SIEMENS

Building Automation

Project: **DESIGO INSIGHT V2.3 - Web plus**

Title: **JavaScript Programming Guideline**

Subject: **Technical Documentation**

This document requires or recommends certain practices for developing programs in the JavaScript language.

Key Words: JavaScript, Code, Standard, Guideline

Revision Date: 20-Apr-2004
Company: Siemens Building Technologies AG / Building Automation

---

Disclaimer: Even though this document is marked "For internal use only", we (R&D department in Siemens Building Technologies) think that it is only beneficial for our firm and for the JavaScript programming community to publish this document. The permission for publication has been given on the 08.11.2006 by SBT's head of R&D.
We publish this document as is, and without any guarantee. The purpose of publication is for information only, and to promote consistent programming practice in the JavaScript community.

Table of Contents

1. Introduction

1.1 Objective

This document requires or recommends certain practices for developing programs in the JavaScript language. The objective of this coding standard is to have a positive effect on:

- Avoidance of errors/bugs, especially the hard-to-find ones.
- Maintainability, by promoting some proven design principles.
- Maintainability, by requiring or recommending a certain unity of style.
- Performance, by dissuading wasteful practices.

1.2 Scope

This standard pertains to the use of the JavaScript language. Certain items that deserve attention have been identified, but have **not** been included in this document because treatment in separate documents appears more appropriate.

Not included items are:

- JavaScript class inheritance
- Rules or recommendations on how to layout brackets, braces, and code in general.

1.3 Scope of Validity

This document is intended for developers of the DESIGO INSIGHT Web Plus project.

1.4 Document References

Ref.	Doc.Number	Author	Title
[1]	PD917-3.38	Jean-Claude Philippin, 3271	C# Programming Guideline

2. General rules

2.1 Overview

Rule or Recommendation	
Rule 2-101:	Don't use Alert(), Confirm() and Prompt() in final code
Rule 2-102:	Add Prototype functions only in utility.js
Rule 2-103:	Avoid global functions
Rec. 2-104:	Use Object instead of Array

2.2 Rules and Recommendations

2.2.1 Rule 2-101: Don't use Alert(), Confirm() and Prompt() in final code

Don't use the functions alert(), confirm() or prompt() in the final code. They block the JavaScript thread and therefore the updates in the web application.

They can still be very useful and it's OK to use them during the implementation phase.

2.2.2 Rule 2-102: Add Prototype functions to intrinsic Classes only in utility.js

Prototype functions must be defined in utility.js

Example: in utility.js

```
String.prototype.trim = function(){ ... }
```

2.2.3 Rule 2-103: Don't use viewer specific global functions and variables

Don't use global variables and functions to avoid name conflicts. Name conflicts are not recognised by the JavaScript compiler. Encapsulate viewer specific global functions and variables in an object with the same name as the viewer prefix:

(**pv**: plant viewer, **ov**: object viewer, **ts**: time scheduler, **tv**: trend viewer, **av**: alarm viewer, **lv**: log viewer)

The names pv, ov, etc. are an exception to Rule 3-105.

Example: for the Web Plant Viewer

```
// container for all Web Plant Viewer specific variables, objects and functions
var pv = new Object();

// viewer specific global variable
pv.bInitialized = false;

// viewer specific global function
pv.myMethod = function(strText)
{
     if (pv.bInitialized)
     ...
}
```

2.2.4 Rec. 2-104: Use Object instead of Array for "Hash Table"

Use Object() if the array is just a hash table with strings or numbers as key.

Use an Array() only if you use Array properties like `length` or Array methods like `push(), slice()` etc.

