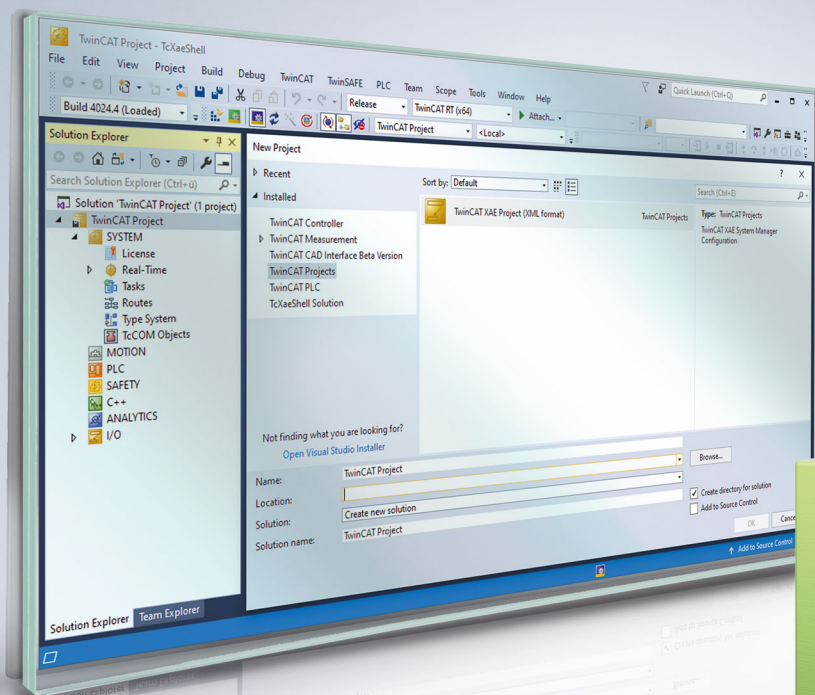


# BECKHOFF New Automation Technology

Manual | EN

# TE1200

TwinCAT 3 | PLC Static Analysis





# Table of contents

<b>1 Foreword</b>	<b>5</b>
1.1 Notes on the documentation	5
1.2 For your safety	5
1.3 Notes on information security	7
1.4 Information on the security risk analysis	8
<b>2 Overview</b>	<b>9</b>
<b>3 Installation</b>	<b>11</b>
3.1 Functionality: Light vs. full	11
3.2 System Requirements	13
3.3 Licensing	13
<b>4 Configuration</b>	<b>14</b>
4.1 Settings	14
4.2 Rules	16
4.2.1 Rules - overview and description	17
4.3 Naming conventions	81
4.3.1 Naming conventions – overview and description	83
4.3.2 Options	90
4.3.3 Placeholder {datatype}	93
4.4 Metrics	94
4.4.1 Metrics - overview and description	95
4.5 Forbidden symbols	110
<b>5 Commands</b>	<b>111</b>
5.1 Command 'Run static analysis'	111
5.1.1 Syntax in the message window	112
5.2 Command 'Run static analysis [Check all objects]'	113
5.3 Command 'View Standard Metrics'	114
5.3.1 Commands in the context menu of the 'Standard Metrics' view	115
5.4 Command 'View Standard Metrics [Check all objects]'	116
5.4.1 Commands in the context menu of the 'Standard Metrics' view	117
5.5 'Show constant propagation values for current editor' command	118
5.6 Command 'Show cognitive complexity for current editor'	119
<b>6 Pragmas and attributes</b>	<b>121</b>
<b>7 Constant propagation</b>	<b>126</b>
<b>8 QuickFix/Precompile</b>	<b>130</b>
<b>9 Automation Interface support</b>	<b>132</b>
<b>10 Examples</b>	<b>135</b>
10.1 Static analysis	135
10.2 Standard metrics	136
<b>11 Support and Service</b>	<b>138</b>



# 1 Foreword

## 1.1 Notes on the documentation

This description is intended exclusively for trained specialists in control and automation technology who are familiar with the applicable national standards.

The documentation and the following notes and explanations must be complied with when installing and commissioning the components.

The trained specialists must always use the current valid documentation.

The trained specialists must ensure that the application and use of the products described is in line with all safety requirements, including all relevant laws, regulations, guidelines, and standards.

### Disclaimer

The documentation has been compiled with care. The products described are, however, constantly under development.

We reserve the right to revise and change the documentation at any time and without notice.

Claims to modify products that have already been supplied may not be made on the basis of the data, diagrams, and descriptions in this documentation.

### Trademarks

Beckhoff®, ATRO®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, MX-System®, Safety over EtherCAT®, TC/BSD®, TwinCAT®, TwinCAT/BSD®, TwinSAFE®, XFC®, XPlanar®, and XTS® are registered and licensed trademarks of Beckhoff Automation GmbH.

If third parties make use of the designations or trademarks contained in this publication for their own purposes, this could infringe upon the rights of the owners of the said designations.



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

### Copyright

© Beckhoff Automation GmbH & Co. KG, Germany.

The distribution and reproduction of this document, as well as the use and communication of its contents without express authorization, are prohibited.

Offenders will be held liable for the payment of damages. All rights reserved in the event that a patent, utility model, or design are registered.

### Third-party trademarks

Trademarks of third parties may be used in this documentation. You can find the trademark notices here: <https://www.beckhoff.com/trademarks>.

## 1.2 For your safety

### Safety regulations

Read the following explanations for your safety.

Always observe and follow product-specific safety instructions, which you may find at the appropriate places in this document.

### Exclusion of liability

All the components are supplied in particular hardware and software configurations which are 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.



## Personnel qualification

This description is only intended for trained specialists in control, automation, and drive technology who are familiar with the applicable national standards.

## Signal words

The signal words used in the documentation are classified below. In order to prevent injury and damage to persons and property, read and follow the safety and warning notices.

### Personal injury warnings

 <b>DANGER</b>
Hazard with high risk of death or serious injury.
 <b>WARNING</b>
Hazard with medium risk of death or serious injury.
 <b>CAUTION</b>
There is a low-risk hazard that could result in medium or minor injury.

### Warning of damage to property or environment

<b>NOTICE</b>
The environment, equipment, or data may be damaged.

### Information on handling the product



This information includes, for example:  
recommendations for action, assistance or further information on the product.

## **1.3 Notes on information security**

The products of Beckhoff Automation GmbH & Co. KG (Beckhoff), insofar as they can be accessed online, are equipped with security functions that support the secure operation of plants, systems, machines and networks. Despite the security functions, the creation, implementation and constant updating of a holistic security concept for the operation are necessary to protect the respective plant, system, machine and networks against cyber threats. The products sold by Beckhoff are only part of the overall security concept. The customer is responsible for preventing unauthorized access by third parties to its equipment, systems, machines and networks. The latter should be connected to the corporate network or the Internet only if appropriate protective measures have been set up.

In addition, the recommendations from Beckhoff regarding appropriate protective measures should be observed. Further information regarding information security and industrial security can be found in our <https://www.beckhoff.com/secguide>.

Beckhoff products and solutions undergo continuous further development. This also applies to security functions. In light of this continuous further development, Beckhoff expressly recommends that the products are kept up to date at all times and that updates are installed for the products once they have been made available. Using outdated or unsupported product versions can increase the risk of cyber threats.

To stay informed about information security for Beckhoff products, subscribe to the RSS feed at <https://www.beckhoff.com/secinfo>.

## 1.4 Information on the security risk analysis

If you have installed this product, Beckhoff will provide you with the following information for a security risk analysis for your system.

### Engineering

#### **Workload: TwinCAT.Standard.XAE**

To be able to use the TE1200 function, you only need to install the standard workload mentioned above.

See also:

- Information on the security risk analysis for the TwinCAT standard XAE workload

### Runtime

No runtime components are installed or required.



## 2 Overview

With the integration of the static code analysis, a further programming tool is available in TwinCAT 3.1 that supports the PLC software development process. The tool is integrated in TwinCAT 3 PLC and can be seen as a supplement to the compiler.

### Function overview

Static Analysis implements more than 100 coding rules, some of which can be parameterized and combined to create individual rule sets. The rule sets defined in "PLCopen Coding Guidelines" are taken into account in some rules. For example, it can report if a pointer variable has not been checked for nonzero before dereferencing. As a result, the user's attention is drawn to possibly inadvertent and erroneous implementations, so that these program points can be optimized at an early stage.

You can also define a naming convention for each possible data type, which is then checked for compliance. In addition, over 20 metrics are available to analyze and characterize the underlying source code. When calculated regularly, the metrics can indicate negative trends and deviations from quality targets. The key figures therefore represent an indicator for assessing software quality. For example, the tabular output contains metrics for the number of statements or the proportion of comments.

The Static Analysis can be triggered manually or performed automatically during the code generation. TwinCAT outputs the result of the analysis, i.e. messages regarding deviations from the specifications and rules, in the message window. In the PLC project properties you can define the parameters to be checked in detail. When configuring the rules, you can also define whether a rule violation is to be output as an error or a warning. You can use pragma statements to exclude particular parts of the code from the check. For errors reported by Static Analysis based on precompile information, there is support in the ST Editor for immediate troubleshooting ([QuickFix/Precompile](#) [► 130]).

### Advantage

Static Analysis helps to write code that is easier to read and to identify potential sources of error during programming. On the one hand, this increases code quality and, on the other, saves a lot of time when developing applications and troubleshooting.

Failure to observe a coding rule generally indicates an implementation weakness; correcting it enables early troubleshooting or error avoidance. The automatic control of the user-specific naming conventions also ensures that the control programs can be developed in a standardized manner with regard to type and variable names. This gives different PLC projects implemented on the basis of the same naming conventions a uniform look and feel, which greatly improves the readability of programs. In addition, the metrics provide an indication of the software quality.

### Functionalities

An overview of the functionalities of "TwinCAT 3 PLC Static Analysis" is provided below:

- Static Analysis:
  - Function: The Static Analysis checks the source code of a project for deviations from certain coding rules and naming conventions, as well as for forbidden symbols. The result is output in the message window.
  - Configuration: The required coding rules, naming conventions and forbidden symbols can be configured in the [Rules](#) [► 16], [Naming conventions](#) [► 81] and [Forbidden symbols](#) [► 110] tabs of the PLC project properties.
- Standard metrics:
  - Function: Certain metrics are applied to your source code, which express the software properties in the form of key figures (e.g. the number of statements or the percentage of comments). They provide an indication of the software quality. The results are output in the **Standard Metrics** view.
  - Configuration: The required metrics can be configured in the [Metrics](#) [► 94] tab of the PLC project properties.

Alternatively, there is an option to use a license-free version of Static Analysis that provides a very much reduced range of functions. A detailed comparison of the functions of the license-free and the licensed version of Static Analysis can be found in chapter [Installation](#) [► 11].

Further information on installation, configuration and execution of the "Static Analysis" can be found on the following pages:

- [Installation](#) [► 11]
- [Configuration of the settings, rules, naming conventions, metrics and forbidden symbols](#) [► 14]
- [Command 'Run static analysis'](#) [► 111]
- [Command 'Run static analysis \[Check all objects\]'](#) [► 113]
- [Command 'View Standard Metrics'](#) [► 114]
- [Command 'View Standard Metrics \[Check all objects\]'](#) [► 116]
- [Pragmas and attributes](#) [► 121]
- [Examples](#) [► 135]
- [Automation Interface support](#) [► 132]



### Libraries

TwinCAT only analyzes the application code of the current PLC project; the referenced libraries are ignored!

If you have opened the library project, however, you can check the elements it contains with the help of the command [Command 'Run static analysis \[Check all objects\]'](#) [► 113].



### Punctual disablement of checks

[Pragmas and attributes](#) [► 121] can be used to disable checks for certain parts of the code.



### Static Analysis via the Automation Interface

Static Analysis can be operated via the Automation Interface (see [Automation Interface support](#) [► 132]).

## 3 Installation

The TE1200 | TwinCAT 3 PLC Static Analysis function is installed when the TwinCAT 3 development environment is installed. Accordingly, there is no separate TF1200 setup/package. The additional TE1200 engineering component only needs to be licensed. Information on a license-free test mode can be found under [Licensing](#) [► 13].

### **TwinCAT Package Manager: Installation (TwinCAT 3.1 Build 4026)**

Detailed instructions on installing products can be found in the chapter [Installing workloads](#) in the [TwinCAT 3.1 Build 4026 installation instructions](#).

Install the following workload to be able to use the product:

- TwinCAT.Standard.XAE

### **TwinCAT setup: Installation (TwinCAT 3.1 Build 4024 and earlier)**

Install the following setup in order to be able to use the product:

- TwinCAT 3.1 eXtended Automation Engineering (XAE) (full installation)

Detailed installation instructions can be found in the [Installation TwinCAT 3.1 Build 4024](#) chapter.

## 3.1 Functionality: Light vs. full

If you do not have an Engineering license for TE1200 you can use the license-free version of Static Analysis (Static Analysis Light), which has some restrictions (see table below). The free Light version enables you to familiarize yourself with the basic handling of the product, for example, based on a heavily reduced set of functions.

### **Static Analysis Light vs. Static Analysis Full**

An overview of the different features of the license-free and license-managed variants of Static Analysis is provided below.

Functional aspect	Static Analysis Light (without TE1200 license)	Static Analysis Full (with TE1200 license)
License required	No, usable free of charge	Yes, TE1200 license required
Save/export and load/import (rule) configuration	Not possible, coupled to PLC project properties	Possible (using the <b>Load/Save</b> buttons in the <a href="#">Settings</a> [► 14])
Execution is coupled to the compilation process	Yes, not configurable	Configurable (using the <b>Perform static analysis automatically</b> option in the <a href="#">Settings</a> [► 14]; Manual execution with the help of the command <a href="#">Command 'Run static analysis'</a> [► 111])
Checking for unused objects (e.g. within a library project)	Not possible	Possible (with the help of the command <a href="#">Command 'Run static analysis [Check all objects]'</a> [► 113])
Maximum number of reported errors	500 (not configurable) (Further information on the significance of 500 as the maximum number of errors can be found in the <a href="#">Settings</a> [► 14])	Configurable (using the setting <b>Maximum number of errors</b> in the <a href="#">Settings</a> [► 14])
Maximum number of reported warnings	Output of warnings not possible (see following line)	Configurable (using the setting <b>Maximum number of warnings</b> in the <a href="#">Settings</a> [► 14])
<a href="#">Rules: Activation options</a> [► 16]	<ul style="list-style-type: none"> <li>• Active and output as error</li> <li>• Inactive</li> </ul>	<ul style="list-style-type: none"> <li>• Active and output as error</li> <li>• Active and output as warning</li> <li>• Inactive</li> </ul>
<a href="#">Rules: scope</a> [► 17]	7 coding rules <ul style="list-style-type: none"> <li>• SA0033: Unused variables</li> <li>• SA0028: Overlapping memory areas</li> <li>• SA0006: Write access to multiple tasks</li> <li>• SA0004: Multiple writes access on output</li> <li>• SA0027: Multiple usage of name</li> <li>• SA0167: Report temporary FunctionBlock instances</li> <li>• SA0175: Suspicious operation on string</li> </ul>	More than 100 coding rules
<a href="#">Rules: Precompile wavy underline, QuickFix</a> [► 130]	Not available	Available
<a href="#">Naming conventions</a> [► 81]	Not available	Available
<a href="#">Metrics</a> [► 94]	Not available	Available
<a href="#">Forbidden symbols</a> [► 110]	Not available	Available

Pragmas and attributes [► 121] for temporary deactivation of rules	Yes, available in the Light scope: <ul style="list-style-type: none"> <li>• Pragma {analysis ...}</li> <li>• Attribute {attribute 'no-analysis'}</li> <li>• Attribute {attribute 'analysis' := '...'}</li> </ul>	Yes, available in full scope: <ul style="list-style-type: none"> <li>• Pragma {analysis ...}</li> <li>• Attribute {attribute 'no-analysis'}</li> <li>• Attribute {attribute 'analysis' := '...'}</li> <li>• Attribute {attribute 'naming' := '...'}</li> <li>• Attribute {attribute 'nameprefix' := '...'}</li> <li>• Attribute {attribute 'analysis:report-multiple-instance-calls'}</li> </ul>
--	--	--

## 3.2 System Requirements

### Engineering (XAE)

Technical data	Requirements
Operating system	<ul style="list-style-type: none"> <li>• Windows 10</li> <li>• Windows 11</li> </ul>
Target platform	<ul style="list-style-type: none"> <li>• x86</li> <li>• x64</li> </ul>
TwinCAT version	TwinCAT 3.1 Build 4022 or higher
Required TwinCAT license	TE1200 Engineering license

## 3.3 Licensing

For information on licensing the TE1200 engineering component, please read the documentation on Licensing.



### Close XAE before (de)activating the TE1200 license

Before activating or deactivating the TE1200 license via TwinCAT restart, please close all open development environments.

### Test mode

Please note that there is no 7-day trial license available for this product. If you do not have an Engineering license for TE1200 you can use the license-free version of Static Analysis (Static Analysis Light), which has some restrictions (see below). The free Light version enables you to familiarize yourself with the basic handling of the product, for example, based on a heavily reduced set of functions.

See also: [Functionality: Light vs. full \[► 11\]](#)

## 4 Configuration

After the [installation \[► 11\]](#) and licensing of "TE1200 | TwinCAT 3 PLC Static Analysis", the category **Static Analysis** in the properties of the PLC project is extended by the additional rules and configuration options.

In the project properties you will then find tabs for the basic configuration and for configuring the rules, conventions, metrics and forbidden symbols, which have to be taken into account in the code analysis.

The properties of a PLC project can be opened via the context menu of PLC project object or via the **Project** menu, if the focus is on a PLC project in the project tree.

The current settings or modifications are saved when you save the PLC project properties. The **Save** button, which can be found in the **Settings** tab, can be used to save the current Static Analysis configuration additionally in an external file. Such a configuration file can be loaded into the development environment via the **Load** button.

The following pages contain further information on the individual tabs of the **Static Analysis** project properties category.

- [Settings \[► 14\]](#)
- [Rules \[► 16\]](#)
- [Naming conventions \[► 81\]](#)
- [Naming conventions \(2\) \[► 90\]](#)
- [Metrics \[► 94\]](#)
- [Forbidden symbols \[► 110\]](#)

---

### ● Scope of the "Static Analysis" configuration

**I** The parameters you set in the category **Static Analysis** of the PLC project properties are referred to as **Solution options** and therefore affect not only the PLC project whose properties you currently edit. The configured settings, rules, naming conventions, metrics and forbidden symbols are applied to all PLC projects in the development environment.

---

### 4.1 Settings

The **Settings** tab can be used to configure whether the static code analysis is automatically performed when the code is generated. The current configuration of the **Static Analysis** can be saved in an external file, or a configuration can be loaded from an external file.

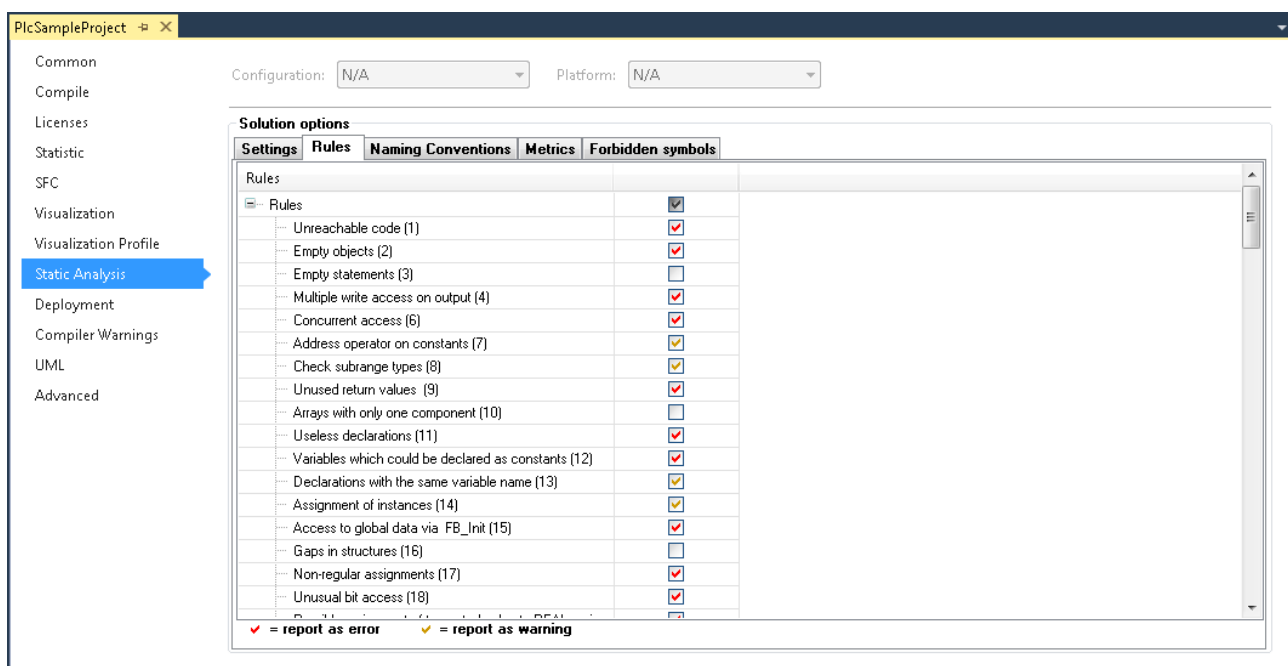
Perform Static Analysis automatically after compilation	If this option is enabled, TwinCAT performs the Static Analysis whenever code is generated without error (e.g. when the command <b>Build Project</b> is executed). The analysis can be started manually via the command <u>Command 'Run static analysis' [► 111]</u> , irrespective of the configuration of this option.
Load	This button opens the standard dialog for a locating of a file. Select the required configuration file *.csa for the Static Analysis, which may previously have been created via <b>Save</b> (see below). Since the Static Analysis properties are "solution options", the project properties for the Static Analysis, as described in the csa file, are applied to all PLC projects in the development environment.
Save	This button is used to save the current project properties for the Static Analysis in an xml file. The standard dialog for saving a file appears, and the file type is preset to "Static analysis files" (*.csa). Such a file can later be applied to the project via the <b>Load</b> button (see above).  Please note that the setting of the error limit "Maximum number of errors" is not saved in this file.
Maximum number of errors	<p>Preset: 500</p> <p>In this box you can enter the desired error limit, which is checked during the execution of the Static Analysis. If either the <b>error limit</b> or the <b>warning limit</b> (see below) is reached, <b>execution</b> of the Static Analysis is <b>canceled</b> and the <b>previous analysis result</b> is <b>output</b>.</p> <p><b>Performance vs. completeness:</b></p> <p>Please note: The more objects are checked by the Static Analysis, the longer the execution of the Static Analysis takes. And the more errors are entered in the output window, the longer the results output of the Static Analysis takes.</p> <p>In the assumed case that there are more than 500 Static Analysis errors in a PLC project, the following use cases arise.</p> <ul style="list-style-type: none"> <li>• Use of a small error limit (e.g. 500): You wish to gradually process the output errors by correcting the respective program code and executing the Static Analysis again to check the correction. In this case it wouldn't be necessary to check all the objects at once and to display all the errors at once. Instead, it is usually sufficient in this case to display a subset as the Static Analysis result, wherein the Static Analysis is executed with a good performance.</li> <li>• Use of a large error limit (e.g. 5000): You wish to output a total report from the Static Analysis in order to be able to roughly estimate the total work required for the correction of the program code. You can attain this goal by increasing the error limit. Please note that, depending on the project situation, the execution of the Static Analysis takes (much) longer the higher the error limit is set.</li> </ul> <p><b>Detailed explanation of the behavior:</b></p> <p>If there are more than 500 Static Analysis errors in a project, then configuring the error limit to 500 does not mean that the Static Analysis outputs exactly 500 errors. In fact, the following happens during the execution of the Static Analysis: Before checking a further POU, a check is performed to see whether the Static Analysis errors found so far already exceed the configured limit. If this is the case, the execution of the Static Analysis is aborted and the analysis result so far is output. If on the other hand the limit has not been reached, this POU is checked by the Static Analysis and the errors found in this POU are added to the analysis result. If this newly formed error total (e.g. 530) exceeds the configured error limit, the execution of the Static Analysis is aborted before the checking of the next POU and the errors found so far (e.g. 530) are output.</p>

Maximum number of warnings	Preset: 500  In this field, you can enter the desired warning limit, which is checked during the execution of Static Analysis. If either the <b>error limit</b> (see above) or the <b>warning limit</b> is reached, <b>execution</b> of the Static Analysis is <b>canceled</b> and the <b>previous analysis result</b> is <b>output</b> .  Further information on the use cases can be found in the description of the "Maximum number of errors" option (see above).
----------------------------	---

## 4.2 Rules

In the **Rules** tab you can configure the rules that are taken into account when the static analysis is performed [► 111]. The rules are displayed as a tree structure in the project properties. Some rules are arranged below organizational nodes.

The rules alert the user to possibly inadvertent and erroneous implementations so that these parts of the programme can be optimized at an early stage.



### Default settings

All rules are enabled by default, with the exception of SA0016, SA0024, SA0073, SA0101, SA0105-SA0107, SA0111-SA0125, SA0133, SA0134, SA0145, SA0147, SA0148, SA0150, SA0162-SA0167 and the "strict" IEC rules.

### Configuring the rules


Individual rules can be enabled or disabled via the checkbox for the respective row. Ticking the checkbox for a subnode affects all entries below this node. Ticking the checkbox for the top node affects all list entries. The entries below a node can be collapsed or expanded by clicking on the minus or plus sign to the left of the node name.

The number in brackets after each rule, for example "Unreachable code (1)", is the rule number that is issued if the rule is not observed.

The following three settings are available, which can be accessed by repeated clicking on the checkbox:

- ☐ : The rule is not checked.
- ☒ : A rule violation results in an error being reported in the message window.



-  : A rule violation results in a warning being reported in the message window.

### Syntax of rule violations in the message window

Each rule has a unique number (shown in parentheses after the rule in the rule configuration view). If a rule violation is detected during the static analysis, the number together with an error or warning description is issued in the message window, based on the following syntax. The abbreviation "SA" stands for "Static Analysis".

Syntax: **"SA<rule number>: <rule description>"**

Sample for rule number 33 (unused variables): "SA0033: Not used: variable 'bSample'"

### Temporary disabling of rules

Rules that are enabled in this dialog can be temporarily disabled in the project via a pragma. For further information please refer to [Pragmas and attributes](#) [► 121].

### Overview and description of the rules

An overview of the rules and a detailed description of the rules can be found at [Rules - overview and description](#) [► 17].

## 4.2.1 Rules - overview and description

### ● Check strict IEC rules

**i** The checks under the node "Check strict IEC rules" determine functionalities and data types that are allowed in TwinCAT, in extension of IEC61131-3.

### ● Checking concurrent/competing access

**i** The following rules exist on this topic:

[SA0006: Write access from multiple tasks](#) [► 24]

Determines variables that are written to by more than one task.

[SA0103: Concurrent access on not atomic data](#) [► 59]

Determines non-atomic variables (for example with data types STRING, WSTRING, ARRAY, STRUCT, FB instances, 64-bit data types) that are used in more than one task.

Please note that only direct access can be recognized. Indirect access operations, for example via pointer/reference, are not listed.

Please also refer to the documentation on the subject "[Multi-task data access synchronization in the PLC](#)", which contains several notes on the necessity and options for data access synchronization.

### Parameterizable rules

### ● Parameterizability

**i** Please note that some rules can be parameterized and, for example, limits can be set individually. You can configure the parameters to be taken into account in the respective check by double-clicking on the row of the corresponding rule in the rule configuration (PLC project properties > "Static Analysis" category > "Rules" tab). You can set the control parameters in the dialog that opens.

The following rules can be parameterized:

- [SA0100: Variables greater than <n> bytes](#) [► 57]
- [SA0101: Names with invalid length](#) [► 57]

- [SA0166: Maximum number of input/output/VAR IN OUT variables \[► 75\]](#)
- [SA0178: Cognitive complexity \[► 80\]](#)
- [SA0179: Coupling between objects \[► 80\]](#)

## Overview

- [SA0001: Unreachable code \[► 22\]](#)
- [SA0002: Empty objects \[► 22\]](#)
- [SA0003: Empty statements \[► 22\]](#)
- [SA0004: Multiple writes access on output \[► 23\]](#)
- [SA0006: Write access from several tasks \[► 24\]](#)
- [SA0007: Address operators on constants \[► 24\]](#)
- [SA0008: Check subrange types \[► 25\]](#)
- [SA0009: Unused return values \[► 25\]](#)
- [SA0010: Arrays with only one component \[► 26\]](#)
- [SA0011: Useless declarations with only one component \[► 26\]](#)
- [SA0012: Variables which could be declared as constants \[► 26\]](#)
- [SA0013: Declarations with the same variable name \[► 27\]](#)
- [SA0014: Assignments of instances \[► 27\]](#)
- [SA0015: Access to global data via FB init \[► 28\]](#)
- [SA0016: Gaps in structures \[► 28\]](#)
- [SA0017: Non-regular assignments to pointer variables \[► 29\]](#)
- [SA0018: Unusual bit access \[► 29\]](#)
- [SA0020: Possibly assignment of truncated value to REAL variable \[► 30\]](#)
- [SA0021: Transporting the address of a temporary variable \[► 30\]](#)
- [SA0022: \(Possibly\) non-rejected return values \[► 31\]](#)
- [SA0023: Complex return values \[► 31\]](#)
- [SA0024: Untyped literals \[► 31\]](#)
- [SA0025: Unqualified enumeration constants \[► 32\]](#)
- [SA0026: Possible truncated strings \[► 32\]](#)
- [SA0027: Multiple usage of name \[► 33\]](#)
- [SA0028: Overlapping memory areas \[► 33\]](#)
- [SA0029: Notation in code different to declaration \[► 34\]](#)
- **List unused objects**
  - [SA0031: Unused signatures \[► 34\]](#)

- [SA0032: Unused enumeration constants \[► 34\]](#)
- [SA0033: Unused variables \[► 35\]](#)
- [SA0035: Unused input variables \[► 35\]](#)
- [SA0036: Unused output variables \[► 35\]](#)
- [SA0034: Enumeration variables with incorrect assignment \[► 36\]](#)
- [SA0037: Write access to input variable \[► 36\]](#)
- [SA0038: Read access to output variable \[► 37\]](#)
- [SA0040: Possible division by zero \[► 37\]](#)
- [SA0041: Possibly loop-invariant code \[► 37\]](#)
- [SA0042: Usage of different access paths \[► 38\]](#)
- [SA0043: Use of a global variable in only one POU \[► 39\]](#)
- [SA0044: Declarations with reference to interface \[► 39\]](#)
- **Conversions**
  - [SA0019: Implicit pointer conversions \[► 40\]](#)
  - [SA0130: Implicit expanding conversions \[► 40\]](#)
  - [SA0133: Explicit narrowing conversions \[► 41\]](#)
  - [SA0134: Explicit signed/unsigned conversions \[► 41\]](#)
- **Usage of direct addresses**
  - [SA0005: Invalid addresses and data types \[► 42\]](#)
  - [SA0047: Access to direct addresses \[► 42\]](#)
  - [SA0048: AT declarations on direct addresses \[► 43\]](#)
- **Rules for operators**
  - [SA0051: Comparison operators on BOOL variables \[► 43\]](#)
  - [SA0052: Unusual shift operation \[► 43\]](#)
  - [SA0053: Too big bitwise shift \[► 44\]](#)
  - [SA0054: Comparisons of REAL/LREAL for equality/inequality \[► 44\]](#)
  - [SA0055: Unnecessary comparison operations of unsigned operands \[► 45\]](#)
  - [SA0056: Constant out of valid range \[► 45\]](#)
  - [SA0057: Possible loss of decimal points \[► 46\]](#)
  - [SA0058: Operations of enumeration variables \[► 46\]](#)
  - [SA0059: Comparison operations always returning TRUE or FALSE \[► 47\]](#)
  - [SA0060: Zero used as invalid operand \[► 48\]](#)
  - [SA0061: Unusual operation on pointer \[► 48\]](#)
  - [SA0062: Expression is constant \[► 49\]](#)

- [SA0063: Possibly not 16-bit-compatible operations \[► 49\]](#)
- [SA0064: Addition of pointer \[► 49\]](#)
- [SA0065: Incorrect pointer addition to base size \[► 50\]](#)
- [SA0066: Use of temporary results \[► 51\]](#)
- **Rules for statements**
  - **FOR statements**
    - [SA0072: Invalid uses of counter variable \[► 52\]](#)
    - [SA0073: Use of non-temporary counter variable \[► 52\]](#)
    - [SA0081: Upper border is not a constant \[► 52\]](#)
  - **CASE statements**
    - [SA0075: Missing ELSE \[► 53\]](#)
    - [SA0076: Missing enumeration constant \[► 54\]](#)
    - [SA0077: Type mismatches with CASE expression \[► 54\]](#)
    - [SA0078: Missing CASE branches \[► 55\]](#)
  - [SA0090: Return statement before end of function \[► 55\]](#)
- [SA0095: Assignments in conditions \[► 56\]](#)
- [SA0100: Variables greater than <n> bytes \[► 57\]](#)
- [SA0101: Names with invalid length \[► 57\]](#)
- [SA0102: Access to program/fb variables from the outside \[► 58\]](#)
- [SA0103: Concurrent access on not atomic data \[► 59\]](#)
- [SA0105: Multiple instance calls \[► 60\]](#)
- [SA0106: Virtual method calls in FB init \[► 60\]](#)
- [SA0107: Missing formal parameters \[► 62\]](#)
- **Check strict IEC rules**
  - [SA0111: Pointer variables \[► 62\]](#)
  - [SA0112: Reference variables \[► 62\]](#)
  - [SA0113: Variables with data type WSTRING \[► 62\]](#)
  - [SA0114: Variables with data type LTIME \[► 63\]](#)
  - [SA0115: Declarations with data type UNION \[► 63\]](#)
  - [SA0117: Variables with data type BIT \[► 63\]](#)
  - [SA0119: Object-oriented features \[► 64\]](#)
  - [SA0120: Program calls \[► 64\]](#)
  - [SA0121: Missing VAR\\_EXTERNAL declarations \[► 65\]](#)
  - [SA0122: Array index defined as expression \[► 65\]](#)

- SA0123: Usages of INI, ADR or BITADR [[▶ 66](#)]
- SA0147: Unusual shift operation - strict [[▶ 66](#)]
- SA0148: Unusual bit access - strict [[▶ 66](#)]

#### - Rules for initializations

- SA0118: Initializations not using constants [[▶ 67](#)]
- SA0124: Dereference access in initializations [[▶ 67](#)]
- SA0125: References in initializations [[▶ 68](#)]

- SA0140: Statements commented out [[▶ 71](#)]

#### - Possible use of uninitialized variables

- SA0039: Possible null pointer dereferences [[▶ 69](#)]
- SA0046: Possible use of not initialized interface [[▶ 70](#)]
- SA0145: Possible use of not initialized reference [[▶ 70](#)]
- SA0150: Violations of lower or upper limits of the metrics [[▶ 71](#)]
- SA0160: Recursive calls [[▶ 72](#)]
- SA0161: Unpacked structure in packed structure [[▶ 73](#)]
- SA0162: Missing comments [[▶ 74](#)]
- SA0163: Nested comments [[▶ 74](#)]
- SA0164: Multi-line comments [[▶ 75](#)]
- SA0166: Maximum number of input/output/VAR\_IN\_OUT variables [[▶ 75](#)]
- SA0167: Report temporary FunctionBlock instances [[▶ 76](#)]
- SA0168: Unnecessary assignments [[▶ 77](#)]
- SA0169: Ignored outputs [[▶ 77](#)]
- SA0170: Address of an output variable should not be used [[▶ 77](#)]
- SA0171: Enumerations should have the 'strict' attribute [[▶ 78](#)]
- SA0172: Possible attempt to access outside the array limits [[▶ 79](#)]
- SA0175: Suspicious operation on string [[▶ 79](#)]
- **Metrics**
  - SA0178: Cognitive complexity [[▶ 80](#)]
  - SA0179: Coupling between objects [[▶ 80](#)]
- SA0180: Index range does not cover the entire array [[▶ 80](#)]

#### Detailed description

**SA0001: Unreachable code**

Function	Determines code that is not executed, for example due to a RETURN or CONTINUE statement.
Reason	Unreachable code should be avoided in any case. The check often indicates the presence of test code, which should be removed.
Importance	High
PLCopen rule	CP2

**Sample 1 – RETURN:**

```

PROGRAM MAIN
VAR
    bReturnBeforeEnd : BOOL;
END_VAR

bReturnBeforeEnd := FALSE;
RETURN;
bReturnBeforeEnd := TRUE;          // => SA0001

```

**Sample 2 – CONTINUE:**

```

FUNCTION F_ContinueInLoop : BOOL
VAR
    nCounter : INT;
END_VAR

F_ContinueInLoop := FALSE;

FOR nCounter := INT#0 TO INT#5 BY INT#1 DO
    CONTINUE;
    F_ContinueInLoop := FALSE;    // => SA0001
END_FOR

```

**SA0002: Empty objects**

Function	Determines POU's, GVLs or data type declarations that do not contain code.
Reason	Empty objects should be avoided. They are often a sign that an object is not fully implemented.  Exception: In some cases, the body of a function block will not assigned code if it is only to be used via interfaces. In other cases, a method is only created because it is required by an interface, without scope for meaningful implementation of the method. In any case, a comment should be included in such a situation.
Importance	Medium

**SA0003: Empty statements**

Function	Determines lines of code containing a semicolon (;) but no statement.
Reason	An empty statement can be an indication of missing code.
Exception	Although there are meaningful uses for empty statements. For example, it may be useful to explicitly program all cases in a CASE statement, including cases in which no action is required. If such an empty CASE statement is commented, the statistical code analysis does not generate an error message.
Importance	Low

**Samples:**

```

;                                // => SA0003
(* comment *);                  // => SA0003
nVar;                           // => SA0003

```

The following sample generates the error "SA0003: Empty statement" for State 2.

```

CASE nVar OF
  1: DoSomething();
  2: ;
  3: DoSomethingElse();
END_CASE

```

The following sample does not generate an SA0003 error.

```

CASE nVar OF
  1: DoSomething();
  2: ; // nothing to do
  3: DoSomethingElse();
END_CASE

```

#### SA0004: Multiple write access on output

Function	Determines outputs that are written at more than one position.
Reason	The maintainability suffers if an output is written in various places in the code. It is then unclear which write access is actually affecting the process. It is good practice to perform the calculation of the output variables in auxiliary variables and to assign the calculated value to a point at the end of the cycle.
Exception	No error is issued if an output variable is written in different branches of IF or CASE statements.
Importance	High
PLCopen rule	CP12



This rule **cannot** be disabled via a pragma or attribute!  
 For more information on attributes, see [Pragmas and attributes](#) [► 121].

