

SIEMENS

Ingenuity for life



Programming style guide for JavaScript in SIMATIC WinCC Unified

SIMATIC WinCC Unified V16
SIMATIC HMI Unified Comfort Panels

<https://support.industry.siemens.com/cs/ww/en/view/109758536>

Siemens
Industry
Online
Support



Legal information

Use of application examples

Application examples illustrate the solution of automation tasks through an interaction of several components in the form of text, graphics and/or software modules. The application examples are a free service by Siemens AG and/or a subsidiary of Siemens AG ("Siemens"). They are non-binding and make no claim to completeness or functionality regarding configuration and equipment. The application examples merely offer help with typical tasks; they do not constitute customer-specific solutions. You yourself are responsible for the proper and safe operation of the products in accordance with applicable regulations and must also check the function of the respective application example and customize it for your system.

Siemens grants you the non-exclusive, non-sublicensable and non-transferable right to have the application examples used by technically trained personnel. Any change to the application examples is your responsibility. Sharing the application examples with third parties or copying the application examples or excerpts thereof is permitted only in combination with your own products. The application examples are not required to undergo the customary tests and quality inspections of a chargeable product; they may have functional and performance defects as well as errors. It is your responsibility to use them in such a manner that any malfunctions that may occur do not result in property damage or injury to persons.

Disclaimer of liability

Siemens shall not assume any liability, for any legal reason whatsoever, including, without limitation, liability for the usability, availability, completeness and freedom from defects of the application examples as well as for related information, configuration and performance data and any damage caused thereby. This shall not apply in cases of mandatory liability, for example under the German Product Liability Act, or in cases of intent, gross negligence, or culpable loss of life, bodily injury or damage to health, non-compliance with a guarantee, fraudulent non-disclosure of a defect, or culpable breach of material contractual obligations. Claims for damages arising from a breach of material contractual obligations shall however be limited to the foreseeable damage typical of the type of agreement, unless liability arises from intent or gross negligence or is based on loss of life, bodily injury or damage to health. The foregoing provisions do not imply any change in the burden of proof to your detriment. You shall indemnify Siemens against existing or future claims of third parties in this connection except where Siemens is mandatorily liable.

By using the application examples you acknowledge that Siemens cannot be held liable for any damage beyond the liability provisions described.

Other information

Siemens reserves the right to make changes to the application examples at any time without notice. In case of discrepancies between the suggestions in the application examples and other Siemens publications such as catalogs, the content of the other documentation shall have precedence.

The Siemens terms of use (<https://support.industry.siemens.com>) shall also apply.

Security information

Siemens provides products and solutions with Industrial Security functions that support the secure operation of plants, systems, machines and networks.

In order to protect plants, systems, machines and networks against cyber threats, it is necessary to implement – and continuously maintain – a holistic, state-of-the-art industrial security concept. Siemens' products and solutions constitute one element of such a concept.

Customers are responsible for preventing unauthorized access to their plants, systems, machines and networks. Such systems, machines and components should only be connected to an enterprise network or the Internet if and to the extent such a connection is necessary and only when appropriate security measures (e.g. firewalls and/or network segmentation) are in place. For additional information on industrial security measures that may be implemented, please visit <https://www.siemens.com/industrialsecurity>.

Siemens' products and solutions undergo continuous development to make them more secure. Siemens strongly recommends that product updates are applied as soon as they are available and that the latest product versions are used. Use of product versions that are no longer supported, and failure to apply the latest updates may increase customer's exposure to cyber threats.

To stay informed about product updates, subscribe to the Siemens Industrial Security RSS Feed at: <https://www.siemens.com/industrialsecurity>.

