# Table of contents

1 Foreword ................................................................................................................................................. 5  
  1.1 Notes on the documentation ........................................................................................................... 5  
  1.2 Safety instructions ............................................................................................................................. 6  
2 Overview .................................................................................................................................................. 7  
3 Installation ............................................................................................................................................... 9  
4 Configuration .......................................................................................................................................... 12  
  4.1 Settings ........................................................................................................................................... 12  
  4.2 Rules .............................................................................................................................................. 14  
    4.2.1 Rules - overview and description ............................................................................................... 15  
  4.3 Naming conventions ............................................................................................................................ 79  
    4.3.1 Naming conventions – overview and description ...................................................................... 80  
    4.3.2 Placeholder {datatype} ............................................................................................................... 87  
  4.4 Naming conventions (2) ...................................................................................................................... 88  
  4.5 Metrics ............................................................................................................................................. 91  
    4.5.1 Metrics - overview and description ............................................................................................ 92  
  4.6 Forbidden symbols ............................................................................................................................ 96  
5 Execution ................................................................................................................................................ 98  
  5.1 Command 'Run static analysis' ......................................................................................................... 98  
    5.1.1 Syntax in the message window .................................................................................................... 99  
  5.2 Command 'Run static analysis [Check all objects]' ..................................................................... 100  
  5.3 Command 'View Standard Metrics' ................................................................................................ 101  
    5.3.1 Commands in the context menu of the 'Standard Metrics' view .............................................. 102  
  5.4 Command 'View Standard Metrics [Check all objects]' ............................................................... 103  
    5.4.1 Commands in the context menu of the 'Standard Metrics' view .............................................. 104  
6 Pragmas and attributes ............................................................................................................................ 106  
7 Examples ................................................................................................................................................ 111  
  7.1 Static analysis .................................................................................................................................. 111  
  7.2 Standard metrics .............................................................................................................................. 112  
8 Automation Interface support .................................................................................................................. 114
1 Foreword

1.1 Notes on the documentation

This description is only intended for the use of trained specialists in control and automation engineering who are familiar with applicable national standards.

It is essential that the documentation and the following notes and explanations are followed when installing and commissioning the components.

It is the duty of the technical personnel to use the documentation published at the respective time of each installation and commissioning.

The responsible staff must ensure that the application or use of the products described satisfy all the requirements for safety, including all the relevant laws, regulations, guidelines and standards.

Disclaimer

The documentation has been prepared 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 prior announcement.

No claims for the modification of products that have already been supplied may be made on the basis of the data, diagrams and descriptions in this documentation.

Trademarks

Beckhoff®, TwinCAT®, EtherCAT®, EtherCAT G®, EtherCAT G10®, EtherCAT P®, Safety over EtherCAT®, TwinSAFE®, XFC®, XTS® and XPlanar® are registered trademarks of and licensed by Beckhoff Automation GmbH.

Other designations used in this publication may be trademarks whose use by third parties for their own purposes could violate the rights of the owners.

Patent Pending

The EtherCAT Technology is covered, including but not limited to the following patent applications and patents:


with corresponding applications or registrations in various other countries.

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

Copyright

© Beckhoff Automation GmbH & Co. KG, Germany.

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

Offenders will be held liable for the payment of damages. All rights reserved in the event of the grant of a patent, utility model or design.
1.2 Safety instructions

Safety regulations

Please note the following safety instructions and explanations!
Product-specific safety instructions can be found on following pages or in the areas mounting, wiring, commissioning etc.

Exclusion of liability

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

Personnel qualification

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

Description of symbols

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

<table>
<thead>
<tr>
<th><strong>DANGER</strong></th>
<th>Serious risk of injury!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure to follow the safety instructions associated with this symbol directly endangers the life and health of persons.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>WARNING</strong></th>
<th>Risk of injury!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure to follow the safety instructions associated with this symbol endangers the life and health of persons.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>CAUTION</strong></th>
<th>Personal injuries!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure to follow the safety instructions associated with this symbol can lead to injuries to persons.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>NOTE</strong></th>
<th>Damage to the environment or devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure to follow the instructions associated with this symbol can lead to damage to the environment or equipment.</td>
<td></td>
</tr>
</tbody>
</table>

| **Tip or pointer** | This symbol indicates information that contributes to better understanding. |
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.

Static code analysis should be regarded as a supplement to the compiler. It facilitates writing clearer code and can help uncover potential sources of error during programming. 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.

Static Analysis is integrated into TwinCAT 3 PLC as a programming tool. It checks the source code of a PLC project for deviations from certain coding rules, naming conventions or unauthorized symbols. The rule set defined in the PLCopen Coding Guidelines is used as the basis, complemented by additional checking options.

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 instructions statements to exclude particular parts of the code from the check.

In addition, you can display selected metrics to assess code quality in a separate view. Static Analysis determines these metrics from your program code. Key parameters are calculated that characterize the various program parts or express the properties of the software. They therefore provide an indication of the software quality. For example, the tabular output contains metrics for the number of statements or the proportion of comments. Another example is the McCabe metric, which measures what is referred to as cyclomatic complexity. This is a measure of the number of execution paths that can be passed during code execution.

Benefits

"Static code analysis" facilitates writing clearer code and can help uncover potential sources of error during programming.

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 "TC3 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, Naming conventions and Forbidden symbols 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 indicators (e.g. the number of code lines). 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 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 licensed versions of Static Analysis can be found [here](#).

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

- Installation [► 9]
- Configuration of the settings, rules, naming conventions, metrics and forbidden symbols [► 12]
- Command ‘Run static analysis’ [► 98]
- Command ‘Run static analysis [Check all objects]’ [► 100]
- Command ‘View Standard Metrics’ [► 101]
- Command ‘View Standard Metrics [Check all objects]’ [► 103]
- Pragmas and attributes [► 106]
- Examples [► 111]
- Automation Interface support [► 114]

**Libraries**

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

When the library project is open, you can check the elements it contains using the command Command ‘Run static analysis [Check all objects]’ [► 100].

**Punctual disablement of checks**

Pragmas and attributes [► 106] 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 [► 114]).
3 Installation

The function "TE1200 | TC3 PLC Static Analysis" is installed together with the TwinCAT 3 development environment (XAE setup) and has been included as release version since TwinCAT version 3.1 build 4022.0. All that is therefore required is licensing of the additional TE1200 engineering component. For further information please refer to the documentation on Licensing.

Please note that there is no 7-day trial license available for this product. Without 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 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.
<table>
<thead>
<tr>
<th>Functional aspect</th>
<th>Static Analysis Light</th>
<th>Static Analysis Full</th>
</tr>
</thead>
<tbody>
<tr>
<td>Save/export and load/import (rule) configuration</td>
<td>Not possible, coupled to PLC project properties</td>
<td>Possible (using the Load/Save buttons in the Settings [12])</td>
</tr>
<tr>
<td>Execution is coupled to the compilation process</td>
<td>Yes, not configurable</td>
<td>Configurable (using the <strong>Perform static analysis automatically</strong> option in the Settings [12]; Manual execution using the command Command 'Run static analysis' [98])</td>
</tr>
<tr>
<td>Checking for unused objects (e.g. within a library project)</td>
<td>Not possible</td>
<td>Possible (using the command Command 'Run static analysis (Check all objects)' [100])</td>
</tr>
<tr>
<td>Maximum number of reported errors</td>
<td>500 (not configurable) (Further information on the significance of 500 as the maximum number of errors can be found in the Settings [12])</td>
<td>Configurable (using the setting <strong>Maximum number of errors</strong> in the Settings [12])</td>
</tr>
<tr>
<td>Maximum number of reported warnings</td>
<td>Output of warnings not possible (see following line)</td>
<td>Configurable (using the setting <strong>Maximum number of warnings</strong> in the Settings [12])</td>
</tr>
<tr>
<td>Rules – activation options [14]</td>
<td>• Active and output as error</td>
<td>• Active and output as error</td>
</tr>
<tr>
<td></td>
<td>• Inactive</td>
<td>• Active and output as warning</td>
</tr>
<tr>
<td></td>
<td>• Inactive</td>
<td>• Inactive</td>
</tr>
<tr>
<td></td>
<td>• SA0033: Unused variables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SA0028: Overlapping memory areas</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SA0006: Write access to multiple tasks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SA0004: Multiple write access on output</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SA0027: Multiple usage of name</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SA0167: Report temporary FunctionBlock instances</td>
<td></td>
</tr>
<tr>
<td>Naming conventions [79]</td>
<td>Not available</td>
<td>Available</td>
</tr>
<tr>
<td>Metrics [91]</td>
<td>Not available</td>
<td>Available</td>
</tr>
<tr>
<td>Forbidden symbols [96]</td>
<td>Not available</td>
<td>Available</td>
</tr>
</tbody>
</table>
### Pragmas and attributes [106] for temporary deactivation of rules

Yes, available according to the Light scope:
- Pragma {analysis ...}
- Attribute {attribute 'no-analysis'}
- Attribute {attribute 'analysis' := '...'}

Yes, available according to the Full scope:
- Pragma {analysis ...}
- Attribute {attribute 'no-analysis'}
- Attribute {attribute 'analysis' := '...'}
- Attribute {attribute 'naming' := '...'}
- Attribute {attribute 'nameprefix' := '...'}
- Attribute {attribute 'analysis:report-multiple-instance-calls'}
4 Configuration

After the installation and licensing of "TE1200 | TC3 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 [12]
- Rules [14]
- Naming conventions [79]
- Naming conventions (2) [88]
- Metrics [91]
- Forbidden symbols [96]

Scope of the "Static Analysis" configuration

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

If this option is enabled, TwinCAT performs the Static Analysis whenever code is generated (e.g. when the command Build Project is executed). The analysis can be started manually via the Command 'Run static analysis' [p. 98], 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 Save (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 Load 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

Preset: 500

In this box you can enter the desired error limit, which is checked during the execution of the Static Analysis. If either the error limit or the warning limit (see below) is reached, execution of the Static Analysis is canceled and the previous analysis result is output.

Performance vs completeness:

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.

In the assumed case that there are more than 500 Static Analysis errors in a PLC project, the following use cases arise.

