -2^{15} \leq x \leq 2^{15}-1 \\ undef., & (-2^{31} \leq x < -2^{15}) \vee (2^{15} \leq x \leq 2^{31}-1) \end{cases}$$

5.8.5.15 I32TOU16

Executes a safe conversion of the signed 32-bit integer type to the unsigned 16-bit integer type.

Safety C function interface

```
UINT16      I32TOU16(INT32 x)
UINT        I32TOU16(DINT x)
safeUINT16  I32TOU16(safeINT32 x)
safeUINT    I32TOU16(safeDINT x)
```

Functional specification

`y = I32TOU16(x):` $f(x) \rightarrow y$

$$x \in \mathbb{Z}, \quad -2^{31} \leq x \leq 2^{31} - 1$$
$$y \in \mathbb{Z}, \quad 0 \leq y \leq 2^{16} - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{16} - 1 \\ undef., & (-2^{31} \leq x < 0) \vee (2^{16} \leq x \leq 2^{31} - 1) \end{cases}$$

5.8.5.16 I32TOU32

Executes a safe conversion of the signed 32-bit integer type to the unsigned 32-bit integer type.

Safety C function interface

```
UINT32      I32TOU32(INT32 x)
UDINT       I32TOU32(DINT x)
safeUINT32  I32TOU32(safeINT32 x)
safeUDINT   I32TOU32(safeDINT x)
```

Functional specification

`y = I32TOU32(x):` $f(x) \rightarrow y$

$$x \in \mathbb{Z}, \quad -2^{31} \leq x \leq 2^{31} - 1$$
$$y \in \mathbb{Z}, \quad 0 \leq y \leq 2^{31} - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{31} - 1 \\ undef., & (-2^{31} \leq x < 0) \vee (2^{31} \leq x \leq 2^{31} - 1) \end{cases}$$

5.8.5.17 U32TOI8

Executes a safe conversion of the signed 32-bit integer type to the unsigned 8-bit integer type.

Safety C function interface

```
INT8      U32TOI8(UINT32 x)
SINT      U32TOI8(UDINT x)
safeINT8  U32TOI8(safeUINT32 x)
safeSINT  U32TOI8(safeUDINT x)
```

Functional specification

`y = U32TOI8(x):` $f(x) \rightarrow y$

$$x \in \mathbb{Z}, \quad 0 \leq x \leq 2^{32} - 1$$
$$y \in \mathbb{Z}, \quad 0 \leq y \leq 2^{7} - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{7} - 1 \\ undef., & 2^{7} \leq x \leq 2^{32} - 1 \end{cases}$$

5.8.5.18 U32TOU8

Executes a safe conversion of the unsigned 32-bit integer type to the unsigned 8-bit integer type.

Safety C function interface

```
UINT8       U32TOU8(UINT32 x)
USINT       U32TOU8(UDINT x)
safeUINT8   U32TOU8(safeUINT32 x)
safeUSINT   U32TOU8(safeUDINT x)
```

Functional specification

`y = U32TOU8(x):` $f(x) \to y$

$$x \in \mathbb{Z}, \qquad 0 \leq x \leq 2^{32} - 1$$
$$y \in \mathbb{Z}, \qquad 0 \leq y \leq 2^{8} - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{8} - 1 \\ undef., & 2^{8} \leq x \leq 2^{32} - 1) \end{cases}$$

5.8.5.19 U32TOI16

Executes a safe conversion of the signed 32-bit integer type to the unsigned 16-bit integer type.

Safety C function interface

```
INT8      U32TOI16(UINT32 x)
SINT      U32TOI16(UDINT x)
safeINT8  U32TOI16(safeUINT32 x)
safeSINT  U32TOI16(safeUDINT x)
```

Functional specification

`y = U32TOI16(x):` $f(x) \to y$

$$x \in \mathbb{Z}, \qquad 0 \leq x \leq 2^{32} - 1$$
$$y \in \mathbb{Z}, \qquad 0 \leq y \leq 2^{15} - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{15} - 1 \\ undef., & 2^{15} \leq x \leq 2^{32} - 1 \end{cases}$$

5.8.5.20 U32TOU16

Executes a safe conversion of the unsigned 32-bit integer type to the unsigned 16-bit integer type.