6.2	Declaring and using variables	19
	ST008 rule: Declare variables (block validity ranges)	19
	ST009 rule: Declare variables at the start of a code block	19
	ST010 rule: Use "var" declared script variables in defined code block	19
	ST011 rule: One line per variable declaration	20
	ST012 recommendation: Avoid unnecessary variable declarations ..	20
	ST013 recommendation: Avoid "var" declaration in sub- functions/instructions	20
6.3	Comparing variables	21
	ST014 rule: Do not compare a variable with itself	21
	ST015 rule: Checking a number for whether it is not a number	21
	ST016 rule: Specify output format (radix argument) with parseInt() function	21
6.4	Entering numbers	22
	ST017 rule: Fully specify floating point numbers	22
	ST018 rule: Do not begin numbers with "0"	22
6.5	Arrays and structures	23
	ST019 rule: Commas between structure and array elements	23
	ST020 recommendation: Avoid empty array elements	23
7	Functions and objects	24
7.1	Functions	24
	FO001 rule: Do not redefine functions	24
	FO002 rule: Define return value	24
	FO003 rule: Do not overwrite call parameters	24
	FO004 rule: Function call with declaration in brackets	24
	FO005 rule: Do not write instructions to a string	25
	FO006 rule: Create function without "new"	25
	FO007 rule: Do not use global object properties as function	25
	FO008 recommendation: Avoid single-line code blocks	25
	FO009 recommendation: Avoid "eval()" function	25
	FO010 recommendation: Avoid unnecessary function binding	25
7.2	Objects	26
	FO011 rule: Save new objects in variables	26
	FO012 rule: Begin object names with capital letters	26
	FO013 rule: No duplicate property declarations for objects	26
	FO014 rule: Do not modify native objects	26
	FO015 recommendation: Avoid object constructor when creating object	27
8	Conditional instructions, branches and loops	28
	IL001 rule: "if": With "return", "else" is superfluous	28
	IL002 rule: Do not use constants alone in conditions	28
	IL003 rule: "for...in": First check properties before working with them	28
	IL004 rule: Define "default" in "for"/"switch" instructions	28
	IL005 rule: No double "case" instructions	29
	IL006 rule: Every "case" has a "break"	29
	IL007 recommendation: Avoid nested ternary operations	29
9	Expressions to avoid	30
	EA001 rule: Avoid "with" expression	30
	EA002 rule: Avoid "arguments.caller" and "arguments.callee" expression	30
	EA003 rule: Avoid "_proto_" property	30

Table of contents

10	Diagnostics and debugging	31
	DD001 rule: Trace() function for diagnostics.....	31
11	Appendix	32
11.1	Service and support	32
11.2	Links and literature	33
11.3	Change documentation	33

1 Introduction

When writing JavaScript code, the task of the programmer is to make the user program as clear and readable as possible.

Each programmer uses his/her own strategy, for example in naming variables or functions, or in writing comments.

The various approaches of programmers result in wildly different user programs, which frequently can only be interpreted by their own creators.

Note

The programming style guide for SIMATIC S7-1200/S7-1500 serves as a basis for this document.

<https://support.industry.siemens.com/cs/ww/en/view/109478084>

1.1 Objectives

The rules and recommendations described here will help you create more consistent code which can be better maintained and reused.

If multiple programmers are working on the same program, it is therefore recommended to define terminology for the whole project, and to adhere to a mutually agreed-upon programming style. In this way, errors can be recognized and/or prevented as early as possible.

For the maintainability and clarity of the source code, it is first of all necessary to maintain a certain surface form. Visual effects do not significantly improve the quality of the software. It is much more important, for example, to define rules that support the developer in the following way:

- Avoid typos and careless mistakes which are then misinterpreted by the compiler.
Objective: The compiler should detect as many errors as possible.
- Support the programmer when diagnosing program errors, e.g. inadvertent reuse of a temporary variable beyond one cycle.
Objective: The identifier quickly indicates problems.
- Standardization of default applications and default libraries
Objective: It should be easy to learn, and the reusability of program code should be increased.
- Simple maintenance; simplification of development
Objective: Changes to program code at individual points of use, scripts on screen objects and in global modules should have minimal impact on the overall configuration. It should be possible for different programmers to make changes to program code in individual modules.

Note

Ensure that the rules and recommendations described in this document are self-consistent and build on each other.

1.2 Advantages of a uniform programming style

- Integrated, uniform style
- Easy to read and understand
- Simple maintenance and reusability
- Simple and fast error identification and correction
- Efficient collaboration between multiple programmers

1.3 Applies to

This document applies to projects and libraries in TIA Portal in connection with the JavaScript scripting environment as per the ECMAScript Language Specification.

Google V8 is used as a script engine, which largely implements ECMAScript 2015, 6th Edition from June 2015 (ECMAScript 6).

ESLint rules

This document uses ESLint rules as a basis. The rules listed here sometimes differ slightly from the ESLint rules, as the primary goal is to achieve consistency in the TIA Portal configuration (interplay with programming style guide for SIMATIC S7-1200/S7-1500).

1.4 Delimitation

This document does not provide a description of:

- HMI configuration in TIA Portal
- Commissioning HMI projects

Sufficient experience in these topics is required in order to sensibly interpret and apply the rules and recommendations. This document is not a replacement for expertise in software development, but rather serves as a reference.

1.5 Deviations from the rules, other specifications

For customer projects, the required standards as well as customer- or industry-specific standards on the customer's side (or the technology used) must be observed and take priority over this style guide or parts thereof.

When combining customer specifications with this style guide, attention must be paid to the integrity of said combination and of the project as a whole. Deviating from the rules must be justified at the corresponding point in the user program, and must be documented. The rules defined by the customer must be documented in a suitable form.