• 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.

• 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.

Detailed explanation of the behavior:

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.
4.2 Rules

In the Rules tab you can configure the rules that are taken into account when the static analysis is performed [p. 98]. The rules are displayed as a tree structure in the project properties. Some rules are arranged below organizational nodes.

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, 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 deactivation 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 [106].

Overview and description of the rules

An overview and a detailed description of the rules can be found under Rules - overview and description [15].

4.2.1 Rules - overview and description

- **Check strict IEC rules**
  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**
  The following rules exist on this topic:
  - **SA0006: Write access from several tasks** [21]
    Determines variables with write access from more than one task.
  - **SA0103: Concurrent access on not atomic data** [60]
    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.

Overview

- **SA0001: Unreachable code** [19]
- **SA0002: Empty objects** [20]
- **SA0003: Empty statements** [20]
- **SA0004: Multiple writes access on output** [20]
- **SA0006: Write access from several tasks** [21]
- **SA0007: Address operators on constants** [22]
- **SA0008: Check subrange types** [22]
- SA0009: Unused return values [p. 23]
- SA0010: Arrays with only one component [p. 23]
- SA0011: Useless declarations [p. 24]
- SA0012: Variables which could be declared as constants [p. 24]
- SA0013: Declarations with the same variable name [p. 25]
- SA0014: Assignments of instances [p. 25]
- SA0015: Access to global data via FB_init [p. 26]
- SA0016: Gaps in structures [p. 26]
- SA0017: Non-regular assignments [p. 27]
- SA0018: Unusual bit access [p. 27]
- SA0020: Possibly assignment of truncated value to REAL variable [p. 28]
- SA0021: Transporting the address of a temporary variable [p. 28]
- SA0022: (Possibly) non-rejected return values [p. 29]
- SA0023: Complex return values [p. 29]
- SA0024: Untyped literals/constants [p. 30]
- SA0025: Unqualified enumeration constants [p. 30]
- SA0026: Possible truncated strings [p. 31]
- SA0027: Multiple usage of name [p. 31]
- SA0028: Overlapping memory areas [p. 31]
- SA0029: Notation in implementation different to declaration [p. 32]
- List unused objects
  - SA0031: Unused signatures [p. 32]
  - SA0032: Unused enumeration constants [p. 32]
  - SA0033: Unused variables [p. 33]
  - SA0035: Unused input variables [p. 33]
  - SA0036: Unused output variables [p. 34]
- SA0034: Enumeration variables with incorrect assignment [p. 34]
- SA0037: Write access to input variable [p. 35]
- SA0038: Read access to output variable [p. 35]
- SA0040: Possible division by zero [p. 36]
- SA0041: Possibly loop-invariant code [p. 36]
- SA0042: Usage of different access paths [p. 37]
- SA0043: Use of a global variable in only one POU [p. 37]
- **SA0044**: Declarations with reference to interface [38]

- **Conversions**
  - **SA0019**: Implicit pointer conversions [38]
  - **SA0130**: Implicit expanding conversions [39]
  - **SA0131**: Implicit narrowing conversions [39]
  - **SA0132**: Implicit signed/unsigned conversions [40]
  - **SA0133**: Explicit narrowing conversions [40]
  - **SA0134**: Explicit signed/unsigned conversions [41]

- **Usage of direct addresses**
  - **SA0005**: Invalid addresses and data types [41]
  - **SA0047**: Access to direct addresses [42]
  - **SA0048**: AT declarations on direct addresses [42]

- **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 [47]
  - **SA0059**: Comparison operations always returning TRUE or FALSE [48]
  - **SA0060**: Zero used as invalid operand [48]
  - **SA0061**: Unusual operation on pointer [49]
  - **SA0062**: Using TRUE and FALSE in expressions [49]
  - **SA0063**: Possibly not 16-bit-compatible operations [49]
  - **SA0064**: Addition of pointer [50]
  - **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]
    - **SA0080**: Loop index variable for array index exceeds array range [53]
- SA0081: Upper border is not a constant [53]

- **CASE statements**
  - SA0075: Missing ELSE [54]
  - SA0076: Missing enumeration constant [54]
  - SA0077: Type mismatches with CASE expression [55]
  - SA0078: Missing CASE branches [56]
  - SA0090: Return statement before end of function [56]

- SA0095: Assignments in conditions [56]
- SA0100: Variables greater than <n> bytes [58]
- SA0101: Names with invalid length [58]
- SA0102: Access to program/fb variables from the outside [59]
- SA0103: Concurrent access on not atomic data [60]
- SA0105: Multiple instance calls [61]
- SA0106: Virtual method calls in FB_init [61]
- SA0107: Missing formal parameters [63]

- **Check strict IEC rules**
  - SA0111: Pointer variables [63]
  - SA0112: Reference variables [63]
  - SA0113: Variables with data type WSTRING [64]
  - SA0114: Variables with data type LTIME [64]
  - SA0115: Declarations with data type UNION [64]
  - SA0117: Variables with data type BIT [65]
  - SA0119: Object-oriented features [65]
  - SA0120: Program calls [66]
  - SA0121: Missing VAR_EXTERNAL declarations [66]
  - SA0122: Array index defined as expression [67]
  - SA0123: Usages of INI, ADR or BITADR [67]
  - SA0147: Unusual shift operation - strict [67]
  - SA0148: Unusual bit access - strict [68]

- **Rules for initializations**
  - SA0118: Initializations not using constants [68]
  - SA0124: Dereference access in initializations [69]
  - SA0125: References in initializations [70]

- SA0140: Statements commented out [73]
- Possible use of uninitialized variables
  - SA0039: Possible null pointer dereferences [ponsive]
  - SA0046: Possible use of not initialized interface [responsive]
  - SA0145: Possible use of not initialized reference [responsive]
- SA0150: Violations of lower or upper limits of the metrics [responsive]
- SA0160: Recursive calls [responsive]
- SA0161: Unpacked structure in packed structure [responsive]
- SA0162: Missing comments [responsive]
- SA0163: Nested comments [responsive]
- SA0164: Multi-line comments [responsive]
- SA0165: Maximum number of input/output/VAR_IN_OUT variables [responsive]
- SA0166: Recursive calls [responsive]

Detailed description

SA0001: Unreachable code

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines code that is not executed, for example due to a RETURN or CONTINUE statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Unreachable code should be avoided in any case. The check often indicates the presence of test code, which should be removed.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP2</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines POU s, GVLs or data type declarations that do not contain code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Empty objects should be avoided. They are often a sign that an object is not fully implemented.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

### SA0003: Empty statements

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines lines of code containing a semicolon (;) but no statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>An empty statement can be an indication of missing code.</td>
</tr>
<tr>
<td>Exception</td>
<td>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 instruction is commented, the statistical code analysis does not generate an error message.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines outputs that are written at more than one position.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Exception</td>
<td>No error is issued if an output variable is written in different branches of IF or CASE statements.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP12</td>
</tr>
</tbody>
</table>

---

This rule cannot be deactivated by a pragma or an attribute!

Further information on attributes can be found under Pragmas and attributes [106].
Sample:

Global variable list:

```plaintext
VAR_GLOBAL
  bVar  AT%QX0.0  : BOOL;
  nSample AT%QW5  : INT;
END_VAR
```

Program MAIN:

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines variables with write access from more than one task.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Exception</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP10</td>
</tr>
</tbody>
</table>

See also rule SA0103 [60].

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'}

Sample:

The two global variables nVar and bVar are written by two tasks.
Global variable list:

<table>
<thead>
<tr>
<th>VAR_GLOBAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>nVar : INT;</td>
</tr>
<tr>
<td>bVar : BOOL;</td>
</tr>
<tr>
<td>END_VAR</td>
</tr>
</tbody>
</table>

Program MAIN_Fast, called from the task PlcTaskFast:

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

Program MAIN_Slow, called from the task PlcTaskSlow:

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

### SA0007: Address operators on constants

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines locations at which the ADR operator is used for a constant.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Exception</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

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:

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines function, method and property calls for which the return value is not used.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Exception</td>
<td>If a return value is of no interest during a call, this can be documented and the assignment can be omitted. Error returns should never be ignored!</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP7/CP17</td>
</tr>
</tbody>
</table>

Sample:

Function F_ReturnBOOL:

FUNCTION F_ReturnBOOL : BOOL
F_ReturnBOOL := TRUE;

Program MAIN:

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

SA0010: Arrays with only one component

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines arrays containing only a single component.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Exception</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

Samples:

PROGRAM MAIN
VAR
  aEmpty1 : ARRAY [0..0] OF INT;        // => SA0010
  aEmpty2 : ARRAY [15..15] OF REAL;    // => SA0010
END_VAR
### SA0011: Useless declarations

**Function**
Determines structures, unions or enumerations with at most one component.

**Reason**
Such a declaration can be confusing for a reader. 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:**

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

---

**Sample 2 – Union:**

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

---

**Sample 3 – Enumeration:**

```plaintext
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:**

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

nVar := nVar + nSample; // => SA0012
```
SA0013: Declarations with the same variable name

<table>
<thead>
<tr>
<th>Function</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>N5/N9</td>
</tr>
</tbody>
</table>

Samples:

Global variable list GVL_App:

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines assignments to function block instances. For instances with pointer or reference variables such assignments may be risky.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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. A copy method can be useful for selectively copying values from one instance to another: inst_First.Copy_From(inst_Second)</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Sample:

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

    fb1();
    fb2 := fb1; // => SA0014
```
### SA0015: Access to global data via FB_init

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines access of a function block to global data via the FB_init method. The value of this variable depends on the order of the initializations!</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Depending on the declaration location of the instance of a function block, a non-initialized variable may be accessed if the rule is violated.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>
TYPE ST_UnpaddedStructure2 :
    STRUCT
        bBOOL : WORD;
        nINT : INT;
        nBYTE : BYTE;
        nWORD : WORD; // => SA0016
    END_STRUCT
END_TYPE

SA0017: Non-regular assignments

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines assignments to pointers, which are not an address (ADR operator, pointer variables) or constant 0.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>If a pointer contains a value that is not a valid address, an access violation exception occurs when dereferencing the pointer.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines bit access to signed variables. However, the IEC 61131-3 standard only permits bit access to bit fields. See also strict rule SA0148 [68].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Exception</td>
<td>Exception for flag enumerations: If an enumeration is declared as flag via the pragma attribute 'flags', the error SA0018 is not issued for bit access with OR, AND or NOT operations.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>
SA0020: Possibly assignment of truncated value to REAL variable

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines operations on integer variables, during which a truncated value may be assigned to a variable of data type REAL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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: lrealvar := dintvar1 * dintvar2. 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: lreal_var := TO_LREAL(dintvar1) * TO_LREAL(dintvar2).</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

Sample:

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines assignments of addresses of temporary variables (variables on the stack) to non-temporary variables.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

Sample:

Method FB_Sample.SampleMethod:

```plaintext
METHOD SampleMethod : XWORD
VAR
  fVar  : LREAL;
END_VAR
SampleMethod := ADR(fVar);
```

Program MAIN:

```plaintext
PROGRAM MAIN
VAR
  nReturn   : XWORD;
  fbSample  : FB_Sample;
END_VAR
nReturn := fbSample.SampleMethod();  // => SA0021
```
SA0022: (Possibly) unassigned return value

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines all functions and methods containing an execution thread without assignment to the return value.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Sample:

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>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 memory space occupied).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Sample:

Structure ST_sample:

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

Example of functions with return value:

```plaintext
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/constants

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines untyped literals/constants (e.g. nCount : INT := 10;).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>TwinCAT assigns the types for literals according to their use. In some cases, this can lead to unexpected situations that are better clarified by a typed literal. Sample: dw := ROL(DWORD#1, i)</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Sample:**

```plaintext
PROGRAM MAIN
VAR
    nVar : INT;
    fVar : LREAL;
END_VAR
nVar := 100;           // => SA0024
nVar := INT#100;      // no error
fVar := 12.5;          // => SA0024
fVar := LREAL#12.5;   // no error
```

SA0025: Unqualified enumeration constants

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines enumeration constants that are not used with a qualified name, i.e. without preceding enumeration name.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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: &quot;eRed&quot;). 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'}.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

**Sample:**

```plaintext
Enumeration E_Color:

TYPE E_Color :
{
    eRed,
    eGreen,
    eBlue
};
END_TYPE

Program MAIN:

PROGRAM MAIN
VAR
    eColor : E_Color;
END_VAR
    eColor := E_Color.eGreen; // no error
    eColor := eGreen;         // => SA0025
```
SA0026: Possible truncated strings

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines string assignments and initializations that do not use an adequate string length.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>If strings of different lengths are assigned, a string may be truncated. The result is then not the expected one.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Samples:

```plaintext
PROGRAM MAIN
VAR
  sVar1 : STRING[10];
  sVar2 : STRING[6];
END_VAR
sVar2 := sVar1;                        // => SA0026
```

SA0027: Multiple usage of name

| Function | Determines multiple use of a variable name/identifier or object name (POU) within the scope of a project. The following cases are covered:
  • The name of an enumeration constant is the same as the name in another enumeration within the application or a referenced library.
  • The name of a variable that is the same as the name of an object within the application or a referenced library.
  • The name of a variable is the same as the name of an enumeration constant within the application or a referenced library.
  • The name of an object is the same as the name of another object within the application or a referenced library. |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Sample:

The following example generates error/warning SA0027, since the library Tc2_Standard is referenced in the project, which provides the function block TON.

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

SA0028: Overlapping memory areas

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines the points due to which two or more variables occupy the same memory space.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>If two variables occupy the same memory 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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>
Sample:

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

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

SA0029: Notation in implementation different to declaration

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines the code positions (in the implementation) at which the notation of an identifier differs from the notation in its declaration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>The IEC 61131-3 standard defines identifiers as not case-sensitive. This means that a variable declared as &quot;varx&quot; can also be used as &quot;VaRx&quot; in the code. However, this can be confusing and misleading and should therefore be avoided.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Samples:

Function F_TEST:

```plaintext
FUNCTION F_TEST : BOOL
```

Program MAIN:

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

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

SA0031: Unused signatures

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

SA0032: Unused enumeration constants

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines enumeration constants that are not used in the compiled program code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Unused enumeration constants result in unnecessary enumeration definition bloat and confusion when the program is read.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP24</td>
</tr>
</tbody>
</table>

Sample:

Enumeration E_Sample:
TYPE E_Sample :
{
    eNull,
    eOne,
    eTwo
};
END_TYPE

Program MAIN:

PROGRAM MAIN
VAR
eSample : E_Sample;
END_VAR
eSample := E_Sample.eNull;
eSample := E_Sample.eTwo;

SA0033: Unused variables

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines variables that are declared but not used within the compiled program code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP22/CP24</td>
</tr>
</tbody>
</table>

SA0035: Unused input variables

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines input variables that are not used externally by any function block instance.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP24</td>
</tr>
</tbody>
</table>

Sample:

Function block FB_Sample:

FUNCTION_BLOCK FB_Sample
VAR_INPUT
    nIn : INT;
    bIn : BOOL;  // => SA0035
END_VAR
VAR_OUTPUT
    nOut : INT;  // => SA0036
END_VAR

Program MAIN:

PROGRAM MAIN
VAR
    fbSample : FB_Sample;
END_VAR
fbSample(nIn := 99);
SA0036: Unused output variables

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines output variables that are not used externally by any function block instance.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP24</td>
</tr>
</tbody>
</table>

Sample:

Function block FB_Sample:

```plaintext
FUNCTION_BLOCK FB_Sample
  VAR_INPUT
    nIn : INT;
    bIn : BOOL; // => SA0035
  END_VAR
  VAR_OUTPUT
    nOut : INT; // => SA0036
  END_VAR
END_FUNCTION_BLOCK

PROGRAM MAIN:

VAR
  fbSample : FB_Sample;
END_VAR

fbSample(nIn := 99);
```

SA0034: Enumeration variables with incorrect assignment

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines values that are assigned to an enumeration variable. Only defined enumeration constants may be assigned to an enumeration variable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

Sample:

Enumeration E_Color:

```plaintext
TYPE E_Color :
  
  { eRed := 1,
    eBlue := 2,
    eGreen := 3
  };
END_TYPE
```

Program MAIN:

```plaintext
PROGRAM MAIN
 
VAR
  eColor : E_Color;
END_VAR

  eColor := E_Color.eRed;
eColor := eBlue;
eColor := 1; // => SA0034
```
SA0037: Write access to input variable

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines input variables (VAR_INPUT) that are subject to write access within the POU.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines output variables (VAR_OUTPUT) that are subject to read access within the POU.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

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 // => SA0038
      bLocal := (nOut > 100); // => SA0038
      nLocal := nOut; // => SA0038
      nLocal := 2*nOut; // => SA0038
    END_IF
```
SA0040: Possible division by zero

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines code positions at which division by zero may occur.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Division by 0 is not allowed. A variable that is used as a divisor should always be checked for 0 first. Otherwise, a &quot;Divide by Zero&quot; exception may occur at runtime.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

**Sample:**

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines code that may be loop-invariant, i.e. code within a FOR, WHILE or REPEAT loop that returns the same result in each loop, in which case repeated execution would be unnecessary. Only calculations are taken into account, no simple assignments.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>This is a performance warning. Code that is executed in a loop but performs a repetitive action in each loop can be executed outside the loop.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

**Sample:**

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

```plaintext
PROGRAM MAIN
VAR
  nTest1 : INT := 5;
nTest2 : INT := nTest1;
nTest3 : INT;
nTest4 : INT;
nTest5 : INT;
nTest6 : INT;
nCounter : INT;
END_VAR
FOR nCounter := 1 TO 100 BY 1 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
```
SA0042: Usage of different access paths

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines the usage of different access paths for the same variable.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

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
  nGlobal1 : INT;
END_VAR
```

Program MAIN:
```
PROGRAM MAIN
VAR
  sVar : STRING;
END_VAR

nGlobal1 := INT#2; // => SA0042
GVL.nGlobal1 := 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines global variables that are only used in a single POU.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>A global variable that is only used at one point should also be declared at this point.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP26</td>
</tr>
</tbody>
</table>

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;
```

Program MAIN:
```
SubProgram();
nGlobal1 := nGlobal2;
```
SA0044: Declarations with reference to interface

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines declarations with REFERENCE TO &lt;interface&gt; and declarations of VAR_IN_OUT variables with the type of an interface (realized implicitly via REFERENCE TO).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

Samples:

I_Sample is an interface defined in the project.

Function block FB_Sample:

```plaintext
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
END_FUNCTION_BLOCK
```

Program MAIN:

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

```

SA0019: Implicit pointer conversions

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines implicitly generated pointer data type conversions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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. Implicit data type conversions are reported with a different message.</td>
</tr>
<tr>
<td>Exception</td>
<td>BOOL ↔ BIT</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP25</td>
</tr>
</tbody>
</table>

Samples:

```plaintext
PROGRAM MAIN
  VAR
    nInt   : INT;
    nByte  : BYTE;
```

Because the C developer tries to avoid unexpected memory overwrites, the compiler reports an error, reporting the rule that prevents such access. The following rule can be deactivated by adding an attribute value to the statement:

```plaintext
{attribute 'analysis' := '-44'}
```

BECKHOFF
SA0130: Implicit expanding conversions

| Reason | 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:

\[
\text{lint} := \text{dint} * \text{dint};
\]

the compiler performs the implicit conversion only after the multiplication:

\[
\text{lint} := \text{TO_LINT}(	ext{dint} * \text{dint});
\]

An overflow is therefore truncated. If you want to prevent this, you can have the conversion performed earlier for the elements:

\[
\text{lint} := \text{TO_LINT}(	ext{dint}) \times \text{TO_LINT}(	ext{dint});
\]

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.

| Exception | BOOL ↔ BIT |
| Importance | low |

Samples:

```
PROGRAM MAIN
VAR
  nBYTE    : BYTE;
  nUSINT   : USINT;
  nUINT    : UINT;
  nINT     : INT;
  nUDINT   : UDINT;
  nDINT    : DINT;
  nULINT   : ULINT;
  nLINT    : LINT;
  nLWORD   : LWORD;
  fLREAL   : LREAL;
END_VAR

nLINT := nINT;  // => SA0130
nULINT := nBYTE; // => SA0130
nLWORD := nUDINT; // => SA0130
fLREAL := nBYTE;  // => SA0130
nDINT := nUINT;  // => SA0130

nBYTE.5 := FALSE; // no error (BIT-BOOL-conversion)
```
Configuration

This message is now obsolete because it is already reported as a warning by the compiler.

Sample:

```plaintext
PROGRAM MAIN
VAR
  fREAL    : REAL;
  flREAL   : LREAL;
END_VAR
fREAL := flREAL;           // => SA0131
nBYTE.5 := FALSE;          // no error (BIT-BOOL-conversion)

SA0132: Implicit signed/unsigned conversions

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines implicitly performed conversions from signed to unsigned data types or vice versa.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

This message is now obsolete because it is already reported as a warning by the compiler.