#### Sample:

Global variable list:

```

VAR_GLOBAL
  bVar      AT%X0.0 : BOOL;
  nSample   AT%QW5   : INT;
END_VAR

```

MAIN program:

```

PROGRAM MAIN
VAR
  nCondition      : INT;
END_VAR

IF nCondition < INT#0 THEN
  bVar := TRUE; // => SA0004
  nSample := INT#12; // => SA0004
END_IF

CASE nCondition OF
  INT#1:
    bVar := FALSE; // => SA0004
  INT#2:
    nSample := INT#11; // => SA0004
ELSE
  bVar := TRUE; // => SA0004
  nSample := INT#9; // => SA0004
END_CASE

```

**SA0006: Write access from several tasks**

Function	Determines variables with write access from more than one task.
Reason	A variable that is written in several tasks may change its value unexpectedly under certain circumstances. This can lead to confusing situations. String variables and, on some 32-bit systems, 64-bit integer variables also may even assume an inconsistent state if the variable is written in two tasks at the same time.
Exception	In certain cases it may be necessary for several tasks to write a variable. Make sure, for example through the use of semaphores, that the access does not lead to an inconsistent state.
Importance	High
PLCopen rule	CP10



See also rule [SA0103](#) [► 59].

**Call corresponds to write access**

Please note that calls are interpreted as write access. For example, calling a method for a function block instance is regarded as a write access to the function block instance. A more detailed analysis of accesses and calls is not possible, e.g. due to virtual calls (pointers, interface).

To deactivate rule SA0006 for a variable (e.g. for a function block instance), the following attribute can be inserted above the variable declaration: {attribute 'analysis' := '-6'}

**Examples:**

The two global variables nVar and bVar are written by two tasks.

Global variable list:

```
VAR_GLOBAL
  nVar  : INT;
  bVar  : BOOL;
END_VAR
```

Program MAIN\_Fast, called from the task PlcTaskFast:

```
nVar := nVar + 1;           // => SA0006
bVar := (nVar > 10);        // => SA0006
```

Program MAIN\_Slow, called from the task PlcTaskSlow:

```
nVar := nVar + 2;           // => SA0006
bVar := (nVar < -50);        // => SA0006
```

**SA0007: Address operators on constants**

Function	Determines locations at which the ADR operator is used for a constant.
Reason	A pointer to a constant variable cancels the CONSTANT property of the variable. The variable can be changed via the pointer without the compiler reporting this.
Exception	In rare cases, it may make sense for pointer to a constant to be passed to a function. If this option is used, measures must be implemented to ensure that the function does not change the value that was passed to it. In this case, use VAR_IN_OUT CONSTANT if possible.
Importance	High



If the option **Replace constants** is enabled in the compiler options of the PLC project properties, the address operator for scalar constants (Integer, BOOL, REAL) is not allowed and a compilation error is issued. (Constant strings, structures and arrays always have an address.)

**Sample:**



```

PROGRAM MAIN
VAR CONSTANT
    cValue : INT := INT#15;
END_VAR
VAR
    pValue : POINTER TO INT;
END_VAR
pValue := ADR(cValue); // => SA0007

```

### SA0008: Check subrange types

Function	Determines range exceedances of subrange types. Assigned literals are checked at an early stage by the compiler. If constants are assigned, the values must be within the defined range. If variables are assigned, the data types must be identical.
Reason	If subrange types are used, make sure that the function remains within the respective subrange. The compiler checks such subrange violations only for assignments of constants.
Importance	Low



The check is not performed for CFC objects, because the code structure does not allow this.

#### Sample:

```

PROGRAM MAIN
VAR
    nSub1 : INT (INT#1..INT#10);
    nSub2 : INT (INT#1..INT#1000);
    nVar : INT;
END_VAR
nSub1 := nSub2; // => SA0008
nSub1 := nVar; // => SA0008

```

### SA0009: Unused return values

Function	Determines function, method and property calls for which the return value is not used.
Reason	If a function or method returns a return value, the value should be evaluated. In many cases the return value contains information to indicate whether the function was executed successfully. If no evaluation is performed, it is subsequently not possible to determine whether the return value was overlooked or whether it is in fact not required.
Exception	If a return value is of no interest during a call, this should be documented and the assignment can be omitted. Error returns should never be ignored!
Importance	Medium
PLCopen rule	CP7/CP17

#### Sample:

Function F\_ReturnBOOL:

```

FUNCTION F_ReturnBOOL : BOOL
F_ReturnBOOL := TRUE;

```

MAIN program:

```

PROGRAM MAIN
VAR
    bVar : BOOL;
END_VAR
F_ReturnBOOL(); // => SA0009
bVar := F_ReturnBOOL();

```

**SA0010: Arrays with only one component**

Function	Determines arrays containing only a single component.
Reason	An array with a component can be replaced by a Base Type variable. Access to such a variable is much faster than access to a variable via an index.
Exception	The length of an array is often determined by a constant and used as a parameter for a program. The program can then work with arrays of different lengths and does not have to be changed if the length is only 1. Such a situation should be documented accordingly.
Importance	Low

**Samples:**

```

PROGRAM MAIN
VAR
    aEmpty1  : ARRAY [0..0] OF INT;           // => SA0010
    aEmpty2  : ARRAY [15..15] OF REAL;        // => SA0010
END_VAR

```

**SA0011: Useless declarations with only one component**

Function	Determines structures, unions, or enumerations with only one component.
Reason	No structures, unions or enumerations with only one component should be declared. Such declarations can be confusing for readers. A structure with only one element can be replaced by an alias type. An enumeration with an element can be replaced by a constant.
Importance	Low
PLCopen rule	CP22/CP24

**Sample 1 – Structure:**

```

TYPE ST_SingleStruct :           // => SA0011
STRUCT
    nPart : INT;
END_STRUCT
END_TYPE

```

**Sample 2 – Union:**

```

TYPE U_SingleUnion :           // => SA0011
UNION
    fVar : LREAL;
END_UNION
END_TYPE

```

**Sample 3 – Enumeration:**

```

TYPE E_SingleEnum :           // => SA0011
(
    eOnlyOne := 1
);
END_TYPE

```

**SA0012: Variables which could be declared as constants**

Function	Determines variables that are not subject to write access and therefore could not be declared as constants.
Reason	If a variable is only written at the declaration point and is otherwise only used in read mode, the static analysis assumes that the variable is to remain unchanged. Declaration as a constant means that the variable is checked for changes in the event of program modifications. Plus, declaration as a constant may lead to faster code.
Importance	Low

**Sample:**

```

PROGRAM MAIN
VAR
    nSample : INT := INT#17;
    nVar    : INT;
END_VAR

nVar := nVar + nSample;           // => SA0012

```

### SA0013: Declarations with the same variable name

Function	Determines variables with the same name as other variables (example: global and local variables with the same name), or the same name as functions, actions, methods or properties within the same access range.
Reason	Identical names can be confusing when the code is read and can lead to errors if the wrong object is accessed accidentally. We therefore recommend using naming conventions that avoid such situations.
Importance	Medium
PLCopen rule	N5/N9

#### Samples:

Global variable list GVL\_App:

```

VAR_GLOBAL
    nVar : INT;
END_VAR

```

MAIN program, containing a method with the name Sample:

```

PROGRAM MAIN
VAR
    bVar : BOOL;
    nVar : INT;           // => SA0013
    Sample : DWORD;       // => SA0013
END_VAR

.nVar := 100;           // Writing global variable "nVar"
nVar := 500;           // Writing local variable "nVar"

METHOD Sample
VAR_INPUT
...

```

### SA0014: Assignments of instances

Function	Determines assignments to function block instances. For instances with pointer or reference variables such assignments may be risky.
Reason	<p>This is a performance warning. When an instance is assigned to another instance, all elements and subelements are copied from one instance to the other. Pointers to data are also copied, but not their referenced data, so that the target instance and the source instance contain the same data after the assignment. Depending on the size of the instances, such an assignment may take a long time. If, for example, an instance is to be passed to a function for processing, it is much better to pass a pointer to the instance.</p> <p>A copy method can be useful for selectively copying values from one instance to another:</p> <pre>fb2.CopyFrom(fb1)</pre>
Importance	Medium

#### Sample:

```

PROGRAM MAIN
VAR
    fb1 : FB_Sample;
    fb2 : FB_Sample;
END_VAR

fb1();
fb2 := fb1;           // => SA0014

```

**SA0015: Access to global data via FB\_init**

Function	Determines access of a function block to global data via the FB_init method. The value of this variables depends on the order of the initializations!
Reason	Depending on the declaration location of the instance of a function block, a non-initialized variable may be accessed if the rule is violated.
Importance	High

**Sample:**

Global variable list GVL\_App:

```
VAR_GLOBAL
    nVar      : INT;
END_VAR
```

Function block FB\_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
    nLocal     : INT;
END_VAR
```

Method FB\_Sample.FB\_init:

```
METHOD FB_init : BOOL
VAR_INPUT
    bInitRetains : BOOL;          // if TRUE, the retain variables are initialized (warm start / cold
start)
    bInCopyCode  : BOOL;          // if TRUE, the instance afterwards gets moved into the copy code
(online change)
END_VAR

nLocal := 2*nVar;                // => SA0015
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
END_VAR
```

**SA0016: Gaps in structures**

Function	Determines gaps in structures or function blocks, caused by the alignment requirements of the currently selected target system. If possible, you should remove gaps by rearranging the structure elements or by filling them with dummy elements. If this is not possible, you can disable the rule for the affected structures using the <u>attribute {attribute 'analysis' := '...'} [► 123]</u> .
Reason	Due to different alignment requirements on different platforms, such structures may have a different layout in the memory. The code may behave differently, depending on the platform.
Importance	Low

**Samples:**

```
TYPE ST_UnpaddedStructure1 :
STRUCT
    bBOOL : BOOL;
    nINT   : INT;          // => SA0016
    nBYTE  : BYTE;
    nWORD  : WORD;
END_STRUCT
END_TYPE

TYPE ST_UnpaddedStructure2 :
STRUCT
    bBOOL : WORD;
    nINT   : INT;
```

```

    nBYTE   : BYTE;
    nWORD   : WORD;           // => SA0016
END_STRUCT
END_TYPE

```

### SA0017: Non-regular assignments to pointer variables

Function	Determines assignments to pointers, which are not an address (ADR operator, pointer variables) or constant 0.
Reason	If a pointer is assigned a value that is not a valid address, dereferencing the pointer leads to an "Access Violation Exception".
Importance	High

#### Sample:

```

PROGRAM MAIN
VAR
    nVar       : INT;
    pInt       : POINTER TO INT;
    nAddress   : XWORD;
END_VAR

nAddress := nAddress + 1;

pInt := ADR(nVar);           // no error
pInt := 0;                   // no error
pInt := nAddress;            // => SA0017

```

### SA0018: Unusual bit access

Function	Determines bit access to signed variables. However, the IEC 61131-3 standard only permits bit access to bit fields. See also strict rule <a href="#">SA0148</a> [► 66].
Reason	Signed data types should not be used as bit fields and vice versa. The IEC 61131-3 standard does not provide for such access. This rule must be observed if the code is to be portable.
Exception	Exception for flag enumerations: If an enumeration is declared as flag via the pragma attribute {attribute 'flags'}, the error SA0018 is not issued for bit access with OR, AND or NOT operations.
Importance	Medium

#### Samples:

```

PROGRAM MAIN
VAR
    nINT       : INT;
    nDINT      : DINT;
    nULINT     : ULINT;
    nSINT      : SINT;
    nUSINT     : USINT;
    nBYTE      : BYTE;
END_VAR

nINT.3 := TRUE;           // => SA0018
nDINT.4 := TRUE;          // => SA0018
nULINT.18 := FALSE;       // no error because this is an unsigned data type
nSINT.2 := FALSE;         // => SA0018
nUSINT.3 := TRUE;         // no error because this is an unsigned data type
nBYTE.5 := FALSE;         // no error because BYTE is a bit field

```

**SA0020: Possibly assignment of truncated value to REAL variable**

Function	Determines operations on integer variables, during which a truncated value may be assigned to a variable of data type REAL.
Reason	<p>The static code analysis returns an error when the result of an integer calculation is assigned to a REAL or LREAL variable. The programmer should be made aware of a possibly incorrect interpretation of such an assignment:</p> <pre>fLEAL := nDINT1 * nDINT2.</pre> <p>Since the value range of LREAL is greater than that of DINT, it could be assumed that the result of the calculation is always displayed in LREAL. But this is not the case. The processor calculates the result of the multiplication as an integer and then casts the result to LREAL. An overflow in the integer calculation would be lost. To avoid this problem, the calculation should be performed as a REAL operation:</p> <pre>fLREAL := TO_LREAL(nDINT1) * TO_LREAL(nDINT2)</pre>
Importance	High

**Sample:**

```
PROGRAM MAIN
VAR
    nVar1 : DWORD;
    nVar2 : DWORD;
    fVar   : REAL;
END_VAR

nVar1 := nVar1 + DWORD#1;
nVar2 := nVar2 + DWORD#2;
fVar  := nVar1 * nVar2;           // => SA0020
```

**SA0021: Transporting the address of a temporary variable**

Function	Determines assignments of addresses of temporary variables (variables on the stack) to non-temporary variables.
Reason	Local variables of a function or method are created on the stack and exist only while the function or method is processed. If a pointer points to such a variable after processing the method or function, then this pointer can be used to access undefined memory or an incorrect variable in another function. This situation must be avoided in any case.
Importance	High

**Sample:**

Method FB\_Sample.SampleMethod:

```
METHOD SampleMethod : XWORD
VAR
    fVar : LREAL;
END_VAR

SampleMethod := ADR(fVar);
```

Program MAIN:

```
PROGRAM MAIN
VAR
    nReturn : XWORD;
    fbSample : FB_Sample;
END_VAR

nReturn := fbSample.SampleMethod();           // => SA0021
```

**SA0022: (Possibly) unassigned return value**

Function	Determines all functions and methods containing an execution thread without assignment to the return value.
Reason	An unassigned return value in a function or method indicates missing code. Even if the return value always has a default value, it is useful to explicitly assign it again in any case, in order to avoid ambiguities.
Importance	Medium

**Sample:**

```

FUNCTION F_Sample : DWORD
VAR_INPUT
    nIn      : UINT;
END_VAR
VAR
    nTemp    : INT;
END_VAR
nIn := nIn + UINT#1;

IF (nIn > UINT#10) THEN
    nTemp    := 1;           // => SA0022
ELSE
    F_Sample := DWORD#100;
END_IF

```

**SA0023: Complex return values**

Function	Determines complex return values that cannot be returned with a simple register copy of the processor. These include structures and arrays as well as return values of the type STRING (irrespective of the size of storage space occupied).
Reason	This is a performance warning. If large values are returned as a result of a function, method, or property, the processor copies them repeatedly when the code is executed. This can lead to runtime problems and should be avoided if possible. Better performance is achieved if a structured value is passed to a function or method as VAR_IN_OUT and filled in the function or method.
Importance	Medium

**Sample:****Structure ST\_sample:**

```

TYPE ST_Sample :
STRUCT
    n1  : INT;
    n2  : BYTE;
END_STRUCT
END_TYPE

```

**Example of functions with return value:**

```

FUNCTION F_MyFunction1 : I_MyInterface           // no error
FUNCTION F_MyFunction2 : ST_Sample                // => SA0023
FUNCTION F_MyFunction3 : ARRAY[0..1] OF BOOL     // => SA0023

```

**SA0024: Untyped literals**

Function	Determines untyped literals that are part of an operation.
Reason	Untyped literals are automatically typed depending on their use. In some cases, such as <code>nDWORD := ROL(DWORD#1, i);</code> , this can lead to unexpected situations where it is better to achieve clear clarification by using a typed literal.
Importance	Low

**Sample:**

```

PROGRAM MAIN
VAR
  nINT    : INT := 10;           // no error as no part of operation
  nDINT   : DINT;
  nLINT   : LINT;
  fREAL   : REAL;
  fLREAL  : LREAL;
END_VAR

nINT := nINT + 34;               // => SA0024
nINT := nINT + INT#34;          // no error

nDINT := nDINT + 23;            // => SA0024
nDINT := nDINT + DINT#23;       // no error

nLINT := nLINT + 124;           // => SA0024
fREAL := fREAL + 1.1;          // => SA0024
fLREAL := fLREAL + 3.4;        // => SA0024

```

### SA0025: Unqualified enumeration constants

Function	Determines enumeration constants that are not used with a qualified name, i.e. without preceding enumeration name.
Reason	Qualified access makes the code more readable and easier to maintain. Without forcing qualified variable names, an additional enumeration could be inserted when the program is extended. This enumeration contains a constant with the same name as an existing enumeration (see the sample below: "eRed"). In this case there would be an ambiguous access in this piece of code.  We recommend using only enumerations that have the {attribute 'qualified-only'}.
Importance	Medium

#### Sample:

Enumeration E\_Color:

```

TYPE E_Color :
(
  eRed,
  eGreen,
  eBlue
);
END_TYPE

```

MAIN program:

```

PROGRAM MAIN
VAR
  eColor : E_Color;
END_VAR

eColor := E_Color.eGreen;       // no error
eColor := eGreen;               // => SA0025

```

### SA0026: Possible truncated strings

Function	Determines string assignments and initializations that do not use an adequate string length.
Reason	If strings of different lengths are assigned, a string may be truncated. The result is then not the expected one.
Importance	Medium

#### Samples:

```

PROGRAM MAIN
VAR
  sVar1 : STRING[10];
  sVar2 : STRING[6];
  sVar3 : STRING[6] := 'abcdefghi';           // => SA0026
END_VAR

```



```
sVar2 := sVar1; // => SA0026
```

### SA0027: Multiple usage of name

Function	<p>Determines multiple use of a variable name/identifier or object name (POU) within the scope of a project. The following cases are covered:</p> <ul style="list-style-type: none"> <li>• The name of an enumeration constant is identical to the name in another enumeration within the application or in an included library.</li> <li>• The name of a variable is identical to the name of another object in the application or in an included library.</li> <li>• The name of a variable is identical to the name of an enumeration constant in an enumeration in the application or in an included library.</li> <li>• The name of an object is identical to the name of another object in the application or in an included library.</li> </ul>
Reason	Identical names can be confusing when reading the code. They can lead to errors if the wrong object is accessed inadvertently. Therefore, define and follow naming conventions in order to avoid such situations.
Exception	Enumerations declared with the 'qualified_only' attribute are exempt from SA0027 checking because their elements can only be accessed in a qualified manner.
Importance	Medium

#### Sample:

The following sample generates error/warning SA0027, since the library Tc2\_Standard is referenced in the project, which provides the function block TON.

```
PROGRAM MAIN
VAR
    ton : INT; // => SA0027
END_VAR
```

### SA0028: Overlapping memory areas

Function	Determines the points due to which two or more variables occupy the same storage space.
Reason	If two variables occupy the same storage space, the code may behave very unexpectedly. This must be avoided in all cases. If the use of a value in different interpretations is unavoidable, for example once as DINT and once as REAL, you should define a UNION. Also, via a pointer you can access a value typed otherwise without converting the value.
Importance	High

#### Sample:

In the following sample both variables use byte 21, i.e. the memory areas of the variables overlap.

```
PROGRAM MAIN
VAR
    nVar1 AT%QB21 : INT; // => SA0028
    nVar2 AT%QD5 : DWORD; // => SA0028
END_VAR
```

**SA0029: Notation in code different to declaration**

Function	Determines the code positions (in the implementation) at which the notation of an identifier differs from the notation in its declaration.
Reason	The IEC 61131-3 standard defines identifiers as not case-sensitive. This means that a variable declared as "varx" can also be used as "VaRx" in the code. However, this can be confusing and misleading and should therefore be avoided.
Importance	Medium

**Samples:****Function F\_TEST:**

```
FUNCTION F_TEST : BOOL
...
```

**Program MAIN:**

```
PROGRAM MAIN
VAR
    nVar      : INT;
    bReturn   : BOOL;
END_VAR

nvar      := nVar + 1;           // => SA0029
bReturn := F_Test();           // => SA0029
```

**SA0031: Unused signatures**

Function	Determines programs, function blocks, functions, data types, interfaces, methods, properties, actions etc., which are not called within the compiled program code.
Reason	Unused objects result in unnecessary project bloat and confusion when the code is read.
Importance	low
PLCopen rule	CP2

**SA0032: Unused enumeration constants**

Function	Determines enumeration constants that are not used in the compiled program code.
Reason	Unused enumeration constants result in unnecessary enumeration definition bloat and confusion when the program is read.
Importance	Low
PLCopen rule	CP24

**Sample:****Enumeration E\_Sample:**

```
TYPE E_Sample :
(
    eNull,
    eOne,           // => SA0032
    eTwo
);
END_TYPE
```

**Program MAIN:**

```
PROGRAM MAIN
VAR
    eSample : E_Sample;
END_VAR

eSample := E_Sample.eNull;
eSample := E_Sample.eTwo;
```

**SA0033: Unused variables**

Function	Determines variables that are declared but not used within the compiled program code.
Reason	Unused variables make a program less easy to read and maintain. Unused variables occupy unnecessary memory space and take up unnecessary runtime during the initialization.
Importance	medium
PLCopen rule	CP22/CP24

**SA0035: Unused input variables**

Function	Determines input variables that are not assigned within the respective function block.
Reason	Unused variables make a program less easy to read and maintain. Unused variables occupy unnecessary memory space and take up unnecessary runtime during the initialization.
Importance	Medium
PLCopen rule	CP24

**Sample:**

Function block FB\_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    bIn1 : BOOL;
    bIn2 : BOOL;           // => SA0035
END_VAR
VAR_OUTPUT
    bOut1 : BOOL;
    bOut2 : BOOL;           // => SA0036
END_VAR
bOut1 := bIn1;
```

**SA0036: Unused output variables**

Function	Determines output variables that are not assigned within the respective function block.
Reason	Unused variables make a program less easy to read and maintain. Unused variables occupy unnecessary memory space and take up unnecessary runtime during the initialization.
Importance	Medium
PLCopen rule	CP24

**Sample:**

Function block FB\_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    bIn1 : BOOL;
    bIn2 : BOOL;           // => SA0035
END_VAR
VAR_OUTPUT
    bOut1 : BOOL;
    bOut2 : BOOL;           // => SA0036
END_VAR
bOut1 := bIn1;
```

**SA0034: Enumeration variables with incorrect assignment**

Function	Determines values that are assigned to an enumeration variable. Only defined enumeration constants may be assigned to an enumeration variable.
Reason	An enumeration type variable should only have the intended values, otherwise code that uses that variable may not work correctly. We recommend using only enumerations that have the {attribute 'strict'}. In this case the compiler checks the correct use of the enumeration components.
Importance	High

**Sample:****Enumeration E\_Color:**

```

TYPE E_Color :
(
    eRed    := 1,
    eBlue   := 2,
    eGreen  := 3
);
END_TYPE

```

**Program MAIN:**

```

PROGRAM MAIN
VAR
    eColor : E_Color;
END_VAR

eColor := E_Color.eRed;
eColor := eBlue;
eColor := 1; // => SA0034

```

**SA0037: Write access to input variable**

Function	Determines input variables (VAR_INPUT) that are subject to write access within the POU.
Reason	According to the IEC 61131-3 standard, an input variable may not be changed within a function block. Such access is also an error source and makes the code more difficult to maintain. It indicates that a variable is used as an input and simultaneously as an auxiliary variable. Such dual use should be avoided.
Importance	Medium

**Sample:****Function block FB\_Sample:**

```

FUNCTION_BLOCK FB_Sample
VAR_INPUT
    bIn  : BOOL := TRUE;
    nIn  : INT  := 100;
END_VAR
VAR_OUTPUT
    bOut : BOOL;
END_VAR

```

**Method FB\_Sample.SampleMethod:**

```

IF bIn THEN
    nIn := 500; // => SA0037
    bOut := TRUE;
END_IF

```

**SA0038: Read access to output variable**

Function	Determines output variables (VAR_OUTPUT) that are subject to read access within the POU.
Reason	The IEC-61131-3 standard prohibits reading an output within a function block. It indicates that the output is not only used as an output but also as a temporary variable for intermediate results. Such dual use should be avoided.
Importance	Low

**Sample:**

Function block FB\_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_OUTPUT
    bOut    : BOOL;
    nOut    : INT;
END_VAR
VAR
    bLocal  : BOOL;
    nLocal  : INT;
END_VAR
```

Method FB\_Sample.SampleMethod:

```
IF bOut THEN
    bLocal := (nOut > 100);    // => SA0038
    nLocal := nOut;           // => SA0038
    nLocal := 2*nOut;         // => SA0038
END_IF
```

**SA0040: Possible division by zero**

Function	Determines code positions at which division by zero may occur.
Reason	Division by 0 is not allowed. A variable that is used as a divisor should always be checked for 0 first. Otherwise, a "Divide by Zero" exception may occur at runtime.
Importance	High

**Sample:**

```
PROGRAM MAIN
VAR CONSTANT
    cSample : INT := 100;
END_VAR
VAR
    nQuotient1 : INT;
    nDividend1 : INT;
    nDivisor1  : INT;

    nQuotient2 : INT;
    nDividend2 : INT;
    nDivisor2  : INT;
END_VAR

nDivisor1 := cSample;
nQuotient1 := nDividend1/nDivisor1;           // no error
nQuotient2 := nDividend2/nDivisor2;           // => SA0040
```

**SA0041: Possibly loop-invariant code**

Function	Determines assignments in (FOR, WHILE, REPEAT) loops that calculate the same value for each loop pass. Such lines of code could be inserted outside the loop.
Reason	This is a performance warning. Code that is executed in a loop, but does the same thing in every loop pass, can be executed outside the loop.
Importance	Medium

**Sample:**

In the following sample SA0041 is output as error/warning, since the variables nTest1 and nTest2 are not used in the loop.

```

PROGRAM MAIN
VAR CONSTANT
    cMax      : INT := 3;
END_VAR
VAR
    nTest1    : INT := 5;
    nTest2    : INT := nTest1;
    nTest3    : INT;
    nTest4    : INT;
    nTest5    : INT;
    nTest6    : INT;
    nIndex    : INT;
    nCounter  : INT;
END_VAR

FOR nCounter := 1 TO 100 DO
    nTest3 := nTest1 + nTest2; // => SA0041
    nTest4 := nTest3 + nCounter; // no loop-invariant code, because nTest3 and nCounter are used
within loop
    nTest6 := nTest5;          // simple assignments are not regarded
END_FOR

FOR nIndex := 1 TO cMax-1 DO // => SA0041 for "cMax-1"
    nCounter := nCounter + 1;
END_FOR

```

**SA0042: Usage of different access paths**

Function	Determines the usage of different access paths for the same variable.
Reason	Different access to the same element reduces the readability and maintainability of a program. We recommend consistent use of {attribute 'qualified-only'} for libraries, global variable lists and enumerations. This forces fully qualified access.
Importance	Low

**Samples:**

In the following sample SA0042 is output as error/warning, because the global variable nGlobal is accessed directly and via the GVL namespace, and because the function CONCAT is accessed directly and via the library namespace.

**Global variables:**

```

VAR_GLOBAL
    nGlobal : INT;
END_VAR

```

**Program MAIN:**

```

PROGRAM MAIN
VAR
    sVar : STRING;
END_VAR

nGlobal := INT#2; // => SA0042
GVL.nGlobal := INT#3; // => SA0042

sVar := CONCAT('ab', 'cd'); // => SA0042
sVar := Tc2_Standard.CONCAT('ab', 'cd'); // => SA0042

```

**SA0043: Use of a global variable in only one POU**

Function	Determines global variables that are only used in one POU.
Reason	A global variable that is only used at one point should also be declared at this point.
Importance	Medium
PLCopen rule	CP26

**Sample:**

The global variable nGlobal1 is only used in the MAIN program.

Global variables:

```
VAR_GLOBAL
    nGlobal1 : INT;           // => SA0043
    nGlobal2 : INT;
END_VAR
```

Program SubProgram:

```
nGlobal2 := 123;
```

MAIN program:

```
SubProgram();
nGlobal1 := nGlobal2;
```

**SA0044: Declarations with reference to interface**

Function	Determines declarations with REFERENCE TO <interface> and declarations of VAR_IN_OUT variables with the type of an interface (realized implicitly via REFERENCE TO).
Reason	An interface type is always implicitly a reference to an instance of a function block that implements this interface. A reference to an interface is therefore a reference to a reference and can lead to unwanted behavior.
Importance	High

**Samples:**

I\_Sample is an interface defined in the project.