2 Definition of terms

2.1 Rules/recommendations

The stipulations in this document are divided into recommendations and rules:

- **Rules** are obligatory and must be observed. They are imperative for reusable, well-performing code. Rules may also be broken in exceptions. But this must be documented appropriately.
- **Recommendations** are specifications that firstly serve for the uniformity of the code, and secondly which are intended as support and information. Recommendations should be observed in principle, but there can certainly be cases in which a recommendation is not followed. Reasons for doing this may be higher efficiency or better readability.

2.2 Numbering of rules

For a unique assignment of rules between various categories, the rules and recommendations are labeled with a prefix according to their category (two characters), and then numbered (3 numbers).

If a rule is missing, the number is not reassigned. If you define additional rules, you can use numbering between 901 and 999.

Table 2-1

Prefix	Category
ES	Engineering system (Programming environment)
GR	General rules
SR	Stylistic rules
ST	Script variables
FO	Functions and objects
IL	Instruction and loops
EA	Expressions to avoid
DD	Diagnostics and debugging

2.3 Terms (block, function and parameter)

Some of the rules in this programming style guide make reference to functions, function parameters, and blocks.

The following descriptions provide a brief explanation of these terms for clarification. This clarification is necessary for a shared understanding of the terms in use moving forward.

Function and parameter

When you create a function in JavaScript or use that function, then you can also pass values to the function. You can do this with the help of "parameters". "Parameters" are often also called "arguments", but the meaning is the identical.

```
const tagValue1, tagValue2;

function doSomething(parameter1, parameter2) {
  // ...function code
}

// ...code
doSomething(tagValue1, tagValue2);
```

Block

A block, or an instruction block, is used to group one or more instructions. A block is identified by the curly brackets that contain it.

The following example shows a block using the example of an If instruction.

```
if (a > 2) {
  tagValue = a * 6;
  HMIRuntime.Trace(a);
}
```

3 Settings in TIA Portal

This chapter describes the rules and recommendations for basic settings in the TIA Portal programming environment.

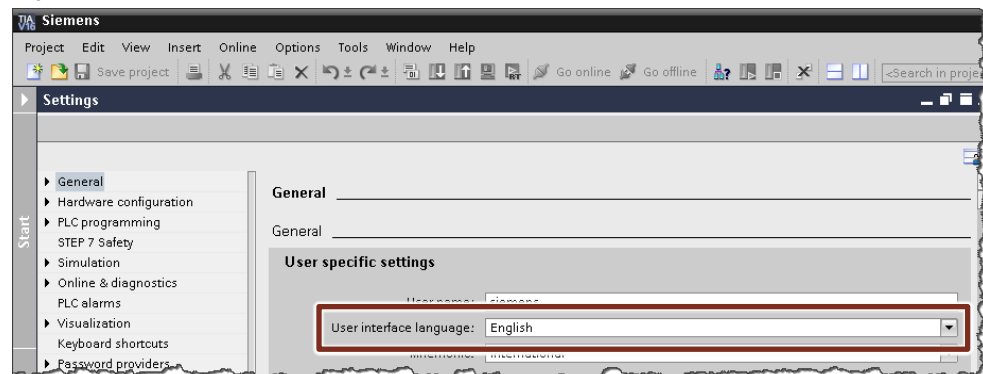
Note The following settings have no direct impact on the JavaScript script editor. However, they are recommended for the sake of consistency with other style guides.

ES001 rule: User interface language "English"

The user interface language in TIA Portal is set to "English". This means that all newly created projects will be automatically created in the editing and reference language, and the system constants will be created in "English".

Reason: The user interface language must be set consistently so that the system constants in all projects are in the same language.

Figure 3-1

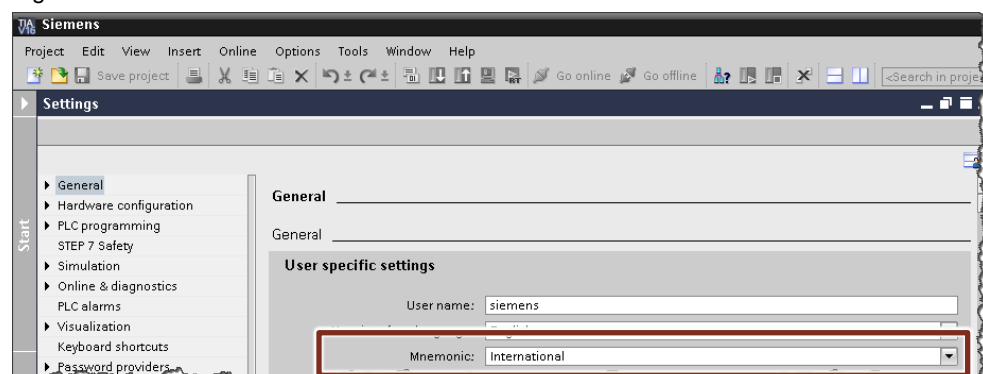


ES002 rule: "International" mnemonics

The mnemonics (language setting for programming language) is set to "International".