Samples:

```plaintext
PROGRAM MAIN
VAR
  nBYTE    : BYTE;
  nUDINT   : UDINT;
  nULINT   : ULINT;
  nWORD    : WORD;
  nLWORD   : LWORD;
  nSINT    : SINT;
  nINT     : INT;
  nDINT    : DINT;
  nLINT    : LINT;
END_VAR
nLINT := nULINT;              // => SA0132
nUDINT := nDINT;              // => SA0132
nSINT  := nBYTE;              // => SA0132

SA0133: Explicit narrowing conversions

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines explicitly performed conversions from a larger to a smaller data type.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

Samples:
**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:**

```plaintext
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**
Determines invalid address and data type specifications.

The following size prefixes are valid for addresses. Deviations from this lead to an invalid address specification.

- **X** for BOOL
- **B** for 1-byte data types
- **W** for 2-byte data types
- **D** for 4-byte data types

**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:**

```plaintext
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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines direct address access operations in the implementation code.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>N1/CP1</td>
</tr>
</tbody>
</table>

**Samples:**

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

```plaintext
  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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines AT declarations on direct addresses.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>N1/CP1</td>
</tr>
</tbody>
</table>

**Samples:**

```plaintext
PROGRAM MAIN
VAR
  b1    AT%IX0.0 : BOOL;  // => SA0048
  b2    AT%I*  : BOOL;    // no error
END_VAR
```
SA0051: Comparison operations on BOOL variables

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines comparison operations on variables of type BOOL.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Sample:

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>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 SA0147[67].</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Exception</td>
<td>Shift operation on bit array data types (byte, DWORD, LWORD, WORD) do not result in a SA0052 error.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Samples:

```plaintext
PROGRAM MAIN
VAR
  nINT   : INT;
  nDINT  : DINT;
  nULINT : ULIMIT;
  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#4);   // 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:

```plaintext
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**: REAL/LREAL values are implemented as floating-point numbers according to 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.

Sample:

```plaintext
lr11 := 1.1;
lr33 := 3.3;
lrVar1 := lr11 + lr11;
lrVar2 := lr33 - lr11;
bTest := lrVar1 = lrVar2;
```

bTest will return FALSE in this case, even if the variables lrVar1 and lrVar2 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:

```plaintext
bTest := ABS(lrVar1 - lrVar2) < 0.1;
```

**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:

```plaintext
PROGRAM MAIN
VAR
  fREAL1 : REAL;
  fREAL2 : REAL;
  fLREAL1 : LREAL;
```
Configuration

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines unnecessary comparisons with unsigned operands. An unsigned data type is never less than zero.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>A comparison revealed by this check provides a constant result and indicates an error in the code.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines literals (constants) outside the valid operator range.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

TE1200  Version: 2.5  45
Samples:

PROGRAM MAIN
VAR
   nBYTE : BYTE;
   rWORD : WORD;
   rDWORD : 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 > UINT#355;        // => SA0056
bResult := nUINT >= UDINT#70000;     // => SA0056
bResult := nUDINT > ULINT#4294967300; // => SA0056

SA0057: Possible loss of decimal points

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines statements with possible loss of decimal points.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>A piece of code of the following type:</td>
</tr>
<tr>
<td></td>
<td>diTemp2 := 1  rTemp1 := TO_REAL(diTemp2 / DINT#2)</td>
</tr>
<tr>
<td></td>
<td>can lead to misinterpretation. This line of code can lead to</td>
</tr>
<tr>
<td></td>
<td>the assumption that the division would be performed as a REAL</td>
</tr>
<tr>
<td></td>
<td>operation and the result in this case would be REAL#0.5.</td>
</tr>
<tr>
<td></td>
<td>However, this is not the case, i.e. the operation is</td>
</tr>
<tr>
<td></td>
<td>performed as an integer operation, the result is cast to</td>
</tr>
<tr>
<td></td>
<td>REAL, and rTemp1 is assigned the value REAL#0. To avoid this,</td>
</tr>
<tr>
<td></td>
<td>you should use a cast to ensure that the operation is</td>
</tr>
<tr>
<td></td>
<td>performed as a REAL operation:</td>
</tr>
<tr>
<td></td>
<td>rTemp1 := TO_REAL(diTemp2) / REAL#2.</td>
</tr>
</tbody>
</table>

| 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines operations on variables of type enumeration. Assignments are permitted.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Enumerations should not be used as normal integer values. Alternatively, an alias data type can be defined or a subrange type can be used.</td>
</tr>
<tr>
<td>Exception</td>
<td>If an enumeration is marked with the attribute (attribute 'strict'), the compiler reports such an operation.</td>
</tr>
<tr>
<td></td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Sample 1:

Enumeration E_Color:

```plaintext
TYPE E_Color :
{
    eRed := 1,
    eBlue := 2,
    eGreen := 3
};
END_TYPE
```

Program MAIN:

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

```plaintext
eColor := E_Color.Green;          // no error
eColor := E_Color.Green + 1;      // ⇒ SA0058
nVar := E_Color.Blue / 2;         // ⇒ SA0058
nVar := E_Color.Green + E_Color.Red; // ⇒ SA0058
```

Sample 2:

Enumeration E_State with attribute 'flags':

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

Program MAIN:

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

```plaintext
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 that always have the result TRUE or FALSE and 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:

```plaintext
PROGRAM MAIN
VAR
   nBYTE   : BYTE;
   nWORD   : WORD;
   nDWORD  : DWORD;
   nLWORD  : LWORD;
   nUSINT  : USINT;
   nUINT   : UINT;
   nUDINT  : UDINT;
   nLINT   : LINT;
   nINT    : INT;
   nDINT   : DINT;
   nLWORD  : LWORD;
   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 := 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:

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines operations on variables of type POINTER TO, which are not = (equality), &lt;&gt; (inequality), + (addition) or ADR.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>E2/E3</td>
</tr>
</tbody>
</table>

**Samples:**

```plaintext
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: Using TRUE and FALSE in expressions

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines the use of the literal TRUE or FALSE in expressions (e.g. b1 AND NOT TRUE).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Such an expression is obviously unnecessary and may indicate an error. In any case, the expression causes an unnecessary load on the runtime.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

**Samples:**

```plaintext
PROGRAM MAIN
VAR
  bVar1 : BOOL;
  bVar2 : BOOL;
END_VAR
bVar1 := bVar1 AND NOT TRUE;      // => SA0062
bVar2 := bVar1 OR TRUE;          // => SA0062
bVar2 := bVar1 OR NOT FALSE;     // => SA0062
bVar2 := bVar1 AND FALSE;        // => SA0062
```

SA0063: Possibly not 16-bit-compatible operations

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines 16-bit operations with intermediate results. Background: 32-bit intermediate results may be truncated on 16-bit systems.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Sample:**

(nVar+10) may exceed 16 bits.
PROGRAM MAIN
VAR
  nVar : INT;
END_VAR

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

SA0064: Addition of pointer

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines all pointer additions.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Pointer arithmetic is generally permitted in TwinCAT and can be used in a meaningful way. However, it is also a source of error. Therefore, there are programming rules that prohibit pointer arithmetic. Such a requirement can be verified with this test.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

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

SA0065: Incorrect pointer addition to base size

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines pointer additions in which the value to be added does not match the basic data size of the pointer. Only literals with the basic size may be added. No multiples of the basic size may be added.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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. Sample in ST:</td>
</tr>
<tr>
<td></td>
<td>This code would work differently in C:</td>
</tr>
<tr>
<td></td>
<td>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 non-aligned memory, which (depending on the processor) can lead to an alignment exception during access.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

Samples:
**SA0066: Use of temporary results**

<table>
<thead>
<tr>
<th>Function</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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: usintTest := 0; bError := usintTest - 1 &lt;&gt; 255; 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#ffffffff is then displayed and this is not equal to 255. To avoid this, you have to explicitly cast the intermediate result: bError := TO_USINT(usintTest - 1) &lt;&gt; 255;</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

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 typcast in all reported situations, the code will be much slower and less readable!

**Sample:**

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines write access operations to a counter variable within a FOR loop.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>L12</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines the use of non-temporary variables in FOR loops.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP21/L13</td>
</tr>
</tbody>
</table>

Sample:
PROGRAM MAIN
VAR
  nIndex : INT;
nSum : INT;
END_VAR

FOR nIndex := 1 TO 10 BY 1 DO    // => SA0073
  nSum := nSum + nIndex;
END_FOR

SA0080: Loop index variable for array index exceeds array range

Function Determines FOR statements in which the index variable is used for access to an array index and exceeds the array index range.

Reason Arrays are typically processed in FOR loops. The start and end value of the counter variable should typically match or at least not exceed the lower and upper limits of the array. Here a typical cause of error is detected if array boundaries are changed and constants are not handled carefully, or if a different value is used by mistake in the FOR loop than in the array declaration.

Importance high

Samples:

PROGRAM MAIN
VAR CONSTANT
c1 : INT := 0;
END_VAR
VAR
  nIndex1 : INT;
nIndex2 : INT;
nIndex3 : INT;
a1 : ARRAY[1..100] OF INT;
a2 : ARRAY[1..9,1..9,1..9] OF INT;
a3 : ARRAY[0..99] OF INT;
END_VAR
// 1 violation of the rule (lower range is exceeded) => 1 error SA0080
FOR nIndex1 := c1 TO INT#100 BY INT#1 DO
  a1[nIndex1] := nIndex1;             // => SA0080
END_FOR
// 6 violations (lower and upper range is exceeded for each array dimension) => 3 errors SA0080
FOR nIndex2 := INT#0 TO INT#10 BY INT#1 DO
  a2[nIndex2, nIndex2, nIndex2] := nIndex2; // => SA0080
END_FOR
// 1 violation (upper range is exceeded by the end result of the index), expressions on index are not evaluated => no error
FOR nIndex3 := INT#0 TO INT#50 BY INT#1 DO
  a3[nIndex3 * INT#2] := nIndex3;      // no error
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
### SA0075: Missing ELSE

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines CASE statements without ELSE branch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>L17</td>
</tr>
</tbody>
</table>

**Sample:**

