



**COPADATA**  
do it your way

# zenon manual

## Programming Interfaces

v.7.60





©2017 Ing. Punzenberger COPA-DATA GmbH

All rights reserved.

Distribution and/or reproduction of this document or parts thereof in any form are permitted solely with the written permission of the company COPA-DATA. Technical data is only used for product description and are not guaranteed qualities in the legal sense. Subject to change, technical or otherwise.

# Contents

<b>1. Welcome to COPA-DATA help .....</b>	<b>6</b>
<b>2. Programming Interfaces.....</b>	<b>6</b>
<b>3. Add-Ins.....</b>	<b>9</b>
3.1 Prior knowledge .....	10
3.2 Terminology .....	10
3.3 Limitations.....	12
3.4 Create add-in.....	12
3.4.1 Basics .....	13
3.4.2 Types of extensions .....	17
3.4.3 Content of add-in packages.....	19
3.4.4 SharpDevelop .....	20
3.4.5 Microsoft Visual Studio .....	22
3.4.6 Source code management.....	24
3.4.7 Add-in analysis and packaging utility (AddInUtility) .....	24
3.4.8 Action in the event of reloading.....	25
3.4.9 Isolation .....	26
3.5 Use of add-ins in Editor and Runtime .....	26
3.5.1 Activate add-ins in zenon .....	28
3.5.2 Action during installation .....	28
3.5.3 Add-ins node in the Project Manager .....	29
3.5.4 Use in the zenon Editor .....	30
3.5.5 Use in zenon Runtime .....	35
3.6 Switch/conversion from VSTA.....	37
<b>4. Macro list .....</b>	<b>38</b>
4.1 VBA toolbar and context menu detail view .....	39
4.2 VBA on 64-bit systems .....	42
4.3 Basics.....	43
4.3.1 Object PROPERTIES.....	43
4.3.2 Object METHODS.....	44
4.3.3 Object EVENTS.....	44

4.3.4	VBA object structure in zenon .....	44
4.3.5	How to use VBA macros .....	46
4.3.6	How to insert an ActiveX element in zenon? .....	47
4.3.7	Access from an external program .....	49
4.3.8	Functionality of online variables .....	50
4.3.9	List of status bits.....	52
4.3.10	Lasso for selecting dynamic elements in the Runtime .....	54
4.4	Macros in the Editor.....	55
4.4.1	Tool bar macro list.....	56
4.4.2	Linking macros.....	57
4.5	Functions in zenon .....	58
4.5.1	Execute VBA macro .....	59
4.6	Developing wizard in VBA .....	60
4.6.1	Using a wizard .....	61
4.6.2	Structure of a wizard .....	61
4.6.3	Integration in VBA .....	62
4.6.4	Developing a wizard .....	62
4.6.5	Updating wizards.....	68
4.7	Frequently asked questions .....	69
4.7.1	Why does the button stay pressed?.....	69
4.7.2	Macro is not performed with the first click.....	69
4.7.3	Macros no longer work in the Runtime?.....	69
4.7.4	Windows CE and VBA .....	70
4.8	Examples .....	70
4.8.1	MouseEvents and ActiveX Control initialization.....	70
4.8.2	Display variable information .....	71
4.8.3	Read and write variable values .....	72
4.8.4	Read and write variables and implement online variables .....	73
4.8.5	Use dialog multiple times .....	75
4.8.6	Alarm – Events and ActiveX Control handling .....	77
4.8.7	Access to alarms .....	79
4.8.8	Set switch (working with process variables) .....	81
<b>5.</b>	<b>VSTA .....</b>	<b>84</b>
5.1	Basics.....	85
5.1.1	Setting up the VSTA environment .....	85

5.1.2	Access to the object model in zenon.....	86
5.1.3	Functions in zenon.....	88
5.1.4	Debugging a VSTA add-in .....	89
5.1.5	Events in VSTA .....	90
5.1.6	Creating a backup of VSTA projects.....	90
5.2	Creating a VSTA project .....	91
5.2.1	VSTA projects in the editor.....	91
5.2.2	VSTA projects in Runtime .....	92
5.2.3	Developing wizards in VSTA .....	93
5.3	Examples .....	94
5.3.1	Creating variables in the zenon editor .....	94
5.3.2	Writing project information in the zenon output window.....	97
5.3.3	Reading in of variables in zenon via regular expressions .....	99
<b>6.</b>	<b>Process Control Engine (PCE).....</b>	<b>103</b>
6.1	The PCE Editor.....	103
6.1.1	The Taskmanager .....	104
6.1.2	The editing area.....	104
6.1.3	The output window .....	104
6.1.4	The menus of the PCE Editor .....	105
6.1.5	The icon bar of the PCE Editor.....	107
6.2	Course of actions.....	108
6.2.1	Creating a task.....	108
6.2.2	Entering code .....	110
6.2.3	Function Show PCE .....	112
6.2.4	Executing tasks .....	113
6.3	VB Script - Introduction.....	114
6.3.1	Data types.....	114
6.3.2	Variables .....	115
6.3.3	Constants.....	118
6.3.4	Operators .....	118
6.3.5	Conditional Statements.....	120
6.3.6	Looping Through Code .....	122
6.3.7	Types of procedures .....	127
6.3.8	Coding Conventions.....	129

# 1. Welcome to COPA-DATA help

## ZENON VIDEO-TUTORIALS

You can find practical examples for project configuration with zenon in our YouTube channel ([https://www.copadata.com/tutorial\\_menu](https://www.copadata.com/tutorial_menu)). The tutorials are grouped according to topics and give an initial insight into working with different zenon modules. All tutorials are available in English.