Safety C function interface

```
UINT16      U32TOU16(UINT32 x)
UINT        U32TOU16(UDINT x)
safeUINT16  U32TOU16(safeUINT32 x)
safeUINT    U32TOU16(safeUDINT x)
```

Functional specification

`y = U32TOU16(x):` $f(x) \to y$

$$x \in \mathbb{Z}, \qquad 0 \leq x \leq 2^{32} - 1$$
$$y \in \mathbb{Z}, \qquad 0 \leq y \leq 2^{16} - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{16} - 1 \\ undef., & 2^{16} \leq x \leq 2^{32} - 1 \end{cases}$$

5.8.5.21 U32TOI32

Executes a safe conversion of the signed 32-bit integer type to the unsigned 32-bit integer type.

Safety C function interface

```
INT32       U32TOI32(UINT32 x)
DINT        U32TOI32(UDINT x)
safeINT32   U32TOI32(safeUINT32 x)
safeDINT    U32TOI32(safeUDINT x)
```

Functional specification

`y = U32TOI32(x):` $f(x) \rightarrow y$

$$x \in \mathbb{Z}, \qquad 0 \leq x \leq 2^{32} - 1$$
$$y \in \mathbb{Z}, \qquad 0 \leq y \leq 2^{31} - 1$$

$$f(x) = \begin{cases} x, & 0 \leq x \leq 2^{31} - 1 \\ undef., & 2^{31} \leq x \leq 2^{32} - 1 \end{cases}$$

6 Graphical application development

ℹ **Note**	**Use of the TwinCAT Safety Editor** The use of the TwinCAT Safety Editor together with the TwinCAT Safety PLC will be implemented in one of the next releases. Currently this is not possible.

7 Appendix

7.1 Support and Service

Beckhoff and their partners around the world offer comprehensive support and service, making available fast and competent assistance with all questions related to Beckhoff products and system solutions.

Beckhoff's branch offices and representatives

Please contact your Beckhoff branch office or representative for local support and service on Beckhoff products!

The addresses of Beckhoff's branch offices and representatives round the world can be found on her internet pages:
http://www.beckhoff.com

You will also find further documentation for Beckhoff components there.

Beckhoff Headquarters

Beckhoff Automation GmbH & Co. KG

Huelshorstweg 20
33415 Verl
Germany

Phone:	+49(0)5246/963-0
Fax:	+49(0)5246/963-198
e-mail:	info@beckhoff.com

Beckhoff Support

Support offers you comprehensive technical assistance, helping you not only with the application of individual Beckhoff products, but also with other, wide-ranging services:

- support
- design, programming and commissioning of complex automation systems
- and extensive training program for Beckhoff system components

Hotline:	+49(0)5246/963-157
Fax:	+49(0)5246/963-9157
e-mail:	support@beckhoff.com