Example: The system id is the key in the hash table `pv.aoTags`

```
pv.aoTags = new Object();
```

```
// create a new Tag and add it to the hash table
var oTag = new Tag("13.4.543.122");
pv.aoTags[oTag.strSystemId] = oTag;

// trace all Tags in the hash table
for (var strKey in pv.aoTags)
     TraceWindow.traceMsg(strKey + ": " + pv.aoTags[strKey]);
```

3. Naming conventions

3.1 Overview

Rule or Recommendation	
Rec. 3-101:	Use US-English for naming identifiers.
Rule 3-102:	Use Pascal and Camel casing for naming identifiers.
Rule 3-103:	Use a prefix, similar to the Hungarian notation
Rule 3-104:	Do not prefix member fields.
Rule 3-105:	Prefix global objects or variables.
Rec. 3-106:	Use abbreviations with care.
Rule 3-107:	Do not use an underscore in identifiers.
Rule 3-108:	Use the object prefix with an underscore for global event handlers.
Rec. 3-111:	Use a noun or a noun phrase to name a `class` or `struct`.
Rule 3-201:	Use `E` and `e` suffix to an enumeration type.
Rule 3-202:	Use singular names for enumeration types.
Rule 3-203:	Use a plural name for enumerations representing bitfields.
Rec. 3-204:	Do not use letters that can be mistaken for digits, and vice versa.
Rule 3-302:	Add `Callback` to callback functions.
Rule 3-401:	Suffix exception classes with `Exception`.
Rule 3-502:	Use small letters for naming source files.

3.2 Rules and Recommendations

3.2.1 Rec. 3-101: Use US-English for naming identifiers

3.2.2 Rule 3-102: Use Pascal, Camel and Capital casing for naming identifiers

- In Pascal casing the first letter of each word in an identifier is capitalized. For example, `BackColor`.
- In Camel casing only the first letter of the second, third, etc. word in a name is capitalized; for example, `backColor`.
- Two-letter abbreviations have both letters capitalized, e.g. `get`**`IE`**`Version()`
- Constant variables with capitals and underscores

The table below provides the casing for the most common types.

IDENTIFIER	CASE	EXAMPLE
Class	Pascal	`function Utility() {}`
Function	Camel	`Utility.getIEVersion = function()`
local variables	Pascal	`var oArrayIndex = 0;`
constant variables	Capital	`pv.MAX_ROWS = 9;`

3.2.3 Rule 3-103: Use a prefix, similar to the Hungarian notation

- Multiple prefixes can be added, e.g. **g_astr**Status
- Conflict with C#: If a JavaScript name has to be the same as a C# name, use the C# name. This can happen in web service methods.

JavaScript Object Type	Prefix
ActiveXObject	o
Array	a
Boolean	b

Date	dt
Enumerator	enum
Error	err
Function	fn
Number	i for int or f for float
Object	o
RegExp	re
String	str

HTML / DOM Object	Prefix
Applet	ap
Button	bn
Checkbox	cb
Combo (Select)	cbo
Directory	dir
Document	doc
File	fl
FileUpload	fup
Form	frm
Frames	fr
Hidden	hd
Image	img
Link	lnk
Node	n
Radiobutton	rb
Radiobutton group	rbg
Span	span
Textarea	ta
TextField	tf
Timer (from setTimeout)	tm
Window und Dialog	win
Svg node	svg

HTML / DOM Object	Prefix
Global object or variable	g_

Example:

```
var bEnable = true;
var svgTime = svgDocument.getElementById("time");
var g_astrStatus = new Array();
```

3.2.4 Rule 3-104: Do not prefix member fields

Do not prefix class members (e.g. **m**_).

3.2.5 Rule 3-105: Prefix global objects or variables

Prefix global objects or variables with **g**_
Example: `g_oToolbar`

Exception:

Rule 2-103, application prefix objects pv, ov, etc.

3.2.6 Rec. 3-106: Use abbrevations with care

Do not contract words in identifiers, but do use well-known abbreviations. For example, do not use `GetWin` instead of `GetWindow`, but **do** use a well-known abbreviation such as `UI` instead of `UserInterface`.

3.2.7 Rule 3-107: Do not use an underscore in identifiers

Exceptions:

Rule 3-102, To separate words in capital letters of constant variables.
Rule 3-108, for the global function `document_onLoad(),document_onUnload() and document_onResize()`

3.2.8 Rule 3-108: Use the object prefix with an underscore for global event handlers

The event handler functions `document_onLoad(),document_onUnload()` and `document_onResize()` are called by the framework. The function names must not be changed.