Reason: All system languages and system parameters are thus clear and language-independent. This ensures smooth collaboration in teams.

Figure 3-2



4 General rules

GR001 rule: Curly brackets for multi-line instructions

If you have instructions (e.g. if, while, function) that extend over multiple lines, always use curly brackets.

Additional information: <http://eslint.org/docs/rules/curly>

GR002 rule: Use valid regular expressions

Only use valid regular expressions in your JavaScript code. You can use the following website for this: <https://regexr.com/>

Additional information: <http://eslint.org/docs/rules/no-invalid-regexp>

GR003 rule: Use more typical equal-to operators

When comparing values, always use the more typical equality operators (e.g. "===", "!=") rather than the counterparts "==" and "!=".

Additional information: <http://eslint.org/docs/rules/eqeqeq>

You should use the more typical equality operators especially if you wish to compare an instruction with "null".

Additional information: <http://eslint.org/docs/rules/no-eq-null>

GR004 recommendation: Avoid assignment operators with multiple meanings in conditional instructions

In conditional instructions (e.g. if, for, while instruction), avoid assignment operators with multiple meanings (=). Instead, use more typical equality operators.

Additional information: <http://eslint.org/docs/rules/no-cond-assign>

```
// Allowed:
if (x === 0) {
  const b = 1;
}

// Disallowed:
if (x = 0) {
  const b = 1;
}
```

GR005 recommendation: Avoid empty instructions

Avoid instructions and functions with no content, as they can cause confusion while reading.

Additional information: <http://eslint.org/docs/rules/no-empty>

GR006 recommendation: Use single quotes

JavaScript allows you to define strings in one of three ways:

- Double quotes ("),

- Single quotes (')
- Backticks (`)

In WinCC Unified, it is recommended by default to use single quotes for strings. If you need additional quote marks in a string, use double quotes.

Additional information: <http://eslint.org/docs/rules/quotes>

```
// Allowed:
const value = 'myValue';
const string = 'a string containing "double" quotes';

// Disallowed:
const value = "myValue";
const string = "a string containing "double" quotes";
```

Note

You can also use backticks (`) as so-called template strings. In this way, you can connect text and variable with each other in one expression. In this case, the variables are specified with Dollar signs and curly brackets "\${Variable}".

```
// Allowed:
const value = 'myValue';
const string = `The tag value is ${value}`;
```

GR007 recommendation: Avoid commas with an expected expression

Avoid commas if only one expression is expected.

The comma operator encompasses multiple expressions, while only one is expected. It evaluates every operand from left to right and returns the value of the last operator. However, this often hides unwanted side-effects, and its use is often by mistake.

Additional information: <http://eslint.org/docs/rules/no-sequences>

5 Stylistic rules

SR001 rule: Two spaces for code structuring

In order to better structure your JavaScript code, it is recommended to use two spaces to indent content in WinCC Unified. This helps the traceability and readability of the code.

Additional information: <http://eslint.org/docs/rules/indent>

```
// Allowed:
if (x === 0) {
  ··const b = 1;
}

// Disallowed:
if (x === 0) {
const b = 1;
}
```

SR002 rule: Maximum of two empty lines in the code

In order to structure your code, you can use empty lines, e.g. to logically and visually set off if instructions. It also improves the readability of the code.

It is recommended to use a maximum of two empty spaces between blocks of code.

Additional information: <http://eslint.org/docs/rules/no-multiple-empty-lines>

SR003 rule: Spaces after every sentence punctuation mark

The following punctuation marks are important elements when creating code in JavaScript:

- Comma (",")
- Colon (":")
- Semicolon (";")

If you use one of these punctuation marks in your code, no spaces should appear before it. A sentence punctuation mark should however have a space after it.

Additional information:

- Comma (","): <http://eslint.org/docs/rules/comma-spacing>
- Colon (":"): <http://eslint.org/docs/rules/key-spacing>
- Semicolon (";"): <http://eslint.org/docs/rules/semi-spacing>

SR004 rule: End code line with semicolon (";")

Every line of code in JavaScript must end with a semicolon, otherwise it will cause an error.

Additional information: <http://eslint.org/docs/rules/semi>

However, try not to use too many semicolons, as this can cause confusion.

Additional information: <http://eslint.org/docs/rules/no-extra-semi>

SR005 rule: Surround key words with spaces

If you use key words (e.g. "if", "else", "switch") in your JavaScript, they should be surrounded by spaces (i.e. there should be a space before and after the key word).

Additional information: <http://eslint.org/docs/rules/keyword-spacing>

```
// Allowed:
if (true) {
  //...
} else {
  //...
}

// Disallowed:
if (true) {
  //...
}else{
  //...
}
```

SR006 recommendation: Space before every instruction block

If you start an instruction block, for instance after an if instruction, you should insert a space.