```plaintext
PROGRAM MAIN
VAR
  nVar : INT;
  bVar : BOOL;
END_VAR
nVar := nVar + INT#1;
CASE nVar OF
  // => SA0075
    INT#1:
      bVar := FALSE;
    INT#2:
      bVar := TRUE;
END_CASE
```

### SA0076: Missing enumeration constant

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines code positions where an enumeration variable is used as condition and not all enumeration values are treated as CASE branches.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Sample:**

```plaintext
PROGRAM MAIN
VAR
  nVar : INT;
  bVar : BOOL;
END_VAR
nVar := nVar + INT#1;
CASE nVar OF
  // => SA0076
    INT#1:
      IF nVar = 10 THEN
        nMax2 := 50;
      END_IF
END_CASE
```
In the following sample the enumeration value eYellow is not treated as a CASE branch.

**Enumeration E_Color:**

```plaintext
TYPE E_Color :
{
  eRed,
  eGreen,
  eBlue,
  eYellow
};
END_TYPE
```

**Program MAIN:**

```plaintext
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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines code positions where the data type of a condition does not match that of the CASE branch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>If the data types between the CASE variable and the CASE case do not match, this could indicate an error.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Sample:**

**Enumeration E_Sample:**

```plaintext
TYPE E_Sample :
{
  eNull,
  eOne,
  eTwo
} DWORD;
END_TYPE
```

**Program MAIN:**

```plaintext
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;
```
**SA0078: Missing CASE branches**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines CASE statements without cases, i.e. with only a single ELSE statement.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>A CASE statement without cases wastes execution time and is difficult to read.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

**Sample:**

```plaintext
PROGRAM MAIN
VAR
  nVar   : DINT;
  bVar   : BOOL;
END_VAR
nVar := nVar + INT#1;
CASE nVar OF                     // => SA0078
  bVar := NOT bVar;
ELSE
  bVar := NOT bVar;
END_CASE
```

**SA0090: Return statement before end of function**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines code positions where the RETURN statement is not the last statement in a function, method, property or program.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP14</td>
</tr>
</tbody>
</table>

**Sample:**

```plaintext
FUNCTION F_TestFunction : BOOL
F_TestFunction := FALSE;
RETURN;         // => SA0090
F_TestFunction := TRUE;
```

**SA0095: Assignments in conditions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines assignments in conditions of IF, CASE, WHILE or REPEAT constructs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

**Samples:**

```plaintext
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
  DoSomething();
END_IF

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

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

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

IF bCond := TRUE THEN
  DoSomething();
END_IF

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

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

// CASE construct
CASE (eSample := E_Sample.eMember0) OF
  E_Sample.eMember0:
    DoSomething();
  E_Sample.eMember1:
    DoSomething();
END_CASE

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

// REPEAT construct
REPEAT
  DoSomething();
UNTIL (bCond = TRUE) OR ((nCond1 := nCond2) = 12)
END_REPEAT
SA0100: Variables greater than <n> bytes

**Function**
Determines variables that use more than n bytes; n is defined by the current configuration.

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 > category "Static Analysis" > "Rules" tab > Rule 100). You can make the following settings in the dialog that appears:

- Upper limit in bytes (default value: 1024)

**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**
Determines names with invalid length. The object names must have a defined length.

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 > category "Static Analysis" > "Rules" tab > Rule 101). You can make the following settings in the dialog that appears:

- Minimum number of characters (default value: 5)
- Maximum number of characters (default value: 30)
- Exceptions

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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines external access to local variables of programs or function blocks.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Samples:

Function block FB_Base:

```plaintext
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:

```plaintext
FUNCTION_BLOCK FB_Sub EXTENDS FB_Base

Method FB_Sub.SampleMethod:

METHOD SampleMethod : INT
VAR_INPUT
END_VAR
nLocal := nLocal + 5;
```

Program PRG_1:

```plaintext
PROGRAM PRG_1
VAR
  bLocal : BOOL;
END_VAR
bLocal := NOT bLocal;
```

Program MAIN:

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Exception</td>
<td>This rule does not apply in the following cases:</td>
</tr>
<tr>
<td></td>
<td>• If the target system has an FPU (floating point unit), the access of several tasks to LREAL variables is not determined and reported.</td>
</tr>
<tr>
<td></td>
<td>• If the target system is a 64-bit processor or &quot;TwinCAT RT (x64)&quot; is selected as the solution platform, the rule does not apply for 64-bit data types.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

See also rule SA0006 [21].

Samples:

Structure ST_sample:

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

Function block FB_Sample:

```plaintext
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优点"
    GVL.aTest := GVL.aTest2;
    GVL.fbTest.fInput := 3;
    GVL.stTest.nTest := GVL.stTest.nTest + 1;

  PROGRAM MAIN2, called by task PlcTask2:
```
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 attribute [attribute 'analysis:report-multiple-instance-call'] 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

Program MAIN:
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:

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

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

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

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

FUNCTION_BLOCK FB_Sub EXTENDS FB_Base
VAR
  nSub : DINT;
END_VAR

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

Program MAIN:

```plaintext
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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines where formal parameters are missing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Code becomes more readable if the formal parameters are specified when the code is called.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Sample:**

**Function F_Sample:**

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

**Program MAIN:**

```plaintext
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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines variables of type POINTER TO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>The IEC 61131-3 standard does not allow pointers.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Sample:**

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

**SA0112: Reference variables**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines variables of type REFERENCE TO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>The IEC 61131-3 standard does not allow references.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>
Sample:

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

**SA0113: Variables with data type WSTRING**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines variables of type WSTRING.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Not all systems support WSTRING. The code becomes easier to port if WSTRING is not used.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

Sample:

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

**SA0114: Variables with data type LTIME**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines variables of type LTIME.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Not all systems support LTIME. The code becomes more portable if LTIME is not used.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

Sample:

```plaintext
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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines declarations of a UNION data type and declarations of variables of the type of a UNION.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>The IEC-61131-3 standard has no provision for unions. The code becomes easier to port if there are no unions.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Samples:**

Union U_Sample:

```plaintext
TYPE U_Sample :
    UNION
       fVar : LREAL;
       nVar : LINT;
END_UNION
END_TYPE
```

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

**SA0117: Variables with data type BIT**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines declarations of variables of type BIT (possible within structure and function block definitions).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>The IEC-61131-3 has no provision for data type BIT. The code becomes easier to port if BIT is not used.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Samples:**

**Structure ST_sample:**

```plaintext
TYPE ST_Sample :
  STRUCT
    bBIT   : BIT;           // => SA0117
    bBOOL  : BOOL;
  END_STRUCT
END_TYPE
```

**Function block FB_Sample:**

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

**SA0119: Object-oriented features**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines the use of object-oriented features such as:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Function block declarations with EXTENDS or IMPLEMENTS</td>
</tr>
<tr>
<td></td>
<td>• Property and interface declarations</td>
</tr>
<tr>
<td></td>
<td>• Use of the THIS or SUPER pointer</td>
</tr>
<tr>
<td>Reason</td>
<td>Not all systems support object-oriented programming. The code becomes easier to port if object orientation is not used.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Samples:**

**Interface I_Sample:**

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

**Function block FB_Base:**

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

**Function block FB_Sub:**

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

**Method FB_Sub.SampleMethod:**

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

Get function of the property FB_Sub.SampleProperty:
**SA0120: Program calls**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines program calls.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Sample:**

Program SubProgram:

```plaintext
PROGRAM SubProgram
```

Program MAIN:

```plaintext
PROGRAM MAIN
SubProgram();  // => SA0120
```

**SA0121: Missing VAR_EXTERNAL declarations**

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

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:

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

Program Prog1:

```plaintext
PROGRAM Prog1
VAR
  nVar : INT;
END_VAR
nVar := nGlobal;  // => SA0121
```

Program Prog2:

```plaintext
PROGRAM Prog2
VAR
  nVar : INT;
```
SA0122: Array index defined as expression

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines the use of expressions in the declaration of array boundaries.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Not all systems allow expressions as array boundaries.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Sample:**

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines the use of the (TwinCAT-specific) operators INI, ADR, BITADR.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>TwinCAT-specific operators prevent portability of the code.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

**Sample:**

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

SA0147: Unusual shift operation - strict

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines bit shift operations that are not performed on bit field data types (BYTE, WORD, DWORD, LWORD).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
</tbody>
</table>

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 [27].

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

Version: 2.5

68
Program MAIN:

```plaintext
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

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines all code locations where dereferenced pointers are used in the declaration part of POUs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Pointers and references should not be used for initializations, because this can lead to access violations at runtime if the pointer has not been initialized.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Samples:

```plaintext
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''
  '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 [69]
- Possible null pointer dereferences in the implementation part => SA0039 [70]

References:
- Use of references in the declaration part => SA0125 [70]
- Possible use of not initialized reference in the implementation part => SA0145 [72]

Interfaces:
- Possible use of not initialized interface in the implementation part => SA0046 [71]
SA0125: References in initializations

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines all reference variables used for initialization in the declaration part of POUs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Pointers and references should not be used for initializations, because this can lead to access violations at runtime if the pointer has not been initialized.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

Samples:

FUNCTION_BLOCK FB_Test
VAR_INPUT
  pStruct : POINTER TO ST_Test;
  refStruct : REFERENCE TO ST_Test;
END_VAR
VAR
  bPointer := pStruct^.bTest;  // => SA0124: Dereference access in initialization
  bRef := 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
- Possible null pointer dereferences in the implementation part => SA0039

References:
- Use of references in the declaration part => SA0125
- Possible use of not initialized reference in the implementation part => SA0145

Interfaces:
- Possible use of not initialized interface in the implementation part => SA0046

SA0039: Possible null pointer dereferences

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines code positions at which a NULL-pointer may be dereferenced.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>A pointer should be checked before each dereferencing to see if it is not equal to 0. Otherwise an access violation may occur at runtime.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>
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
• Possible null pointer dereferences in the implementation part => SA0039

References:

• Use of references in the declaration part => SA0125
• Possible use of not initialized reference in the implementation part => SA0145

Interfaces:

• Possible use of not initialized interface in the implementation part => SA0046

SA0046: Possible use of not initialized interface

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines the use of interfaces that may not have been initialized before the use.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>An interface reference should be checked for &lt;&gt; 0 before it is used, otherwise an access violation may occur at runtime.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

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 [69]
- Possible null pointer dereferences in the implementation part => SA0039 [70]

References:
- Use of references in the declaration part => SA0125 [70]
- Possible use of not initialized reference in the implementation part => SA0145 [72]

Interfaces:
- Possible use of not initialized interface in the implementation part => SA0046 [71]

SA0145: Possible use of not initialized reference

<table>
<thead>
<tr>
<th>Function</th>
<th>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 POUs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>A reference should be checked for validity before it is accessed, otherwise an access violation may occur at runtime.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

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 [69]
- Possible null pointer dereferences in the implementation part => SA0039 [70]

References:
- Use of references in the declaration part => SA0125 [70]
- Possible use of not initialized reference in the implementation part => SA0145 [72]

Interfaces:
- Possible use of not initialized interface in the implementation part => SA0046 [71]

SA0140: Statements commented out

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines statements that are commented out.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>C4</td>
</tr>
</tbody>
</table>

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

SA0150: Violations of lower or upper limits of the metrics

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines function blocks that violate the enabled metrics at the lower or upper limit.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Code that adheres to certain metrics is easier to read, easier to maintain and easier to test.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP9</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines recursive calls of programs, actions, methods and properties. Determines possible recursions through virtual function calls and interface calls.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>Recursions lead to non-deterministic behavior and are therefore a source of errors.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP13</td>
</tr>
</tbody>
</table>

**Sample 1:**

Method `FB_Sample.SampleMethod1`:

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

Method `FB_Sample.SampleMethod2`:

```plaintext
METHOD SampleMethod2 : BOOL
VAR_INPUT
END_VAR
```

Program `MAIN`:

```plaintext
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`:

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

Set function of the property `SampleProperty`:

```plaintext
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`:

```plaintext
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**

<table>
<thead>
<tr>
<th>Function</th>
<th>Determines unpacked structures that are used in packed structures.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>An unpacked structure is usually placed by the compiler 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.</td>
</tr>
<tr>
<td>Importance</td>
<td>high</td>
</tr>
</tbody>
</table>

**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.

```plaintext
{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 | Determines points in the program that are not commented. Comments are required for:
|          | • the declaration of variables. The comments are shown above or to the right.
|          | • the declaration of POUs, DUTs, GVLs or interfaces. The comments are shown above
|          | the declaration (in the first row).
| 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 for the variable b1: "SA0162: Missing comment for 'b1'".

```c
// 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 '<...>'".

```c
(* 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

<table>
<thead>
<tr>
<th>Function</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>low</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>C5</td>
</tr>
</tbody>
</table>

You can deactivate this check with the `pragma (analysis ...) [^107]`, including for comments in the declaration part.

#### Samples:

```plaintext
(* 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 | 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. 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 > category "Static Analysis" > "Rules" tab > Rule 166). You can make the following settings in the dialog that appears:  
• Maximum number of inputs (default value: 10)  
• Maximum number of outputs (default value: 10)  
• Maximum number of inputs/outputs (default value: 10) |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
<tr>
<td>PLCopen rule</td>
<td>CP23</td>
</tr>
</tbody>
</table>

#### 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**

<table>
<thead>
<tr>
<th>Function</th>
<th>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.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reason</td>
<td>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. Secondly, function block instances are frequently large and require a great deal of space on the stack (which is usually limited on controllers). Thirdly, the initialization and often also the scheduling of the function block can take up quite a lot of time.</td>
</tr>
<tr>
<td>Importance</td>
<td>medium</td>
</tr>
</tbody>
</table>

**Samples:**

**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
```

**Program MAIN:**

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

In the naming conventions tab you can define naming conventions. Their compliance is accounted for in the static analysis execution [98]. 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.

**Configuration of the naming conventions**

<table>
<thead>
<tr>
<th>Names</th>
<th>Nodes and elements for which a prefix can be defined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>The number in brackets after each element, for example &quot;PROGRAM (102)&quot;, is the prefix convention number that is output if the naming convention is not followed.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prefix</th>
<th>You can define the naming conventions by entering the required prefix in this column. Please note the following notes and options:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Several possible prefixes per line</td>
</tr>
<tr>
<td></td>
<td>• Multiple prefixes can be entered separated by commas.</td>
</tr>
<tr>
<td></td>
<td>• Sample: &quot;x, b&quot; as prefixes for variables of data type BOOL. &quot;x&quot; and &quot;b&quot; may be used as prefix for Boolean variables.</td>
</tr>
<tr>
<td></td>
<td>• Regular expressions</td>
</tr>
<tr>
<td></td>
<td>• You can also use regular expressions (RegEx) for the prefix. In this case you have to use @ as additional prefix.</td>
</tr>
<tr>
<td></td>
<td>• Sample: &quot;:@b[a-dA-D]&quot; as prefix for variables of data type BOOL. The name of the boolean variable must start with &quot;b&quot;, and may be followed by a character in the range &quot;a-dA-D&quot;.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Prefixes for variables</th>
<th>Organizational node for all data types and scopes of variables for which a prefix can be defined.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefixes for POU{}{}{}</td>
<td>Organizational node for all POU{}{}{} types and method validity ranges for which a prefix can be defined.</td>
</tr>
<tr>
<td>Prefixes for DUTs{}{}{}{}</td>
<td>Organizational node for all DUT{}{}{} data types for which a prefix can be defined.</td>
</tr>
</tbody>
</table>
Formation of the expected prefix

The prefix expected for the different declarations is formed depending on the configuration of the options found in the Naming conventions (2) page. On the Naming conventions (2) 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\}, 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.

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 further information please refer to Pragmas and attributes.

Overview of naming conventions

An overview of naming conventions can be found under Naming conventions – overview and description.

4.3.1 Naming conventions – overview and description

Overview

- Prefixes for variables
  - Prefixes for types
    - NC0003: BOOL
    - NC0004: BIT
    - NC0005: BYTE
    - NC0006: WORD
    - NC0007: DWORD
    - NC0008: LWORD
- NC0013: SINT [83]
- NC0014: INT [83]
- NC0015: DINT [83]
- NC0016: LINT [83]
- NC0009: USINT [83]
- NC0010: UINT [83]
- NC0011: UDINT [83]
- NC0012: ULINT [83]
- NC0017: REAL [83]
- NC0018: LREAL [83]
- NC0019: STRING [83]
- NC0020: WSTRING [83]
- NC0021: TIME [83]
- NC0022: LTIME [83]
- NC0023: DATE [83]
- NC0024: DATE_AND_TIME [83]
- NC0025: TIME_OF_DAY [83]
- NC0026: POINTER [83]
- NC0027: REFERENCE [83]
- NC0028: SUBRANGE [84]
- NC0030: ARRAY [84]
- NC0031: Function block instance [84]
- NC0036: Interface [84]
- NC0032: Structure [84]
- NC0029: ENUM [85]
- NC0033: Alias [85]
- NC0034: Union [85]
- NC0035: XWORD [83]
- NC0037: UXINT [83]
- NC0038: XINT [83]

- Prefixes for scopes
  - NC0051: VAR_GLOBAL [86]
  - NC0070: VAR_GLOBAL CONSTANT [86]
- NC0071: VAR_GLOBAL RETAIN [▶ 86]
- NC0072: VAR_GLOBAL PERSISTENT [▶ 86]
- NC0073: VAR_GLOBAL RETAIN PERSISTENT [▶ 86]

**VAR**

- NC0053: Program variables [▶ 86]
- NC0054: Function block variables [▶ 86]
- NC0055: Function/method variables [▶ 86]

- NC0056: VAR_INPUT [▶ 86]
- NC0057: VAR_OUTPUT [▶ 86]
- NC0058: VAR_IN_OUT [▶ 86]
- NC0059: VAR_STAT [▶ 86]
- NC0061: VAR_TEMP [▶ 86]
- NC0062: VAR CONSTANT [▶ 86]
- NC0063: VAR PERSISTENT [▶ 86]
- NC0064: VAR RETAIN [▶ 86]
- NC0065: I/O variables [▶ 86]

- **Prefixes for POUs**

  - **Prefixes for POU type**
    - NC0102: PROGRAM [▶ 86]
    - NC0103: FUNCTIONBLOCK [▶ 86]
    - NC0104: FUNCTION [▶ 86]
    - NC0105: METHOD [▶ 86]
    - NC0106: ACTION [▶ 86]
    - NC0107: PROPERTY [▶ 86]
    - NC0108: INTERFACE [▶ 86]

  - **Method/property scope**
    - NC0121: PRIVATE [▶ 87]
    - NC0122: PROTECTED [▶ 87]
    - NC0123: INTERNAL [▶ 87]
    - NC0124: PUBLIC [▶ 87]

- **Prefixes for DUTs**
  - NC0151: Structure [▶ 87]
  - NC0152: Enumeration [▶ 87]
  - NC0153: Union [▶ 87]
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 Naming conventions (2) [88].

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 [98].