Function block FB\_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR_INPUT
    iInput : I_Sample;
END_VAR
VAR_OUTPUT
    iOutput : I_Sample;
END_VAR
VAR_IN_OUT
    iInOut1 : I_Sample;           // => SA0044
    {attribute 'analysis' := '-44'}
    iInOut2 : I_Sample;           // no error SA0044 because rule is deactivated via
attribute
END_VAR
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
    iSample : I_Sample;
    refItf : REFERENCE TO I_Sample; // => SA0044
END_VAR
```

**SA0019: Implicit pointer conversions**

Function	Determines implicitly generated pointer data type conversions.
Reason	<p>Pointers are not strictly typed in TwinCAT and can be assigned to each other as required. This is a commonly used option and therefore not reported by the compiler. However, it can also unintentionally lead to unexpected access. If a POINTER TO BYTE is assigned to a POINTER TO DWORD, it is possible that the last pointer will unintentionally overwrite memory. Therefore, always check this rule and suppress the message only in cases where you deliberately want to access a value with a different type.</p> <p>Implicit data type conversions are reported with a different message.</p>
Exception	BOOL ↔ BIT
Importance	High
PLCopen rule	CP25

**Samples:**

```

PROGRAM MAIN
VAR
    nInt    : INT;
    nByte   : BYTE;

    pInt    : POINTER TO INT;
    pByte   : POINTER TO BYTE;
END_VAR

pInt := ADR(nInt);
pByte := ADR(nByte);

pInt := ADR(nByte);           // => SA0019
pByte := ADR(nInt);          // => SA0019

pInt := pByte;                // => SA0019
pByte := pInt;                // => SA0019

```

**SA0130: Implicit expanding conversions**

Function	Determines code positions where conversions from smaller to larger data types are implicitly carried out during arithmetic operations.
Reason	<p>The compiler allows any assignment of different types if the range of the source type is fully within the range of the target type. However, the compiler will build a conversion into the code as late as possible. For an assignment of the following type:</p> <pre>nLINT := nDINT * nDINT;</pre> <p>the compiler performs the implicit conversion only after the multiplication:</p> <pre>nLINT := TO_LINT(nDINT * nDINT);</pre> <p>An overflow is therefore truncated. If you want to prevent this, you can have the conversion performed earlier for the elements:</p> <pre>nLINT := TO_LINT(nDINT) * TO_LINT(nDINT);</pre> <p>Therefore, it may be useful to report points where the compiler implements implicit conversions in order to check whether these are exactly what is intended. In addition, explicit conversions can serve to improve portability to other systems if they have more restrictive type checks.</p>
Exception	BOOL ↔ BIT
Importance	Low

**Samples:**

```

PROGRAM MAIN
VAR
    nDINT   : DINT;
    nLINT   : LINT;
    nUSINT  : USINT;
    nUINT   : UINT;

```



```

nUDINT : UDINT;
nULINT : ULINT;
nLWORD : LWORD;
fLREAL : LREAL;
nBYTE  : BYTE;
END_VAR

nDINT := UINT_TO_DINT(nUINT) * UINT_TO_DINT(nUINT); // no error
nDINT := nUINT * nUINT;                               // => SA0130

nLINT := nDINT * nDINT;                               // => SA0130
nULINT := nUSINT * nUSINT;                             // => SA0130
nLWORD := nUDINT * nUDINT;                             // => SA0130
fLREAL := nBYTE * nBYTE;                               // => SA0130

```

### SA0133: Explicit narrowing conversions

Function	Determines explicitly performed conversions from a larger to a smaller data type.
Reason	A large number of type conversions can mean that incorrect data types have been selected for variables. There are therefore programming guidelines that require an explicit justification for data type conversions.
Importance	Low

#### Samples:

```

PROGRAM MAIN
VAR
    nSINT : SINT;
    nDINT : DINT;
    nLINT : LINT;
    nBYTE : BYTE;
    nUINT : UINT;
    nDWORD : DWORD;
    nLWORD : LWORD;
    fREAL : REAL;
    fLREAL : LREAL;
END_VAR

nSINT := LINT_TO_SINT(nLINT); // => SA0133
nBYTE := DINT_TO_BYTE(nDINT); // => SA0133
nSINT := DWORD_TO_SINT(nDWORD); // => SA0133
nUINT := LREAL_TO_UINT(fLREAL); // => SA0133
fREAL := LWORD_TO_REAL(nLWORD); // => SA0133

```

### SA0134: Explicit signed/unsigned conversions

Function	Determines explicitly performed conversions from signed to unsigned data types or vice versa.
Reason	Excessive use of type conversions may mean that incorrect data types have been selected for variables. There are therefore programming guidelines that require an explicit justification for data type conversions.
Importance	Low

#### Samples:

```

PROGRAM MAIN
VAR
    nBYTE : BYTE;
    nUDINT : UDINT;
    nULINT : ULINT;
    nWORD : WORD;
    nLWORD : LWORD;
    nSINT : SINT;
    nINT : INT;
    nDINT : DINT;
    nLINT : LINT;
END_VAR

```

```

nLINT  := ULINT_TO_LINT(nULINT); // => SA0134
nUDINT := DINT_TO_UDINT(nDINT);  // => SA0134
nSINT  := BYTE_TO_SINT(nBYTE);   // => SA0134
nWORD  := INT_TO_WORD(nINT);     // => SA0134
nLWORD := SINT_TO_LWORD(nSINT);  // => SA0134

```

### SA0005: Invalid addresses and data types

Function	<p>Determines invalid address and data type specifications.</p> <p>The following size prefixes are valid for addresses. Deviations from this lead to an invalid address specification.</p> <ul style="list-style-type: none"> <li>• X for BOOL</li> <li>• B for 1-byte data types</li> <li>• W for 2-byte data types</li> <li>• D for 4-byte data types</li> </ul>
Reason	Variables that lie on direct addresses should, if possible, be associated with an address that corresponds to their data type width. It can be confusing for the reader of the code if, for example, a DWORD is placed on a BYTE address.
Importance	Low



If the recommended placeholders %I\* or %Q\* are used, TwinCAT automatically performs flexible and optimized addressing.

#### Samples:

```

PROGRAM MAIN
VAR
    nOK    AT%QW0    : INT;
    bOK    AT%QX5.0  : BOOL;

    nNOK   AT%QD10   : INT;          // => SA0005
    bNOK   AT%QB15   : BOOL;        // => SA0005
END_VAR

```

### SA0047: Access to direct addresses

Function	Determines direct address access operations in the implementation code.
Reason	Symbolic programming is always preferred: A variable has a name that can also have a meaning. An address does not provide an indication of what it is used for.
Importance	High
PLCopen rule	N1/CP1

#### Samples:

```

PROGRAM MAIN
VAR
    bBOOL   : BOOL;
    nBYTE   : BYTE;
    nWORD   : WORD;
    nDWORD  : DWORD;
END_VAR

bBOOL := %IX0.0;          // => SA0047
%QX0.0 := bBOOL;         // => SA0047
%QW2   := nWORD;         // => SA0047
%QD4   := nDWORD;        // => SA0047
%MX0.1 := bBOOL;         // => SA0047
%MB1   := nBYTE;         // => SA0047
%MD4   := nDWORD;        // => SA0047

```

**SA0048: AT declarations on direct addresses**

Function	Determines AT declarations on direct addresses.
Reason	The use of direct addresses in the code is an error source and leads to poorer readability and maintainability of the code.  We therefore recommend using the placeholders %I* or %Q*, for which TwinCAT automatically carries out flexible and optimized addressing.
Importance	High
PLCopen rule	N1/CP1

**Samples:**

```

PROGRAM MAIN
VAR
    b1    AT%IX0.0 : BOOL;        // => SA0048
    b2    AT%I*    : BOOL;        // no error
END_VAR

```

**SA0051: Comparison operations on BOOL variables**

Function	Determines comparison operations on variables of type BOOL.
Reason	TwinCAT allows such comparisons, but they are rather unusual and can be confusing. The IEC-61131-3 standard does not provide for these comparisons, so you should avoid them.
Importance	Medium

**Sample:**

```

PROGRAM MAIN
VAR
    b1      : BOOL;
    b2      : BOOL;
    bResult : BOOL;
END_VAR

bResult := (b1 > b2);           // => SA0051
bResult := NOT b1 AND b2;
bResult := b1 XOR b2;

```

**SA0052: Unusual shift operation**

Function	Determines shift operation (bit shift) on signed variables. However, the IEC 61131-3 standard only permits shift operations to bit fields. See also strict rule <a href="#">SA0147</a> [► 66].
Reason	TwinCAT allows shift operations on signed data types. However, such operations are unusual and can be confusing. The IEC-61131-3 standard does not provide for such operations, so you should avoid them.
Exception	Shift operation on bit array data types (byte, DWORD, LWORD, WORD) do not result in a SA0052 error.
Importance	Medium

**Samples:**

```

PROGRAM MAIN
VAR
    nINT   : INT;
    nDINT  : DINT;
    nULINT : ULINT;
    nSINT  : SINT;
    nUSINT : USINT;
    nLINT  : LINT;

    nDWORD : DWORD;
    nBYTE  : BYTE;
END_VAR

```

```

nINT    := SHL(nINT, BYTE#2);    // => SA0052
nDINT   := SHR(nDINT, BYTE#4);   // => SA0052
nULINT  := ROL(nULINT, BYTE#1);  // no error because this is an unsigned data type
nSINT   := ROL(nSINT, BYTE#2);   // => SA0052
nUSINT  := ROR(nUSINT, BYTE#3);  // no error because this is an unsigned data type
nLINT   := ROR(nLINT, BYTE#2);   // => SA0052

nDWORD  := SHL(nDWORD, BYTE#3);  // no error because DWORD is a bit field data type
nBYTE   := SHR(nBYTE, BYTE#1);   // no error because BYTE is a bit field data type

```

### SA0053: Too big bitwise shift

Function	Determines whether the data type width was exceeded in bitwise shift of operands.
Reason	If a shift operation exceeds the data type width, a constant 0 is generated. If a rotation shift exceeds the data type width, it is difficult to read and the rotation value should therefore be shortened.
Importance	High

#### Samples:

```

PROGRAM MAIN
VAR
    nBYTE   : BYTE;
    nWORD   : WORD;
    nDWORD  : DWORD;
    nLWORD  : LWORD;
END_VAR

nBYTE := SHR(nBYTE, BYTE#8);    // => SA0053
nWORD := SHL(nWORD, BYTE#45);   // => SA0053
nDWORD := ROR(nDWORD, BYTE#78); // => SA0053
nLWORD := ROL(nLWORD, BYTE#111); // => SA0053

nBYTE := SHR(nBYTE, BYTE#7);    // no error
nWORD := SHL(nWORD, BYTE#15);   // no error

```

### SA0054: Comparisons of REAL/LREAL for equality/inequality

Function	Determines where the comparison operators = (equality) and <> (inequality) compare operands of type REAL or LREAL.
Reason	<p>REAL/LREAL values are implemented as floating point numbers in accordance with the IEEE 754 standard. This standard implies that certain seemingly simple decimal numbers cannot be represented exactly. As a result, the same decimal number may have different LREAL representations.</p> <p>Sample:</p> <pre> fLREAL_11 := 1.1; fLREAL_33 := 3.3; fLREAL_a := fLREAL_11 + fLREAL_11; fLREAL_b := fLREAL_33 - fLREAL_11; bTest := fLREAL_a = fLREAL_b; </pre> <p>bTest will return FALSE in this case, even if the variables fLREAL_a and fLREAL_b both return the monitoring value "2.2". This is not a compiler error, but a property of the floating point units of all common processors. You can avoid this by specifying a minimum value by which the values may differ:</p> <pre> bTest := ABS(fLREAL_a - fLREAL_b) &lt; 0.1; </pre>
Exception	A comparison with 0.0 is not reported by this analysis. For 0 there is an exact representation in the IEEE 754 standard and therefore the comparison normally works as expected. For better performance, it therefore makes sense to allow a direct comparison here.
Importance	High
PLCopen rule	CP54

**Samples:**

```

PROGRAM MAIN
VAR
    fREAL1 : REAL;
    fREAL2 : REAL;
    fLREAL1 : LREAL;
    fLREAL2 : LREAL;
    bResult : BOOL;
END_VAR

bResult := (fREAL1 = fREAL1); // => SA0054
bResult := (fREAL1 = fREAL2); // => SA0054
bResult := (fREAL1 <> fREAL2); // => SA0054
bResult := (fLREAL1 = fLREAL1); // => SA0054
bResult := (fLREAL1 = fLREAL2); // => SA0054
bResult := (fLREAL2 <> fLREAL2); // => SA0054

bResult := (fREAL1 > fREAL2); // no error
bResult := (fLREAL1 < fLREAL2); // no error

```

**SA0055: Unnecessary comparison operations of unsigned operands**

Function	Determines unnecessary comparisons with unsigned operands. An unsigned data type is never less than zero.
Reason	A comparison revealed by this check provides a constant result and indicates an error in the code.
Importance	High

**Samples:**

```

PROGRAM MAIN
VAR
    nBYTE : BYTE;
    nWORD : WORD;
    nDWORD : DWORD;
    nLWORD : LWORD;
    nUSINT : USINT;
    nUINT : UINT;
    nUDINT : UDINT;
    nULINT : ULINT;

    nSINT : SINT;
    nINT : INT;
    nDINT : DINT;
    nLINT : LINT;

    bResult : BOOL;
END_VAR

bResult := (nBYTE >= BYTE#0); // => SA0055
bResult := (nWORD < WORD#0); // => SA0055
bResult := (nDWORD >= DWORD#0); // => SA0055
bResult := (nLWORD < LWORD#0); // => SA0055
bResult := (nUSINT >= USINT#0); // => SA0055
bResult := (nUINT < UINT#0); // => SA0055
bResult := (nUDINT >= UDINT#0); // => SA0055
bResult := (nULINT < ULINT#0); // => SA0055

bResult := (nSINT < SINT#0); // no error
bResult := (nINT < INT#0); // no error
bResult := (nDINT < DINT#0); // no error
bResult := (nLINT < LINT#0); // no error

```

**SA0056: Constant out of valid range**

Function	Determines literals (constants) outside the valid operator range.
Reason	The message is output in cases where a variable is compared with a constant that lies outside the value range of this variable. The comparison then returns a constant TRUE or FALSE. This indicates a programming error.
Importance	High

**Samples:**

```

PROGRAM MAIN
VAR
    nBYTE   : BYTE;
    nWORD   : WORD;
    nDWORD  : DWORD;
    nUSINT  : USINT;
    nUINT   : UINT;
    nUDINT  : UDINT;

    bResult : BOOL;
END_VAR

bResult := nBYTE >= 355;           // => SA0056
bResult := nWORD > UDINT#70000;   // => SA0056
bResult := nDWORD >= ULINT#4294967300; // => SA0056
bResult := nUSINT >= ULINT#355;   // => SA0056
bResult := nUINT >= UDINT#70000;  // => SA0056
bResult := nUDINT > ULINT#4294967300; // => SA0056

```

**SA0057: Possible loss of decimal points**

Function	Determines statements with possible loss of decimal points.
Reason	<p>A piece of code of the following type:</p> <pre>nDINT := 1; fREAL := TO_REAL(nDINT / DINT#2);</pre> <p>can lead to misinterpretation. This line of code can lead to the assumption that the division would be performed as a REAL operation and the result in this case would be REAL#0.5. However, this is not the case, i.e. the operation is performed as an integer operation, the result is cast to REAL, and fREAL is assigned the value REAL#0. To avoid this, you should use a cast to ensure that the operation is performed as a REAL operation:</p> <pre>fREAL := TO_REAL(nDINT) / REAL#2;</pre>
Importance	Medium

**Samples:**

```

PROGRAM MAIN
VAR
    fREAL : REAL;
    nDINT : DINT;
    nLINT : LINT;
END_VAR

nDINT := nDINT + DINT#11;
fREAL := DINT_TO_REAL(nDINT / DINT#3); // => SA0057
fREAL := DINT_TO_REAL(nDINT) / 3.0;    // no error
fREAL := DINT_TO_REAL(nDINT) / REAL#3.0; // no error

nLINT := nLINT + LINT#13;
fREAL := LINT_TO_REAL(nLINT / LINT#7); // => SA0057
fREAL := LINT_TO_REAL(nLINT) / 7.0;    // no error
fREAL := LINT_TO_REAL(nLINT) / REAL#7.0; // no error

```

**SA0058: Operations of enumeration variables**

Function	Determines operations on variables of type enumeration. Assignments are permitted.
Reason	Enumerations should not be used as normal integer values. Alternatively, an alias data type can be defined or a subrange type can be used.
Exception	<p>If an enumeration is marked with the attribute {attribute 'strict'}, the compiler reports such an operation.</p> <p>If an enumeration is declared as a flag via the pragma attribute {attribute 'flags'}, no SA0058 error is issued for operations with AND, OR, NOT, XOR.</p>
Importance	Medium

**Sample 1:****Enumeration E\_Color:**

```

TYPE E_Color :
(
    eRed    := 1,
    eBlue   := 2,
    eGreen  := 3
);
END_TYPE

```

**MAIN program:**

```

PROGRAM MAIN
VAR
    nVar    : INT;
    eColor  : E_Color;
END_VAR

eColor := E_Color.eGreen;           // no error
eColor := E_Color.eGreen + 1;       // => SA0058
nVar    := E_Color.eBlue / 2;       // => SA0058
nVar    := E_Color.eGreen + E_Color.eRed; // => SA0058

```

**Sample 2:****Enumeration E\_State with attribute 'flags':**

```

{attribute 'flags'}
TYPE E_State :
(
    eUnknown := 16#00000001,
    eStopped  := 16#00000002,
    eRunning  := 16#00000004
) DWORD;
END_TYPE

```

**MAIN program:**

```

PROGRAM MAIN
VAR
    nFlags : DWORD;
    nState : DWORD;
END_VAR

IF (nFlags AND E_State.eUnknown) <> DWORD#0 THEN // no error
    nState := nState AND E_State.eUnknown;         // no error

ELSIF (nFlags OR E_State.eStopped) <> DWORD#0 THEN // no error
    nState := nState OR E_State.eRunning;          // no error
END_IF

```

**SA0059: Comparison operations always returning TRUE or FALSE**

Function	Determines comparisons with literals whose result is always TRUE or FALSE and which can already be evaluated during compilation.
Reason	An operation that consistently returns TRUE or FALSE is an indication of a programming error.
Importance	High

**Samples:**

```

PROGRAM MAIN
VAR
    nBYTE   : BYTE;
    nWORD   : WORD;
    nDWORD  : DWORD;
    nLWORD  : LWORD;
    nUSINT  : USINT;
    nUINT   : UINT;
    nUDINT  : UDINT;
    nULINT  : ULINT;
    nSINT   : SINT;
    nINT    : INT;
    nDINT   : DINT;

```

```

    nLINT   : LINT;
    bResult : BOOL;
END_VAR

bResult := nBYTE   <= 255;           // => SA0059
bResult := nBYTE   <= BYTE#255;      // => SA0059
bResult := nWORD   <= WORD#65535;    // => SA0059
bResult := nDWORD  <= DWORD#4294967295; // => SA0059
bResult := nLWORD  <= LWORD#18446744073709551615; // => SA0059
bResult := nUSINT  <= USINT#255;      // => SA0059
bResult := nUINT   <= UINT#65535;     // => SA0059
bResult := nUDINT  <= UDINT#4294967295; // => SA0059
bResult := nULINT  <= ULINT#18446744073709551615; // => SA0059
bResult := nSINT   >= -128;           // => SA0059
bResult := nSINT   >= SINT#-128;      // => SA0059
bResult := nINT    >= INT#-32768;     // => SA0059
bResult := nDINT   >= DINT#-2147483648; // => SA0059
bResult := nLINT   >= LINT#-9223372036854775808; // => SA0059

```

### SA0060: Zero used as invalid operand

Function	Determines operations in which an operand with value 0 results in an invalid or meaningless operation.
Reason	Such an expression may indicate a programming error. In any case, it causes unnecessary runtime.
Importance	Medium

#### Samples:

```

PROGRAM MAIN
VAR
    nBYTE   : BYTE;
    nWORD   : WORD;
    nDWORD  : DWORD;
    nLWORD  : LWORD;
END_VAR

nBYTE := nBYTE + 0;           // => SA0060
nWORD := nWORD - WORD#0;      // => SA0060
nDWORD := nDWORD * DWORD#0;   // => SA0060
nLWORD := nLWORD / 0;         // Compile error: Division by zero

```

### SA0061: Unusual operation on pointer

Function	Determines operations on variables of type POINTER TO, which are not = (equality), <> (inequality), + (addition) or ADR.
Reason	Pointer arithmetic is generally permitted in TwinCAT and can be used in a meaningful way. The addition of a pointer with an integer value is therefore classified as a common operation on pointers. This makes it possible to process an array of variable length using a pointer. All other (unusual) operations with pointers are reported with SA0061.
Importance	High
PLCopen rule	E2/E3

#### Samples:

```

PROGRAM MAIN
VAR
    pINT : POINTER TO INT;
    nVar : INT;
END_VAR

pINT := ADR(nVar);           // no error
pINT := pINT * DWORD#5;      // => SA0061
pINT := pINT / DWORD#2;      // => SA0061
pINT := pINT MOD DWORD#3;    // => SA0061
pINT := pINT + DWORD#1;      // no error
pINT := pINT - DWORD#1;      // => SA0061

```



**SA0062: Expression is constant**

Function	Determines constant expressions that always return TRUE or FALSE, regardless of the values of any variables used.
Reason	Such an expression is obviously unnecessary and may indicate an error. In any case, the expression unnecessarily affects the readability and possibly also the runtime.
Importance	Medium

**Samples:**

```

PROGRAM MAIN
VAR
    bVar1  : BOOL;
    bVar2  : BOOL;
    nVar   : INT;
END_VAR

IF MAX(nVar,1) >= 1 THEN           // => SA0062
;
END_IF

bVar1 := bVar1 AND NOT TRUE;       // => SA0062
bVar2 := bVar1 OR TRUE;           // => SA0062
bVar2 := bVar1 OR NOT FALSE;      // => SA0062
bVar2 := bVar1 AND FALSE;         // => SA0062

IF (bVar1 = FALSE) THEN           // => SA0062
;
END_IF

IF NOT bVar1 THEN                 // => no error
;
END_IF

nVar := 0;
IF nVar <> 0 THEN                  // => SA0062
;
END_IF

```

**SA0063: Possibly not 16-bit-compatible operations**

Function	Determines 16-bit operations with intermediate results. Background: On 16-bit systems, 32-bit temporary results can be truncated.
Reason	This message is intended to protect against problems in the very rare case when code is written that is intended to run on both a 16-bit processor and a 32-bit processor.
Importance	Low

**Sample:**

(nVar+10) can exceed 16 bits.

```

PROGRAM MAIN
VAR
    nVar  : INT;
END_VAR

nVar := (nVar + 10) / 2;           // => SA0063

```

**SA0064: Addition of pointer**

Function	Determines all pointer additions.
Reason	In TwinCAT, pointer arithmetic is generally permitted and can be used sensibly. However, this also represents a source of error. Therefore, there are programming rules that prohibit pointer arithmetic. Such a requirement can be verified with this test.
Importance	Medium

**Samples:**

```

PROGRAM MAIN
VAR
    aTest : ARRAY[0..10] OF INT;
    pINT  : POINTER TO INT;
    nIdx  : INT;
END_VAR

pINT := ADR(aTest[0]);
pINT^ := 0;
pINT := ADR(aTest) + SIZEOF(INT);           // => SA0064
pINT^ := 1;
pINT := ADR(aTest) + 6;                     // => SA0064
pINT := ADR(aTest[10]);

FOR nIdx := 0 TO 10 DO
    pINT^ := nIdx;
    pINT := pINT + 2;                       // => SA0064
END_FOR

```

### SA0065: Incorrect pointer addition to base size

Function	Determines pointer additions in which the value to be added does not match the basic data size of the pointer. Only literals of the base data size and multiples thereof can be added without error.
Reason	<p>In TwinCAT (in contrast to C and C++), when a pointer with an integer value is added, only this integer value is added as the number of bytes, not the integer value multiplied by the base size.</p> <pre>pINT := ADR(array_of_int[0]); pINT := pINT + 2 ; // in TwinCAT zeigt pINT anschließend auf array_of_int[1]</pre> <p>This code would work differently in C:</p> <pre>short* pShort pShort = &amp;(array_of_short[0]) pShort = pShort + 2; // in C zeigt pShort anschließend auf array_of_short[2]</pre> <p>In TwinCAT, a multiple of the basic size of the pointer should therefore always be added to a pointer. Otherwise, the pointer may point to a not aligned memory, which (depending on the processor) can lead to an alignment exception on access.</p>
Importance	High

### Samples:

```

PROGRAM MAIN
VAR
    pUDINT : POINTER TO UDINT;
    nVar   : UDINT;
    pREAL  : POINTER TO REAL;
    fVar   : REAL;
END_VAR

pUDINT := ADR(nVar) + 4;
pUDINT := ADR(nVar) + (2 + 2);
pUDINT := ADR(nVar) + SIZEOF(UDINT);
pUDINT := ADR(nVar) + 3;           // => SA0065
pUDINT := ADR(nVar) + 2*SIZEOF(UDINT); // => SA0065
pUDINT := ADR(nVar) + (3 + 2);    // => SA0065

pREAL := ADR(fVar);
pREAL := pREAL + 4;
pREAL := pREAL + (2 + 2);
pREAL := pREAL + SIZEOF(REAL);
pREAL := pREAL + 1;           // => SA0065
pREAL := pREAL + 2;           // => SA0065
pREAL := pREAL + 3;           // => SA0065
pREAL := pREAL + (SIZEOF(REAL) - 1); // => SA0065
pREAL := pREAL + (1 + 4);      // => SA0065

```

**SA0066: Use of temporary results**

Function	Determines applications of intermediate results in statements with a data type that is smaller than the register size. In this case, the implicit cast may lead to undesirable results.
Reason	<p>For performance reasons, TwinCAT carries out operations across the register width of the processor. Intermediate results are not truncated. This can lead to misinterpretations, as in the following case:</p> <pre>usintTest := 0; bError := usintTest - 1 &lt;&gt; 255;</pre> <p>In TwinCAT, bError is TRUE in this case, because the operation usintTest - 1 is typically executed as a 32-bit operation and the result is not cast to the size of bytes. In the register the value 16#ffffff is then displayed and this is not equal to 255. To avoid this, you have to explicitly cast the intermediate result:</p> <pre>bError := TO_USINT(usintTest - 1) &lt;&gt; 255;</pre>
Importance	Low



If this message is enabled, a large number of rather unproblematic situations in the code will be reported. Although a problem can only arise if the operation produces an overflow or underflow in the data type, the Static Analysis cannot differentiate between the individual situations.

If you include an explicit typecast in all reported situations, the code will be much slower and less readable!

**Sample:**

```
PROGRAM MAIN
VAR
    nBYTE    : BYTE;
    nDINT    : DINT;
    nLINT    : LINT;
    bResult  : BOOL;
END_VAR

//
=====
=
// type size smaller than register size
// use of temporary result + implicit casting => SA0066
bResult := ((nBYTE - 1) <> 255); // => SA0066

// correcting this code by explicit cast so that the type size is equal to or bigger than register
size
bResult := ((BYTE_TO_LINT(nBYTE) - 1) <> 255); // no error
bResult := ((BYTE_TO_LINT(nBYTE) - LINT#1) <> LINT#255); // no error

//
=====
=
// result depends on solution platform
bResult := ((nDINT - 1) <> 255); // no error on x86 solution platform
// => SA0066 on x64 solution platform

// correcting this code by explicit cast so that the type size is equal to or bigger than register
size
bResult := ((DINT_TO_LINT(nDINT) - LINT#1) <> LINT#255); // no error

//
=====
=
// type size equal to or bigger than register size
// use of temporary result and no implicit casting => no error
bResult := ((nLINT - 1) <> 255); // no error

//
=====
```

**SA0072: Invalid uses of counter variable**

Function	Determines write access operations to a counter variable within a FOR loop.
Reason	Manipulating the counter variable in a FOR loop can easily lead to an infinite loop. To prevent the execution of the loop for certain values of the counter variables, use CONTINUE or simply IF.
Importance	High
PLCopen rule	L12

**Sample:**

```

PROGRAM MAIN
VAR_TEMP
  nIndex : INT;
END_VAR
VAR
  aSample : ARRAY[1..10] OF INT;
  nLocal : INT;
END_VAR
FOR nIndex := 1 TO 10 BY 1 DO
  aSample[nIndex] := nIndex;           // no error
  nLocal := nIndex;                   // no error

  nIndex := nIndex - 1;                // => SA0072
  nIndex := nIndex + 1;                // => SA0072
  nIndex := nLocal;                   // => SA0072
END_FOR

```

**SA0073: Use of non-temporary counter variable**

Function	Determines the use of non-temporary variables in FOR loops.
Reason	This is a performance warning. A counter variable is always initialized each time a programming block is called. You can create such a variable as a temporary variable (VAR_TEMP). This may result in faster access, and the variable does not occupy permanent storage space.
Importance	Medium
PLCopen rule	CP21/L13

**Sample:**

```

PROGRAM MAIN
VAR
  nIndex : INT;
  nSum : INT;
END_VAR
FOR nIndex := 1 TO 10 BY 1 DO // => SA0073
  nSum := nSum + nIndex;
END_FOR

```

**SA0081: Upper border is not a constant**

Function	Determines FOR statements in which the upper limit is not defined with a constant value.
Reason	If the upper limit of a loop is a variable value, it is no longer possible to see how often a loop is executed. This can lead to serious problems at runtime, in the worst case to an infinite loop.
Importance	High

**Samples:**

```

PROGRAM MAIN
VAR_CONSTANT
  cMax : INT := 10;
END_VAR
VAR
  nIndex : INT;

```

```

    nVar    : INT;
    nMax1   : INT := 10;
    nMax2   : INT := 10;
END_VAR

FOR nIndex := 0 TO 10 DO           // no error
    nVar := nIndex;
END_FOR

FOR nIndex := 0 TO cMax DO         // no error
    nVar := nIndex;
END_FOR

FOR nIndex := 0 TO nMax1 DO        // => SA0081
    nVar := nIndex;
END_FOR

FOR nIndex := 0 TO nMax2 DO        // => SA0081
    nVar := nIndex;

    IF nVar = 10 THEN
        nMax2 := 50;
    END_IF
END_FOR

```

### SA0075: Missing ELSE

Function	Determines CASE statements without ELSE branch.
Reason	Defensive programming requires the presence of an ELSE in every CASE statement. If no action is required in the ELSE case, you should indicate this with a comment. The reader of the code is then aware that the case was not simply overlooked.
Exception	A missing ELSE branch is not reported as missing if an enumeration declared with the 'strict' attribute is used in the CASE statement, and if all enumeration constants are listed in that CASE statement.
Importance	Low
PLCopen rule	L17

#### Sample:

```

{attribute 'qualified_only'}
{attribute 'strict'}
{attribute 'to_string'}
TYPE E_Sample :
(
    eNull,
    eOne,
    eTwo
);
END_TYPE

PROGRAM MAIN
VAR
    eSample : E_Sample;
    nVar    : INT;
END_VAR

CASE eSample OF
    E_Sample.eNull: nVar := 0;
    E_Sample.eOne:  nVar := 1;
    E_Sample.eTwo:  nVar := 2;
END_CASE

CASE eSample OF                // => SA0075
    E_Sample.eNull: nVar := 0;
    E_Sample.eTwo:  nVar := 2;
END_CASE

```

**SA0076: Missing enumeration constant**

Function	Determines whether each enumeration constant is used as a condition in CASE statements and queried in a CASE branch.
Reason	Defensive programming requires the processing of all possible values of an enumeration. If no action is required for a particular enumeration value, you should indicate this explicitly with a comment. This makes it clear that the value was not simply overlooked.
Importance	Low

**Sample:**

In the following sample the enumeration value eYellow is not treated as a CASE branch.

Enumeration E\_Color:

```
TYPE E_Color :
(
    eRed,
    eGreen,
    eBlue,
    eYellow
);
END_TYPE
```

**MAIN program:**

```
PROGRAM MAIN
VAR
    eColor : E_Color;
    bVar   : BOOL;
END_VAR

eColor := E_Color.eYellow;

CASE eColor OF
    E_Color.eRed:
        bVar := FALSE;

    E_Color.eGreen,
    E_Color.eBlue:
        bVar := TRUE;

ELSE
    bVar := NOT bVar;
END_CASE
```

**SA0077: Type mismatches with CASE expression**

Function	Determines code positions where the data type of a condition does not match that of the CASE branch.
Reason	If the data types between the CASE variable and the CASE case do not match, this could indicate an error.
Importance	Low

**Sample:**

Enumeration E\_Sample:

```
TYPE E_Sample :
(
    eNull,
    eOne,
    eTwo
) DWORD;
END_TYPE
```

Program MAIN:

```

PROGRAM MAIN
VAR
    nDINT    : DINT;
    bVar     : BOOL;
END_VAR

nDINT := nDINT + DINT#1;

CASE nDINT OF
    DINT#1:
        bVar := FALSE;

    E_Sample.eTwo,           // => SA0077
    DINT#3:
        bVar := TRUE;

ELSE
    bVar := NOT bVar;
END_CASE

```

### SA0078: Missing CASE branches

Function	Determines CASE statements without cases, i.e. with only a single ELSE statement.
Reason	A CASE statement without cases wastes execution time and is difficult to read.
Importance	Medium

#### Sample:

```

PROGRAM MAIN
VAR
    nVar     : DINT;
    bVar     : BOOL;
END_VAR

nVar := nVar + INT#1;

CASE nVar OF
                                // => SA0078
ELSE
    bVar := NOT bVar;
END_CASE

```

### SA0090: POU's should have a only one exit point

Function	Detects code positions where the RETURN statement is not the last statement in a function, method, property or program. It also recognizes places where there is a RETURN within an IF branch.
Reason	A RETURN in the code leads to poorer maintainability, testability and readability of the code. A RETURN in the code is easily overlooked. You must insert code, which should be executed in any case when a function exits, before each RETURN. This is often overlooked.
Importance	Medium
PLCopen rule	CP14

#### Sample:

```

FUNCTION F_TestFunction : DINT
VAR_INPUT
    bTest    : BOOL;
END_VAR

IF bTest THEN
    RETURN;           // => SA0090
END_IF

F_TestFunction := 99;

```

**SA0095: Assignments in conditions**

Function	Determines assignments in conditions of IF, CASE, WHILE or REPEAT constructs.
Reason	An assignment (:=) and a comparison (=) can easily be confused. An assignment in a condition can therefore easily be unintentional and is therefore reported. This can also confuse readers of the code.
Importance	High

**Samples:**

```

PROGRAM MAIN
VAR
    bTest    : BOOL;
    bResult  : BOOL;
    bValue   : BOOL;

    b1       : BOOL;
    n1       : INT;
    n2       : INT;

    nCond1   : INT := INT#1;
    nCond2   : INT := INT#2;
    bCond    : BOOL := FALSE;
    nVar     : INT;
    eSample  : E_Sample;
END_VAR

// IF constructs
IF (bTest := TRUE) THEN                // => SA0095
    DoSomething();
END_IF

IF (bResult := F_Sample(bInput := bValue)) THEN // => SA0095
    DoSomething();
END_IF

b1 := ((n1 := n2) = 99);                // => SA0095

IF INT_TO_BOOL(nCond1 := nCond2) THEN // => SA0095
    DoSomething();
ELSIF (nCond1 := 11) = 11 THEN          // => SA0095
    DoSomething();
END_IF

IF bCond := TRUE THEN                  // => SA0095
    DoSomething();
END_IF

IF (bCond := FALSE) OR (nCond1 := nCond2) = 12 THEN // => SA0095
    DoSomething();
END_IF

IF (nVar := nVar + 1) = 120 THEN        // => SA0095
    DoSomething();
END_IF

// CASE construct
CASE (eSample := E_Sample.eMember0) OF // => SA0095
    E_Sample.eMember0:
        DoSomething();

    E_Sample.eMember1:
        DoSomething();
END_CASE

// WHILE construct
WHILE (bCond = TRUE) OR (nCond1 := nCond2) = 12 DO // => SA0095
    DoSomething();
END_WHILE

// REPEAT construct
REPEAT
    DoSomething();
UNTIL
    (bCond = TRUE) OR ((nCond1 := nCond2) = 12) // => SA0095
END_REPEAT

```



**SA0100: Variables greater than <n> bytes**

Function	<p>Determines variables that use more than n bytes; n is defined by the current configuration.</p> <p>You can configure the parameter that is taken into account in the check by double-clicking on the row for rule 100 in the rule configuration (PLC Project Properties &gt; category "Static Analysis" &gt; "Rules" tab &gt; Rule 100). You can make the following settings in the dialog that appears:</p> <ul style="list-style-type: none"> <li>• Upper limit in bytes (default value: 1024)</li> </ul>
Reason	Some programming guidelines specify a maximum size for a single variable. This function facilitates a corresponding check.
Importance	Low

**Sample:**

In the following sample the variable aSample is greater than 1024 bytes.