Additional information: <http://eslint.org/docs/rules/space-before-blocks>

```
// Allowed:
if (true) {
  //...
}

// Disallowed:
if (true){
  //...
}
```

SR007 recommendation: Space between operands

If you have a mathematical calculation in your code, then every operand and every operator should be followed by a space.

Additional information: <http://eslint.org/docs/rules/space-infix-ops>

```
// Allowed:
a = b + c;

// Disallowed:
a=b+c;
```


SR008 recommendation: Avoid comma at line start

It is recommended to use commas (",") at the end of a line rather than at the line start.

Additional information: <http://eslint.org/docs/rules/comma-style>

```
// Allowed:
const foo = 1, bar = 2;
const value1 = 1,
      value2 = 2;

// Disallowed:
const foo = 1
, bar = 2;
const value1 = 1
, value2 = 2;
```

SR009 recommendation: No spaces at line end

When you are writing and commenting your code, you should make sure not to end the line with a space.

This is not an error in general, but can lead to misinterpretation.

Additional information: <http://eslint.org/docs/rules/no-trailing-spaces>

```
// Allowed:
const foo = 1, bar = 2; //comment

// Disallowed:
const foo = 1, bar = 2; //comment...
//...
```

SR010 recommendation: No empty lines at block starts and classes

Avoid an empty line at the start of a block or class.

Additional information: <http://eslint.org/docs/rules/padded-blocks>

SR011 recommendation: Avoid space between function and identifier

Ensure that you do not place a space between the function and the identifier.

Example

```
// Allowed:
myFunction();

// Disallowed:
myFunction ();
```

Additional information: <http://eslint.org/docs/rules/no-spaced-func>

SR012 recommendation: Space before function brackets

Always leave a space between the function and the curly function brackets.

Additional information: <http://eslint.org/docs/rules/space-before-function-paren>

```
// Allowed:
myFunction(a) {};

// Disallowed:
myFunction(a){};
```

SR013 recommendation: Begin comments with space

When you comment a line of code, then insert a space before you begin the comment text.

This rule does not apply to structuring lines ("/******") for visually separating code lines from one another.

Additional information: <http://eslint.org/docs/rules/spaced-comment>

```
// Allowed:
// This is an example comment
/* This is an example comment */
/******

// Disallowed:
//This is an example comment
/*This is an example comment*/
```

SR014 recommendation: Use normal tabs and spaces

Only use the Tab or Space key to insert spaces.

Additional information: <http://eslint.org/docs/rules/no-irregular-whitespace>

SR015 recommendation: Limit strings to one line

Do not enter strings in your code that are longer than a line. If you nevertheless need multi-line strings, then you can implement this with a line break "\n".

Additional information: <http://eslint.org/docs/rules/no-multi-str>

```
// Allowed:
const x = "Line 1\n" +
         "Line 2";

// Disallowed:
const x = "Line 1 \
         Line 2";
```

6 Script variables, arrays and structures

6.1 General rules for script variables

ST001 rule: Use "camelCase" notation

Use the camelCase notation for writing variables.

Additional information: <http://eslint.org/docs/rules/camelcase>

```
// Allowed:
let machineState;
let standardValue;

// Disallowed:
let machine_State;
let StandardValue;
```

ST002 rule: Unique variable naming

When you define variables, ensure that they have unique names and are only used once.

Additional information: <http://eslint.org/docs/rules/no-redeclare>

Within a function, variables should likewise be unique and not overlap with global variables.

Additional information: <http://eslint.org/docs/rules/no-shadow>

ST003 rule: Use dot notation

Use dot notation to access properties of objects and elements.

Additional information: <http://eslint.org/docs/rules/dot-notation>

```
Screen.Items('Rectangle1').BackColor = 0xFF00FF00;
```

ST004 rule: Do not use key words for functions or variables

Do not use native key words or object names (e.g. "length", "top", "undefined", "NaN", "Infinity", "eval" or "arguments") as the name of a function or a variable. This can cause confusion and hinders traceability of the code.

Additional information: <http://eslint.org/docs/rules/no-native-reassign>

<http://eslint.org/docs/rules/no-shadow-restricted-names>

ST005 rule: Do not create any new primitive wrappers

In JavaScript, there are three primitive data types which have wrapper objects:

- String
- Numbers
- Boolean.

Avoid creating your own manual wrapper instances, even if JavaScript allows it.

Additional information: <http://eslint.org/docs/rules/no-new-wrappers>

ST006 recommendation: Prohibit octal escape sequence in string literals

Since JavaScript version ECMAScript 5 specification, octal escape sequences in string literals are obsolete and should no longer be used. Instead, Unicode escape sequences should be used.

Additional information: <http://eslint.org/docs/rules/no-octal-escape>

ST007 recommendation: Avoid unnecessary Boolean operations

With comparisons or assignments, avoid unnecessary Boolean operations, such as double negation.