```plaintext
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
- 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 [98].

```plaintext
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 [98].

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 [98].

```
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 [98].

```
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 [98].

```
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 [98].

```
iSample   : I_Sample;
```

**NC0032: Structure**

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

Declaration of a structure:

```pascal
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 [98].

```pascal
stSample : ST_Sample;
```

**NC0029: ENUM**

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

Sample declaration:

Declaration of an enumeration:

```pascal
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 [98].

```pascal
eSample : E_Sample;
```

**NC0033: Alias**

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

Sample declaration:

Declaration of an alias:

```pascal
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 [98].

```pascal
tMessage : T_Message;
```

**NC0034: Union**

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

Sample declaration:

Declaration of a union:

```pascal
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 [98].

```pascal
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 [98].

```plaintext
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 [98].

```plaintext
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
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

### 4.3.2 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 POUs 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 Naming conventions (2)

The Naming Conventions (2) tab contains options that extend the settings of the Naming conventions tab. You can use these options to configure how the expected overall prefix for variables/declarations is to be composed.

The observance of the naming conventions is checked during the execution of the Static Analysis.

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.
   • Standard setting: disabled

Examples:
• Variable "bvar" with the expected prefix "b"
• Function block "FB_sample" with the expected prefix "FB_"

<table>
<thead>
<tr>
<th>Option</th>
<th>State</th>
<th>Result of the static analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>First character after prefix should be an upper case letter</td>
<td>Enabled</td>
<td>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 &quot;bVar&quot; and &quot;FB_Sample&quot;.</td>
</tr>
<tr>
<td></td>
<td>Disabled</td>
<td>The identifiers &quot;bvar&quot; and &quot;FB_sample&quot; are permissible. No upper/lower case error is output.</td>
</tr>
</tbody>
</table>

2) Recursive prefixes for combinable data types
   • If enabled: Variables of the combinable data types (POINTER, REFERENCE, ARRAY, SUBRANGE) must have a composite date 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 date type prefix.
   • Standard 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 namespace** 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 namespace** 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.

• Standard setting: enabled

**Examples:**

• Prefix configuration for data types:
  - POINTER (26) = "p"
  - ARRAY (30) = "a"
  - INT (14) = "n"
  - BOOL (3) = "b"

• Prefix configuration for namespace/area of validity/scope
  - Case 1: Function block variables (54) = "_local_
  - Case 2: Function block variables (54) = empty box/not configured

Further examples of a namespace/area of validity/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;
  END_VAR

**Option scenario 1:**

<table>
<thead>
<tr>
<th>Option</th>
<th>State</th>
<th>Expected overall prefix for case 1 (NC0054 = &quot;<em>local</em>&quot;)</th>
<th>Expected overall prefix for case 2 (NC0054 = empty)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Recursive prefixes for combinable data types</td>
<td>Enabled</td>
<td>For var1: 'pan'</td>
<td>For var1: 'pan'</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For var2: 'aab'</td>
<td>For var2: 'aab'</td>
</tr>
<tr>
<td>Combine scope prefix with data type prefix</td>
<td>Enabled</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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 1:

<table>
<thead>
<tr>
<th>Option</th>
<th>State</th>
<th>Expected overall prefix for case 1 (NC0054 = &quot;<em>local</em>&quot;)</th>
<th>Expected overall prefix for case 2 (NC0054 = empty)</th>
</tr>
</thead>
</table>
| 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 thus "p" or "a" respectively.

- 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:

<table>
<thead>
<tr>
<th>Option</th>
<th>State</th>
<th>Expected overall prefix for case 1 (NC0054 = &quot;<em>local</em>&quot;)</th>
<th>Expected overall prefix for case 2 (NC0054 = empty)</th>
</tr>
</thead>
</table>
| 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" respectively 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" respectively as the expected overall prefix.

### Option scenario 4:

<table>
<thead>
<tr>
<th>Option</th>
<th>State</th>
<th>Expected overall prefix for case 1 (NC0054 = &quot;<em>local</em>&quot;)</th>
<th>Expected overall prefix for case 2 (NC0054 = empty)</th>
</tr>
</thead>
</table>
| 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" respectively 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" respectively as the expected overall prefix.

**Further notes/samples:**

For POUs 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**. Examples:

- 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.

### 4.5 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' is executed.

#### 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 | You can enable or disable the individual metrics using the checkbox for the respective row. When command 'View Standard Metrics' [101] is executed, the metrics that are enabled in the respective configuration are shown for each programming block in the Standard Metrics view.  
|        | • [ ]: The metric is disabled and is not displayed in the Standard Metrics view when the command View Standard Metrics is executed.  
|        | • [✓]: The metric is enabled and is displayed in the Standard Metrics view when the command View Standard Metrics is executed.  

| Lower limit | For each metric you can define an individual upper and lower limit by entering the required number in the respective metric row.  
| Upper limit | 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. |

**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' [101].  
  - 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 [14] tab.  
  - Perform the static analysis (see: Command 'Run static analysis' [98]).  
  - 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 under Metrics - overview and description [92].

### 4.5.1 Metrics - overview and description

**Overview**
<table>
<thead>
<tr>
<th>Column abbreviation in Standard Metrics view</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code size</td>
<td>Code size [number of bytes] (&quot;code size&quot;) [93]</td>
</tr>
<tr>
<td>Variables size</td>
<td>Variables size [number of bytes] (&quot;variables size&quot;) [93]</td>
</tr>
<tr>
<td>Stack size</td>
<td>Stack size [number of bytes] (&quot;stack size&quot;) [93]</td>
</tr>
<tr>
<td>Calls</td>
<td>Number of calls (&quot;calls&quot;) [94]</td>
</tr>
<tr>
<td>Tasks</td>
<td>Called in tasks (&quot;tasks&quot;) [94]</td>
</tr>
<tr>
<td>Globals</td>
<td>Used different global variables (&quot;Globals&quot;) [94]</td>
</tr>
<tr>
<td>IOs</td>
<td>Number of direct address accesses (&quot;IOs&quot;) [94]</td>
</tr>
<tr>
<td>Locals</td>
<td>Number of local variables (&quot;locals&quot;) [94]</td>
</tr>
<tr>
<td>Inputs</td>
<td>Number input variables (&quot;inputs&quot;) [94]</td>
</tr>
<tr>
<td>Outputs</td>
<td>Number output variables (&quot;outputs&quot;) [94]</td>
</tr>
<tr>
<td>NOS</td>
<td>Number of statements (&quot;NOS&quot;) [94]</td>
</tr>
<tr>
<td>Comments</td>
<td>Percentage of comments (&quot;comments&quot;) [94]</td>
</tr>
<tr>
<td>McCabe</td>
<td>Complexity (McCabe) (&quot;McCabe&quot;) [94]</td>
</tr>
<tr>
<td>Prather</td>
<td>Complexity of nesting (Prather) (&quot;Prather&quot;) [94]</td>
</tr>
<tr>
<td>DIT</td>
<td>Depth of inheritance tree (&quot;DIT&quot;) [95]</td>
</tr>
<tr>
<td>NOC</td>
<td>Number of children (&quot;NOC&quot;) [95]</td>
</tr>
<tr>
<td>RFC</td>
<td>Response for class (&quot;RFC&quot;) [95]</td>
</tr>
<tr>
<td>CBO</td>
<td>Coupling between objects (&quot;CBO&quot;) [95]</td>
</tr>
<tr>
<td>Elshof</td>
<td>Complexity of reference (&quot;Elshof&quot;) [95]</td>
</tr>
<tr>
<td>LCOM</td>
<td>Lack of cohesion of methods (&quot;LCOM&quot;) [95]</td>
</tr>
<tr>
<td>n1 (Halstead)</td>
<td>Halstead – number of different used operators (n1) [95]</td>
</tr>
<tr>
<td>N1 (Halstead)</td>
<td>Halstead – number of operators (N1) [95]</td>
</tr>
<tr>
<td>n2 (Halstead)</td>
<td>Halstead – number of different used operands (n2) [95]</td>
</tr>
<tr>
<td>N2 (Halstead)</td>
<td>Halstead – number of operands (N2) [95]</td>
</tr>
<tr>
<td>HL (Halstead)</td>
<td>Halstead – length (HL) [95]</td>
</tr>
<tr>
<td>HV (Halstead)</td>
<td>Halstead – volume (HV) [95]</td>
</tr>
<tr>
<td>D (Halstead)</td>
<td>Halstead – difficulty (D) [95]</td>
</tr>
<tr>
<td>SFC branches</td>
<td>Number of SFC branches [96]</td>
</tr>
<tr>
<td>SFC steps</td>
<td>Number of SFC steps [96]</td>
</tr>
</tbody>
</table>

**Detailed description**

**Code size [number of bytes] ("code size")**

Code size as number of bytes.

**Variables size [number of bytes] ("variable size")**

Variables size as number of bytes.

**Stack size [number of bytes] ("stack size")**

Stack size as number of bytes.
**Number of calls ("calls")**
Number of function block calls within the application.

**Called in tasks ("tasks")**
Number of tasks calling the function block.

**Used different global variables ("Globals")**
Number of different global variables used in the function block.

**Number of direct address accesses ("IOs")**
Number of IO access operations in the function block = number of all read and write access operations to a direct address.

**Example:**
The number of direct address access operations for the MAIN program is 2.