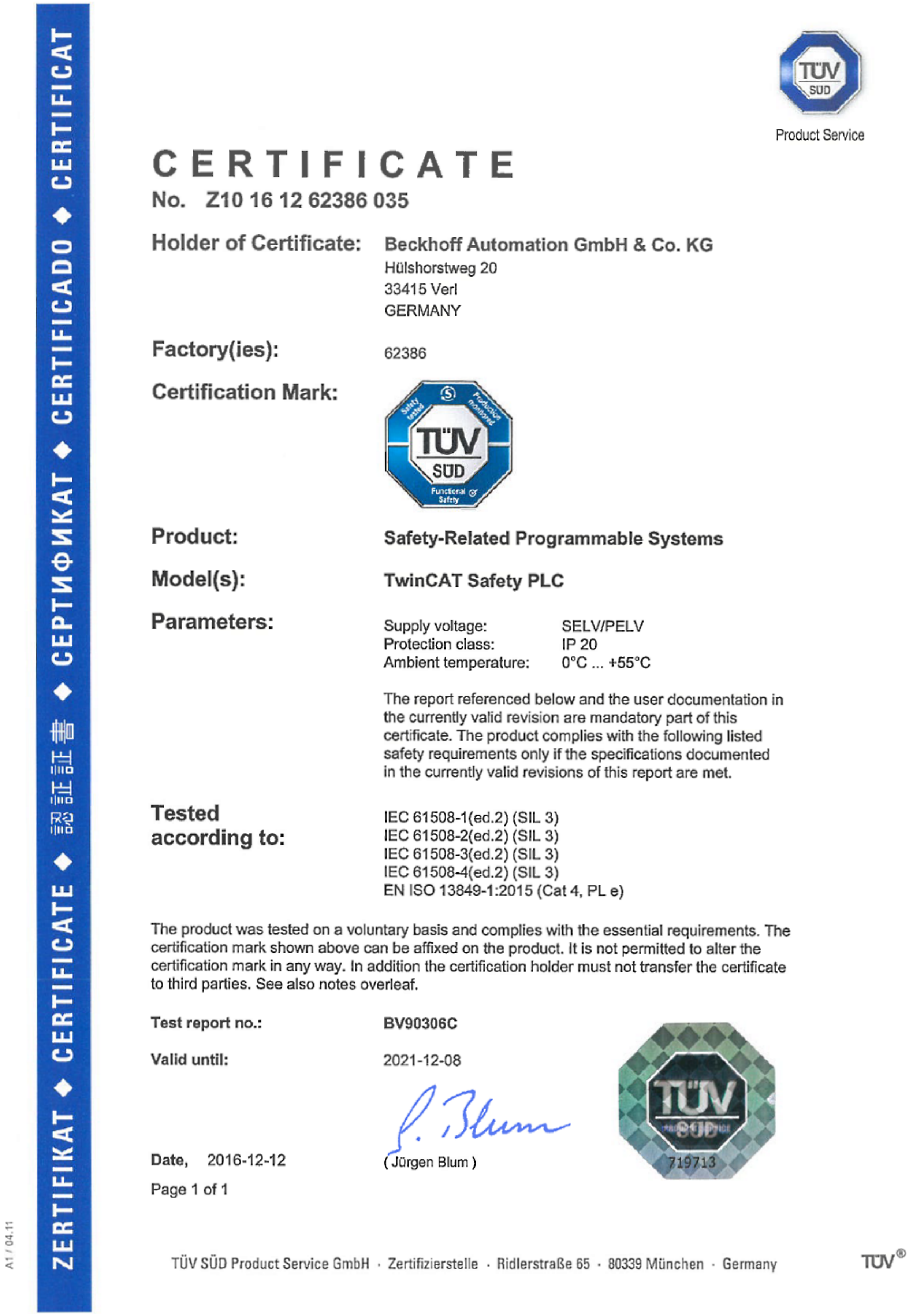
Beckhoff Service

The Beckhoff Service Center supports you in all matters of after-sales service:

- on-site service
- repair service
- spare parts service
- hotline service

Hotline:	+49(0)5246/963-460
Fax:	+49(0)5246/963-479
e-mail:	service@beckhoff.com

7.2 Certificates



ZERTIFIKAT ◆ CERTIFICATE ◆ 認証証書 ◆ СЕРТИФИКАТ ◆ CERTIFICADO ◆ CERTIFICAT

A1 / 04.11

TÜV SÜD

Product Service

CERTIFICATE

No. Z10 16 12 62386 035

Holder of Certificate: **Beckhoff Automation GmbH & Co. KG**
Hülshorstweg 20
33415 Verl
GERMANY

Factory(ies): 62386

Certification Mark:

Product: **Safety-Related Programmable Systems**

Model(s): **TwinCAT Safety PLC**

Parameters:

Supply voltage:	SELV/PELV
Protection class:	IP 20
Ambient temperature:	0°C ... +55°C

The report referenced below and the user documentation in the currently valid revision are mandatory part of this certificate. The product complies with the following listed safety requirements only if the specifications documented in the currently valid revisions of this report are met.

Tested according to:

IEC 61508-1(ed.2) (SIL 3)
IEC 61508-2(ed.2) (SIL 3)
IEC 61508-3(ed.2) (SIL 3)
IEC 61508-4(ed.2) (SIL 3)
EN ISO 13849-1:2015 (Cat 4, PL e)

The product was tested on a voluntary basis and complies with the essential requirements. The certification mark shown above can be affixed on the product. It is not permitted to alter the certification mark in any way. In addition the certification holder must not transfer the certificate to third parties. See also notes overleaf.

Test report no.: **BV90306C**

Valid until: 2021-12-08

J. Blum

Date, 2016-12-12

(Jürgen Blum)

TÜV SÜD 719713

Page 1 of 1

TÜV SÜD Product Service GmbH · Zertifizierstelle · Ridlerstraße 65 · 80339 München · Germany

TÜV®

List of figures