```
PROGRAM MAIN
VAR
    aSample : ARRAY [0..1024] OF BYTE;           // => SA0100
END_VAR
```

**SA0101: Names with invalid length**

Function	<p>Determines names with invalid length. The object names must have a defined length.</p> <p>You can configure the parameters that are taken into account in the check by double-clicking on the row for rule 101 in the rule configuration (PLC Project Properties &gt; category "Static Analysis" &gt; "Rules" tab &gt; Rule 101). You can make the following settings in the dialog that appears:</p> <ul style="list-style-type: none"> <li>• Minimum number of characters (default value: 5)</li> <li>• Maximum number of characters (default value: 30)</li> <li>• Exceptions</li> </ul>
Reason	Some programming guidelines specify a minimum length for variable names. Compliance can be verified with this analysis.
Importance	Low
PLCopen rule	N6

**Samples:**

Rule 101 is configured with the following parameters:

- Minimum number of characters: 5
- Maximum number of characters: 30
- Exceptions: MAIN, i

**Program PRG1:**

```
PROGRAM PRG1                               // => SA0101
VAR
END_VAR
```

**Program MAIN:**

```
PROGRAM MAIN                               // no error due to configured exceptions
VAR
    i      : INT;                           // no error due to configured exceptions
    b      : BOOL;                           // => SA0101
    nVar1  : INT;
END_VAR

PRG1 ();
```

**SA0102: Access to program/fb variables from the outside**

Function	Determines read accesses from outside to local variables of programs or function blocks.
Reason	TwinCAT determines external write access operations to local variables of programs or function blocks as compilation errors. Since read access operations to local variables are not intercepted by the compiler and this violates the basic principle of data encapsulation (concealing of data) and contravenes the IEC 61131-3 standard, this rule can be used to determine read access to local variables.
Importance	Medium

**Samples:****Function block FB\_Base:**

```
FUNCTION_BLOCK FB_Base
VAR
    nLocal : INT;
END_VAR
```

**Method FB\_Base.SampleMethod:**

```
METHOD SampleMethod : INT
VAR_INPUT
END_VAR

nLocal := nLocal + 1;
```

**Function block FB\_Sub:**

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
```

**Method FB\_Sub.SampleMethod:**

```
METHOD SampleMethod : INT
VAR_INPUT
END_VAR

nLocal := nLocal + 5;
```

**Program PRG\_1:**

```
PROGRAM PRG_1
VAR
    bLocal : BOOL;
END_VAR

bLocal := NOT bLocal;
```

**MAIN program:**

```
PROGRAM MAIN
VAR
    bRead      : BOOL;
    nReadBase  : INT;
    nReadSub   : INT;
    fbBase     : FB_Base;
    fbSub      : FB_Sub;
END_VAR

bRead  := PRG_1.bLocal;    // => SA0102
nReadBase := fbBase.nLocal; // => SA0102
nReadSub  := fbSub.nLocal;  // => SA0102
```

**SA0103: Concurrent access on not atomic data**

Function	Determines non-atomic variables (for example with data types STRING, WSTRING, ARRAY, STRUCT, FB instances, 64-bit data types) that are used in more than one task.
Reason	If no synchronization occurs during access, inconsistent values may be read when reading in one task and writing in another task at the same time.
Exception	This rule does not apply in the following cases: <ul style="list-style-type: none"> <li>• If the target system has an FPU (floating point unit), the access of several tasks to LREAL variables is not determined and reported.</li> <li>• If the target system is a 64-bit processor or "TwinCAT RT (x64)" is selected as the solution platform, the rule does not apply for 64-bit data types.</li> </ul>
Importance	Medium



See also rule [SA0006](#) [► 24].

**Samples:****Structure ST\_sample:**

```

TYPE ST_Sample :
STRUCT
    bMember : BOOL;
    nTest   : INT;
END_STRUCT
END_TYPE

```

**Function block FB\_Sample:**

```

FUNCTION_BLOCK FB_Sample
VAR_INPUT
    fInput : LREAL;
END_VAR

```

**GVL:**

```

{attribute 'qualified_only'}
VAR_GLOBAL
    fTest : LREAL;           // => no error SA0103: Since the target system has a FPU, SA0103
does not apply.
    nTest : LINT;           // => error reporting depends on the solution platform:
                           // - SA0103 error if solution platform is set to "TwinCAT
RT(x86) "
                           // - no error SA0103 if solution platform is set to "TwinCAT
(x64) "
    sTest : STRING;         // => SA0103
    wsTest : WSTRING;      // => SA0103
    aTest : ARRAY[0..2] OF INT; // => SA0103
    aTest2 : ARRAY[0..2] OF INT; // => SA0103
    fbTest : FB_Sample;    // => SA0103
    stTest : ST_Sample;    // => SA0103
END_VAR

```

**Program MAIN1, called by task PlcTask1:**

```

PROGRAM MAIN1
VAR
END_VAR

GVL.fTest := 5.0;
GVL.nTest := 123;
GVL.sTest := 'sample text';
GVL.wsTest := "sample text";
GVL.aTest := GVL.aTest2;
GVL.fbTest.fInput := 3;
GVL.stTest.nTest := GVL.stTest.nTest + 1;

```

**Program MAIN2, called by task PlcTask2:**

```

PROGRAM MAIN2
VAR
    fLocal : LREAL;
    nLocal : LINT;

```

```

sLocal   : STRING;
wsLocal  : WSTRING;
aLocal   : ARRAY[0..2] OF INT;
aLocal2  : ARRAY[0..2] OF INT;
fLocal2  : LREAL;
nLocal2  : INT;
END_VAR

fLocal   := GVL.fTest + 1.5;
nLocal   := GVL.nTest + 10;
sLocal   := GVL.sTest;
wsLocal  := GVL.wsTest;
aLocal   := GVL.aTest;
aLocal2  := GVL.aTest2;
fLocal2  := GVL.fbTest.fInput;
nLocal2  := GVL.stTest.nTest;

```

### SA0105: Multiple instance calls

Function	Determines and reports instances of function blocks that are called more than once. To ensure that an error message for a repeatedly called function block instance is generated, the <u>attribute</u> {attribute 'analysis:report-multiple-instance-call'} [► 125] must be added in the declaration part of the function block.
Reason	Some function blocks are designed such that they can only be called once in a cycle. This test checks whether a call is made at several points.
Importance	Low
PLCopen rule	CP16/CP20

#### Sample:

In the following sample the Static Analysis will issue an error for fb2, since the instance is called more than once, and the function block is declared with the required attribute.

Function block FB\_Test1 without attribute:

```
FUNCTION_BLOCK FB_Test1
```

Function block FB\_Test2 with attribute:

```
{attribute 'analysis:report-multiple-instance-calls'}
FUNCTION_BLOCK FB_Test2
```

MAIN program:

```

PROGRAM MAIN
VAR
    fb1   : FB_Test1;
    fb2   : FB_Test2;
END_VAR

fb1();
fb1();
fb2();           // => SA0105
fb2();           // => SA0105

```

### SA0106: Virtual method calls in FB\_init

Function	Determines method calls in the method FB_init of a basic function block, which are overwritten by a function block derived from the basic FB.
Reason	In such cases it may happen that the variables in overwritten methods are not initialized in the base FB.
Importance	High

#### Sample:

- Function block FB\_Base has the methods FB\_init and MyInIt. FB\_init calls MyInIt for initialization.
- Function block FB\_Sub is derived from FB\_Base.

- FB\_Sub.MyInit overwrites or extends FB\_Base.MyInit.
- MAIN instantiates FB\_Sub. During this process it uses the instance variable nSub before it was initialized, due to the call sequence during the initialization.

#### Function block FB\_Base:

```
FUNCTION_BLOCK FB_Base
VAR
    nBase          : DINT;
END_VAR
```

#### Method FB\_Base.FB\_init:

```
METHOD FB_init : BOOL
VAR_INPUT
    bInitRetains : BOOL;
    bInCopyCode  : BOOL;
END_VAR
VAR
    nLocal       : DINT;
END_VAR

nLocal := MyInit();           // => SA0106
```

#### Method FB\_Base.MyInit:

```
METHOD MyInit : DINT

nBase := 123;                 // access to member of FB_Base
MyInit := nBase;
```

#### Function block FB\_Sub:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
VAR
    nSub          : DINT;
END_VAR
```

#### Method FB\_Sub.MyInit:

```
METHOD MyInit : DINT

nSub := 456;                 // access to member of FB_Sub
SUPER^.MyInit();            // call of base implementation
MyInit := nSub;
```

#### MAIN program:

```
PROGRAM MAIN
VAR
    fbBase      : FB_Base;
    fbSub       : FB_Sub;
END_VAR
```

The instance MAIN.fbBase has the following variable values after the initialization:

- nBase is 123

The instance MAIN.fbSub has the following variable values after the initialization:

- nBase is 123
- nSub is 0

The variable MAIN.fbSub.nSub is 0 after the initialization, because the following call sequence is used during the initialization of fbSub:

- Initialization of the basic function block:
  - implicit initialization
  - explicit initialization: FB\_Base.FB\_init
  - FB\_Base.FB\_init calls FB\_Sub.MyInit → **SA0106**
  - FB\_Sub.MyInit calls FB\_Base.MyInit (via SUPER pointer)
- Initialization of the derived function block:
  - implicit initialization

**SA0107: Missing formal parameters**

Function	Determines where formal parameters are missing.
Reason	Code becomes more readable if the formal parameters are specified when it is called.
Importance	Low

**Sample:**

Function F\_Sample:

```
FUNCTION F_Sample : BOOL
VAR_INPUT
    bIn1 : BOOL;
    bIn2 : BOOL;
END_VAR
F_Sample := bIn1 AND bIn2;
```

MAIN program:

```
PROGRAM MAIN
VAR
    bReturn : BOOL;
END_VAR
bReturn := F_Sample(TRUE, FALSE);           // => SA0107
bReturn := F_Sample(TRUE, bIn2 := FALSE);   // => SA0107
bReturn := F_Sample(bIn1 := TRUE, bIn2 := FALSE); // no error
```

**SA0111: Pointer variables**

Function	Determines variables of type POINTER TO.
Reason	The IEC 61131-3 standard does not allow pointers.
Importance	Low

**Sample:**

```
PROGRAM MAIN
VAR
    pINT : POINTER TO INT;    // => SA0111
END_VAR
```

**SA0112: Reference variables**

Function	Determines variables of type REFERENCE TO.
Reason	The IEC 61131-3 standard does not allow references.
Importance	Low

**Sample:**

```
PROGRAM MAIN
VAR
    refInt : REFERENCE TO INT;    // => SA0112
END_VAR
```

**SA0113: Variables with data type WSTRING**

Function	Determines variables of type WSTRING.
Reason	Not all systems support WSTRING. The code becomes easier to port if WSTRING is not used.
Importance	Low

**Sample:**

```
PROGRAM MAIN
VAR
  wsVar : WSTRING;           // => SA0113
END_VAR
```

**SA0114: Variables with data type LTIME**

Function	Determines variables of type LTIME.
Reason	Not all systems support LTIME. The code becomes more portable if LTIME is not used.
Importance	Low

**Sample:**

```
PROGRAM MAIN
VAR
  tVar : LTIME;              // => SA0114
END_VAR

// no error SA0114 for the following code line:
tVar := tVar + LTIME#1000D15H23M12S34MS2US44NS;
```

**SA0115: Declarations with data type UNION**

Function	Determines declarations of a UNION data type and declarations of variables of the type of a UNION.
Reason	The IEC-61131-3 standard has no provision for unions. The code becomes easier to port if there are no unions.
Importance	Low

**Samples:****Union U\_Sample:**

```
TYPE U_Sample :              // => SA0115
UNION
  fVar : LREAL;
  nVar : LINT;
END_UNION
END_TYPE
```

**MAIN program:**

```
PROGRAM MAIN
VAR
  uSample : U_Sample;        // => SA0115
END_VAR
```

**SA0117: Variables with data type BIT**

Function	Determines declarations of variables of type BIT (possible within structure and function block definitions).
Reason	The IEC-61131-3 has no provision for data type BIT. The code becomes easier to port if BIT is not used.
Importance	Low

**Samples:****Structure ST\_sample:**

```
TYPE ST_Sample :
STRUCT
  bBIT : BIT;                // => SA0117
```

```

    bBOOL : BOOL;
END_STRUCT
END_TYPE

```

#### Function block FB\_Sample:

```

FUNCTION_BLOCK FB_Sample
VAR
    bBIT : BIT;                // => SA0117
    bBOOL : BOOL;
END_VAR

```

### SA0119: Object-oriented features

Function	Determines the use of object-oriented features such as: <ul style="list-style-type: none"> <li>Function block declarations with EXTENDS or IMPLEMENTS</li> <li>Property and interface declarations</li> <li>Use of the THIS or SUPER pointer</li> </ul>
Reason	Not all systems support object-oriented programming. The code becomes easier to port if object orientation is not used.
Importance	Low

#### Samples:

##### Interface I\_Sample:

```
INTERFACE I_Sample // => SA0119
```

##### Function block FB\_Base:

```
FUNCTION_BLOCK FB_Base IMPLEMENTS I_Sample // => SA0119
```

##### Function block FB\_Sub:

```
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base // => SA0119
```

##### Method FB\_Sub.SampleMethod:

```
METHOD SampleMethod : BOOL // no error
```

##### Get function of the property FB\_Sub.SampleProperty:

```

VAR // => SA0119
END_VAR

```

##### Set function of the property FB\_Sub.SampleProperty:

```

VAR // => SA0119
END_VAR

```

### SA0120: Program calls

Function	Determines program calls.
Reason	According to the IEC 61131-3 standard, programs can only be called in the task configuration. The code becomes easier to port if program calls elsewhere are avoided.
Importance	Low

#### Sample:

##### Program SubProgram:

```
PROGRAM SubProgram
```

##### Program MAIN:

```

PROGRAM MAIN
SubProgram(); // => SA0120

```



**SA0121: Missing VAR\_EXTERNAL declarations**

Function	Determines the use of a global variable in the function block, without it being declared as VAR_EXTERNAL (required according to the standard).
Reason	According to the IEC 61131-3 standard, access to global variables is only permitted via an explicit import through a VAR_EXTERNAL declaration.
Importance	Low
PLCopen rule	CP18



In TwinCAT 3 PLC it is not necessary for variables to be declared as external. The keyword exists in order to maintain compatibility with IEC 61131-3.

**Sample:**

Global variables:

```
VAR_GLOBAL
    nGlobal : INT;
END_VAR
```

Program Prog1:

```
PROGRAM Prog1
VAR
    nVar : INT;
END_VAR

nVar := nGlobal; // => SA0121
```

Program Prog2:

```
PROGRAM Prog2
VAR
    nVar : INT;
END_VAR
VAR_EXTERNAL
    nGlobal : INT;
END_VAR

nVar := nGlobal; // no error
```

**SA0122: Array index defined as expression**

Function	Determines the use of expressions in the declaration of array boundaries.
Reason	Not all systems allow expressions as array boundaries.
Importance	Low

**Sample:**

```
PROGRAM MAIN
VAR CONSTANT
    cSample : INT := INT#15;
END_VAR
VAR
    aSample1 : ARRAY[0..10] OF INT;
    aSample2 : ARRAY[0..10+5] OF INT; // => SA0122
    aSample3 : ARRAY[0..cSample] OF INT;
    aSample4 : ARRAY[0..cSample + 1] OF INT; // => SA0122
END_VAR
```

**SA0123: Usages of INI, ADR or BITADR**

Function	Determines the use of the (TwinCAT-specific) operators INI, ADR, BITADR.
Reason	TwinCAT-specific operators prevent portability of the code.
Importance	Low

**Sample:**

```

PROGRAM MAIN
VAR
    nVar : INT;
    pINT : POINTER TO INT;
END_VAR
pINT := ADR(nVar);           // => SA0123

```

**SA0147: Unusual shift operation - strict**

Function	Determines bit shift operations that are not performed on bit field data types (BYTE, WORD, DWORD, LWORD).
Reason	The IEC 61131-3 standard only allows bit access to bit field data types. However, the TwinCAT 3 compiler also allows bit shift operations with unsigned data types.
Importance	Low



See also non-strict rule [SA0052](#) [► 43].

**Samples:**

```

PROGRAM MAIN
VAR
    nBYTE      : BYTE := 16#45;
    nWORD      : WORD := 16#0045;
    nUINT      : UINT;
    nDINT      : DINT;
    nResBYTE   : BYTE;
    nResWORD   : WORD;
    nResUINT   : UINT;
    nResDINT   : DINT;
    nShift     : BYTE := 2;
END_VAR
nResBYTE := SHL(nByte,nShift); // no error because BYTE is a bit field
nResWORD := SHL(nWORD,nShift); // no error because WORD is a bit field
nResUINT := SHL(nUINT,nShift); // => SA0147
nResDINT := SHL(nDINT,nShift); // => SA0147

```

**SA0148: Unusual bit access - strict**

Function	Determines bit access operations that are not performed on bit field data types (BYTE, WORD, DWORD, LWORD).
Reason	The IEC 61131-3 standard only allows bit access to bit field data types. However, the TwinCAT 3 compiler also allows bit access to unsigned data types.
Importance	Low



See also non-strict rule [SA0018](#) [► 29].

**Samples:**

```

PROGRAM MAIN
VAR
    nINT      : INT;

```

```

nDINT      : DINT;
nULINT     : ULINT;
nSINT      : SINT;
nUSINT     : USINT;
nBYTE      : BYTE;
END_VAR

nINT.3     := TRUE;           // => SA0148
nDINT.4     := TRUE;           // => SA0148
nULINT.18  := FALSE;          // => SA0148
nSINT.2     := FALSE;          // => SA0148
nUSINT.3    := TRUE;           // => SA0148
nBYTE.5     := FALSE;          // no error because BYTE is a bitfield

```

### SA0118: Initializations not using constants

Function	Determines initializations that do not assign constants.
Reason	Initializations should be as consistent as possible and should not refer to other variables. In particular, you should avoid function calls during initialization, since this can lead to access to uninitialized data.
Importance	Medium

#### Samples:

Function F\_ReturnDWORD:

```
FUNCTION F_ReturnDWORD : DWORD
```

Program MAIN:

```

PROGRAM MAIN
VAR CONSTANT
    c1 : DWORD := 100;
END_VAR
VAR
    n1 : DWORD := c1;
    n2 : DWORD := F_ReturnDWORD();           // => SA0118
    n3 : DWORD := 150;
    n4 : DWORD := n3;                         // => SA0118
END_VAR

```

### SA0124: Dereference access in initializations

Function	Determines all code positions where dereferenced pointers are used in the declaration part of POU's.
Reason	Pointers and references should not be used for initialization because this can lead to access violations at runtime if the pointer has not been initialized.
Importance	Medium

#### Samples:

```

FUNCTION_BLOCK FB_Test
VAR_INPUT
    pStruct      : POINTER TO ST_Test;
    refStruct    : REFERENCE TO ST_Test;
END_VAR
VAR
    bPointer     : BOOL := pStruct^.bTest; // => SA0124: Dereference access in initialization
    bRef         : BOOL := refStruct.bTest; // => SA0125: Reference used in initialization
END_VAR

bPointer := pStruct^.bTest;           // => SA0039: Possible null pointer dereference 'pStruct^'
bRef     := refStruct.bTest;          // => SA0145: Possible use of not initialized reference 'refStruct'

IF pStruct <> 0 THEN
    bPointer := pStruct^.bTest;        // no error SA0039 as the pointer is checked for unequal 0
END_IF

IF __ISVALIDREF(refStruct) THEN

```

```

    bRef      := refStruct.bTest;          // no error SA0145 as the reference is checked via
    __ISVALIDREF
END_IF

```

### Overview of the rules on "dereferencing".

#### Pointers:

- Dereferencing of pointers in the declaration part => [SA0124](#) [► 67]
- Possible null pointer dereferences in the implementation part => [SA0039](#) [► 69]

#### References:

- Use of references in the declaration part => [SA0125](#) [► 68]
- Possible use of not initialized reference in the implementation part => [SA0145](#) [► 70]

#### Interfaces:

- Possible use of not initialized interface in the implementation part => [SA0046](#) [► 70]

### SA0125: References in initializations

Function	Determines all reference variables used for initialization in the declaration part of POU's.
Reason	Pointers and references should not be used for initialization because this can lead to access violations at runtime if the pointer has not been initialized.
Importance	Medium

#### Samples:

```

FUNCTION_BLOCK FB_Test
VAR_INPUT
    pStruct      : POINTER TO ST_Test;
    refStruct    : REFERENCE TO ST_Test;
END_VAR
VAR
    bPointer     : BOOL := pStruct^.bTest; // => SA0124: Dereference access in initialization
    bRef         : BOOL := refStruct.bTest; // => SA0125: Reference used in initialization
END_VAR

bPointer := pStruct^.bTest; // => SA0039: Possible null pointer dereference 'pStruct^'
bRef     := refStruct.bTest; // => SA0145: Possible use of not initialized reference 'refStruct'

IF pStruct <> 0 THEN
    bPointer := pStruct^.bTest; // no error SA0039 as the pointer is checked for unequal 0
END_IF

IF __ISVALIDREF(refStruct) THEN
    bRef := refStruct.bTest; // no error SA0145 as the reference is checked via __ISVALIDREF
END_IF

```

### Overview of the rules on "dereferencing".

#### Pointers:

- Dereferencing of pointers in the declaration part => [SA0124](#) [► 67]
- Possible null pointer dereferences in the implementation part => [SA0039](#) [► 69]

#### References:

- Use of references in the declaration part => [SA0125](#) [► 68]
- Possible use of not initialized reference in the implementation part => [SA0145](#) [► 70]

#### Interfaces:

- Possible use of not initialized interface in the implementation part => [SA0046](#) [► 70]

**SA0039: Possible null pointer dereferences**

Function	Determines code positions at which a NULL-pointer may be dereferenced.
Reason	A pointer should be checked before each dereferencing to see if it is not equal to 0. Otherwise, access violations may occur at runtime.
Importance	High

**Sample 1:**

```

PROGRAM MAIN
VAR
  pInt1      : POINTER TO INT;
  pInt2      : POINTER TO INT;
  pInt3      : POINTER TO INT;
  nVar1      : INT;
  nCounter   : INT;
END_VAR

nCounter := nCounter + INT#1;

pInt1 := ADR(nVar1);
pInt1^ := nCounter;           // no error

pInt2^ := nCounter;           // => SA0039
nVar1 := pInt3^;              // => SA0039

```

**Sample 2:**

```

FUNCTION_BLOCK FB_Test
VAR_INPUT
  pStruct    : POINTER TO ST_Test;
  refStruct  : REFERENCE TO ST_Test;
END_VAR
VAR
  bPointer   : BOOL := pStruct^.bTest; // => SA0124: Dereference access in initialization
  bRef       : BOOL := refStruct.bTest; // => SA0125: Reference used in initialization
END_VAR

bPointer := pStruct^.bTest;           // => SA0039: Possible null pointer dereference 'pStruct^'
bRef := refStruct.bTest;              // => SA0145: Possible use of not initialized reference
'refStruct'

IF pStruct <> 0 THEN
  bPointer := pStruct^.bTest;         // no error SA0039 as the pointer is checked for unequal 0
END_IF

IF __ISVALIDREF(refStruct) THEN
  bRef := refStruct.bTest;            // no error SA0145 as the reference is checked via
__ISVALIDREF
END_IF

```

**Overview of the rules on "dereferencing".****Pointers:**

- Dereferencing of pointers in the declaration part => [SA0124 \[► 67\]](#)
- Possible null pointer dereferences in the implementation part => [SA0039 \[► 69\]](#)

**References:**

- Use of references in the declaration part => [SA0125 \[► 68\]](#)
- Possible use of not initialized reference in the implementation part => [SA0145 \[► 70\]](#)

**Interfaces:**

- Possible use of not initialized interface in the implementation part => [SA0046 \[► 70\]](#)

**SA0046: Possible use of not initialized interfaces**

Function	Determines the use of interfaces that may not have been initialized before the use.
Reason	An interface reference should be checked for $\neq 0$ before it is used, otherwise an access violation may occur at runtime.
Importance	High

**Samples:****Interface I\_Sample:**

```
INTERFACE I_Sample
METHOD SampleMethod : BOOL
VAR_INPUT
    nInput : INT;
END_VAR
```

**Function block FB\_Sample:**

```
FUNCTION_BLOCK FB_Sample IMPLEMENTS I_Sample
METHOD SampleMethod : BOOL
VAR_INPUT
    nInput : INT;
END_VAR
```

**Program MAIN:**

```
PROGRAM MAIN
VAR
    fbSample      : FB_Sample;
    iSample       : I_Sample;
    iSampleNotSet : I_Sample;
    nParam        : INT;
    bReturn       : BOOL;
END_VAR

iSample := fbSample;
bReturn := iSample.SampleMethod(nInput := nParam); // no error

bReturn := iSampleNotSet.SampleMethod(nInput := nParam); // => SA0046
```

**Overview of the rules on "dereferencing".****Pointers:**

- Dereferencing of pointers in the declaration part => [SA0124](#) [► 67]
- Possible null pointer dereferences in the implementation part => [SA0039](#) [► 69]

**References:**

- Use of references in the declaration part => [SA0125](#) [► 68]
- Possible use of not initialized reference in the implementation part => [SA0145](#) [► 70]

**Interfaces:**

- Possible use of not initialized interface in the implementation part => [SA0046](#) [► 70]

**SA0145: Possible use of not initialized references**

Function	Determines all reference variables that may not be initialized before they are used and were not checked by the __ISVALIDREF operator. This rule is applied in the implementation part of POU's.
Reason	A reference should be checked for validity before it is accessed, otherwise an access violation may occur at runtime.
Importance	High

**Samples:**

```

FUNCTION_BLOCK FB_Test
VAR_INPUT
    pStruct    : POINTER TO ST_Test;
    refStruct  : REFERENCE TO ST_Test;
END_VAR
VAR
    bPointer   : BOOL := pStruct^.bTest; // => SA0124: Dereference access in initialization
    bRef       : BOOL := refStruct.bTest; // => SA0125: Reference used in initialization
END_VAR

bPointer := pStruct^.bTest; // => SA0039: Possible null pointer dereference 'pStruct^'
bRef     := refStruct.bTest; // => SA0145: Possible use of not initialized reference
'refStruct'

IF pStruct <> 0 THEN
    bPointer := pStruct^.bTest; // no error SA0039 as the pointer is checked for unequal 0
END_IF

IF __ISVALIDREF(refStruct) THEN
    bRef := refStruct.bTest; // no error SA0145 as the reference is checked via
    __ISVALIDREF
END_IF

```

### Overview of the rules on "dereferencing".

#### Pointers:

- Dereferencing of pointers in the declaration part => [SA0124](#) [[▶ 67](#)]
- Possible null pointer dereferences in the implementation part => [SA0039](#) [[▶ 69](#)]

#### References:

- Use of references in the declaration part => [SA0125](#) [[▶ 68](#)]
- Possible use of not initialized reference in the implementation part => [SA0145](#) [[▶ 70](#)]

#### Interfaces:

- Possible use of not initialized interface in the implementation part => [SA0046](#) [[▶ 70](#)]

### SA0140: Statements commented out

Function	Determines statements that are commented out.
Reason	Code is often commented out for debugging purposes. When such a comment is enabled, it is not clear at a later point in time whether the code should be deleted or whether it was only commented out for debugging purposes and was inadvertently not commented in again.
Importance	High
PLCopen rule	C4

#### Sample:

```
//bStart := TRUE; // => SA0140
```

### SA0150: Violations of lower or upper limits of the metrics

Function	Determines function blocks that violate the enabled metrics at the lower or upper limit.
Reason	Code that adheres to certain metrics is easier to read, easier to maintain and easier to test.
Importance	High
PLCopen rule	CP9

#### Sample:

The metric "Number of calls" is enabled and configured in the metrics configuration enabled (PLC Project Properties > category "Static Analysis" > "Metrics" tab).

- Lower limit: 0
- Upper limit: 3
- Function block Prog1 is called 5 times

During the execution of the Static Analysis the violation of SA0150 is issued as an error or warning in the message window.

```
// => SA0150: Metric violation for 'Prog1'. Result for metric 'Calls' (5) > 3"
```

### SA0160: Recursive calls

Function	Determines recursive calls of programs, actions, methods and properties. Determines possible recursions through virtual function calls and interface calls.
Reason	Recursions lead to non-deterministic behavior and are therefore a source of errors.
Importance	Medium
PLCopen rule	CP13

#### Sample 1:

Method FB\_Sample.SampleMethod1:

```
METHOD SampleMethod1
VAR_INPUT
END_VAR

SampleMethod1(); (* => SA0160: Recursive call:
                  'MAIN -> FB_Sample.SampleMethod1 -> FB_Sample.SampleMethod1' *)
```

Method FB\_Sample.SampleMethod2:

```
METHOD SampleMethod2 : BOOL
VAR_INPUT
END_VAR

SampleMethod2 := THIS^.SampleMethod2(); (* => SA0160: Recursive call:
                                          'MAIN -> FB_Sample.SampleMethod2 ->
FB_Sample.SampleMethod2' *)
```

Program MAIN:

```
PROGRAM MAIN
VAR
    fbSample : FB_Sample;
    bReturn  : BOOL;
END_VAR

fbSample.SampleMethod1();
bReturn := fbSample.SampleMethod2();
```

#### Sample 2:

Please note regarding properties:

For a property, a local input variable is implicitly created with the name of the property. The following Set function of a property thus assigns the value of the implicit local input variables to the property of an FB variable.

Function block FB\_Sample:

```
FUNCTION_BLOCK FB_Sample
VAR
    nParameter : INT;
END_VAR
```

Set function of the property SampleProperty:

```
nParameter := SampleProperty;
```



In the following Set function, the implicit input variable of the property is assigned to itself. The assignment of a variable to itself does not constitute a recursion, so that this Set function does not generate an SA0160 error.

Set function of the property SampleProperty:

```
SampleProperty := SampleProperty;           // no error SA0160
```

However, access to a property using the THIS pointer is qualified. By using the THIS pointer, the instance and thus the property is accessed, rather than the implicit local input variable. This means that the shading of implicit local input variables and the property itself is lifted. In the following Set function, a new call to the property is generated, which leads to a recursion and thus to error SA0160.

Set function of the property SampleProperty:

```
THIS^.SampleProperty := SampleProperty;     // => SA0160
```

### SA0161: Unpacked structure in packed structure

Function	Determines unpacked structures that are used in packed structures.
Reason	The compiler normally places an unpacked structure on an address that allows aligned access to all elements within the structure. If you create this structure in a packed structure, aligned access is no longer possible, and access to an element in the unpacked structure can lead to a misalignment exception at runtime.
Importance	High

#### Sample:

The structure ST\_SingleDataRecord is packed but contains instances of the unpacked structures ST\_4Byte and ST\_9Byte. This results in a SA0161 error message.

```
{attribute 'pack_mode' := '1'}
TYPE ST_SingleDataRecord :
STRUCT
    st9Byte      : ST_9Byte; // => SA0161
    st4Byte      : ST_4Byte; // => SA0161
    n1           : UDINT;
    n2           : UDINT;
    n3           : UDINT;
    n4           : UDINT;
END_STRUCT
END_TYPE
```

#### Structure ST\_9Byte:

```
TYPE ST_9Byte :
STRUCT
    nRotorSlots   : USINT;
    nMaxCurrent    : UINT;
    nVelocity      : USINT;
    nAcceleration  : UINT;
    nDeceleration  : UINT;
    nDirectionChange : USINT;
END_STRUCT
END_TYPE
```

#### Structure ST\_4Byte:

```
TYPE ST_4Byte :
STRUCT
    fDummy        : REAL;
END_STRUCT
END_TYPE
```

**SA0162: Missing comments**

Function	Detects uncommented locations in the program. Comments are required for: <ul style="list-style-type: none"> <li>the declaration of variables. The comments are shown above or to the right.</li> <li>the declaration of POU, DUTs, GVLs or interfaces. The comments are shown above the declaration (in the first row).</li> </ul>
Reason	Full commentary is required by many programming guidelines. It increases the readability and maintainability of the code.
Importance	Low
PLCopen rule	C2

**Samples:**

The following sample generates the error "SA0162: Missing comment for 'b1'" for variable b1.