3.2.9 Rec. 3-111: Use a noun or a noun phrase to name a `class`

Also, if the class involved is a derived class, it is a good practice to use a compound name. For example, if you have a class named `Button`, deriving from this class may result in a class named `BeveledButton`.

3.2.10 Rule 3-201: Use `E` and e suffix to an enumeration type

Example:

```
ToolbarImageButton.EState = new Object();
ToolbarImageButton.EState.eNormal = 0;
ToolbarImageButton.EState.eHover = 1;
```

3.2.11 Rule 3-202: Use singular names of enumeration types

For example, do not name an enumeration type `Protocol`**`s`** but name it `Protocol` instead. See example above.

3.2.12 Rule 3-203: Use a plural name for enumerations representing bitfields

Use a plural name for such enumeration types. The following code snippet is a good example of an enumeration that allows combining multiple options.

```
Utility.EFormats = new Object();
Utility.EFormats.eZero   = 0x01    // Format has leading zeros
Utility.EFormats.eLeft   = 0x02    // Format is left justified
Utility.EFormats.eUnit   = 0x04    // Format has eng units
Utility.EFormats.eExp    = 0x10    // Format is real Exp notation
```

3.2.13 Rec. 3-204: Do not use letters that can be mistaken for digits, and vice versa

To create obfuscated code, use very short, meaningless names formed from the letters `O`, `o`, `l`, `I` and the digits `0` and `1`. Anyone reading code like

```
bool b001 = (lo == 10 ? I1 == 11 : 101 != 101);
```

will marvel at your creativity.

3.2.14 Rule 3-302: Add `Callback` to callback functions

Functions that are used to pass a reference to a callback function must be suffixed with `Callback`. For example:

```
...
pv.callWebService(pv.queryTagCallback, "QueryTag", strTagname);
...

pv.queryTagCallback = function(oResult)
```

```
{
     ...
}
```

3.2.15 Rule 3-401: Suffix exception classes with `Exception`

For example:

```
function InsightException( iErrorCode, strMessage )
{
  this.iErrorCode = iErrorCode;
  this.strMessage = strMessage;
}
```

3.2.16 Rule 3-502: Use small letters for naming source files

JavaScript files will be transferred to the client. Do not use the capital and underscore characters for files names. Use dashes '-' to separate words.
Example: date-picker.js

4. Comments and embedded documentation

4.1 Overview

Rule or Recommendation	
Rule 4-101:	Each file shall contain a header block.
Rule 4-102:	Use // for comments.
Rule 4-103:	All comments habe to be written in US English.
Rule 4-104:	Use XML tags for documenting types and members.

4.2 Rules and Recommendations

4.2.1 Rule 4-101: Each file shall contain a header block

```
//  Copyright ©2004 by Siemens Building Technologies, Building Automation
//--------------------------------------------------------------------------
//  Filename                : infobar.js
//--------------------------------------------------------------------------
//  Project                 : DESIGO INSIGHT Web Plus
//  Language                : ECMAScript V3
//  Author                  : Laurent Bugnion, 3457
//  Date of creation        : 09.03.2004
//--------------------------------------------------------------------------
//  Contains the javascript code used for the Infobar custom control.
//  Use Infobar.setText( ... ) to set the text displayed in the right-most
/// panel. The text displayed will be preserved over round-trips.
//
//  Revisions:
//
//--------------------------------------------------------------------------
```

4.2.2 Rule 4-102: Use // for comments

If the characters // are consistently used for writing comments, then the combination /* */ can be used to make comments out of entire section of code during development and debugging phases, because JavaScript does not allow comments to be nested using /* */.

4.2.3 Rule 4-103: All comments have to be written in US English

Exceptions:
ToDo notes which are removed when the reason is solved.