Additional information: <http://eslint.org/docs/rules/no-extra-boolean-cast>

6.2 Declaring and using variables

ST008 rule: Declare variables (block validity ranges)

In order to use variables in your script, you first have to declare them.

A basic distinction is made between the types of declaration:

Table 6-1

Declaration	Explanation
var	Declares a variable, has no block validity range.
let	Declares a variable with validity in the current block.
const	Declares a constant with validity in the current block. An error will occur if an attempt is made to overwrite the variable.

Note

The "var" declaration is outdated and should be avoided for new JavaScript scripts.

A rule of thumb for declaring variables:

- By default, use the variable declaration "const".
- If the value or object of the variable changes during the script, the declaration type "const" will be replaced by "let".

```
var a = 5;
let b = 50;
const c = 500;

if (a > 2) {
  a = 6;
  let b = 60;           //only in the block visible
  HMIRuntime.Trace(a); //Trace output: 6
  HMIRuntime.Trace(b); //Trace output: 60
  HMIRuntime.Trace(c); //Trace output: 500
}

HMIRuntime.Trace(a); //Trace output: 6
HMIRuntime.Trace(b); //Trace output: 50
HMIRuntime.Trace(c); //Trace output: 500
```

ST009 rule: Declare variables at the start of a code block

When writing code, make sure to declare the variables at the beginning of a block of code.

Additional information: <http://eslint.org/docs/rules/vars-on-top>

ST010 rule: Use "var" declared script variables in defined code block

Please avoid the "var" declaration type. If you nevertheless need "var", then always use the script variable that was declared with "var" inside the code block where you defined it, otherwise error messages may occur.

Additional information: <http://eslint.org/docs/rules/block-scoped-var>

```
// Allowed:
function doSomething() {
  var result;

  if (x > 10) {
    result = true;
  } else {
    result = false;
  }
}

// Disallowed:
function doSomething() {
  if (x > 10) {
    var result = true;
  } else {
    var result = false;
  }
}
```

ST011 rule: One line per variable declaration

Use a separate line for every variable declaration.

Additional information: <http://eslint.org/docs/rules/one-var>

ST012 recommendation: Avoid unnecessary variable declarations

Only define as many variables as you need in your code. Too many defined variables easily give an impression of overcomplexity.

Additional information: <http://eslint.org/docs/rules/no-unused-vars>

ST013 recommendation: Avoid "var" declaration in sub-functions/instructions

Please avoid the "var" declaration type. If you nevertheless need "var", the always define variables and functions as "var" in the main part of a function and avoid defining them in nested sub-functions or instructions.

Additional information: <http://eslint.org/docs/rules/no-inner-declarations>

6.3 Comparing variables

ST014 rule: Do not compare a variable with itself

Never compare a variable with itself, as this will cause an error. An exception is checking for the function "isNaN()".

Additional information: <http://eslint.org/docs/rules/no-self-compare>

ST015 rule: Checking a number for whether it is not a number

If you are checking the value of a variable for whether it is not a number, then use the function "isNaN()" in place of the expression "NaN" (abbreviation for "Not a Number").

Additional information: <http://eslint.org/docs/rules/use-isnan>

```
// Allowed:
if (isNaN(x))

// Disallowed:
if (x == NaN)
```

ST016 rule: Specify output format (radix argument) with parseInt() function

Besides the expression to be converted, the parseInt() function also has a second argument (the radix) for the output format of the expression to be converted. By default, the parseInt() function automatically detects whether this is an integer or a hexadecimal number (using the 0x prefix).

However, in order to specify the arguments clearly, it is recommended to include the output format in the function.

Additional information: <http://eslint.org/docs/rules/radix>

6.4 Entering numbers

ST017 rule: Fully specify floating point numbers

If you enter floating point numbers, always use the established number formats in order to obtain a clear representation as a decimal number. This prevents misinterpretations when reading the code.

Additional information: <http://eslint.org/docs/rules/no-floating-decimal>

```
// Allowed:  
const x = -0.2;  
const y = 5.0;  
  
// Disallowed:  
const x = -.2;  
const y = 5.;
```

ST018 rule: Do not begin numbers with "0"

If you use numbers in your script code, then make sure to enter them without leading zeroes, otherwise they could be interpreted as octal numbers.

Additional information: <http://eslint.org/docs/rules/no-octal>

6.5 Arrays and structures

ST019 rule: Commas between structure and array elements

The enumeration of elements in an array is separated by commas. There must not be a comma after the final element.

Additional information: <http://eslint.org/docs/rules/comma-dangle>

ST020 recommendation: Avoid empty array elements

When creating an array, make sure that you do not insert any empty array elements.