```
// Comment for MAIN program
PROGRAM MAIN
VAR
    b1    : BOOL;
    // Comment for variable b2
    b2    : BOOL;
    b3    : BOOL;           // Comment for variable b3
END_VAR
```

**SA0163: Nested comments**

Function	Determines code positions with nested comments.
Reason	Nested comments are difficult to read and should be avoided.
Importance	Low
PLCopen rule	C3

**Samples:**

The four nested comments identified accordingly in the following sample each result in the error: "SA0163: Nested comment '<...>'".

```
(* That is
(* nested comment number 1 *)
*)
PROGRAM MAIN
VAR
    (* That is
    // nested comment
    number 2 *)
    a      : DINT;
    b      : DINT;

    (* That is
    (* nested comment number 3 *) *)
    c      : BOOL;
    nCounter : INT;
END_VAR

(* That is // nested comment number 4 *)

nCounter := nCounter + 1;

(* This is not a nested comment *)
```

**SA0164: Multi-line comments**

Function	Determines code positions at which the multi-line comment operator (**) is used. Only the two single-line comment operators are allowed: // for standard comments, /// for documentation comments.
Reason	Some programming guidelines prohibit multi-line comments in the code, because the beginning and end of a comment could get out of sight and the closing comment bracket could be deleted by mistake.
Importance	Low
PLCopen rule	C5



You can disable this check with the `pragma {analysis ...} [► 122]`, including for comments in the declaration part.

**Samples:**

```
(*
This comment leads to error:
"SA0164 ..."
*)
PROGRAM MAIN
VAR
    /// Documentation comment not reported by SA0164
    nCounter1: DINT;
    nCounter2: DINT;           // Standard single-line comment not reported by SA0164
END_VAR

(* This comment leads to error: "SA0164 ..." *)
nCounter1 := nCounter1 + 1;
nCounter2 := nCounter2 + 1;
```

**SA0166: Maximum number of input/output/VAR\_IN\_OUT variables**

Function	<p>The check determines whether a defined number of input variables (VAR_INPUT), output variables (VAR_OUTPUT) or VAR_IN_OUT variables is exceeded in a function block.</p> <p>You can configure the parameters that are taken into account in the check by double-clicking on the row for rule 166 in the rule configuration (PLC Project Properties &gt; category "Static Analysis" &gt; "Rules" tab &gt; Rule 166). You can make the following settings in the dialog that appears:</p> <ul style="list-style-type: none"> <li>• Maximum number of inputs (default value: 10)</li> <li>• Maximum number of outputs (default value: 10)</li> <li>• Maximum number of inputs/outputs (default value: 10)</li> </ul>
Reason	This is about checking individual programming guidelines. Many programming guidelines stipulate a maximum number of parameters for function blocks. Too many parameters make the code unreadable and the function blocks difficult to test.
Importance	Medium
PLCopen rule	CP23

**Corresponding metrics available**

The following metrics are available for calculating the input and output variables as part of the metrics table:

Number of input variables [► 98]

Number of output variables [► 98]

**Sample:**

Rule 166 is configured with the following parameters:

- Maximum number of inputs: 0
- Maximum number of outputs: 10
- Maximum number of inputs/outputs: 1

The following function block therefore reports two SA0166 errors, since too many inputs (> 0) and too many inputs/outputs (> 1) are declared.

Function block FB\_Sample:

```
FUNCTION_BLOCK FB_Sample           // => SA0166
VAR_INPUT
    bIn      : BOOL;
END_VAR
VAR_OUTPUT
    bOut     : BOOL;
END_VAR
VAR_IN_OUT
    bInOut1  : BOOL;
    bInOut2  : BOOL;
END_VAR
```

### SA0167: Report temporary FunctionBlock instances

Function	Determines function block instances that are declared as temporary variables. This applies to instances that are declared in a method, in a function or as VAR_TEMP, and which are reinitialized in each processing cycle or each function block call.
Reason	<ul style="list-style-type: none"> <li>• Function blocks have a state that is usually retained over several PLC cycles. An instance on the stack exists only for the duration of the function call. It is therefore only rarely useful to create an instance as a temporary variable.</li> <li>• Secondly, function block instances are frequently large and require a great deal of space on the stack (which is usually limited on controllers).</li> <li>• Thirdly, the initialization and often also the scheduling of the function block can take up quite a lot of time.</li> </ul>
Importance	Medium

### Examples:

Method FB\_Sample.SampleMethod:

```
METHOD SampleMethod : INT
VAR_INPUT
END_VAR
VAR
    fbTrigger : R_TRIG;           // => SA0167
END_VAR
```

Function F\_Sample:

```
FUNCTION F_Sample : INT
VAR_INPUT
END_VAR
VAR
    fbSample  : FB_Sample;        // => SA0167
END_VAR
```

MAIN program:

```
PROGRAM MAIN
VAR_TEMP
    fbSample  : FB_Sample;        // => SA0167
    nReturn   : INT;
END_VAR
nReturn := F_Sample();
```

**SA0168: Unnecessary assignments**

Function	Determines assignments to variables that have no effects in the code.
Reason	If several values are assigned to a variable without the variable being evaluated between the assignments, the first assignments do not have any effect on the program.
Importance	Low

**Sample:**

```

PROGRAM MAIN
VAR
    nVar1    : DWORD;
    nVar2    : DWORD;
END_VAR

nVar1 := 1;

IF nVar2 > 100 THEN
    nVar2 := 0;
    nVar2 := nVar2 + 1;
END_IF

nVar1 := 2;                                // => SA0168

```

**SA0169: Ignored outputs**

Function	Determines the outputs of methods and functions that are not specified when calling the method or function.
Reason	Ignored outputs can be an indication of unhandled errors or nonsensical function calls, as the results are not used.
Importance	Medium

**Sample:****Function F\_Sample:**

```

FUNCTION F_Sample : BOOL
VAR_INPUT
    bIn      : BOOL;
END_VAR
VAR_OUTPUT
    bOut     : BOOL;
END_VAR

```

**Program MAIN:**

```

PROGRAM MAIN
VAR
    bReturn  : BOOL;
    bFunOut  : BOOL;
END_VAR

bReturn := F_Sample(bIn := TRUE , bOut => bFunOut);
bReturn := F_Sample(bIn := TRUE);                    // => SA0169

```

**SA0170: Address of an output variable should not be used**

Function	Determines code positions where the address of an output variable (VAR_OUTPUT, VAR_IN_OUT) of a function block is used.
Reason	It is not allowed to use the address of a function block output in the following way: <ul style="list-style-type: none"> <li>• By means of the ADR operator</li> <li>• By means of REF=</li> </ul>
Exception	No error is reported if the output variable is used within the same function block.
Importance	Medium

**Sample:**

## Function block FB\_Sample:

```

FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nIn      : INT;
END_VAR
VAR_OUTPUT
    nOut     : INT;
END_VAR
VAR
    pFB      : POINTER TO FB_Sample;
    pINT     : POINTER TO INT;
END_VAR

IF pFB <> 0 THEN
    pINT := ADR(pFB^.nOut);           // => SA0170
END_IF

nOut := nIn;
pINT := ADR(THIS^.nOut);             // no error due to internal usage
pINT := ADR(nOut);                   // no error due to internal usage

```

Accesses within another function block, in this case in the MAIN program:

```

PROGRAM MAIN
VAR
    fbSample      : FB_Sample;
    pExternal     : POINTER TO INT;
    refExternal   : REFERENCE TO INT;
END_VAR

pExternal := ADR(fbSample.nOut);      // => SA0170
refExternal REF= fbSample.nOut;      // => SA0170

```

**SA0171: Enumerations should have the 'strict' attribute**

Function	Detects declarations of enumerations which are not provided with the {attribute 'strict'} attribute.
Reason	The {attribute 'strict'} attribute causes compiler errors to be issued if the code violates strict programming rules for enumerations. By default, when a new enumeration is created, the declaration is automatically assigned the 'strict' attribute.
Importance	High

For more information see: PLC > Reference programming > Pragmas > Attribute pragmas > Attribute 'strict'

**Sample:**

```

{attribute 'qualified_only'}
{attribute 'strict'}
TYPE E_TrafficLight :
(
    eRed := 0,
    eYellow,
    eGreen
);
END_TYPE

{attribute 'qualified_only'}
TYPE E_MachineStates :           // => SA0171
(
    eStopped := 0,
    eRunning,
    eError
);
END_TYPE

```

**SA0172: Possible attempt to access outside the array limits**

Function	Determines possible accesses to an array index outside the array limits.
Reason	The range of the array index is often exceeded in FOR loops in which the index variable is used to access an array index.
Importance	High

**Sample:**

```

PROGRAM MAIN
VAR_TEMP
  nIndex      : INT;
END_VAR
VAR
  aSample     : ARRAY[0..10] OF INT;
END_VAR

FOR nIndex := INT#0 TO INT#50 DO
  aSample[nIndex] := 0;                      // => SA0172
END_FOR

```

**SA0175: Suspicious operation on string**

Function	Determines code positions that are suspicious for UTF-8 encoding.
Captured constructs	<ol style="list-style-type: none"> <li>Index access to a single-byte string <ul style="list-style-type: none"> <li>Example: sVar[2]</li> <li>Message: Suspicious operation on string: index access '&lt;expression&gt;'</li> </ul> </li> <li>Address access to a single-byte string <ul style="list-style-type: none"> <li>Example: ADR(sVar)</li> <li>Message: Suspicious operation on string: Possible index access '&lt;expression&gt;'</li> </ul> </li> <li>Call of a string function of the Tc2_Standard library except CONCAT and LEN <ul style="list-style-type: none"> <li>Example: FIND(sVar, 'a');</li> <li>Message: Suspicious operation on string: Possible index access '&lt;expression&gt;'</li> </ul> </li> <li>Single byte literal containing non-ASCII characters <ul style="list-style-type: none"> <li>Examples: <pre>sVar := '99€'; sVar := 'Ä';</pre> </li> <li>Message: Suspicious operation on string: literal '&lt;literal&gt;' contains non-ASCII characters</li> </ul> </li> </ol>
Importance	Medium

**Examples:**

```

VAR
  sVar : STRING;
  pVar : POINTER TO STRING;
  nVar : INT;
END_VAR

// 1) SA0175: Suspicious operation on string: Index access
sVar[2];                      // => SA0175

// 2) SA0175: Suspicious operation on string: Possible index access
pVar := ADR(sVar);            // => SA0175

// 3) SA0175: Suspicious operation on string: Possible index access
nVar := FIND(sVar, 'a');       // => SA0175

// 4) SA0175: Suspicious operation on string: Literal '<...>' contains Non-ASCII character
sVar := '99€';                // => SA0175
sVar := 'Ä';                  // => SA0175

```

**SA0178: Cognitive complexity**

Function	<p>The check determines whether a defined limit of cognitive complexity is exceeded in a function block.</p> <p>You can configure the parameter that is taken into account in the check by double-clicking on the row for rule 178 in the rule configuration (PLC Project Properties &gt; category "Static Analysis" &gt; "Rules" tab &gt; Rule 178). You can make the following settings in the dialog that appears:</p> <ul style="list-style-type: none"> <li>• Complexity limit (default value: 20)</li> </ul>
Reason	This is about checking individual programming guidelines. Some programming guidelines stipulate a maximum value for the cognitive complexity of function blocks. Excessive cognitive complexity makes the code difficult to read and maintain.
Importance	Medium

**i** **Corresponding metric available**

The following metric is available for calculating cognitive complexity as part of the metrics table:  
[Cognitive complexity \[► 101\]](#)

**Sample:**

See metric: [Cognitive complexity \[► 101\]](#)

**SA0179: Coupling between objects**

Function	<p>The check determines whether a defined limit of the coupling between objects in a function block is exceeded.</p> <p>You can configure the parameter that is taken into account in the check by double-clicking on the row for rule 179 in the rule configuration (PLC Project Properties &gt; category "Static Analysis" &gt; "Rules" tab &gt; Rule 179). You can make the following settings in the dialog that appears:</p> <ul style="list-style-type: none"> <li>• Coupling limit (default value: 30)</li> </ul>
Reason	This is about checking individual programming guidelines. Some programming guidelines stipulate a maximum value for the coupling between objects for function blocks. Too much coupling between objects makes the code difficult to maintain.
Importance	Medium
Synonym	CBO: <b>C</b> oupling <b>B</b> etween <b>O</b> bjects

**i** **Corresponding metric available**

The following metric is available for calculating cognitive complexity as part of the metrics table:  
[Coupling Between Objects \(CBO\) \[► 105\]](#)

**Sample:**

See metric: [Coupling Between Objects \(CBO\) \[► 105\]](#)

**SA0180: Index range does not cover the entire array**

Function	Determines arrays with an index range that is not completely covered.
Reason	Arrays are often handled in loops, with the loop index indexing the array in such a way that all components of the array are accessed without gaps. This is the case if the loop index and the array index are the same in all dimensions. If the index range does not completely cover the array, this indicates components in the array that have not been processed.
Importance	Medium



**Sample:**

```
PROGRAM MAIN
VAR_TEMP
  nIndex      : INT;
END_VAR
VAR
  aSample     : ARRAY[0..10] OF INT;
END_VAR
FOR nIndex := INT#1 TO INT#10 DO
  aSample[nIndex] := 0;                // => SA0180
END_FOR
```

## 4.3 Naming conventions

In the **Naming Conventions** tab you can define naming conventions. Their compliance is accounted for in the [Static Analysis execution \[► 111\]](#). You define mandatory prefixes for the different data types of variables as well as for different scopes, function block types, and data type declarations. The names of all objects for which a convention can be specified are displayed in the project properties as a tree structure. The objects are arranged below organizational nodes.

In the **Naming Conventions** tab, you will also find [options \[► 90\]](#) that extend the configuration of the prefixes. You can use these options to configure how the expected overall prefix for variables/declarations should be composed.

## Configuration of the naming conventions

Names	<p>Nodes and elements for which a prefix can be defined</p> <p>The number in brackets after each element, for example "PROGRAM (102)", is the prefix convention number that is output if the naming convention is not followed.</p>
Prefix	<p>You can define the naming conventions by entering the required prefix in this column.</p> <p>Please note the following notes and options:</p> <ul style="list-style-type: none"> <li>• Several possible prefixes per line <ul style="list-style-type: none"> <li>◦ Multiple prefixes can be entered separated by commas.</li> <li>◦ Example: "x, b" as prefixes for variables of data type BOOL. "x" and "b" may be used as prefix for Boolean variables.</li> </ul> </li> <li>• Regular expressions <ul style="list-style-type: none"> <li>◦ You can also use regular expressions (RegEx) for the prefix. In this case you have to use @ as additional prefix.</li> <li>◦ Example: "@b[a-dA-D]" as prefix for variables of data type BOOL. The name of the boolean variable must start with "b", and may be followed by a character in the range "a-dA-D".</li> </ul> </li> <li>• Data type placeholder <ul style="list-style-type: none"> <li>◦ For variables of the Alias data type and for properties you can use the data type placeholder "{datatype}" as prefix.</li> <li>◦ Example: Prefix for the variable data type Alias (33) = "{datatype}"</li> </ul> </li> </ul>
Prefixes for variables	Organizational node for all variables for which a prefix dependent on their data type or scope can be defined
Prefixes for POUs	Organizational node for all POU types and method validity ranges for which a prefix can be defined
Prefixes for DUTs	Organizational node for the DUT data types Structure, Enumeration, Alias or Union for which a prefix can be defined
Prefixes for user-defined types (NC0160)	<p>Available from TC3.1 Build 4026</p> <p>Organizational node for special user-defined types, especially those from libraries or for read-only types (e.g. PVOID, HRESULT)</p> <ul style="list-style-type: none"> <li>• You can expand the list with conventions: click the blank line below. Then enter the name of a user-defined type or select a user-defined type in the "Input Assistant" dialog.</li> <li>• You can delete a convention by selecting it and choosing the [Del] key.</li> </ul> <p>Note: These conventions take priority over the prefixes defined with the {attribute 'nameprefix' := '&lt;prefix&gt;'} attribute.</p> <p>Sample:</p> <ul style="list-style-type: none"> <li>• In the "Name" column, enter the read-only system data type "PVOID" in an empty line below the prefixes for user-defined types. In the same line in the "Prefix" column, enter the desired prefix, e.g. "p". Variables of type PVOID are checked for this prefix when running Static Analysis.</li> <li>• More examples of user-defined types whose desired prefix you can configure at this point: <ul style="list-style-type: none"> <li>◦ System data type HRESULT</li> <li>◦ TON function block from the Tc2_System library</li> </ul> </li> </ul>

### ● Formation of the expected prefix



The prefix expected for the different declarations is formed depending on the configuration of the options found in the [Options \[► 90\]](#) dialog.

On the [Options \[► 90\]](#) page you will also find explanations on how the expected prefix is formed, as well as some samples.

### ● Placeholder {datatype} with alias variables and properties



Please also note the possibilities of the placeholder {datatype} [\[► 93\]](#), which you can use for the prefix definition of alias variables and properties.

### ● Local prefix definition for structured types



For variables of structured types, you can specify a prefix locally in the data type declaration using the 'nameprefix' attribute [\[► 124\]](#).

## Syntax of convention violations in the message window

Each naming convention has a unique number (shown in parentheses after the convention in the naming convention configuration view). If a violation of a convention or a preset is detected during the static analysis, the number is output in the error list together with an error description based on the following syntax. The abbreviation "NC" stands for "Naming Convention".

Syntax: "NC<prefix convention number>: <convention description>"

Sample for convention number 151 (DUTs of type Structure): "NC0151: Invalid type name 'STR\_Sample'. Expected prefix 'ST\_'"

## Temporary deactivation of naming conventions

Individual conventions can be disabled temporarily, i.e. for particular code lines. To this end you can add a pragma or an attribute in the declaration or implementation part of the code. For variables of structured types you may specify a prefix locally via an attribute in the data type declaration. For more information see: [Pragmas and attributes \[► 121\]](#).

## Overview of naming conventions

For an overview of naming conventions, see [Naming conventions – overview and description \[► 83\]](#).

## 4.3.1 Naming conventions – overview and description

### Overview

#### - Prefixes for variables

##### - Prefixes for types

- NC0003: BOOL [\[► 86\]](#)
- NC0004: BIT [\[► 86\]](#)
- NC0005: BYTE [\[► 86\]](#)
- NC0006: WORD [\[► 86\]](#)
- NC0007: DWORD [\[► 86\]](#)
- NC0008: LWORD [\[► 86\]](#)
- NC0013: SINT [\[► 86\]](#)

- [NC0014: INT \[► 86\]](#)
- [NC0015: DINT \[► 86\]](#)
- [NC0016: LINT \[► 86\]](#)
- [NC0009: USINT \[► 86\]](#)
- [NC0010: UINT \[► 86\]](#)
- [NC0011: UDINT \[► 86\]](#)
- [NC0012: ULINT \[► 86\]](#)
- [NC0017: REAL \[► 86\]](#)
- [NC0018: LREAL \[► 86\]](#)
- [NC0019: STRING \[► 86\]](#)
- [NC0020: WSTRING \[► 86\]](#)
- [NC0021: TIME \[► 86\]](#)
- [NC0022: LTIME \[► 86\]](#)
- [NC0023: DATE \[► 86\]](#)
- [NC0024: DATE AND TIME \[► 86\]](#)
- [NC0025: TIME OF DAY \[► 86\]](#)
- [NC0026: POINTER \[► 86\]](#)
- [NC0027: REFERENCE \[► 86\]](#)
- [NC0028: SUBRANGE \[► 87\]](#)
- [NC0030: ARRAY \[► 87\]](#)
- [NC0031: Function block instance \[► 87\]](#)
- [NC0036: Interface \[► 87\]](#)
- [NC0032: Structure \[► 88\]](#)
- [NC0029: ENUM \[► 88\]](#)
- [NC0033: Alias \[► 88\]](#)
- [NC0034: Union \[► 88\]](#)
- [NC0035: XWORD \[► 86\]](#)
- [NC0037: UXINT \[► 86\]](#)
- [NC0038: XINT \[► 86\]](#)

**- Prefixes for scopes**

- [NC0051: VAR GLOBAL \[► 89\]](#)
- [NC0070: VAR GLOBAL CONSTANT \[► 89\]](#)
- [NC0071: VAR GLOBAL RETAIN \[► 89\]](#)

- [NC0072: VAR GLOBAL PERSISTENT \[► 89\]](#)
- [NC0073: VAR GLOBAL RETAIN PERSISTENT \[► 89\]](#)
- **VAR**
  - [NC0053: Program variables \[► 89\]](#)
  - [NC0054: Function block variables \[► 89\]](#)
  - [NC0055: Function/method variables \[► 89\]](#)
- [NC0056: VAR INPUT \[► 89\]](#)
- [NC0057: VAR OUTPUT \[► 89\]](#)
- [NC0058: VAR IN OUT \[► 89\]](#)
- [NC0059: VAR STAT \[► 89\]](#)
- [NC0061: VAR TEMP \[► 89\]](#)
- [NC0062: VAR CONSTANT \[► 89\]](#)
- [NC0063: VAR PERSISTENT \[► 89\]](#)
- [NC0064: VAR RETAIN \[► 89\]](#)
- [NC0065: I/O variables \[► 89\]](#)

## - Prefixes for POU

### - Prefixes for POU type

- [NC0102: PROGRAM \[► 89\]](#)
- [NC0103: FUNCTIONBLOCK \[► 89\]](#)
- [NC0104: FUNCTION \[► 89\]](#)
- [NC0105: METHOD \[► 89\]](#)
- [NC0106: ACTION \[► 89\]](#)
- [NC0107: PROPERTY \[► 89\]](#)
- [NC0108: INTERFACE \[► 89\]](#)

### - Method/property scope

- [NC0121: PRIVATE \[► 90\]](#)
- [NC0122: PROTECTED \[► 90\]](#)
- [NC0123: INTERNAL \[► 90\]](#)
- [NC0124: PUBLIC \[► 90\]](#)

## - Prefixes for DUTs

- [NC0151: Structure \[► 90\]](#)
- [NC0152: Enumeration \[► 90\]](#)
- [NC0153: Union \[► 90\]](#)
- [NC0154: Alias \[► 90\]](#)

## - Prefixes for user-defined types

- [NC0160: User-defined type](#) [► 90]

### Detailed description

The following sections contain explanations and examples of which declarations (i.e. at which point in the project) use the individual naming conventions. The declarations samples illustrate cases for which the corresponding prefix would be expected if a prefix was defined with the corresponding naming convention. It should become clear where and how a type or variable can be declared so that the naming convention NC<xxxx> is checked at this point. However, the samples do not show which concrete prefix is defined for the individual naming conventions and would therefore be expected in the sample declarations. There is therefore no OK/NOK comparison.

For concrete examples with a defined prefix, please refer to the page [Options](#) [► 90].

### Basic data types:

#### NC0003: BOOL

Configuration of a prefix for a variable declaration of type BOOL.

#### Sample declarations:

For the following variable declarations the prefix configured for NC0003 is used for the formation of the overall prefix, compliance with which is checked during [execution of the static analysis](#) [► 111].

```
bStatus      : BOOL;
abVar        : ARRAY[1..2] OF BOOL;
IbInput  AT%I* : BOOL;
```

The description of "NC0003: BOOL" is transferrable to the other basic data types:

- NC0004: BIT, NC0005: BYTE
- NC0006: WORD, NC0007: DWORD, NC0008: LWORD
- NC0013: SINT, NC0014: INT, NC0015: DINT, NC0016: LINT, NC0009: USINT, NC0010: UINT, NC0011: UDINT, NC0012: ULINT
- NC0017: REAL, NC0018: LREAL
- NC0019: STRING, NC0020: WSTRING
- NC0021: TIME, NC0022: LTIME, NC0023: DATE, NC0024: DATE\_AND\_TIME, NC0025: TIME\_OF\_DAY
- NC0035: \_\_XWORD, NC0037: \_\_UXINT, NC0038: \_\_XINT

### Nested data types:

#### NC0026: POINTER

Configuration of a prefix for a variable declaration of type POINTER TO.

#### Sample declaration:

For the following variable declaration the prefix configured for NC0026 is used for the formation of the overall prefix, compliance with which is checked during [execution of the static analysis](#) [► 111].

```
pnID : POINTER TO INT;
```

#### NC0027: REFERENCE

Configuration of a prefix for a variable declaration of type REFERENCE TO.

**Sample declaration:**

For the following variable declaration the prefix configured for NC0027 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
reffCurrentPosition : REFERENCE TO REAL;
```

**NC0028: SUBRANGE**

Configuration of a prefix for a variable declaration of a subrange type. A subrange type is a data type whose value range only covers a subset of a base type.

Possible basic data types for a subrange type: SINT, USINT, INT, UINT, DINT, UDINT, BYTE, WORD, DWORD, LINT, ULINT, LWORD.

**Sample declarations:**

For the following variable declaration the prefix configured for NC0028 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
subiRange : INT(3..5);  
sublwRange : LWORD(100..150);
```

**NC0030: ARRAY**

Configuration of a prefix for a variable declaration of type ARRAY[...] OF.

**Sample declaration:**

For the following variable declaration the prefix configured for NC0030 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
anTargetPositions : ARRAY[1..10] OF INT;
```

**Instance-based data types:****NC0031: Function block instance**

Configuration of a prefix for a variable declaration of a function block type.

**Sample declaration:**

Declaration of a function block:

```
FUNCTION_BLOCK FB_Sample  
...
```

For the following variable declaration the prefix configured for NC0031 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
fbSample : FB_Sample;
```

**NC0036: Interface**

Configuration of a prefix for a variable declaration of an interface type.

**Sample declaration:**

Interface declaration:

```
INTERFACE I_Sample
```

For the following variable declaration the prefix configured for NC0036 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
iSample : I_Sample;
```

**NC0032: Structure**

Configuration of a prefix for a variable declaration of a structure type.

**Sample declaration:**

Declaration of a structure:

```
TYPE ST_Sample :  
STRUCT  
    bVar : BOOL;  
    sVar : STRING;  
END_STRUCT  
END_TYPE
```

For the following variable declaration the prefix configured for NC0032 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
stSample : ST_Sample;
```

**NC0029: ENUM**

Configuration of a prefix for a variable declaration of an enumeration type.

**Sample declaration:**

Declaration of an enumeration:

```
TYPE E_Sample :  
(  
    eMember1 := 1,  
    eMember2  
);  
END_TYPE
```

For the following variable declaration the prefix configured for NC0029 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
eSample : E_Sample;
```

**NC0033: Alias**

Configuration of a prefix for a variable declaration of an alias type.

**Sample declaration:**

Declaration of an alias:

```
TYPE T_Message : STRING; END_TYPE
```

For the following variable declaration the prefix configured for NC0033 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
tMessage : T_Message;
```

**NC0034: Union**

Configuration of a prefix for a variable declaration of a union type.

**Sample declaration:**

Declaration of a union:

```
TYPE U_Sample :  
UNION  
    n1 : WORD;  
    n2 : INT;  
END_UNION  
END_TYPE
```

For the following variable declaration the prefix configured for NC0034 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
uSample : U_Sample;
```



**Scopes of variable declarations:****NC0051: VAR\_GLOBAL**

Configuration of a prefix for a variable declaration between the keywords VAR\_GLOBAL and END\_VAR.

**Sample declaration:**

For the following declaration of a global variable, the prefix configured for NC0051 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
VAR_GLOBAL
    gbErrorAcknowledge : BOOL;
END_VAR
```

The description of "NC0051: VAR\_GLOBAL" is transferrable to other scopes of variable declarations:

- NC0070: VAR\_GLOBAL CONSTANT
- NC0071: VAR\_GLOBAL RETAIN
- NC0072: VAR\_GLOBAL PERSISTENT
- NC0073: VAR\_GLOBAL RETAIN PERSISTENT
- NC0053: Program variables (VAR within a program)
- NC0054: Function block variables (VAR within a function block)
- NC0055: Function/method variables (VAR within a function/method)
- NC0056: VAR\_INPUT
- NC0057: VAR\_OUTPUT
- NC0058: VAR\_IN\_OUT
- NC0059: VAR\_STAT
- NC0061: VAR\_TEMP
- NC0062: VAR CONSTANT
- NC0063: VAR PERSISTENT
- NC0064: VAR RETAIN

**NC0065: I/O variables**

Configuration of a prefix for a variable declaration with AT declaration.

**Sample declarations:**

For the following variable declarations with AT declaration, the prefix configured for NC0065 is used for the formation of the overall prefix, compliance with which is checked during execution of the static analysis [► 111].