```plaintext
PROGRAM MAIN
VAR
  OnOutput AT%QB1 : INT;
  nVar            : INT;
END_VAR

OnOutput := 123;
nVar     := OnOutput;
```

**Number of local variables ("local")**
Number of local variables in the function block (VAR).

**Number input variables ("inputs")**
Number of input variables in the function block (VAR_INPUT).

**Number output variables ("outputs")**
Number of output variables in the function block (VAR_OUTPUT).

**Number of statements ("NOS")**
NOS: Number Of executable Statements
NOS = number of executable statements in the function block

**Percentage of comments ("comments")**
Comment proportion = number of comments / number of statements in a function block
For the purpose of this definition, statements also include declaration statements, for example.

**Complexity (McCabe) ("McCabe")**
Complexity = number of binary branches in the control flow graph for the function block (e.g. the number of branches in IF and CASE statements and loops)

**Complexity of nesting (Prather) ("Prather")**
Nesting weight = statements * nesting depth
Complexity of nesting = nesting weight / number statements
Nesting through IF/ELSEIF or CASE/ELSE statements, for example.
Depth of inheritance tree ("DIT")

DIT: Depth of Inheritance Tree

DIT = inheritance depth or maximum path length from the root to the class under consideration

Number of children ("NOC")

NOC: Number Of Children

NOC = number of child classes or number of direct class specializations

Response for class ("RFC")

RFC: Response For Class

RFC = number of methods that can potentially be executed, if an object of the class under consideration responds to a received message

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.

Coupling between objects ("CBO")

CBO: Coupling Between Objects

CBO = number of classes coupled with the class under consideration

The value is used to indicate the coupling between object classes. Coupling refers to a situation where a class uses instance variables (variables of an instantiated class) and the methods of another class.

Complexity of reference (Elshof) ("Elshof")

Complexity of reference = referenced data (number of variables) / number of data references

Lack of cohesion of methods (LCOM) ("LCOM")

Cohesion = pairs of methods without common instance variables minus pairs of methods with common instance variables

This cohesion value is a measure for the encapsulation of a class. The higher the value, the poorer the encapsulation. Reciprocal method and property calls (without init or exit) are also taken into account.

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, case label, 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 used operators - Halstead (n1),**
  **Number of different used operands - Halstead (n2):**
  ◦ Number of different used operators (h₁) and operands (h₂); together they form the vocabulary size h.
• **Number of operators - Halstead (N1),**
  **Number of operands - Halstead (N2):**
  ◦ Number of total used operators (N₁) and operands (N₂); 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 = h₁ * log₂ h₁ + h₂ * log₂ h₂
  ◦ HV = N * log₂ h

Various indicators 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 = h₁/2 * N₂/h₂
• **Effort:**
  ◦ E = D * V

The indicators 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.

**Number of SFC branches**

If the function block is implemented in the Sequential Function Chart language (SFC), this code metric indicates the number of branches in the function block.

**Number of SFC steps**

If the function block is implemented in the Sequential Function Chart language (SFC), this code metric indicates the number of steps in the function block.

### 4.6 Forbidden symbols

In the *Forbidden symbols* tab, you can specify the keywords, symbols and identifiers that must not be used in the project code. The forbidden symbols are checked during the static analysis. [98]
Configuration of forbidden symbols

You can enter these symbols directly in the row or select them via the input assistant. During the static analysis the code is checked for the presence of these terms. Any hits result in an error being issued in the message window.

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 Execution

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, naming conventions and forbidden symbols to be taken into account in the static analysis can be configured 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).

See also: Syntax in the message window

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]' and the differences between the two commands described in the following table.
### Differences

| Command 'Run static analysis' | Command 'Run static analysis
[Check all objects]' |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scope</strong></td>
<td></td>
</tr>
<tr>
<td>The objects used in the PLC project are checked. Objects that are not used are not checked with this command. The scope of this command thus corresponds to the build commands <strong>Build Project/Solution</strong> or <strong>Rebuild Project/Solution</strong> respectively. If you also wish to have the unused objects checked by the static analysis, which is useful, for example, when processing library projects, you can use the command <strong>Run static analysis [check all objects]</strong>.</td>
<td>All objects in the project tree of the PLC project are checked. This is primarily useful when creating libraries or when processing library projects. The scope of this command thus corresponds to the build command <strong>Check all objects</strong>.</td>
</tr>
</tbody>
</table>

### 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 ([Settings] tab). If the option **Perform static analysis automatically** is enabled, TwinCAT performs the static analysis after each successful code generation (with the command **Build project**, for example). | The "Check all objects" variant cannot be executed implicitly. It can only be executed explicitly via the command. |

### 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 [14], naming conventions [79] and forbidden symbols [96] to be taken into account in the static analysis can be configured [12] 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 [14]).

See also: Syntax in the message window [99]

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' [98] and the differences between the two commands described in the following table.
### Differences

<table>
<thead>
<tr>
<th>Differences</th>
<th>Command 'Run static analysis'</th>
<th>Command 'Run static analysis [Check all objects]'</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scope</strong></td>
<td>The objects used in the PLC project are checked. Objects that are not used are not checked with this command. The scope of this command thus corresponds to the build commands <strong>Build Project/Solution</strong> or <strong>Rebuild Project/Solution</strong> respectively. If you also wish to have the unused objects checked by the static analysis, which is useful, for example, when processing library projects, you can use the command <strong>Run static analysis [check all objects]</strong>.</td>
<td>All objects in the project tree of the PLC project are checked. This is primarily useful when creating libraries or when processing library projects. The scope of this command thus corresponds to the build command <strong>Check all objects</strong>.</td>
</tr>
<tr>
<td><strong>Execution options for the command</strong></td>
<td>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 (Settings [12] tab). If the option <strong>Perform static analysis automatically</strong> is enabled, TwinCAT performs the static analysis after each successful code generation (with the command <strong>Build project</strong>, for example).</td>
<td>The &quot;Check all objects&quot; variant cannot be executed implicitly. It can only be executed explicitly via the command.</td>
</tr>
</tbody>
</table>

---

### 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** [91]).

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]' [103] and the differences between the two commands are described in the following table.
### Differences

<table>
<thead>
<tr>
<th>Command 'View Standard Metrics'</th>
<th>Command 'View Standard Metrics [Check all objects]'</th>
</tr>
</thead>
<tbody>
<tr>
<td>The standard metrics are created for the objects used in the PLC project. Objects that are not used are not considered with this command. The scope of this command thus corresponds to the build commands <strong>Build Project/Solution</strong> or <strong>Rebuild Project/Solution</strong> respectively. If you want to create default metrics for unused objects, which is useful when editing library projects, you can use the command 'View Standard Metrics [Check all objects]'</td>
<td>The standard metrics are created for all objects located in the project tree of the PLC project. This is primarily useful when creating libraries or when processing library projects. The scope of this command thus corresponds to the build command <strong>Check all objects</strong>.</td>
</tr>
</tbody>
</table>

#### 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.

---

**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 [91] 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:** 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). 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 [91]).

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' [101] and the differences between the two commands, which are described in the following table.

<table>
<thead>
<tr>
<th>Differences</th>
<th>Command 'View Standard Metrics'</th>
<th>Command 'View Standard Metrics [Check all objects]'</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>The standard metrics are created for the objects used in the PLC project. Objects that are not used are not considered with this command. The scope of this command thus corresponds to the build commands Build Project/Solution or Rebuild Project/Solution respectively. If you want to create default metrics for unused objects, which is useful when editing library projects, you can use the command 'View Standard Metrics [Check all objects]'</td>
<td>The standard metrics are created for all objects located in the project tree of the PLC project. This is primarily useful when creating libraries or when processing library projects. The scope of this command thus corresponds to the build command Check all objects.</td>
</tr>
</tbody>
</table>

### 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**

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 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.
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 [14]
- Naming conventions [79]

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 ...} [107]**
  - for disabling coding rules in the implementation part
  - can be used for individual code lines

- **Attribute {attribute 'no-analysis'} [107]**
  - 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' := '...' [108]**
  - for disabling coding rules in the declaration part
  - can be used for individual declarations or for whole programming objects

- **Attribute {attribute 'naming' := '...' [108]**
  - for disabling naming conventions in the declaration part
  - can be used for individual declarations or for whole programming objects

- **Attribute {attribute 'nameprefix' := '...' [109]**
  - 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'} [110]**
  - for specifying that a function block instance should only be called once
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
```
Pragmas and attributes

[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.

```plaintext
{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.

```plaintext
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.

```plaintext
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.

```plaintext
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' := '...'}

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 [ 79] 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 statistical 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.

```plaintext
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;      // => Invalid variable name 'st2'. Expect prefix 'st'
  pt2  : ST_Test;
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 [14] 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 Examples

7.1 Static analysis

During execution of the static analysis [98], compliance with the coding rules [14], naming conventions [79] and forbidden symbols [96] 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.

7.2 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 [91]). In addition, rule 150 (SA0150: Violation of lower or upper metrics limits) is enabled and configured as warning.

When the command 'View Standard Metrics' [101] 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.

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 [98] has been performed, the violation of the two upper limits is therefore reported as a warning in the message window.
## Error List

<table>
<thead>
<tr>
<th>Description</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>13   SA0150: Metric violation for 'MAIN'. Result for metric 'Code size' (688) &gt; 650</td>
<td>MAIN.TcPOU</td>
</tr>
<tr>
<td>14   SA0150: Metric violation for 'SampleProgram'. Result for metric 'Inputs' (? ) &gt; 5</td>
<td>SampleProgram.TcPOU</td>
</tr>
</tbody>
</table>
8 Automation Interface support

The static analysis can partly be operated via the Automation Interface (AI).

Please also refer to the Automation Interface documentation:
TE1000 XAE > Technologies > Automation Interface > Overview > Product description

Explicit execution of static analysis via the AI

This means that the following two commands can be called explicitly via the Automation Interface.

- Command 'Run static analysis' [98]
- Command 'Run static analysis [Check all objects]' [100]

PowerShell sample:
```powershell
$p = $sysMan.LookupTreeItem("TIPC^MyPlcProject^MyPlcProject Project")
$p.RunStaticAnalysis()
```

For the RunStaticAnalysis() method, bCheckAll can be specified as an optional parameter, although the method can also be called without parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>RunStaticAnalysis()</td>
<td>Execution of the command Run static analysis</td>
</tr>
<tr>
<td>RunStaticAnalysis(bCheckAll = FALSE)</td>
<td>Execution of the command Run static analysis [Check all objects]</td>
</tr>
<tr>
<td>RunStaticAnalysis(bCheckAll = TRUE)</td>
<td>Execution of the command Run static analysis [Check all objects]</td>
</tr>
</tbody>
</table>

Implicit execution of static analysis via the AI

Alternatively, the setting [12] 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.