Additional information: <http://eslint.org/docs/rules/no-sparse-arrays>

```
// Allowed:
const colors = ['red', 'green', 'blue'];

// Disallowed:
const colors = ['red', , 'blue'];
```

7 Functions and objects

7.1 Functions

FO001 rule: Do not redefine functions

If you have declared functions, make sure not to redefine them in another place.

Additional information: <http://eslint.org/docs/rules/no-func-assign>

```
// Disallowed:
function multiply(x, y) {
  return x * y;
}
multiply = 22;
```

If, however, you assign an anonymous function to a variable, then you can overwrite the variable in another place.

```
// Allowed:
let multiply = function (x, y) {
  return x * y;
}
multiply = 22;
```

FO002 rule: Define return value

In a function, define whether that function has a return value or not. If the function has a return value, then define what it is.

Additional information: <http://eslint.org/docs/rules/consistent-return>

FO003 rule: Do not overwrite call parameters

Make sure not to overwrite the call parameters of a function or within a function.

Additional information: <http://eslint.org/docs/rules/no-ex-assign>

Additional information: <http://eslint.org/docs/rules/no-param-reassign>

FO004 rule: Function call with declaration in brackets

If a declared function is to be invoked immediately, you must wrap it with brackets.

Additional information: <http://eslint.org/docs/rules/wrap-iife>

FO005 rule: Do not write instructions to a string

Avoid assigning JavaScript instructions to a string, as this will cause an error.

Additional information: <http://eslint.org/docs/rules/no-implied-eval>

FO006 rule: Create function without "new"

When creating a function, ensure that you create it without "new".

Additional information: <http://eslint.org/docs/rules/no-new-func>

FO007 rule: Do not use global object properties as function

Do not use global object properties as a function (Math, JSON,...).

Additional information: <http://eslint.org/docs/rules/no-obj-calls>

FO008 recommendation: Avoid single-line code blocks

Avoid one-line code blocks in your script, as they are easy to miss and difficult to trace.

Additional information: <http://eslint.org/docs/rules/no-lone-blocks>

```
// Allowed:
function doSomething () {
  test();
}

// Disallowed:
function doSomething () {
  {
    test();
  }
}
```

FO009 recommendation: Avoid "eval()" function

The "eval()" function evaluates JavaScript code and represents it as a string. This function is potentially dangerous and often misused. Avoid this function and use alternatives as described in the link.

Additional information: <http://eslint.org/docs/rules/no-eval>

FO010 recommendation: Avoid unnecessary function binding

Avoid unnecessary use of the "bind()" function.

Additional information: <http://eslint.org/docs/rules/no-extra-bind>

7.2 Objects

FO011 rule: Save new objects in variables

When you create a new object, then it is recommended to save it in a script variable.

Additional information: <http://eslint.org/docs/rules/no-new>

```
// Allowed:
const myObject = new Object();
myObject();

// Disallowed:
new object();
```

FO012 rule: Begin object names with capital letters

The object name after the "new" operator starts with an uppercase letter.

Additional information: <http://eslint.org/docs/rules/new-cap>

```
// Allowed:
const myObject = new Object();

// Disallowed:
const myObject = new object();
```

FO013 rule: No duplicate property declarations for objects

Do not assign values to a variable multiple times, as this can cause unexpected behavior.

Additional information: <http://eslint.org/docs/rules/no-dupe-keys>

```
// Allowed:
const testObject = {
  value1 = 'red',
  value2 = 'green'
};

// Disallowed:
const testObject = {
  value = 'red',
  value = 'green'
};
```

FO014 rule: Do not modify native objects

JavaScript allows objects to be extended, including native objects. This can cause the behavior of the objects to change, which in turn can lead to irritation if you use the "native" object in another code block.

Therefore, avoid modifying or extending "native" objects.

Additional information: <http://eslint.org/docs/rules/no-extend-native>

F0015 recommendation: Avoid object constructor when creating object

When you create a new object, you should make sure to prefer the object-literal syntax, meaning that the object constructor should be avoided.

Additional information: <http://eslint.org/docs/rules/no-new-object>

```
// Allowed:  
const a = {};  
  
// Disallowed:  
const a = new Object();
```

8 Conditional instructions, branches and loops

IL001 rule: "if": With "return", "else" is superfluous

If an "if" instruction contains a return instruction, the "else" block is superfluous. The content can thus be placed outside of the block.

Additional information: <http://eslint.org/docs/rules/no-else-return>

IL002 rule: Do not use constants alone in conditions

If you use an instruction with a condition, make sure that you always use a comparison, e.g. with a constant, with the condition.

If you specify a constant without a comparison in a condition, then the instruction will either be executed always or never.