```
ioVar1  AT%I*    : INT;
ioVar2  AT%IX1.0 : BOOL;
ioVar3  AT%Q*    : INT;
ioVar4  AT%QX2.0 : BOOL;
```

**POU types:****NC0102: PROGRAM**

Configuration of a prefix for the declaration of a program (name of the program in the project tree).

The description of "NC0102: PROGRAM" is transferrable to the other POU types:

- NC0103: FUNCTIONBLOCK
- NC0104: FUNCTION
- NC0105: METHOD
- NC0106: ACTION
- NC0107: PROPERTY
- NC0108: INTERFACE

#### Scopes of methods and properties:

##### NC0121: PRIVATE

Configuration of a prefix for the declaration of a method or a property (name of the method/property in the project tree), whose access modifier is PRIVATE.

The description of "NC121: PRIVATE" is transferrable to the other scopes of methods and properties:

- NC0122: PROTECTED
- NC0123: INTERNAL
- NC0124: PUBLIC

#### DUTs:

##### NC0151: Structure

Configuration of a prefix for the declaration of a structure (name of the structure in the project tree).

The description of "NC0151: Structure" is transferrable to the other DUT types:

- NC0152: Enumeration
- NC0153: Union
- NC0154: Alias

#### User-defined types:

##### NC0160: User defined type

Configuration of a prefix for a user-defined type, e.g. for variables of type PVOID or for instances of the library function block Tc2\_System.TON.

For more information on the input options in this area, visit [Naming conventions](#) [► 81].

## 4.3.2 Options

In the **Naming Conventions** tab, you will find options that extend the configuration of the prefixes. You can use these options to configure how the expected overall prefix for variables/declarations should be composed.

#### 1) First character after prefix should be an upper case letter

- If enabled: The static code analysis reports an error for a variable if the first character of the variable name after the defined prefix is not an upper case letter.

- If disabled: Upper case/lower case spelling is not checked.
- Default setting: disabled

**Samples:**

- Variable "bvar" with the expected prefix "b"
- Function block "FB\_sample" with the expected prefix "FB\_"

Option	State	Result of the static analysis
First character after prefix should be an upper case letter	Enabled	For the definitions mentioned above, an error will be reported in each case that the first letter after the prefix must be an upper case letter. Correct identifiers would be "bVar" and "FB_Sample".
	Disabled	The identifiers "bvar" and "FB_sample" are permissible. No upper/lower case error is output.

**2) Recursive prefixes for combinable data types**

- If enabled: Variables of combinable data types (POINTER, REFERENCE, ARRAY, SUBRANGE) must have a composite **data type prefix**. The composite prefix is formed from the individual prefixes configured for the individual components of the combined data type.
- If disabled: Only the prefix of the outermost data type is expected as the **data type prefix**.
- Default setting: enabled
- Examples: see below

**3) Combine scope prefix with data type prefix**

(namespace = scope)

- If enabled: A variable must have the **prefix for its scope** defined in the naming conventions, followed by its **data type prefix**.
- If disabled: The expected overall prefix depends on whether or not a scope prefix is defined for a variable.
  - If the associated **scope prefix is defined** for a variable, the variable must have **only** the **prefix for its scope** defined in the naming conventions. The **data type prefix** is **not** expected after the scope prefix.
  - If the associated **scope prefix is not defined** for a variable, the variable must have **only** the **data type prefix** defined for it.
- Default setting: enabled
- Examples: see below

**Samples**

- Prefix configuration for data types:
  - POINTER (26) = "p"
  - ARRAY (30) = "a"
  - INT (14) = "n"
  - BOOL (3) = "b"
- Prefix configuration for scope
  - Case 1: Function block variables (54) = "\_local\_"
  - Case 2: Function block variables (54) = empty field/not configured
  - **INFO:** Further examples of a scope include VAR\_GLOBAL (51), VAR\_INPUT (56) and VAR\_CONSTANT (62).
- Declaration:

```
FUNCTION_BLOCK FB_Sample
VAR
    var1 : POINTER TO ARRAY[1..3] OF INT;
    var2 : ARRAY[10..20] OF ARRAY[3..5] OF BOOL;
END_VAR
```

**Option scenario 1:**

Option	State	Expected overall prefix for case 1 (NC0054 = "_local_")	Expected overall prefix for case 2 (NC0054 = empty)
Recursive prefixes for combinable data types	Enabled	For var1: '_local_pan' For var2: '_local_aab'	For var1: 'pan' For var2: 'aab'
Combine scope prefix with data type prefix	Enabled		

Explanation:

- As the option "Recursive prefixes for combinable data types" is enabled, the prefix composed of the individual prefixes is expected as the **data type prefix**. Consequently, the sub-prefixes "p" for POINTER, "a" for ARRAY and "n" for INT are combined to form the data type prefix "pan", or the sub-prefixes "a" for ARRAY, "a" for ARRAY again and "b" for BOOL are combined to form the data type prefix "aab".
- As the option "Combine scope prefix with data type prefix" is also enabled, the combination of **scope prefix** and **data type prefix** is expected as the **overall prefix**.
  - Case 1: \_local\_ + pan = \_local\_pan
  - Case 2: <empty> + pan = pan

**Option scenario 2:**

Option	State	Expected overall prefix for case 1 (NC0054 = "_local_")	Expected overall prefix for case 2 (NC0054 = empty)
Recursive prefixes for combinable data types	Disabled	For var1: '_local_p' For var2: '_local_a'	For var1: 'p' For var2: 'a'
Combine scope prefix with data type prefix	Enabled		

Explanation:

- As the option "Recursive prefixes for combinable data types" is disabled, only the prefix of the outermost data type is expected as the **data type prefix**. The expected data type prefix is therefore "p" or "a".
- As the option "Combine scope prefix with data type prefix" is enabled, the combination of **scope prefix** and **data type prefix** is expected as the **overall prefix** for variables.
  - Case 1: \_local\_ + p = \_local\_p
  - Case 2: <empty> + p = p

**Option scenario 3:**

Option	State	Expected overall prefix for case 1 (NC0054 = "_local_")	Expected overall prefix for case 2 (NC0054 = empty)
Recursive prefixes for combinable data types	Enabled	For var1: '_local_' For var2: '_local_'	For var1: 'pan' For var2: 'aab'
Combine scope prefix with data type prefix	Disabled		

Explanation:

- See option scenario 1: As the option "Recursive prefixes for combinable data types" is enabled, the prefix composed of the individual prefixes is expected as the **data type prefix**. This results in "pan" or "aab" as the data type prefix.
- As the option "Combine scope prefix with data type prefix" is disabled, the expected **overall prefix** depends on whether or not a scope prefix is defined for a variable.

- If **scope prefix is defined** (case 1): The variable must **only** have the **scope prefix**. The data type prefix is not expected after the scope prefix. This results for both variables in "\_local\_" as the expected overall prefix.
- If **scope prefix is not defined** (case 2): The variable must only have the data type prefix. This results in "pan" or "aab" as the expected overall prefix.

#### Option scenario 4:

Option	State	Expected overall prefix for case 1 (NC0054 = "_local_")	Expected overall prefix for case 2 (NC0054 = empty)
Recursive prefixes for combinable data types	Disabled	For var1: '_local_' For var2: '_local_'	For var1: 'p' For var2: 'a'
Combine scope prefix with data type prefix	Disabled		

#### Explanation:

- See option scenario 2: As the option "Recursive prefixes for combinable data types" is disabled, only the prefix of the outermost data type is expected as the **data type prefix**. This results in "p" or "a" as the data type prefix.
- As the option "Combine scope prefix with data type prefix" is disabled, the expected **overall prefix** depends on whether or not a scope prefix is defined for a variable.
  - If **scope prefix is defined** (case 1): The variable must **only** have the **scope prefix**. The data type prefix is not expected after the scope prefix. This results for both variables in "\_local\_" as the expected overall prefix.
  - If **scope prefix is not defined** (case 2): The variable must only have the data type prefix. This results in "p" or "a" as the expected overall prefix.

#### Further notes/examples:

For POU's with an access modifier (methods or properties), the combination of the **prefix for the scope** (NC0121-NC0124: PRIVATE/PROTECTED/INTERNAL/PUBLIC) and the **prefix for the POU type** (NC0105 for method, NC0107 for property) is expected as the **overall prefix**. Samples:

- If the prefix "priv\_" has been configured for PRIVATE (121) and the prefix "M\_" for METHOD (105), the **overall prefix** "priv\_M\_" is expected for a PRIVATE method.
- If the prefix "M\_" is still configured for METHOD (105), but no prefix has been configured for PRIVATE (121), that is, if the field is empty in the naming conventions, the **overall prefix** "M\_" is expected for a PRIVATE method.

### 4.3.3 Placeholder {datatype}

For variables of type Alias and for properties, the placeholder "{datatype}" can be defined as a prefix in the "Naming Conventions" tab. The placeholder {datatype} is thereby replaced by the prefix that is defined for the data type of the alias or for the data type of the property. The static analysis thus reports errors for all alias variables that do not possess the prefix for the data type of the alias or for all properties that do not possess the prefix for the data type of the property.

The placeholder "{datatype}" can also be combined with further prefixes in the prefix definition, e.g. to "P\_{datatype}\_".

#### Example 1 for an alias variable:

- In the project there is an alias "TYPE MyMessageType : STRING; END\_TYPE" as well as a variable of this type (var : MyMessageType;).
- Prefix definitions
  - Prefix for the variable data type alias (33) = "{datatype}"
  - Prefix for the variable data type STRING (19) = "s"

- In the prefix definitions mentioned the data type prefix "s" is expected for a variable of the alias type "MyMessageType" (e.g. for the variable "var").

#### Example 2 for an alias variable:

- Same situation as in example 1 for an alias variable, the only difference being:
  - Prefix for the variable data type alias (33) = "al\_{datatype}"
- In this case the data type prefix "al\_s" is expected for a variable of the alias type "MyMessageType".

#### Example of a property:

- Prefix definitions
  - Prefix for the method/property scope PRIVATE (121) = "priv\_"
  - Prefix for the POU type PROPERTY (107) = "P\_{datatype}"
  - Prefix for the variable data type LREAL (18) = "f"
- Note: For POU's with an access modifier (methods or properties), the combination of the prefix for the scope (NC0121-NC0124: PRIVATE/PROTECTED/INTERNAL/PUBLIC) and the prefix for the POU type (NC0105 for method, NC0107 for property) is expected as the overall prefix.
- With the prefix definitions mentioned the overall prefix "priv\_P\_f" is thus expected for a property with the access modifier PRIVATE and the data type LREAL.

## 4.4 Metrics

In the **Metrics** tab you can select and configure the metrics to be displayed for each function block in the **Standard Metrics** view when the command 'View Standard Metrics' [► 114] is executed.

You have over 20 metrics at your disposal to analyze and characterize the underlying source code. When calculated regularly, the metrics can indicate negative trends and deviations from quality targets. The key figures therefore represent an indicator for assessing software quality. For example, the tabular output contains metrics for the number of statements or the proportion of comments.

### ● Analysis of libraries



The following metrics are also output for the libraries integrated in the project: code size, variable size, stack size and number of calls.

### ● Compilation errors for violations of upper/lower limits



You can use rule SA0150 of the static code analysis to output violations of the upper and lower limits of the activated metrics as compilation errors.

#### Configuration of the metrics

Active	<p>You can enable or disable the individual metrics using the checkbox for the respective row. When <u>command 'View Standard Metrics'</u> [► 114] is executed, the metrics that are enabled in the respective configuration are shown for each programming block in the <b>Standard Metrics</b> view.</p> <ul style="list-style-type: none"> <li>• <input type="checkbox"/> : The metric is disabled and is not displayed in the <b>Standard Metrics</b> view when the command <b>View Standard Metrics</b> is executed.</li> <li>• <input checked="" type="checkbox"/> : The metric is enabled and is displayed in the <b>Standard Metrics</b> view when the command <b>View Standard Metrics</b> is executed.</li> </ul>
Lower limit	<p>For each metric you can define an individual upper and lower limit by entering the required number in the respective metric row.</p> <p>If a metric is only limited in one direction, you can leave the configuration for the other direction blank. In other words, you may specify either only the lower limit or only the upper limit.</p>
Upper limit	

## Evaluation of the upper and lower limits

The set upper and lower limits you can be evaluated in two ways.

- **Standard Metrics** view:
  - Enable the metric whose configured upper and lower limits you want to evaluate.
  - Execute the [Command 'View Standard Metrics'](#) [► 114].
  - TwinCAT shows the enabled metrics for each programming block in the tabular **Standard Metrics** view.
  - If a value is outside the range defined by an upper and/or lower limit in the configuration, the table cell is shown in red.
- Static Analysis:
  - Enable rule 150 as error or warning in the [Rules](#) [► 16] tab.
  - Run the Static Analysis (see: [Command 'Run static analysis'](#) [► 111]).
  - Violations of the upper and/or lower limits are issued as error or warning in the message window.

## Overview and description of the metrics

An overview of the metrics and a detailed description of the rules can be found in the next chapter.

### 4.4.1 Metrics - overview and description

Title in the "Standard metrics" view	Description
Code size	<a href="#">Code size [number of bytes]</a> [► 96]
Variables size	<a href="#">Variables size [number of bytes]</a> [► 96]
Stack size	<a href="#">Stack size [number of bytes]</a> [► 96]
Calls	<a href="#">Number of calls</a> [► 97]
Tasks	<a href="#">Called in tasks</a> [► 97]
Globals	<a href="#">Number of global variables used</a> [► 98]
IOs	<a href="#">Number of address accesses</a> [► 98]
Locals	<a href="#">Number of local variables</a> [► 98]
Inputs	<a href="#">Number of input variables</a> [► 98]
Outputs	<a href="#">Number of output variables</a> [► 98]
NOS	<a href="#">Number Of Statements (NOS)</a> [► 99]
Comments	<a href="#">Percentage of comment</a> [► 100]
McCabe	<a href="#">Complexity (McCabe)</a> [► 100]
Complexity	<a href="#">Cognitive complexity</a> [► 101]
DIT	<a href="#">Depth of Inheritance Tree (DIT)</a> [► 103]
NOC	<a href="#">Number Of Children (NOC)</a> [► 103]
RFC	<a href="#">Response For Class (RFC)</a> [► 104]
CBO	<a href="#">Coupling Between Objects (CBO)</a> [► 105]
Elshof	<a href="#">Complexity of reference (Elshof)</a> [► 105]
LCOM	<a href="#">Lack of Cohesion Of Methods - LCOM</a> [► 106]
SFC branches	<a href="#">Number of SFC branches</a> [► 107]
SFC steps	<a href="#">Number of SFC steps</a> [► 107]

## Detailed description

**Code size [number of bytes]**

Title short form	Code size
Categories	Informative, efficiency
Definition	Number of bytes that a function block contributes to the application code
Further information	The number also depends on the code generator. For example, the code generator for Arm® processors generally generates more bytes than the code generator for x86 processors.

**Variables size [number of bytes]**

Title short form	Variables size
Categories	Informative, efficiency
Definition	Size of the static memory used by the object
Further information	For function blocks, this is the size used for an instance of this function block (which may also contain memory gaps depending on the memory alignment). For programs, functions and global variable lists, this is the sum of the size of all static variables.

**Sample:**

```

FUNCTION F_Sample : INT
VAR_INPUT
    a, b    : INT;
END_VAR
VAR
    c, d    : INT;
END_VAR
VAR_STAT
    f, g, h : INT;
END_VAR

```

The function has three static variables of type INT (f, g, h), each of which requires 2 bytes of memory. F\_Sample therefore has a variable size of 6 bytes.

**Stack size [number of bytes]**

Title short form	Stack size
Categories	Informative, efficient, reliable
Definition	Number of bytes required to call a function or function block
Further information	<p>Input variables and output variables are aligned with the memory. This can create a gap between these variables and the local variables. This gap is counted.</p> <p>Return values from called functions that do not fit into a register are pushed onto the stack. The largest of these values determines the additionally allocated memory, which is also counted. Functions or function blocks that are called within the POU's under consideration have their own stack frame. Therefore, the memory for such calls does not count.</p> <p>Depending on the code generator used, intermediate results of calculations also use the stack. These results are not counted.</p>

**Sample:**

```

FUNCTION F_Sample : INT
VAR_INPUT
    a, b    : INT;
END_VAR
VAR
    c, d, e : INT;
END_VAR
VAR_STAT
    f, g, h : INT;
END_VAR

```



```
c := b;
d := a;
e := a + b;
```

Assumption: The "TwinCAT RT (x86)" solution platform is used for the calculation. The device has a stack alignment of 4 bytes, which can result in gaps between the variables.

The total stack size of `F_Sample` is 16 bytes and consists of:

- 2 input variables with 2 bytes each = 4 bytes
- No padding bytes
- Return value INT = 2 bytes
- Padding bytes for the stack alignment = 2 bytes
- 3 local variables with 2 bytes each = 6 bytes
- Padding bytes for the stack alignment = 2 bytes

`VAR_STAT` is not stored on the stack and therefore does not increase the stack size of a POU.

### Number of calls

Title short form	Calls
Categories	Informative
Definition	Number of calls to the program organization unit (POU) within the application
Further information	If a program is called in a task, this call is also counted.

### Called in tasks

Title short form	Tasks
Categories	Maintainability, reliability
Definition	Number of tasks in which the program organization unit (POU) is called up
Further information	For function blocks, the number of tasks in which the function block itself or any function block in the function block's inheritance tree is called is counted. For methods and actions, the number of tasks in which the (higher-level) function block is called is displayed.

### Sample:

```
FUNCTION_BLOCK FB1
```

```
FUNCTION_BLOCK FB2 EXTENDS FB1
```

```
FUNCTION_BLOCK FB3 EXTENDS FB2
```

Each function block is instantiated and called in a separate program. In addition, each program is called in a separate task.

The metric **Called in tasks** therefore results:

- For FB3: 1
- For FB2: 2, as the calls from FB2 and FB3 (`EXTENDS FB2`) are counted
- For FB1: 3, as the calls from FB1, FB2 and FB3 are counted

**Number of global variables used**

Title short form	Globals
Categories	Maintainability, reusability
Definition	Number of different global variables used in the program organization unit (POU)

**Number of address accesses**

Title short form	IOs
Categories	Reusability, maintainability
Definition	Number of address accesses in the implementation of the object

**Sample:**

```

PROGRAM MAIN
VAR
    bVar      : BOOL;
    bIn       AT%I* : BOOL;
    bOut      AT%Q* : BOOL;
END_VAR

bVar := TRUE;
bOut := bIn;
bOut := NOT bOut AND bIn;

```

The number of address accesses for the program `MAIN` is 5 and is made up of 2 write and 3 read accesses.

**Number of local variables**

Title short form	Locals
Categories	Informative, efficiency
Definition	Number of variables declared in the VAR area of the program organization unit (POU)
Further information	Inherited local variables are not counted.

**Number of input variables**

Number of input variables of the function block (VAR\_INPUT).

Title short form	Inputs
Categories	Maintainability, reusability
Definition	Number of variables declared in the VAR_INPUT area of the program organization unit (POU)
Further information	Inherited input variables are not counted.
Standard upper limit for the associated rule SA0166 [► 75]	10

**Number of output variables**

Number of output variables of the function block (VAR\_OUTPUT).

Title short form	Outputs
Categories	Maintainability, reusability
Definition	Number of variables declared in the VAR_OUTPUT area of the program organization unit (POU)
Further information	For function blocks, this is the number of user-defined output variables (VAR_OUTPUT). For methods and functions, this is the number of user-defined output variables (VAR_OUTPUT) plus one if they have a return value. The return value is also counted. Inherited output variables are not counted.  A high number of output variables is a sign of a violation of the principle of clear responsibility.
Standard upper limit for the associated rule <a href="#">SA0166 [► 75]</a>	10

**Sample:**

```
METHOD METH : BOOL
VAR_OUTPUT
  a : INT;
  b : LREAL;
END_VAR
```

The method METH has three outputs:

- Return value METH
- a
- b

```
METHOD METH1
VAR_OUTPUT
  a : ARRAY[0..10] OF INT;
  b : LREAL;
END_VAR
```

The method METH1 has two outputs:

- a
- b

**Number Of Statements (NOS)**

Title short form	NOS
Categories	Informative
Definition	Number of executable statements in the implementation of a function block, function or method
Further information	NOS = <b>N</b> umber <b>O</b> f executable <b>S</b> tatements Statements in the declaration, empty statements or pragmas are not counted.

**Sample:**

```
FUNCTION_BLOCK FB_Sample
VAR_OUTPUT
  nTest : INT;
  i      : INT;
END_VAR
VAR
  bVar : BOOL;
  c    : INT := 100;    // statements in the declaration are not counted
END_VAR

IF bVar THEN           //if statement: +1
  nTest := 0;          // +1
END_IF
```

```

WHILE nTest = 1 DO           //while statement: +1
;                           // empty statements do not add to the statement count
END_WHILE

FOR c := 0 TO 10 BY 2 DO     //for statement: +1
  i := i+1;                 // +1
END_FOR

{text 'simple text pragma'} //pragmas are not counted
nTest := 2;                 //+1

```

The sample has six statements.

### Percentage of comment

Title short form	Comments
Categories	Maintainability
Definition	<p>Percentage of comments in the source code</p> <p>This number is calculated using the following formula:</p> $\text{Percentage} = 100 * \frac{\text{letters in comments}}{\text{letters in source code and comments together}}$
Further information	<p>Multiple consecutive spaces in the source code are counted as one space, which prevents a high weighting of indented source code. A percentage of 0 is returned for empty objects (no source code and no comments).</p> <p>The statements also include declaration statements, for example.</p>

### Complexity (McCabe)

Title short form	McCabe
Categories	Testability
Definition	<p>Number of binary branches in the control flow of the POU</p> <p>(for example, the number of branches for IF and CASE statements and loops)</p>
Further information	<p>McCabe's cyclomatic complexity is a measure of the readability and testability of source code. It is calculated by counting the number of binary branches in the control flow of the POU. Cyclomatic complexity penalizes high branching because high branching increases the number of test cases required for high test coverage.</p>
Recommended upper limit	10

The following samples show how complexity is calculated according to McCabe.

#### Sample: IF statement

```

// every POU has an initial cyclomatic complexity of 1, since it has at least 1 branch
IF b1 THEN           // +1 for the THEN branch
;
ELSIF b2 THEN         // +1 for the THEN branch of the IF inside the ELSE
;
ELSE
  IF b3 OR b4 THEN    // +1 for the THEN branch
  ;
  END_IF
END_IF

```

The code snippet has a cyclomatic complexity of 4.

#### Sample: CASE statement

```

// every POU has an initial cyclomatic complexity of 1, since it has at least 1 branch
CASE a OF
  1: ;                // +1
  2: ;                // +1

```

```

    3,4,5: ;           // +1
ELSE                // the ELSE statement does not increase the cyclomatic complexity
;
END_CASE

```

The code snippet has a cyclomatic complexity of 4.

### Sample: Loop statement

```

// every POU has an initial cyclomatic complexity of 1, since it has at least 1 branch
WHILE b1 DO          // +1 for the WHILE loop
;
END_WHILE

REPEAT                // +1 for the REPEAT loop
;
UNTIL b2
END_REPEAT

FOR a := 0 TO 100 BY 2 DO // +1 for the REPEAT loop
;
END_FOR

```

The code snippet has a cyclomatic complexity of 4.

### Sample: Other statements

The following statements also lead to an increase in cyclomatic complexity:

```

FUNCTION FUN : STRING
VAR_INPUT
    bReturn : BOOL;
    bJump   : BOOL;
END_VAR

// every POU has an initial cyclomatic complexity of 1, since it has at least 1 branch
JMP(bJump) lbl;           //Conditional jumps increase the cyclomatic complexity by 1

FUN := 'u';
RETURN(condition_return); //Conditional returns increase the cyclomatic complexity by 1, too

lbl:
    FUN := 't';

```

The code snippet has a cyclomatic complexity of 3.

## Cognitive complexity

Title short form	Cognitive complexity
Categories	Maintainability
Definition	Sum of partial complexities resulting, for example, from branches in the control flow of the POU and from sophisticated Boolean expressions
Further information	Cognitive complexity is a measure of the readability and comprehensibility of source code introduced by Sonarsource™ in 2016. It penalizes a strong nesting of the control flow and sophisticated Boolean expressions. The cognitive complexity is only calculated for structured text implementations.
Standard upper limit for the associated rule	20
SA0178 <a href="#">▶ 80</a>	



### Tip

You can also use [Command 'Show cognitive complexity for current editor' \[▶ 119\]](#) to display the increments for Structured Text directly in the editor.

The following samples show how cognitive complexity is calculated.

### Sample: Control flow

Statements that manipulate the control flow increase the cognitive complexity by 1.

```
IF TRUE THEN                //+1 cognitive complexity
;
END_IF

WHILE TRUE DO               //+1 cognitive complexity
;
END_WHILE

FOR i := 0 TO 10 BY 1 DO    //+1 cognitive complexity
;
END_FOR

REPEAT                      //+1 cognitive complexity
;
UNTIL TRUE
END_REPEAT
```

The code snippet has a cognitive complexity of 4.

### Sample: Nesting of the control flow

When nesting the control flow, an increment of 1 is added for each level of nesting.

```
IF TRUE THEN                //+1 cognitive complexity
    WHILE TRUE DO           //+2 (+1 for the loop itself, +1 for the nesting inside the IF)
        FOR i := 0 TO 10 BY 1 DO //+3 (+1 for the FOR loop itself, +2 for the nesting inside the
            WHILE and the IF)
                ;
            END_FOR
        END_WHILE

        REPEAT               //+2 (+1 for the loop itself, +1 for the nesting inside the IF)
            ;
        UNTIL TRUE
        END_REPEAT
    END_IF
```

The code snippet has a cognitive complexity of 8.

### Sample: Boolean expression

Since Boolean expressions play a major role in understanding source code, they are also taken into account when calculating cognitive complexity.

Understanding Boolean expressions that are connected with the same Boolean operator is not as difficult as understanding a Boolean expression that contains alternating Boolean operators. Therefore, each chain of equal Boolean operators in an expression increases the cognitive complexity.

```
b := b1;                    //+0: a simple expression, containing no operators, has no
increment
```

The simple expression without operator has an increment of 0.

```
b := b1 AND b2;             //+1: one chain of AND operators
```

The expression with an AND operation has an increment of 1.

```
b := b1 AND b2 AND b3;      //+1: one more AND, but the number of chains of operators does
not change
```

The expression has an AND more. But since it is the same operator, the number of chains formed with identical operators does not change.

```
b := b1 AND b2 OR b3;       //+2: one chain of AND operators and one chain of OR operators
```

The expression has a chain of AND operators and a chain of OR operators. This results in an increment of 2.

```
b := b1 AND b2 OR b3 AND b4 AND b5; //+3: one chain of AND operators, one chain of OR operators and another chain of AND operators
```

The code snippet has an increment of 3.

```
b := b1 AND NOT b2 AND b3; //+1: the unary NOT operator is not considered in the cognitive complexity
```

The unary operator NOT is not taken into account in the cognitive complexity.

### Sample: Further statements with increment

Structured Text has additional statements and expressions that change the control flow.

The following statements are penalized with an increment in cognitive complexity:

```
aNewLabel:
  x := MUX(i, a,b,c); //+1 for MUX operator
  y := SEL(b, i,j); //+1 for SEL operator
JMP aNewLabel; //+1 for JMP to label
```

EXIT and RETURN statements do not increase cognitive complexity.

### Depth of Inheritance Tree (DIT)

Title short form	DIT
Categories	Maintainability
Definition	Number of inheritances until a function block is reached that does not extend any other function block
Further information	DIT = <b>Depth of Inheritance Tree</b>

#### Sample:

```
FUNCTION_BLOCK FB_Base
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
FUNCTION_BLOCK FB_SubSub EXTENDS FB_Sub
```

The metric **Depth of Inheritance Tree** is:

- For FB\_Base: 0, as it is itself a function block that does not extend any other function block.
- For FB\_Sub: 1, as one step is required to get to FB\_Base.
- For FB\_SubSub: 2, as one step to FB\_Sub and another to FB\_Base is required.

### Number Of Children (NOC)

Title short form	NOC
Categories	Reusability, maintainability
Definition	Number of function blocks that extend the given basic function block. Function blocks that indirectly extend a basic function block are not counted.
Further information	NOC = <b>Number Of Children</b>

#### Sample:

```
FUNCTION_BLOCK FB_Base
```

```

FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
FUNCTION_BLOCK FB_SubSub1 EXTENDS FB_Sub
FUNCTION_BLOCK FB_SubSub2 EXTENDS FB_Sub

```

The metric **Number Of Children** is:

- For FB\_Base: 1 child (FB\_Sub)
- For FB\_Sub: 2 children (FB\_SubSub1, FB\_SubSub2)
- For FB\_SubSub1: 0 children
- For FB\_SubSub2: 0 children

## Response For Class (RFC)

Title short form	RFC
Categories	Maintainability, reusability
Definition	Number of different POU's, methods or actions that can be called by a POU
Further information	<p>RFC = <b>R</b>esponse <b>F</b>or <b>C</b>lass</p> <p>The value is used for measuring the complexity (in terms of testability and maintainability). All possible direct and indirect method calls can be reached via associations are taken into account. These can be used to respond to an incoming message or to respond to an event that has occurred.</p>

### Sample:

#### Function block FB1:

```

FUNCTION_BLOCK FB1
VAR
    d,x,y : INT;
END_VAR
x := METH(d+10);
y := FUN(42, 0.815);

```

#### Method FB1.METH:

```

METHOD METH : INT
VAR_INPUT
    i : INT;
END_VAR
METH := FUN(CUBE(i), 3.1415);

```

#### Function Cube:

```

FUNCTION CUBE : INT
VAR_INPUT
    i : INT;
END_VAR
CUBE := i*i*i;

```

#### Function FUN:

```

FUNCTION FUN : INT
VAR_INPUT
    a : INT;
    f : LREAL;
END_VAR
FUN := LREAL_TO_INT(f*10)*a;

```

- FUN, CUBE: These functions have an RFC of 0, as neither function calls other functions, function blocks or methods for their calculations.
- FB1.METH: The method uses FUN and CUBE, which results in an RFC of 2.
- FB1:
  - The function block FB1 calls METH and FUN, which increases its RFC by 2.



- For FB1, its method METH must also be taken into account. METH uses FUN and CUBE. FUN has already been added to the RFC of FB1 (see previous point). Thus, only the use of CUBE in METH increases the RFC for FB1 to 3.

### Coupling Between Objects (CBO)

Title short form	CBO
Categories	Maintainability, reusability
Definition	Number of additional function blocks that are instantiated and used in a function block
Further information	CBO = <b>Coupling Between Objects</b> A function block with a high level of coupling between objects is likely to be involved in many different tasks and therefore violates the principle of clear responsibility.
Standard upper limit for the associated rule SA0179 <a href="#">▶ 80</a>	30

#### Sample:

```
FUNCTION_BLOCK FB_Base
VAR
    fb3 : FB3; // +1 instantiated here
END_VAR

FUNCTION_BLOCK FB_Sub EXTENDS FB_Base // +0 for EXTENDS
VAR
    fb1 : FB1; // +1: instantiated here
    fb2 : FB2; // +1: instantiated here
END_VAR

fb3(); // +0: instantiated in FB_Base, no increment for call
```

- The extension of a function block does not increase the coupling between objects.
- fb3 is instantiated in the implementation of FB\_Base and inherited by FB\_Sub. The call in FB\_Sub does not increase the coupling between the objects.
- The metric **Coupling Between Objects** for FB\_Sub is therefore : 2

### Complexity of reference (Elshof)

Complexity of reference = referenced data (number of variables) / number of data references

Title short form	Elshof
Categories	Efficiency, maintainability, reusability
Definition	Complexity of the data flow of a POU The complexity of reference is calculated using the following formula: <i>&lt;Number of variables used&gt; / &lt;Number of variable accesses&gt;</i>
Further information	Only variable accesses in the implementation part of the POU are taken into account.

#### Sample:

```
PROGRAM MAIN
VAR
    i, j : INT;
    k : INT := GVL.m;
    b, c : BOOL;
    fb : FB_Sample;
END_VAR

fb(paramA := b); // +3 accesses (fb, paramA and b)
i := j; // +2 accesses (i and j)
j := GVL.d; // +2 accesses (j and GVL.d)
```

For the metric **Complexity of reference (Elshof)**, MAIN:

- Number of variables used = 6
- Number of variable accesses = 7
- Complexity of reference (Elshof) = number of variables used/number of variable accesses = 6/7 = 0.85

**Attention:**

- c and k are not used and therefore do not count as "variables used".
- The assignment `k : INT := GVL.m;` is not counted as it is part of the declaration of the program.

### Lack of Cohesion Of Methods - LCOM

Title short form	LCOM
Categories	Maintainability, reusability
Definition	<p>Cohesion = pairs of methods without common instance variables minus pairs of methods with common instance variables</p> <p>The metric is calculated using the following formula:</p> <p><i>MAX(0, &lt;number of object pairs without cohesion&gt; - &lt;number of object pairs with cohesion&gt; )</i></p>
Further information	<p>LCOM: <b>L</b>ack of <b>C</b>ohesion of <b>M</b>ethods</p> <p>The cohesion between function blocks, their actions, transitions and methods describes whether they access the same variables.</p> <p>The lack of cohesion of methods describes how strongly the objects of a function block are connected to each other. The lower the lack of cohesion, the stronger the connection between the objects.</p> <p>Function blocks with a high lack of cohesion are likely to be involved in many different tasks and therefore violate the principle of unambiguous responsibility.</p>

**Sample:**

Function block FB:

```
FUNCTION_BLOCK FB
VAR_INPUT
    a    : BOOL;
END_VAR
VAR
    i, b : BOOL;
END_VAR
```

Action FB.ACT:

```
i := FALSE;
```

Method FB.METH:

```
METHOD METH : BOOL
VAR_INPUT
    c    : BOOL;
END_VAR
METH := c;
i := TRUE;
```

Method FB.METH2:

```
METHOD METH2 : INT
VAR_INPUT
END_VAR
METH2 := SEL(b, 3, 4);
```

For the metric **Lack of Cohesion Of Methods (LCOM)** , the result for FB:

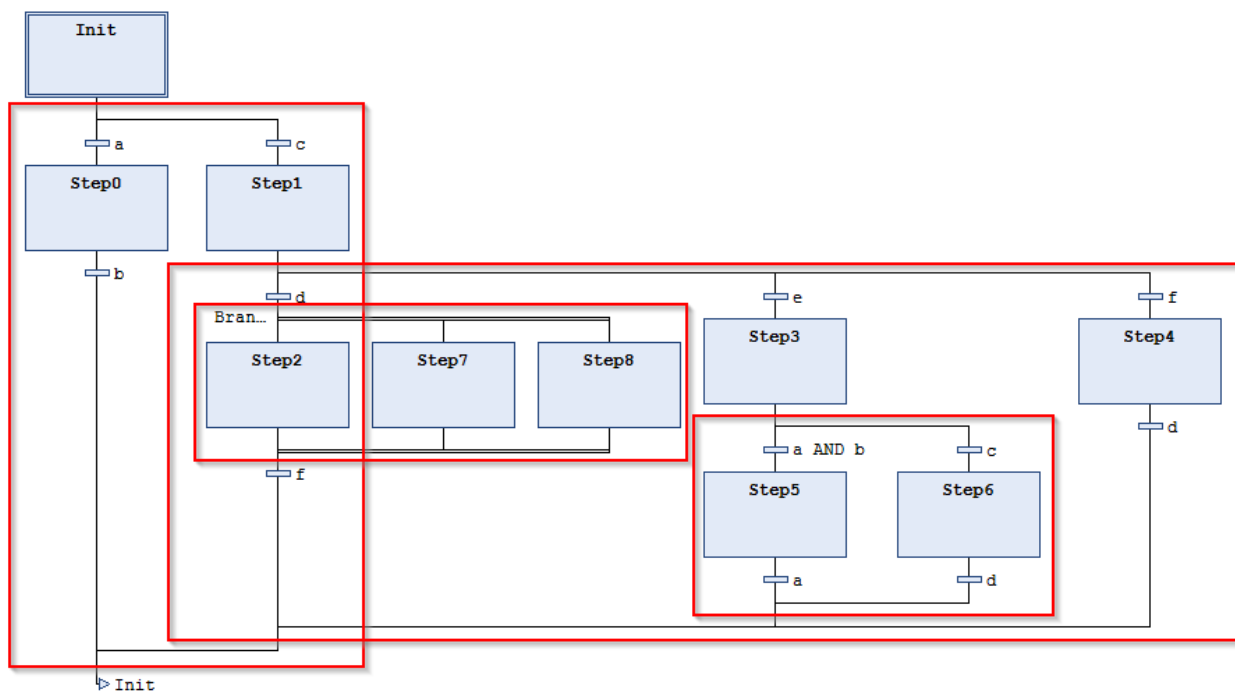
- Object pairs without cohesion (5 pairs):
  - FB, FB.ACT

- FB, FB.METH
- FB, FB.METH2
- FB.ACT, FB.METH2
- FB.METH, FB.METH2
- Object pairs with cohesion (1 pair):
  - FB.ACT, FB.METH (both use i)
- LCOM = number of object pairs without cohesion - number of object pairs with cohesion = 5 - 1 = 4

### Number of SFC branches

Title short form	SFC branches
Categories	Testability, maintainability
Definition	Number of alternative and parallel branches of a POU of the implementation language SFC (Sequential Function Chart)

### Sample:

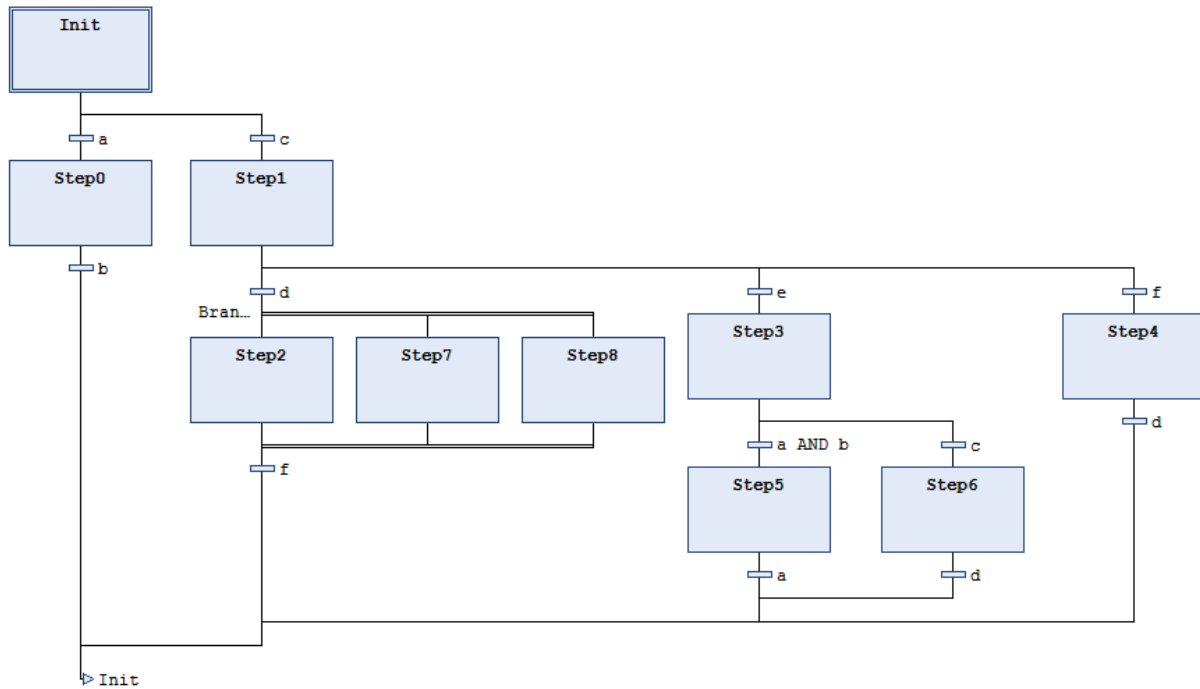


The above code snippet in SFC has 4 branches: 3 alternative and 1 parallel branch.

### Number of SFC steps

If the function block is implemented in Sequential Function Chart (SFC), this code metric indicates the number of steps in the function block.

Title short form	SFC steps
Categories	Maintainability
Definition	Number of steps in a POU of the implementation language SFC (Sequential Function Chart)
Further information	Only the steps contained in the POU programmed in SFC are counted. Steps in the implementations of actions or transitions called in POUs are not counted.

**Sample:**

The above code snippet in SFC has 10 steps.

**Metrics that are available in TwinCAT versions < 3.1.4026.14:**

Column abbreviation in Standard Metrics view	Description
Prather	<a href="#">Complexity of nesting (Prather) [► 108]</a>
n1 (Halstead)	<a href="#">Halstead – number of different used operators (n1) [► 108]</a>
N1 (Halstead)	<a href="#">Halstead – number of operators (N1) [► 108]</a>
n2 (Halstead)	<a href="#">Halstead – number of different used operands (n2) [► 108]</a>
N2 (Halstead)	<a href="#">Halstead – number of operands (N2) [► 108]</a>
HL (Halstead)	<a href="#">Halstead – length (HL) [► 108]</a>
HV (Halstead)	<a href="#">Halstead – volume (HV) [► 108]</a>
D (Halstead)	<a href="#">Halstead – difficulty (D) [► 108]</a>

**Complexity of nesting (Prather)**

Nesting weight = statements \* nesting depth

Complexity of nesting = nesting weight / number statements

Nesting through IF/ELSEIF or CASE/ELSE statements, for example.

**Halstead (n1, N1, n2, N2, HL, HV, D)**

The following metrics are part of the "Halstead" range:

- Number of different used operators - Halstead (n1)
- Number of operators - Halstead (N1)
- Number of different used operands - Halstead (n2)
- Number of operands - Halstead (N2)
- Length - Halstead (HL)
- Volume - Halstead (HV)
- Difficulty - Halstead (D)

Background information:

- Relationship between operators and operands (number, complexity, test effort)
- Based on the assumption that executable programs consist of operators and operands.
- Operands in TwinCAT: variables, constants, components, literals and IEC addresses.
- Operators in TwinCAT: keywords, logical and comparison operators, assignments, IF, FOR, BY, ^, ELSE, CASE, Caselabel, BREAK, RETURN, SIN, +, labels, calls, pragmas, conversions, SUPER, THIS, index access, component access, etc.

For each program the following basic parameters are formed:

- **Number of different operators used - Halstead (n1),  
Number of different operands used - Halstead (n2):**
  - Number of different used operators ( $h_1$ ) and operands ( $h_2$ ); together they form the vocabulary size  $h$ .
- **Number of operators - Halstead (N1),  
Number of operands - Halstead (N2):**
  - Number of total used operators ( $N_1$ ) and operands ( $N_2$ ); together they form the implementation class  $N$ .
- (Language complexity = operators/operator occurrences \* operands/operand occurrences)

These parameters are used to calculate the Halstead length (HL) and Halstead volume (HV):

- **Length - Halstead (HL),  
Volume - Halstead (HV):**
  - $HL = h_1 * \log_2 h_1 + h_2 * \log_2 h_2$
  - $HV = N * \log_2 h$

Various key figures are calculated from the basic parameters:

- **Difficulty - Halstead (D):**
  - Describes the difficulty to write or understand a program (during a code review, for example)
  - $D = h_1/2 * N_2/h_2$
- Effort:
  - $E = D * V$

The key figures usually match the actual measured values very well. The disadvantage is that the method only applies to individual functions and only measures lexical/textual complexity.



## 5 Commands

### 5.1 Command 'Run static analysis'

**Symbol:** 

**Function:** The command starts the static code analysis for the currently active PLC project and outputs the results in the message window.

**Call:** **Build** menu or context menu of the PLC project object

During execution of the static analysis, compliance with the coding rules, naming conventions and forbidden symbols is checked. This command can be used to trigger a static analysis manually (explicit execution), or the analysis can be performed automatically during code generation (implicit execution, see below for more information).

TwinCAT issues the result of the static analysis, i.e. messages relating to rule violations, in the message window. The [rules](#) [► 16], [naming conventions](#) [► 81] and [forbidden symbols](#) [► 110] to be taken into account in the static analysis can be [configured](#) [► 14] in the PLC project properties. You can also define whether the violation of a coding rule should appear as an error or a warning in the message window (see: [Rules](#) [► 16]).

See also: [Syntax in the message window](#) [► 112]



Please note that the selected PLC project is created before this command is executed. Checking via the static analysis is only started if the code generation was successful, i.e. if the compiler did not detect any compilation errors.

Please also note the [Command 'Run static analysis \[Check all objects\]'](#) [► 113] and the differences between the two commands described in the following table.

Differences	'Run static analysis' command	'Run static analysis [Check all objects]' command
Scope of application/ mode of operation	<b>Objects used:</b> The activated rules are applied to the objects that are used in the PLC project. <b>Unused objects:</b> Unused objects are not checked with this command.	<b>Objects used:</b> The activated rules are applied to the objects that are used in the PLC project. <b>Unused objects:</b> The rules that are activated and that can be checked in the precompile are applied to the unused objects. See also: <a href="#">QuickFix/Precompile [► 130]</a>
Note	If you also wish to have the unused objects checked by the static analysis, you can use the <b>'Run static analysis [check all objects]' command</b> .	The command is primarily useful when creating libraries or when processing library projects.
Execution options for the command	Static analysis can be performed either explicitly using the command or implicitly. Implicit execution of the static analysis during each code generation can be enabled or disabled in the PLC project properties ( <a href="#">Settings [► 14]</a> tab). If the <b>Perform static analysis automatically</b> option is enabled, TwinCAT performs the static analysis after each successful code generation (with the <b>Build project</b> command, for example). The command can also be called up via the Automation Interface. See also: <a href="#">Automation Interface support [► 132]</a>	The "Check all objects" variant cannot be executed implicitly. It can only be executed explicitly via the command. The command can also be called up via the Automation Interface. See also: <a href="#">Automation Interface support [► 132]</a>

### 5.1.1 Syntax in the message window

#### Syntax of rule violations in the message window

Each rule has a unique number (shown in parentheses after the rule in the rule configuration view). If a rule violation is detected during the static analysis, the number together with an error or warning description is issued in the message window, based on the following syntax. The abbreviation "SA" stands for "Static Analysis".

Syntax: **"SA<rule number>: <rule description>"**

Sample for rule number 33 (unused variables): "SA0033: Not used: variable 'bSample'"

#### Syntax of convention violations in the message window

Each naming convention has a unique number (shown in parentheses after the convention in the naming convention configuration view). If a violation of a convention or a preset is detected during the static analysis, the number is output in the error list together with an error description based on the following syntax. The abbreviation "NC" stands for "Naming Convention".

Syntax: **"NC<prefix convention number>: <convention description>"**

Sample for convention number 151 (DUTs of type Structure): "NC0151: Invalid type name 'STR\_Sample'. Expected prefix 'ST\_'"

#### Syntax of symbol violations in the message window

If a symbol is used in the code that is configured as a forbidden symbol, an error is issued in the message window after the static analysis has been performed.



Syntax: **"Forbidden symbol '<symbol>'"**

Sample for the symbol XOR: "Forbidden symbol 'XOR'"

## 5.2 Command 'Run static analysis [Check all objects]'

Symbol: 

**Function:** The command starts the static code analysis for all objects of the currently active PLC project and outputs the results in the message window.

**Call:** **Build** menu or context menu of the PLC project object

During execution of the static analysis, compliance with the coding rules, naming conventions and forbidden symbols is checked. This command can be used to trigger the static analysis manually (explicit execution).

TwinCAT issues the result of the static analysis, i.e. messages relating to rule violations, in the message window. The [rules](#) [► 16], [naming conventions](#) [► 81] and [forbidden symbols](#) [► 110] to be taken into account in the static analysis can be [configured](#) [► 14] in the PLC project properties. You can also define whether the violation of a coding rule should appear as an error or a warning in the message window (see: [Rules](#) [► 16]).

See also: [Syntax in the message window](#) [► 112]



Please note that the selected PLC project is created before this command is executed. Checking via the static analysis is only started if the code generation was successful, i.e. if the compiler did not detect any compilation errors.

Please also note the [Command 'Run static analysis'](#) [► 111] and the differences between the two commands described in the following table.

Differences	'Run static analysis' command	'Run static analysis [Check all objects]' command
Scope of application/ mode of operation	<b>Objects used:</b> The activated rules are applied to the objects that are used in the PLC project. <b>Unused objects:</b> Unused objects are not checked with this command.	<b>Objects used:</b> The activated rules are applied to the objects that are used in the PLC project. <b>Unused objects:</b> The rules that are activated and that can be checked in the precompile are applied to the unused objects. See also: <a href="#">QuickFix/Precompile [► 130]</a>
Note	If you also wish to have the unused objects checked by the static analysis, you can use the <b>'Run static analysis [check all objects]' command</b> .	The command is primarily useful when creating libraries or when processing library projects.
Execution options for the command	Static analysis can be performed either explicitly using the command or implicitly. Implicit execution of the static analysis during each code generation can be enabled or disabled in the PLC project properties ( <a href="#">Settings [► 14]</a> tab). If the <b>Perform static analysis automatically</b> option is enabled, TwinCAT performs the static analysis after each successful code generation (with the <b>Build project</b> command, for example). The command can also be called up via the Automation Interface. See also: <a href="#">Automation Interface support [► 132]</a>	The "Check all objects" variant cannot be executed implicitly. It can only be executed explicitly via the command. The command can also be called up via the Automation Interface. See also: <a href="#">Automation Interface support [► 132]</a>

## 5.3 Command 'View Standard Metrics'

**Symbol:** 

**Function:** The command starts the static metric code analysis for the currently active PLC project and represents the metrics for the programming blocks used in a table.

**Call:** **Build** menu or context menu of the PLC project object

The command starts the code generation for the selected PLC project (with the command **Build project**, for example). In a tabular view, **Standard Metrics**, TwinCAT then displays the desired metrics (parameters) for each programming block used. The metrics to be displayed are activated in the project properties (see [Configuration of the metrics \[► 94\]](#)).

If a value is outside the range defined by a lower and/or upper limit in the configuration, the table cell is shown in red.

The table can be sorted by columns by clicking on the respective column header.



Please note that the selected PLC project is created before this command is executed. Creation of the standard metrics is only started if the code generation was successful, i.e. if the compiler did not detect any compilation errors.

Please also note the [Command 'View Standard Metrics \[Check all objects\]' \[► 116\]](#) and the differences between the two commands are described in the following table.

Differences	Command 'View Standard Metrics'	Command 'View Standard Metrics [Check all objects]'
Scope	<p>The standard metrics are created for the objects used in the PLC project. Objects that are not used are not considered with this command.</p> <p>The scope of this command thus corresponds to the build commands <b>Build Project/Solution</b> or <b>Rebuild Project/Solution</b> respectively.</p> <p>If you want to create default metrics for unused objects, which is useful when editing library projects, you can use the <b>command 'View Standard Metrics [Check all objects]'</b>.</p>	<p>The standard metrics are created for all objects located in the project tree of the PLC project.</p> <p>This is primarily useful when creating libraries or when processing library projects.</p> <p>The scope of this command thus corresponds to the build command <b>Check all objects</b>.</p>

### 5.3.1 Commands in the context menu of the 'Standard Metrics' view

Right-click in the **Standard Metrics** view to open a context menu that offers several commands.

The context menu offers options for updating, printing or exporting the metrics table, or to copy to the clipboard. Via the context menu you can also navigate to a view for configuring the metrics – just like in the PLC project properties. In addition, you can generate a Kiviat diagram for the selected function blocks or open the block in the corresponding editor. A prerequisite for generating a Kiviat diagram is that at least three metrics are configured with a defined value range (lower and upper limit).

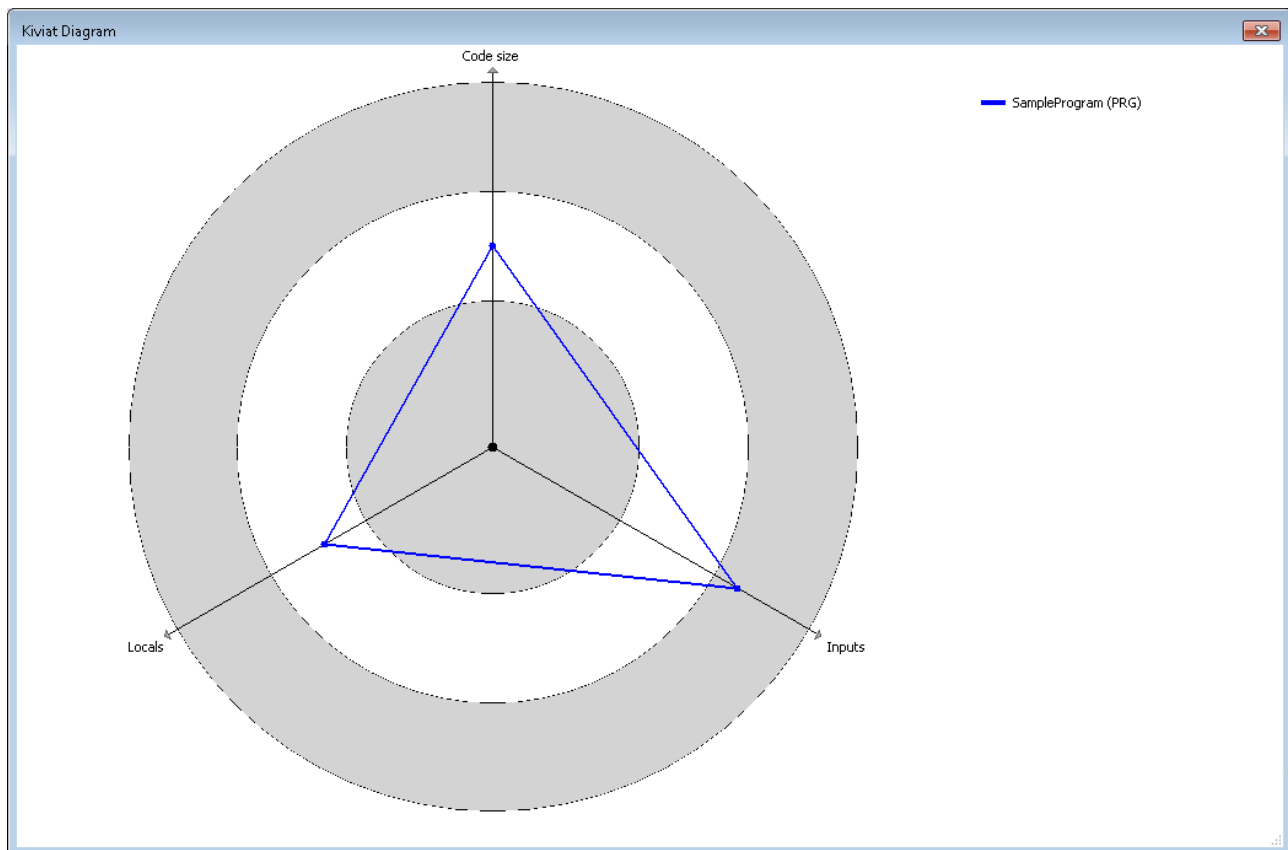
The following commands are available:

- **Calculate:** The values are updated.
- **Print table:** The standard dialog for setting up the print job appears.
- **Copy table:** The data are copied to the clipboard, separated by tabs. From there you can paste the table directly in a spreadsheet or a word processor.
- **Export table:** The data are exported into a text file (\*.csv), separated by semicolons.
- **Kiviat diagram:** A radar chart is created for the selected function block. This is a graphical representation of the function blocks, for which the metrics define a lower and upper limit. It is used to visualize how well the code for the programming unit matches a particular standard. Each metric is shown as an axis in a circle, which starts in the center (value 0) and runs through three ring zones. The inner ring zone represents the range below the lower limit defined for the metric, the outer ring zone represents the range above the upper limit. The axes for the respective metrics are distributed evenly around the circle.  
The current values for the individual metrics on their axes are linked with lines. Ideally, the whole line should be within the central ring zone.

#### **i** Prerequisite for using a Kiviat diagram

At least three metrics with a define value range must be configured.

The following diagram shows an example for 3 metrics with defined ranges (the name of the metric is shown at the end of each axis, the name of the function block at the top right):



- **Configure:** A table opens in which the metrics can be configured. The view, functionality and settings correspond to the [metrics configuration](#) [► 94] in the PLC project properties. If you make a change in this table, it is automatically applied to the PLC project properties.
- **Open POU:** The programming block opens in the corresponding editor.

## 5.4 Command 'View Standard Metrics [Check all objects]'

**Symbol:**

**Function:** The command starts the static metric code analysis for the currently active PLC project and displays the metrics for all programming blocks in a table.

**Call:** Menu **Build** or context menu of the PLC project object

The command starts the code generation for the selected PLC project (with the command **Build project**, for example). TwinCAT shows the selected metrics for each programming block in the tabular **Standard Metrics** view. The metrics to be displayed are activated in the project properties (see [Configuration of the metrics](#) [► 94]).

If a value is outside the range defined by a lower and/or upper limit in the configuration, the table cell is shown in red.

The table can be sorted by columns by clicking on the respective column header.

**i** Please note that the selected PLC project is built before this command is executed. Creation of the standard metrics is only started if the code generation was successful, i.e. if the compiler did not detect any compilation errors.

### **i** Calculation of the "Code size" metric not possible using this command

The calculation of the [Code size](#) [number of bytes] [► 96] metric is only possible via the 'View Standard Metrics' command [► 114]. When executing the **View Standard Metrics [Check all objects]** command, the **Code size** field remains empty.

Please also note the [command 'View Standard Metrics' \[► 114\]](#) and the differences between the two commands, which are described in the following table.

Differences	Command 'View Standard Metrics'	Command 'View Standard Metrics [Check all objects]'
Scope	<p>The standard metrics are created for the objects used in the PLC project. Objects that are not used are not considered with this command.</p> <p>The scope of this command thus corresponds to the build commands <b>Build Project/Solution</b> or <b>Rebuild Project/Solution</b> respectively.</p> <p>If you want to create default metrics for unused objects, which is useful when editing library projects, you can use the <b>command 'View Standard Metrics [Check all objects]'</b>.</p>	<p>The standard metrics are created for all objects located in the project tree of the PLC project.</p> <p>This is primarily useful when creating libraries or when processing library projects.</p> <p>The scope of this command thus corresponds to the build command <b>Check all objects</b>.</p>

### 5.4.1 Commands in the context menu of the 'Standard Metrics' view

Right-click in the **Standard Metrics** view to open a context menu that offers several commands.

The context menu offers options for updating, printing or exporting the metrics table, or to copy to the clipboard. Via the context menu you can also navigate to a view for configuring the metrics – just like in the PLC project properties. In addition, you can generate a Kiviat diagram for the selected function blocks or open the block in the corresponding editor. A prerequisite for generating a Kiviat diagram is that at least three metrics are configured with a defined value range (lower and upper limit).

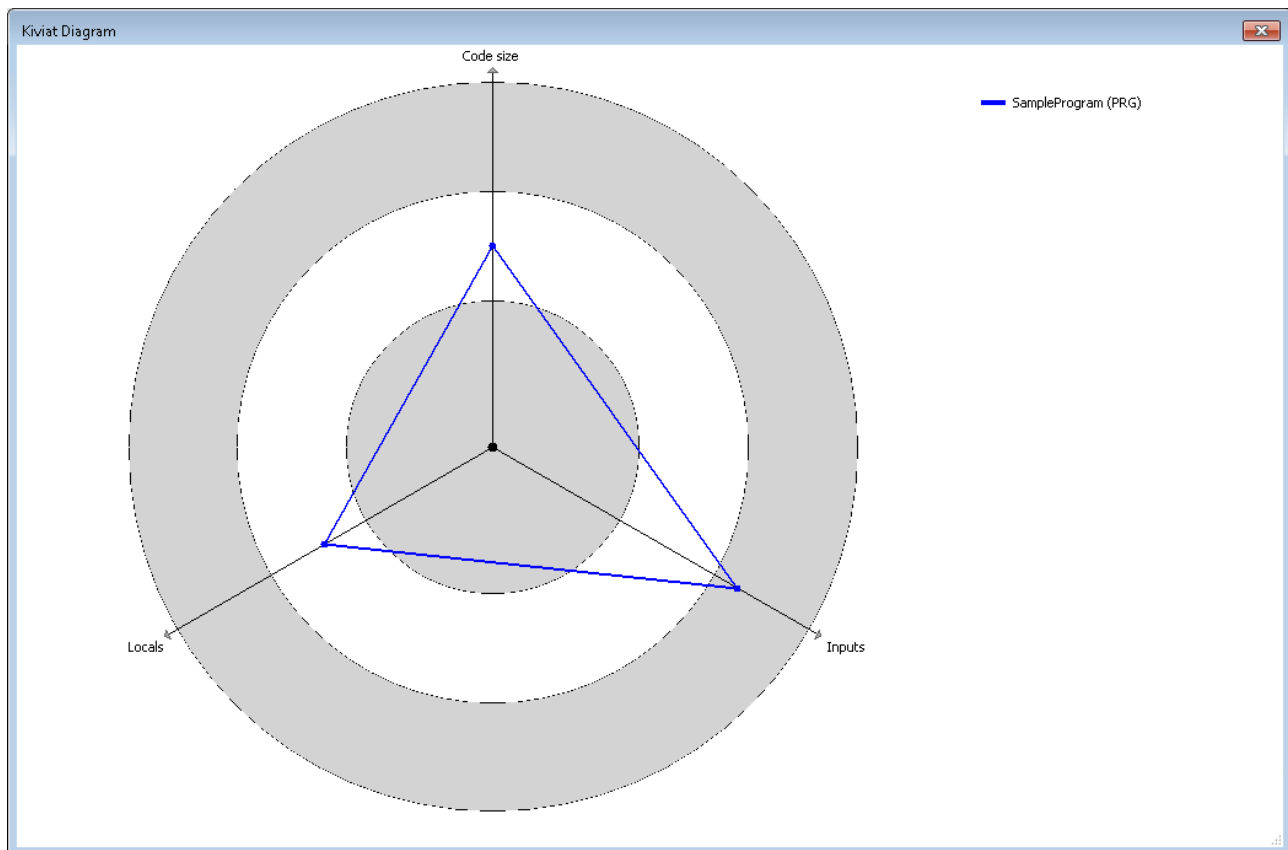
The following commands are available:

- **Calculate:** The values are updated.
- **Print table:** The standard dialog for setting up the print job appears.
- **Copy table:** The data are copied to the clipboard, separated by tabs. From there you can paste the table directly in a spreadsheet or a word processor.
- **Export table:** The data are exported into a text file (\*.csv), separated by semicolons.
- **Kiviat diagram:** A radar chart is created for the selected function block. This is a graphical representation of the function blocks, for which the metrics define a lower and upper limit. It is used to visualize how well the code for the programming unit matches a particular standard. Each metric is shown as an axis in a circle, which starts in the center (value 0) and runs through three ring zones. The inner ring zone represents the range below the lower limit defined for the metric, the outer ring zone represents the range above the upper limit. The axes for the respective metrics are distributed evenly around the circle. The current values for the individual metrics on their axes are linked with lines. Ideally, the whole line should be within the central ring zone.

#### ● Prerequisite for using a Kiviat diagram

**i** At least three metrics with a define value range must be configured.

The following diagram shows an example for 3 metrics with defined ranges (the name of the metric is shown at the end of each axis, the name of the function block at the top right):



- **Configure:** A table opens in which the metrics can be configured. The view, functionality and settings correspond to the [metrics configuration](#) [► 94] in the PLC project properties. If you make a change in this table, it is automatically applied to the PLC project properties.
- **Open POU:** The programming block opens in the corresponding editor.

## 5.5 'Show constant propagation values for current editor' command



Available from TwinCAT 3.1.4026.14

**Symbol:**

**Function:** The command starts the static code analysis and calculates a measured value for the constant propagation of the code in the current editor. The dialog that opens visualizes the result. The analyzed code is listed and the determined measured values are displayed.

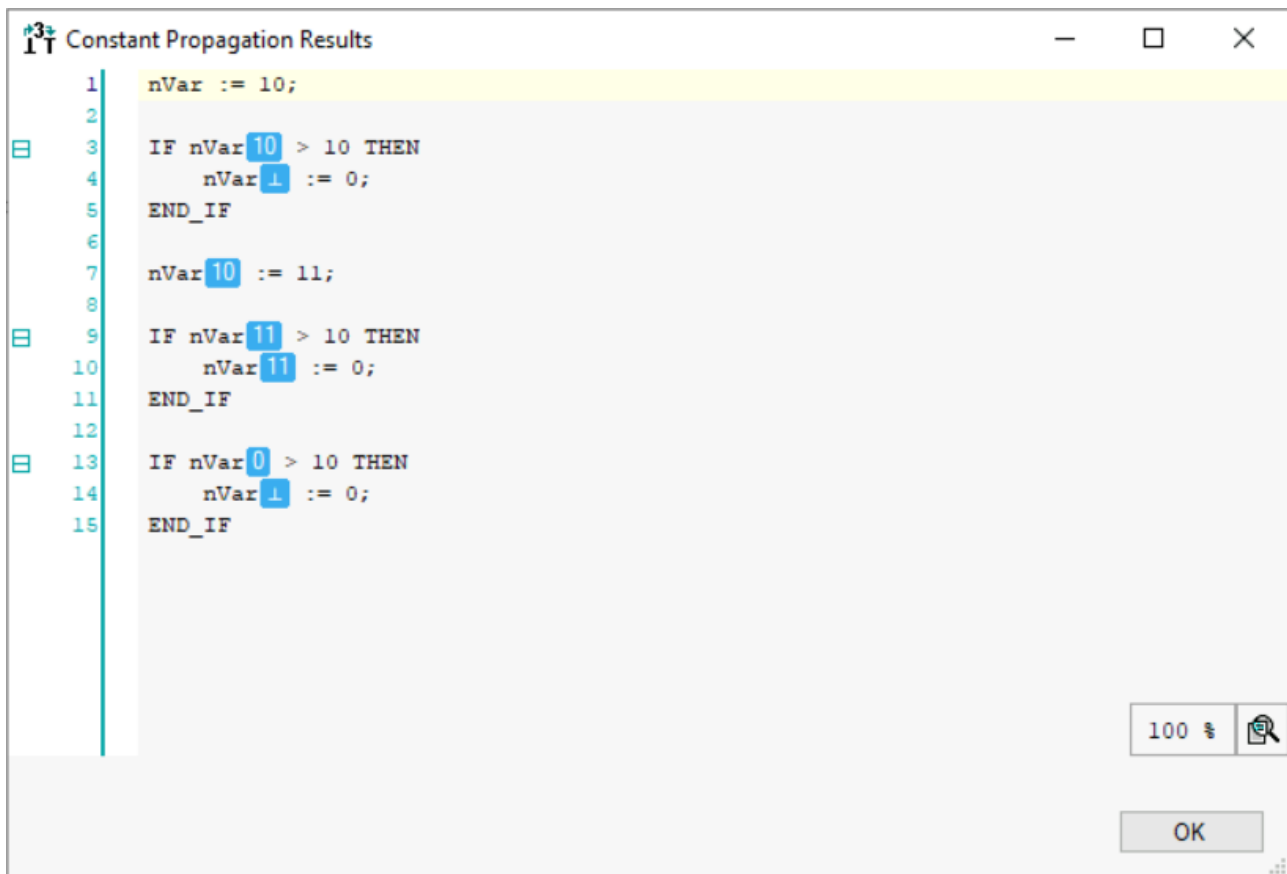
**Call:** **Build** menu or context menu of the ST editor

**Requirement:** A programming object in the ST implementation language is open in the editor.

For more information, see: [Constant propagation](#) [► 126]

**Dialog:** **Results of constant propagation**

Sample:



## 5.6 Command 'Show cognitive complexity for current editor'



Available from TwinCAT 3.1.4026.14

Symbol:

**Function:** The command starts the static code analysis and calculates an increment for the cognitive complexity of the code in the current editor. The dialog that opens visualizes the result and specifies the measured value sum in the title. The analyzed code is listed and displayed with the detected complexities.

**Call:** **Build** menu or context menu of the ST editor

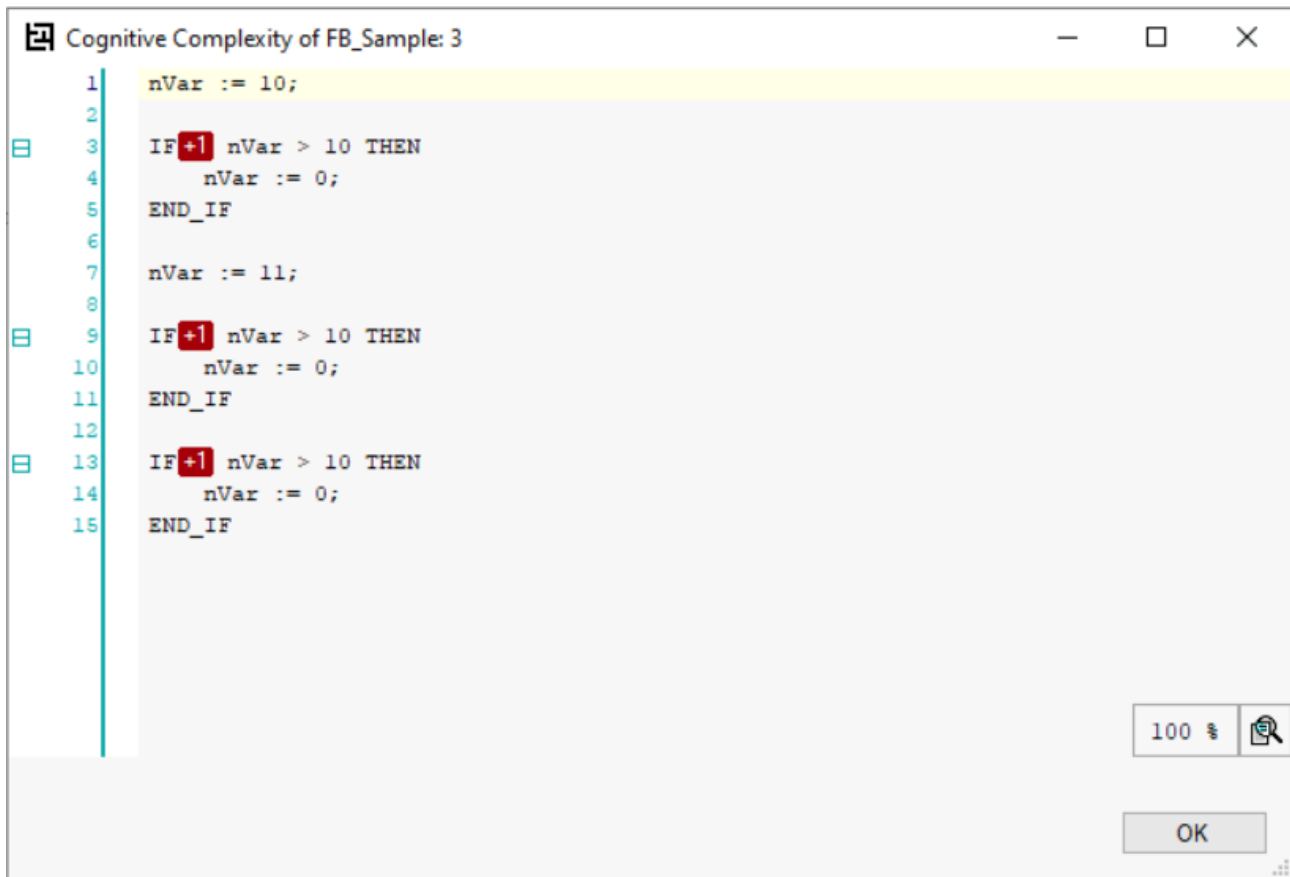
**Requirement:** A programming object in the ST implementation language is open in the editor.

For more information, see also the documentation of the [Cognitive Complexity \[► 101\]](#) metric.

**Dialog:** **Cognitive complexity**

**Sample:**

"Cognitive complexity of FB\_Sample: 3"



Cognitive Complexity of FB\_Sample: 3

```
1  nVar := 10;
2
3  IF +1 nVar > 10 THEN
4      nVar := 0;
5  END_IF
6
7  nVar := 11;
8
9  IF +1 nVar > 10 THEN
10     nVar := 0;
11 END_IF
12
13 IF +1 nVar > 10 THEN
14     nVar := 0;
15 END_IF
```

100 % 🔍

OK



## 6 Pragmas and attributes

A pragma and various attributes are available to temporarily disable individual rules or naming conventions for the static analysis, i.e. to exclude certain code lines or program units from the evaluation.

Requirement: The rules or conventions are enabled or defined in the PLC-project properties. See also:

- [Rules \[► 16\]](#)
- [Naming conventions \[► 81\]](#)

Attributes are inserted in the declaration part of a programming block in order to deactivate certain rules for a complete programming object.

Pragmas are used in the implementation part of a programming block in order to deactivate certain rules for individual code lines. The exception to this is rule SA0164, which can also be deactivated in the declaration part by a pragma.



Rules that are disabled in the project properties cannot be activated by a pragma or attribute.

---



Rule SA0004 cannot be disabled by a pragma or an attribute.

---



### Pragmas in the implementation editor

If you want to use a pragma in the implementation editor, this is currently possible in the ST and FBD/LD/IL editors.

In FBD/LD/IL the desired pragma must be entered in a label.

---

The following section provides an overview and a detailed description of the available pragmas and attributes.

### Overview

- [Pragma {analysis ...} \[► 122\]](#)
  - for disabling coding rules in the implementation part
  - can be used for individual code lines
- [Attribute {attribute 'no-analysis'} \[► 122\]](#)
  - for excluding programming objects (e.g. POU, GVL, DUT) from the static analysis (coding rules, naming conventions, forbidden symbols)
  - can only be used for whole programming objects
- [Attribute {attribute 'analysis' := '...'} \[► 123\]](#)
  - for disabling coding rules in the declaration part
  - can be used for individual declarations or for whole programming objects
- [Attribute {attribute 'naming' := '...'} \[► 123\]](#)
  - for disabling naming conventions in the declaration part
  - can be used for individual declarations or for whole programming objects
- [Attribute {attribute 'nameprefix' := '...'} \[► 124\]](#)
  - for defining prefixes for instances of a structured data type
  - can be used in the declaration part of a structured data type
- [Attribute {attribute 'analysis:report-multiple-instance-calls'} \[► 125\]](#)
  - for specifying that a function block instance should only be called once

- can be used in the declaration part of a function block

## Detailed description

### Pragma {analysis ...}

You can use the pragma {analysis -/+<rule number>} in the implementation part of a programming block in order to disregard individual coding rules for the following code lines. Coding rules are deactivated by specifying the rule numbers preceded by a minus sign ("-"). For activation they are preceded by a plus sign ("+" ). You can specify any number of rules in the pragma with the help of comma separation.

#### Insertion location:

- Deactivation of rules: In the implementation part of the first code line from which the code analysis is disabled with {analysis - ...}.
- Activation of rules: After the last line of the deactivation with {analysis + ...}.
- For rule SA0164, the pragma can also be inserted in the declaration part before a comment.

#### Syntax:

- Deactivation of rules:
  - one rule: {analysis -<rule number>}
  - several rules: {analysis -<rule number>, -<further rule number>, -<further rule number>}
- Activation of rules:
  - one rule: {analysis +<rule number>}
  - several rules: {analysis +<rule number>, +<further rule number>, +<further rule number>}

#### Samples:

Rule 24 (only typed literals permitted) is to be disabled for one line (i.e. in these lines it is not necessary to write "nTest := DINT#99") and then enabled again:

```
{analysis -24}
nTest := 99;
{analysis +24}
nVar := INT#2;
```

#### Specification of several rules:

```
{analysis -10, -24, -18}
```

### Attribute {attribute 'no-analysis'}

You can use the {attribute 'no-analysis'} attribute to exclude an entire programming object from the static analysis check. For this programming object no checks are carried out for the coding rules, naming conventions and forbidden symbols.

#### Insertion location:

above the declaration of a programming object

#### Syntax:

```
{attribute 'no-analysis'}
```

#### Samples:

```
{attribute 'qualified_only'}
{attribute 'no-analysis'}
VAR_GLOBAL
...
END_VAR

{attribute 'no-analysis'}
PROGRAM MAIN
VAR
...
END_VAR
```

**Attribute {attribute 'analysis' := '...'}** 

You can use the attribute {attribute 'analysis' := '-<rule number>'} to switch off certain rules for individual declarations or for a complete programming object. The code rule is deactivated by specifying the rule number(s) with a minus sign in front. You can specify any number of rules in the attribute.

**Insertion location:**

above the declaration of a programming object or in the line above a variable declaration

**Syntax:**

- one rule: {attribute 'analysis' := '-<rule number>'}
- several rules: {attribute 'analysis' := '-<rule number>, -<further rule number>, -<further rule number>'}

**Samples:**

Rule 33 (unused variables) is to be disabled for all variables of the structure.