4.2.4 Rule 4-104: Use XML tags for documenting types and members

All functions, global variables, classes and class members shall be documented using XML tags according to Rule 4-104 [1].
Extension to [1]: The **default values** of function arguments and the data type of the arguments and **return value** can be specified in the <param> tag as an extension to the C# Guideline. The data type of the arguments is optional since the argument has a prefix with the type.
Example:

```
/// <summary>Formats a data point value into a string</summary>
/// <param name="iWidth">The total width of the string </param>
/// <param name="iDigits">The number of decimal places</param>
```

```
/// <param name="iFlags">The flags, how to format the value</param>
///  <list>Utility.EFormats.eZero, Format has leading zeros
/// Utility.EFormats.eLeft, Format is left justified
/// Utility.EFormats.eUnit, Format has eng units
/// Utility.EFormats.eExp, Format is real Exp notation</list>
/// <param name="strUnits">The engineering units</param>
/// <returns type="string">The result string</returns>
/// <remarks>Don't use directly the Utility function</remarks>
/// <example>(4.567).toFormatString(6,2,5,"deg") returns "004.57 deg"</example>
Utility.toFormatString = function(iWidth, iDigits, iFlags, strUnits)
{
     ...
}
```

If the data type can vary, use "**o**" as prefix and specify all data types, which are allowed.
Example:

```
/// <param name="oValue" type="number,string">A number or a string</param>
pv.myFunction = function(oValue)
```

5. Object lifecycle

5.1 Overview

Rule or Recommendation	
Rec. 5-101:	Declare and initialize variables close to where they are used.
Rec. 5-102:	If possible, initialize variables at the point of declaration.
Rule 5-103:	Declare each variable in a separate declaration statement.

5.2 Rules and Recommendations

5.2.1 Rec. 5-101 Declare and initialize variables close to where they are used.

A variable ought to be declared within the smallest possible scope to improve the readability of the code and so that variables are not unnecessary allocated. When a variable that is declared at the beginning of a function is used somewhere in the code, it is not easy to directly see the type of the variable. In addition, there is a risk that such a variable is inadvertently hidden if a local variable, having the same name, is declared in an internal block.

5.2.2 Rec. 5-102: If possible, initialize variables at the point of declaration

```
var bInitialized = checkCondition() ();
var iCount = bInitialized ? 1 : 0;
```

5.2.3 Rule 5-103: Declare each variable in a separate declaration statement.

```
var iMin = 100, iMax = 200;    // wrong, bad style

var iMin = 100;                   // ok
var iMax = 200;
```

6. Control flow

The same rules and recommendations of the C# programming guideline apply also for JavaScript. Refer to chapter 6 of [1].

7. Coding Style

7.1 Overview

RULE OR RECOMMENDATION	
Rule 11-101:	Do not mix coding styles within a group of closely related classes or within a module.
Rule 11-202:	Write unary, increment, decrement, function call, subscript, and access operators together with their operands.
Rule 11-203:	Each new block is indented by tow spaces.
Rule 11-204:	Do not create source lines longer than 80 characters.

7.1.1 Rule 11-101: Do not mix coding styles within a group of closely related classes or within a module.

This coding standard gives you some room in choosing a certain style. Do keep the style consistent within a certain scope. That scope is not rigidly defined here, but is at least as big as a source file.

7.1.2 Rule 11-202: Write unary, increment, decrement, function call, subscript, and access operators together with their operands.

This concerns the following operators:

unary:	`& * + - ~ !`
increment and decrement:	`-- ++`
function call and subscript:	`() []`
access:	`.`

It is not allowed to add spaces in between these operators and their operands.

It is not allowed to separate a unary operator from its operand with a newline.

Note: this rule does **not** apply to the **binary** versions of the `& * + -` operators.

Example:

```
a = -- b;           // wrong
a = --c;            // right

a = -b - c;         // right
a = (b1 + b2) +
    (c1 - c2) +
    d – e - f;      // also fine: make it as readable as possible
```

7.1.3 Rule 11-203: Each new block is indented by two spaces.

Spaces or tabs may be used. If tabs are used the tab size must be set to two spaces.

7.1.4 Rule 11-204: Do not create source lines longer than 120 characters.