This mainly affects the following instructions:

- "if" instruction
- "for" instruction
- "while" instruction
- "do...while" instruction

Additional information: <http://eslint.org/docs/rules/no-constant-condition>

```
// Allowed:
if (value === 0) {
  myFunction();
}

// Disallowed:
if (false) {
  myFunction();
}
```

IL003 rule: "for...in": First check properties before working with them

First check in a "for...in" loop whether a property exists before you continue working with that property.

Additional information: <http://eslint.org/docs/rules/guard-for-in>

IL004 rule: Define "default" in "for"/"switch" instructions

To the extent possible, always define a "default" state in a "for" or "switch" instruction.

If the application does not allow for it, then add an appropriate comment to the code, "// No Default".

Additional information: <http://eslint.org/docs/rules/default-case>

IL005 rule: No double "case" instructions

Make sure that you haven't duplicated a "case" condition in a "switch...case" instruction (e.g. due to a copy & paste error).

Additional information: <http://eslint.org/docs/rules/no-duplicate-case>

```
// Allowed:
switch (value) {
  case 1:
    break;
  case 2:
    break;
  case 3:
    break;
  default:
    break;
}

// Disallowed:
switch (value) {
  case 1:
    break;
  case 2:
    break;
  case 1:      // duplicate test expression
    break;
  default:
    break;
}
```

IL006 rule: Every "case" has a "break"

In a "switch" instruction, every case or "case" instruction should be given "break". This prevents a so-called "fall through" through all "case" instructions. If a fall through is desired, add the comment "//fall through".

Additional information: <http://eslint.org/docs/rules/no-fallthrough>

IL007 recommendation: Avoid nested ternary operations

Avoid nested ternary operations within a line. Employ the "if" instruction as an alternative.

Additional information: <http://eslint.org/docs/rules/no-nested-ternary>

```
// Allowed:
const a = b ? 1 : c;

// Disallowed:
const a = b ? 1 : c ? 2 : 3;
```

9 Expressions to avoid

EA001 rule: Avoid "with" expression

Avoid the instruction "with()" because you are adding members of an object to the current range and this makes it impossible to recognize what a variable is actually referring to within the block.

Additional information: <http://eslint.org/docs/rules/no-with>

EA002 rule: Avoid "arguments.caller" and "arguments.callee" expression

Avoid obsolete and sub-optimal code such as "arguments.caller" and "arguments.callee".

Additional information: <http://eslint.org/docs/rules/no-caller>

EA003 rule: Avoid "_proto_" property

The "_proto_" property has been obsolete since ECMAScript 3.1 and should not be used in code anymore. Instead, use "Object.getPrototypeOf" and "Object.setPrototypeOf"

Additional information: <http://eslint.org/docs/rules/no-proto>

10 Diagnostics and debugging

DD001 rule: Trace() function for diagnostics

In order to debug your code and remedy any errors, use exclusively the SIMATIC WinCC Unified Trace() function.

When finished, make sure to remove all unnecessary Trace() messages from your code, otherwise the code will quickly become confusing.

Additional information: <http://eslint.org/docs/rules/no-unreachable>
<http://eslint.org/docs/rules/no-console>
<http://eslint.org/docs/rules/no-debugger>

11 Appendix

11.1 Service and support

Industry Online Support

Do you have any questions or need assistance?

Siemens Industry Online Support offers round the clock access to our entire service and support know-how and portfolio.

The Industry Online Support is the central address for information about our products, solutions and services.

Product information, manuals, downloads, FAQs, application examples and videos – all information is accessible with just a few mouse clicks:

support.industry.siemens.com

Technical Support

The Technical Support of Siemens Industry provides you fast and competent support regarding all technical queries with numerous tailor-made offers – ranging from basic support to individual support contracts.

Please send queries to Technical Support via Web form:

support.industry.siemens.com/cs/my/src

SITRAIN – Digital Industry Academy

We support you with our globally available training courses for industry with practical experience, innovative learning methods and a concept that's tailored to the customer's specific needs.

For more information on our offered trainings and courses, as well as their locations and dates, refer to our web page:

siemens.com/sitrain

Service offer

Our range of services includes the following:

- Plant data services
- Spare parts services
- Repair services
- On-site and maintenance services
- Retrofitting and modernization services
- Service programs and contracts

You can find detailed information on our range of services in the service catalog web page:

support.industry.siemens.com/cs/sc

Industry Online Support app

You will receive optimum support wherever you are with the "Siemens Industry Online Support" app. The app is available for iOS and Android:

support.industry.siemens.com/cs/ww/en/sc/2067

11.2 Links and literature

Table 11-1

No.	Subject
\1\	Siemens Industry Online Support https://support.industry.siemens.com
\2\	Link to the article page of the application example https://support.industry.siemens.com/cs/ww/en/view/109758536
\3\	Find and fix problems in your JavaScript code https://eslint.org/

11.3 Change documentation

Table 11-2

Version	Date	Change
V1.0	11/2020	First edition