```
{attribute 'analysis' := '-33'}
TYPE ST_Sample :
STRUCT
    bMember    : BOOL;
    nMember    : INT;
END_STRUCT
END_TYPE
```

Checking of rules 28 (overlapping memory areas) and 33 (unused variables) is to be disabled for variable nVar1.

```
PROGRAM MAIN
VAR
    {attribute 'analysis' := '-28, -33'}
    nVar1 AT%QB21 : INT;
    nVar2 AT%QD5  : DWORD;

    nVar3 AT%QB41 : INT;
    nVar4 AT%QD10 : DWORD;
END_VAR
```

Rule 6 (concurrent access) is to be disabled for a global variable, so that no error message is generated if write access to the variable occurs from more than one task.

```
VAR_GLOBAL
    {attribute 'analysis' := '-6'}
    nVar : INT;
    bVar : BOOL;
END_VAR
```

**Attribute {attribute 'naming' := '...'}** 

The attribute {attribute 'naming' := '...'} can be used in the declaration part in order to exclude individual declaration lines from the check for compliance with the current naming conventions.

**Insertion location:**

- Deactivation: in the declaration part above the relevant lines
- Activation: after the last line of the deactivation

**Syntax:**

{attribute 'naming' := '<off|on|omit>'}

- off, on: the check is disabled for all rows between the "off" and "on" statements
- omit: only the next row is excluded from the check

**Sample:**

It is assumed that the following naming conventions are defined:

- The identifiers of INT variables must have a prefix "n" (naming convention NC0014), e.g. "nVar1".
- Function block names must start with "FB\_" (naming convention NC0103), e.g. "FB\_Sample".

For the code shown below, the static analysis then only issues messages for the following variables: cVar, aVariable, bVariable.

```
PROGRAM MAIN
VAR
  {attribute 'naming' := 'off'}
  aVar  : INT;
  bVar  : INT;
  {attribute 'naming' := 'on'}

  cVar  : INT;

  {attribute 'naming' := 'omit'}
  dVar  : INT;

  fb1   : SampleFB;
  fb2   : FB;
END_VAR

{attribute 'naming' := 'omit'}
FUNCTION_BLOCK SampleFB
...

{attribute 'naming' := 'off'}
FUNCTION_BLOCK FB
VAR
  {attribute 'naming' := 'on'}
  aVariable : INT;
  bVariable : INT;
  ...
```

### Attribute {attribute 'nameprefix' := '...'}<sup>1</sup>

The attribute {attribute 'nameprefix' := '...'} defines a prefix for variables of a structured data type. A naming convention then applies to the effect that identifiers for instances of this type must have this prefix.

#### Insertion location:

above the declaration of a structured data type

#### Syntax:

```
{attribute 'nameprefix' := '<prefix>'}
```

#### Example:

The following naming conventions are defined in the category [Naming conventions](#) [► 81] in the PLC project properties:

- Variables of the type of a structure (NC0032): st
- Structures (NC0151): ST\_

Conversely, variables of the type "ST\_Point" should not begin with the prefix "st", but with the prefix "pt".

In the following sample, the statistic analysis will output a message for "a1" and "st1" of the type "ST\_Point" because the variable names do not begin with "pt". For variables of the type "ST\_Test", conversely, the prefix "st" is expected.

```
TYPE ST_Test :
STRUCT
  ...
END_STRUCT
END_TYPE

{attribute 'nameprefix' := 'pt'}
TYPE ST_Point :
STRUCT
  x : INT;
  y : INT;
END_STRUCT
END_TYPE
```

```

PROGRAM MAIN
VAR
    a1   : ST_Point;    // => Invalid variable name 'a1'. Expect prefix 'pt'
    st1  : ST_Point;    // => Invalid variable name 'st1'. Expect prefix 'pt'
    pt1  : ST_Point;

    a2   : ST_Test;     // => Invalid variable name 'a2'. Expect prefix 'st'
    st2  : ST_Test;
    pt2  : ST_Test;     // => Invalid variable name 'st2'. Expect prefix 'st'
END_VAR

```

### Attribute {attribute 'analysis:report-multiple-instance-calls'}

The attribute {attribute 'analysis:report-multiple-instance-calls'} identifies a function block for a check for rule 105: Only function blocks with this attribute are checked to ascertain whether the instances of the function block are called several times. The attribute has no effect if rule 105 is disabled in the [Rules \[► 16\]](#) category in the PLC project properties.

#### Insertion location:

above the declaration of a function block

#### Syntax:

```
{attribute 'analysis:report-multiple-instance-calls'}
```

#### Sample:

In the following sample the static analysis will issue an error for fb2, since the instance is called more than once.

Function block FB\_Test1 without attribute:

```

FUNCTION_BLOCK FB_Test1
...

```

Function block FB\_Test2 with attribute:

```

{attribute 'analysis:report-multiple-instance-calls'}
FUNCTION_BLOCK FB_Test2
...

```

#### Program MAIN:

```

PROGRAM MAIN
VAR
    fb1   : FB_Test1;
    fb2   : FB_Test2;
END_VAR

fb1();
fb1();
fb2();           // => SA0105: Instance 'fb2' called more than once
fb2();           // => SA0105: Instance 'fb2' called more than once

```

## 7 Constant propagation



Available from TwinCAT 3.1.4026.14

Static code analysis is based on constant propagation, the results of which are used for various checks. For example, it checks whether pointers are not equal to 0 or whether array indices are outside the valid range.

They can effectively support static analysis if they know how this analysis works and where its limits lie.

See also: ['Show constant propagation values for current editor' command](#) [► 118]

### Constant propagation

Static analysis attempts to determine the value of a variable based on its use.

#### Sample:

```
PROGRAM MAIN
VAR
    x      : INT;
    bTest  : BOOL;
END_VAR

x := 99;

IF x < 100 THEN
    bTest := TRUE;
END_IF
```

In the implementation in line 1, the constant propagation sets the value 99 for the variable `x` in order to use this value for further analyses. The analysis then recognizes that the expression in the subsequent IF statement is constantly TRUE.

### Locally performed constant propagation

A value is only determined locally in the function block. It is irrelevant how an input is transferred. The results of function calls are also irrelevant.

#### Sample:

```
FUNCTION Func : BOOL
VAR_INPUT
    bTest : BOOL;
END_VAR

IF bTest THEN
    Func := OtherFunc(TRUE);
END_IF
```

If the parameter `bTest` is set to TRUE for each call, this has no effect on the constant propagation. Even if `OtherFunc(TRUE)` always returns TRUE, this has no effect on the constant propagation.

### Only temporary variables have initial values

Static local variables in programs and function blocks do not have an assumed initial value. The variables retain their values from the last call and can therefore be "anything" in principle.

Local variables in functions and temporary variables have an initial value each time they are called. The constant propagation calculates with this initial value.

#### Sample:

```
PROGRAM MAIN
VAR
    x      : INT := 6;
    bTest  : BOOL;
END_VAR
VAR_TEMP
    y      : INT := 8;
END_VAR
```

```
bTest := x < y;
```

The variable `y` will have the value 8 each time MAIN is executed. However, the variable `x` will not necessarily. Therefore, the constant propagation will only assume a value for `y`, but not for `x`.

It is advisable to declare variables that are always written first and then read as temporary variables.

### Constant propagation determines value ranges for numerical data types

To reduce complexity, a value range with upper and lower limits is determined for each variable.

#### Sample:

```
PROGRAM MAIN
VAR
  x      : INT := 6;
  bTest  : BOOL;
  y      : INT;
END_VAR

IF bTest THEN
  x := 1;
ELSE
  x := 100;
END_IF

IF x = 77 THEN
  y := 13;
END_IF
```

The value range `[1..100]` is determined here for the variable `x`. As a result, the comparison `x = 77` is not recognized as a constant expression in line 7, as 77 is within the value range.

### Recurring sophisticated expressions are not recognized as the same variable

Sophisticated expressions may not have a value assigned. If such expressions occur more than once, it is helpful to introduce an auxiliary variable.

#### Sample:

```
PROGRAM MAIN
VAR
  x      : DINT;
  py     : POINTER TO INT;
  y      : INT;
  testArray : ARRAY [0..4] OF DINT;
END_VAR

IF py <> 0 THEN
  IF py^ >= 0 AND py^ <= 4 THEN
    x := testArray[py^];
  END_IF

  y := py^;

  IF y <= 0 AND y <= 4 THEN
    x := testArray[y];
  END_IF
END_IF
```

In line 3, an error is output for a possible access to a value via pointer, although the area to which the pointer points is checked. If the value is first copied into a local variable and its range is checked, then the constant propagation can determine the value range for this variable and allows access to the array in line 9.

### Branches

For branches, individual branches are calculated separately. Value ranges from the individual ranges are then combined to form a new value range.

#### Sample:

```
IF func(TRUE) THEN
  x := 1;
ELSE
  x := 10;
END_IF
```

```

IF func(FALSE) THEN
    y := x;
ELSE
    y := 2*x;
END_IF

```

In line 6,  $x$  has the range  $[1..10]$ . After line 11,  $y$  has the value range  $[1..20]$ . This results from the union of the two value ranges  $[1..10]$  and  $[2..20]$ .

## Conditions

### Sample:

Conditions can restrict the value range of a variable in a code block. Several conditions can be combined. Mutually exclusive conditions can also result in an empty value range.

```

IF y > 0 AND y < 10 THEN
    x := y;
ELSE
    x := 0;
END_IF

IF x < 0 THEN
    i := 99;
END_IF

```

$y$  has the value range  $[1..9]$  in line 2. This results in the value range  $[0..9]$  for  $x$  in line 6. Combined with the condition  $x < 0$ , this results in an empty set of possible values in line 9 for  $x$ . The code is not accessible. The static analysis will report that the condition  $x < 0$  always results in FALSE at this point.

## Loops

The constant propagation will execute loops in the code until the values of the variables in the loop no longer change. It is assumed that a loop can be run through as often as required. The values determined so far are combined with the previous values. Variables that are changed within the loop have a successively growing range. The constant propagation does not assume all possible values for ranges, but only uses limits that occur in the code and also the values 0, 1, 2, 3 and 10, as these often play a role.

The easiest way to illustrate the procedure is with an example.

### Sample:

```

PROGRAM MAIN
VAR
    x : DINT;
    i : DINT;
    y : DINT;
END_VAR

x := 0;
y := 0;

FOR i := 0 TO 5 DO
    x := x + 1;
    y := i;
END_FOR

```

The constant propagation knows the following about the loop:

$i$ ,  $x$ , and  $y$  are 0 at the beginning of the first execution of the loop. The condition  $i \leq 5$  applies to the code in the loop. The condition  $i > 5$  applies to the code after the loop.

The constant propagation determines the following values for the variables in the loop:

	<b>i</b>	<b>x</b>	<b>y</b>	
	$[0..5]$	$[0..MAXDINT]$	$[0..5]$	

The following intermediate steps are carried out in detail:



Run	i	x	y	
1	0	[0..1]	0	i was initialized with 0, y always gets the same values as i
2	[0..1]	[0..2]	[0..1]	
6	[0..5]	[0..6]	[0..5]	First, the range [0..6] is actually calculated for i. However, it is known that $i < 5$ is a condition. Therefore, the value for the code in the loop is limited to this value.
7	[0..5]	[0..7]	[0..5]	
10	[0..5]	[0..10]	[0..5]	x is always incremented. From 10, however, the value is "rounded up" to MAXDINT.
11	[0..5]	[0..MAXDINT]	[0..5]	MAXDINT + 1 results in MAXDINT
from 11				From the 11th run, the values in the loop will no longer change. The propagation is finished.

In addition, the following applies to the code after this loop:  $i = 6$ . The range [0..6] is determined in the loop and this is combined with the condition  $i > 5$ , which results in the exact value 6.

## 8 QuickFix/Precompile



Available from TwinCAT 3.1 Build 4026

Some rules from Static Analysis can already be checked during precompilation. For the detection of such rule violations no explicit execution of the Static Analysis is necessary, but the check already takes place on the basis of the precompile information during editing. The checking of a rule during precompilation takes place only if the rule is enabled in the Static Analysis settings.

### Precompile: Wavy underline and display in the message window

When a rule violation occurs, it is immediately indicated by wavy underline in the declaration editor or in the ST editor. Additionally - as long as the editor is open - error messages or warnings appear in the message window in the category "IntelliSense". These contain the note "(precompile)" before the rule number.

### QuickFix commands

In addition, for some rules that can be checked during precompilation, there is the possibility of a QuickFix in the declaration editor and the ST editor. You can perform automatic, immediate error handling directly at the affected code positions. For quick error handling, click on the wavy underlined code in the editor and then click on the light bulb icon.

Depending on the error, the following QuickFix commands are offered:

- Ignore error message/warning:  
The command causes pragmas or attributes to be automatically inserted into the code that exclude checking the associated rule for that line of code.
- Ignore error message/warning globally for the POU:  
The command causes an attribute to be automatically inserted at the beginning of the declaration part of the programming object. Then a check of the associated rule for this programming object is excluded.
- Disable checking:  
The command causes the checking of the associated rule to be disabled in the settings.
- Fix error by suggesting to change ST code:  
Example for "SA0033: Unused variables": The declaration of the unused variables is removed from the declaration editor.

### Available rules

#### Not available:

Please note that the following rules **cannot** be checked during precompilation.

- SA0004
- SA0006
- SA0013
- SA0016
- SA0027
- SA0028
- SA0042
- SA0100
- SA0103
- SA0105
- SA0150
- SA0160
- SA0161

- SA0175

**Available:**

All other rules are checked on the basis of the precompile information.

## 9 Automation Interface support

The Static Analysis can partly be operated via the Automation Interface (AI). AI support includes the following commands/actions:

- [Explicit execution of Static Analysis via the Automation Interface \[► 132\]](#)
- [Implicit execution of Static Analysis via the Automation Interface \[► 132\]](#)
- [Save settings/configuration via Automation Interface \[► 132\]](#)
- [Load settings/configuration via Automation Interface \[► 133\]](#)
- [Export metrics \[► 133\]](#)

Please also refer to the Automation Interface documentation:  
[Product description](#)

### Explicit execution of Static Analysis via the Automation Interface

The two following commands can be called explicitly via the Automation Interface:

- [Command 'Run static analysis' \[► 111\]](#)
- [Command 'Run static analysis \[Check all objects\]' \[► 113\]](#)

bCheckAll can be specified as optional parameter for the method `RunStaticAnalysis()`. However, the method can also be called without parameters.

Parameter	Call
<code>RunStaticAnalysis()</code>	Execution of the <b>Run static analysis [Check all objects]</b> command
<code>RunStaticAnalysis(bCheckAll = TRUE)</code>	
<code>RunStaticAnalysis(bCheckAll = FALSE)</code>	Execution of the <b>Run static analysis</b> command

#### PowerShell sample:

```
$p = $sysMan.LookupTreeItem("TIPC^MyPlcProject^MyPlcProject Project")
$p.RunStaticAnalysis()
```

#### C# sample for TC3.1 version >= Build 4024:

```
ITcPlcIECProject3 plcIec3 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject3;
plcIec3.RunStaticAnalysis();
```

#### C# sample for TC3.1 version >= Build 4026:

```
ITcPlcIECProject4 plcIec4 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject4;
plcIec4.RunStaticAnalysis();
```

### Implicit execution of Static Analysis via the Automation Interface

Alternatively, the [setting \[► 14\] Perform static analysis automatically](#) can be enabled, and the project can be created via the Automation Interface, so that the Static Analysis is implicitly performed during the project creation process.

### Save settings/configuration via Automation Interface



Available from TwinCAT 3.1 Build 4026

The [settings \[► 14\]](#) from Static Analysis can be saved or exported to a \*.csa file via Automation Interface.

For the method `SaveStaticAnalysisSettings(string bstrFilename)` the destination path of the file must be specified as a transfer parameter.



The `RunStaticAnalysis` method is available from the `ITcPlcIECProject3` interface. The methods `SaveStaticAnalysisSettings` and `LoadStaticAnalysisSettings` are offered from the interface `ITcPlcIECProject4`.

#### C# sample for TC3.1 version >= Build 4026:

```
// Path to the location to export the SAN configuration
string saveCsaPath = @"C:\Users\UserName\Desktop\SaveTest.csa";
[...]
// Navigate to PLC project
ITcPlcIECProject4 plcIec4 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject4;
// Save SAN configuration
plcIec4.SaveStaticAnalysisSettings(saveCsaPath);
```

#### Load settings/configuration via Automation Interface



Available from TwinCAT 3.1 Build 4026

A ready-made Static Analysis configuration (\*.csa file) can be loaded into the PLC project via Automation Interface. The [settings \[► 14\]](#) loaded by this can then be checked by AI by running the Static Analysis (see above).

For the method `LoadStaticAnalysisSettings(string bstrFilename)` the path of the file to be loaded must be specified as a transfer parameter.



The `RunStaticAnalysis` method is available from the `ITcPlcIECProject3` interface. The methods `SaveStaticAnalysisSettings` and `LoadStaticAnalysisSettings` are offered from the interface `ITcPlcIECProject4`.

#### C# sample for TC3.1 version >= Build 4026:

```
// Path to load a SAN configuration
string loadCsaPath = @"C:\Users\UserName\Desktop\LoadTest.csa";
[...]
// Navigate to PLC project
ITcPlcIECProject4 plcIec4 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject4;
// Load SAN configuration
plcIec4.LoadStaticAnalysisSettings(loadCsaPath);
// Optionally run SAN afterwards
plcIec4.RunStaticAnalysis();
```

#### Export metrics



Available from TwinCAT 3.1 Build 4026.4

The standard metrics can be exported to a text file (\*.csv) via the Automation Interface. A current calculation of the metrics is performed implicitly. If this process was executed manually, it would include the following two commands:

- [Command 'View Standard Metrics' \[► 114\]](#)
- **Export table** command, see [Commands in the context menu of the 'Standard Metrics' view \[► 115\]](#)

For the `ExportStandardMetrics(string bstrFilename)` method, the path that the export file is saved on must be specified as a parameter value.



The `ExportStandardMetrics` method is available from the `ITcPlcIECProject5` interface.

#### C# sample for TC3.1 version >= Build 4026.4:

```
// Path to save the csv file
string savePath = @"C:\Users\UserName\Desktop\Metrics.csv";
[...]
// Navigate to PLC project
ITcPlcIECProject5 plcIec5 = sysMan.LookupTreeItem("TIPC^Untitled1^Untitled1
Project") as ITcPlcIECProject5;
// Export standard metrics
plcIec5.ExportStandardMetrics(savePath);
```

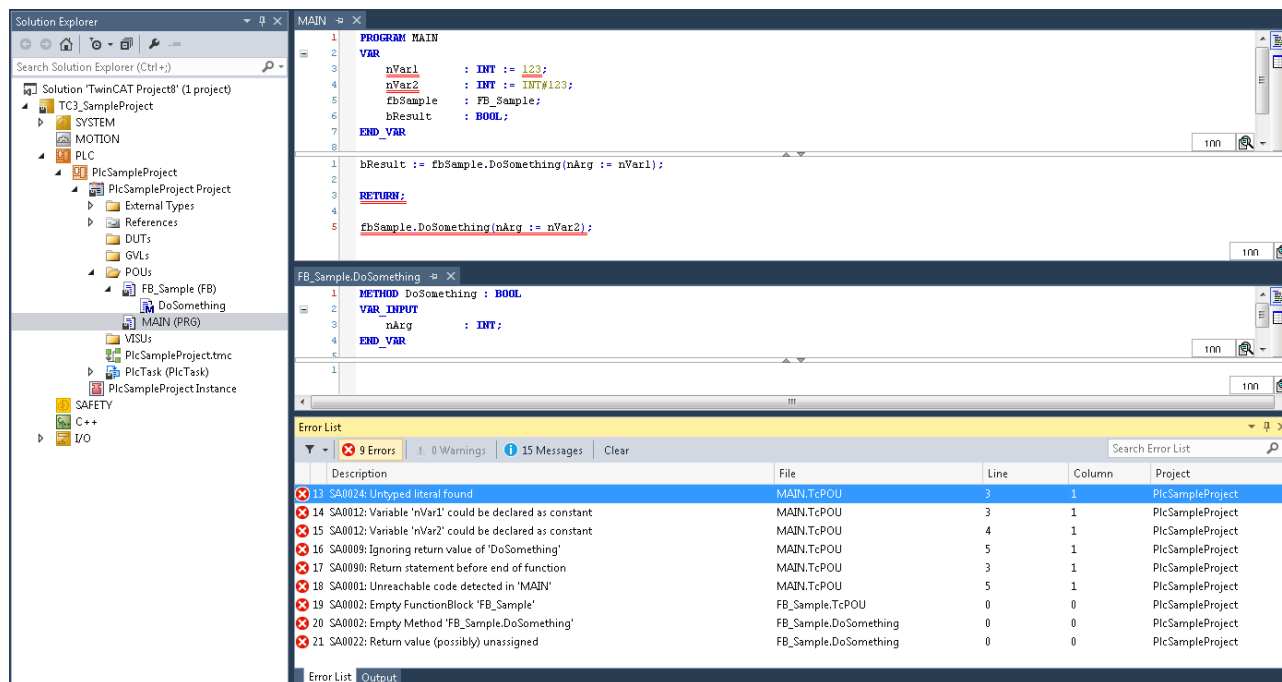
# 10 Examples

## 10.1 Static analysis

During execution of the static analysis [► 111], compliance with the coding rules [► 16], naming conventions [► 81] and forbidden symbols [► 110] is checked. The following section provides a sample for each of these aspects.

### 1) Coding rules

In this sample some coding rules are configured as error. The violations of this coding rules are therefore reported as an error after the static analysis has been performed. Further information is shown in the following diagram.

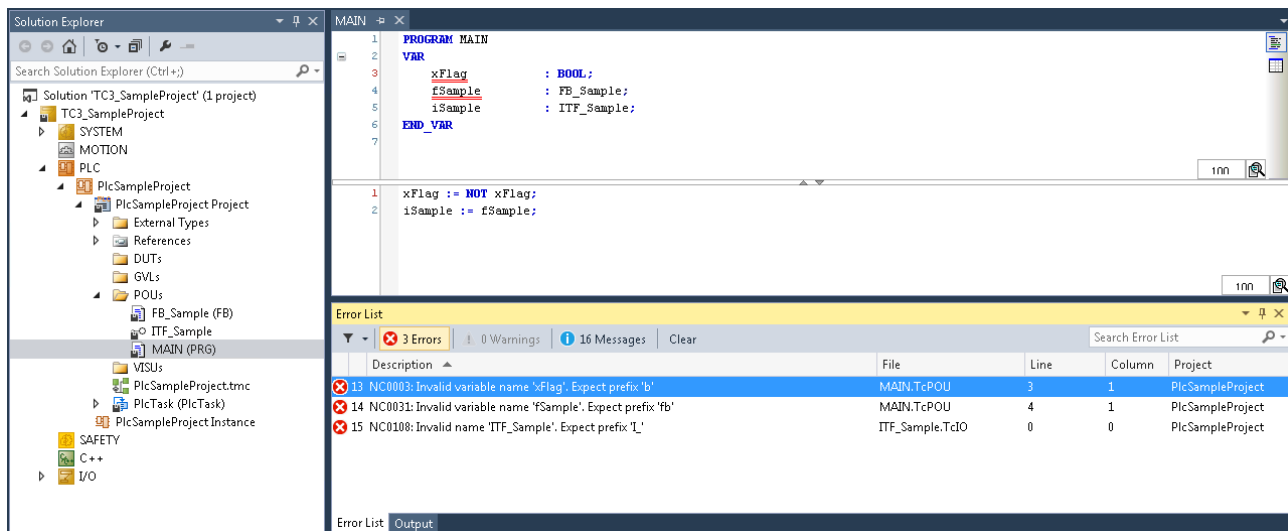


### 2) Naming conventions

The following naming conventions are configured:

- Prefix "b" for variables of type BOOL (NC0003)
- Prefix "fb" for function block instances (NC0031)
- Prefix "FB\_" for function blocks (NC0103)
- Prefix "I\_" for interfaces (NC0108)

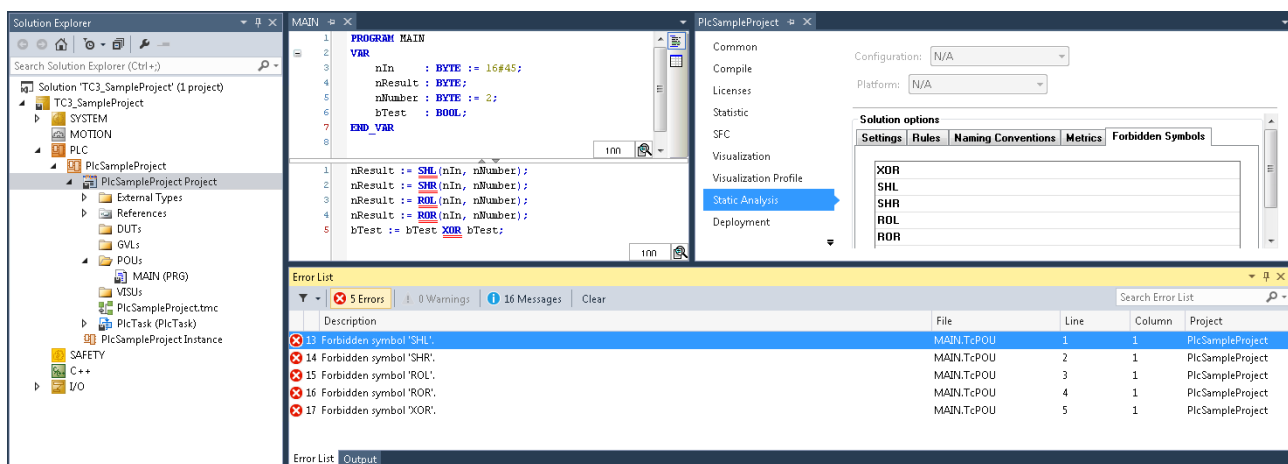
This naming conventions are not adhered to in the declaration of Boolean variables ("x"), the instantiation of function block ("f") and the declaration of the interface type ("ITF\_"). These code positions are reported as an error after the static analysis has been performed.



### 3) Forbidden symbols

The bit string operator XOR and the bit shift-operators SHL, SHR, ROL and ROR are configured as forbidden symbols. These operators should not be used in the code.

Accordingly, any use of these operators is reported as an error after the static analysis has been performed.



## 10.2 Standard metrics

A sample for dealing with the standard metrics is provided below.

In this sample "650" (= 650 bytes) is defined as upper limit for the metric "code size" and "5" as upper limit for the metric "number of input variables" (see: [Configuration of the metrics](#) [► 94]). In addition, rule 150 (SA0150: Violation of lower or upper metric limits) is enabled and configured as warning.

When the [command 'View Standard Metrics'](#) [► 114] is issued, the metric view opens and the indicators that were determined are displayed in tabular form. Since the size of the MAIN program is 688 bytes and the program SampleProgram has 7 input variables, these indicators exceed the defined upper limit in each case, so that the corresponding table cells are shown in red.

Standard metrics												
Program unit	Code size	Variables size	Stack size	Calls	Tasks	Globals	IOs	Locals	Inputs	Outputs	NOS	Co...
MAIN (PRG)	688	10	0	1	1	0	0	7	3	0	67	0
SampleProgram (PRG)	352	12	0	1	1	0	0	5	7	0	33	0

In this sample, the fact that the defined upper limits are exceeded is not only apparent in the metric view. Since rule 150 is configured as warning, the static analysis checks for violations of lower and upper metric limits. After the [static analysis](#) [► 111] has been performed, the violation of the two upper limits is therefore reported as a warning in the message window.



Error List		
<div><div><div></div></div><div>0 Errors</div><div><div></div>2 Warnings</div><div><div></div>16 Messages</div><div>Clear</div></div>		
	Description	File
<div></div> 13	SA0150: Metric violation for 'MAIN'. Result for metric 'Code size' (688) > 650	MAIN.TcPOU
<div></div> 14	SA0150: Metric violation for 'SampleProgram'. Result for metric 'Inputs' (7) > 5	SampleProgram.TcPOU

# 11 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.

## Download finder

Our download finder contains all the files that we offer you for downloading. You will find application reports, technical documentation, technical drawings, configuration files and much more.

The downloads are available in various formats.

## 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 our internet page: [www.beckhoff.com](http://www.beckhoff.com)

You will also find further documentation for Beckhoff components there.

## 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 5246 963-157  
e-mail: [support@beckhoff.com](mailto: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 5246 963-460  
e-mail: [service@beckhoff.com](mailto:service@beckhoff.com)

## Beckhoff Headquarters

Beckhoff Automation GmbH & Co. KG

Huelshorstweg 20  
33415 Verl  
Germany

Phone: +49 5246 963-0  
e-mail: [info@beckhoff.com](mailto:info@beckhoff.com)  
web: [www.beckhoff.com](http://www.beckhoff.com)

## **Trademark statements**

Beckhoff®, TwinCAT®, TwinCAT/BSD®, TC/BSD®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® and XPlanar® are registered trademarks of and licensed by Beckhoff Automation GmbH.

## **Third-party trademark statements**

Arm, Arm9 and Cortex are trademarks or registered trademarks of Arm Limited (or its subsidiaries or affiliates) in the US and/or elsewhere.

Intel, the Intel logo, Intel Core, Xeon, Intel Atom, Celeron and Pentium are trademarks of Intel Corporation or its subsidiaries.

Microsoft, Microsoft Azure, Microsoft Edge, PowerShell, Visual Studio, Windows and Xbox are trademarks of the Microsoft group of companies.

More Information:  
**[www.beckhoff.com/te1200](http://www.beckhoff.com/te1200)**

Beckhoff Automation GmbH & Co. KG  
Hülshorstweg 20  
33415 Verl  
Germany  
Phone: +49 5246 9630  
[info@beckhoff.com](mailto:info@beckhoff.com)  
[www.beckhoff.com](http://www.beckhoff.com)