## GENERAL HELP

If you cannot find any information you require in this help chapter or can think of anything that you would like added, please send an email to [documentation@copadata.com](mailto:documentation@copadata.com) (<mailto:documentation@copadata.com>).

## PROJECT SUPPORT

You can receive support for any real project you may have from our Support Team, who you can contact via email at [support@copadata.com](mailto:support@copadata.com) (<mailto:support@copadata.com>).

## LICENSES AND MODULES

If you find that you need other modules or licenses, our staff will be happy to help you. Email [sales@copadata.com](mailto:sales@copadata.com) (<mailto:sales@copadata.com>).

# 2. Programming Interfaces

Different interfaces to integrate your own programs or to automate planning are available in zenon:

- **Macro List (on page 38) (VBA)**

► **VSTA (on page 84)**

► Add-In Framework (on page 26) (Add-Ins)

► **Process Control Engine (PCE)**

Starting from version 7.20, PCE will not be supported anymore and it will not be shown in the module tree of zenon anymore. While converting projects from versions lower than 7.20, which contain PCE tasks, the node PCE will be shown for these projects again. PCE will not further be developed or further documented.

**Recommendation:** Please use **zenon Logic** instead of PCE



### License information

*Part of the standard license of the Editor and Runtime.*

## CONTEXT MENU

Menu item	Action
<b>Open VBA Editor</b>	Opens the VBA editor
<b>Open the VSTA Editor with Projectaddin</b>	Opens the VSTA editor.
<b>Help</b>	Opens online help.



### Information

*you can find information on the creation and implementation of controls (ActiveX, .NET, WPF) in the Controls manual.*

*You can find information on engineering and use of the SAP interface in the SAP interface chapter.*

## OPEN EDITORS

### VBA-EDITOR FOR THE MACRO LIST

VBA starts the same development environment for **Workspace** and **Project**.

To open the VBA Editor:

1. In the zenon Editor, navigate to the **Programming interface** node.
2. Expand the view of this node by clicking on **[+]**.  
The view of the node is expanded.
3. Right-click on **Macro list**

4. Select the entry **Open VBA Editor** in the context menu.

**Alternative:** press the short cut **Ctrl+F11**

## VSTA EDITOR

VSTA provides separate development environments for **Workspace** and **project**. You can only use one of them at a time. At the start every other VSTA development environment which is open will be close.

To open the VSTA Editor for the workspace:

1. Press the short cut **Alt+F10**.

The code for the workspace and all loaded projects is displayed.

To open the VSTA Editor for the currently loaded project:

1. Navigate to the **Programming interfaces** node
2. Expand the view of this node by clicking on **[+]**.  
The view of the node is expanded.

3. Right-click on **VSTA**

In the context menu, select **Open VSTA editor with ProjectAddin**.  
The editor is opened for the currently-loaded project.

## API FROM VERSION 7.10:

For the use of zenon programming interfaces, the following is applicable from zenon 7.10:

- ▶ VSTA/.NET: **.NET Framework 3.5** must be installed.
- ▶ VBA: If, in the VBA code, Windows API or other imported DLL functions are accessed, these calls must be adapted to the 64-bit environment. In general, the following applies: A VBA file created with a 32-bit version cannot be used without changes in a 64-bit version of VBA.



### Attention

Errors in applications such as ActiveX, PCE, VBA, VSTA, WPF and external applications that access zenon via the API can also influence the stability of Runtime.

### 3. Add-Ins

Add-Ins provide possibilities to expand the functionality of zenon in the Editor and in Runtime with different development environments.

The following are available to you to create Add-Ins:

- ▶ Microsoft Visual Studio (on page 22)
- ▶ SharpDevelop (included in the installation package) (on page 20)

Add-Ins can be created with each .NET programming language. IDE support is available for the programming languages C# and Visual Basic.NET.

**Note:**

- ▶ All lists and interfaces support **IEnumerable**.  
Reference: [https://msdn.microsoft.com/de-de/library/9eehta0\(v=vs.110\).aspx](https://msdn.microsoft.com/de-de/library/9eehta0(v=vs.110).aspx)  
([https://msdn.microsoft.com/de-de/library/9eehta0\(v=vs.110\).aspx](https://msdn.microsoft.com/de-de/library/9eehta0(v=vs.110).aspx) )
- ▶ **LINQ** (Language Integrated Query) can be used.  
Reference: <https://msdn.microsoft.com/en-us/library/mt693024.aspx>  
(<https://msdn.microsoft.com/en-us/library/mt693024.aspx>)



#### Attention

*For Add-Ins, the .NET Naming Guidelines are used.*

*The object names previously used for VSTA and VBA are amended for this. In VSTA and VBA, the previous names remain the same for compatibility reasons. For Add-Ins, naming according to the .NET Naming Guidelines has been used.*

You can find an overview of the respective object names in the **API\_Naming\_Conversion.xlsx** Excel table. You can find this in the zenon installation path in the **\HELP\AddIns** folder.

#### FUNCTIONALITY

Add-Ins allow you to do the following, among other things:

- ▶ Provide project configuration aids (such as wizards)
- ▶ To enhance the project in Runtime with additional functionality

You can find Add-Ins:

- ▶ For the Editor in the **Extras** menu under the following entries:
  - **Manage Editor Add-Ins**
  - **Start Editor Wizards**
  - **Manage Editor services...**

- ▶ For Runtime in the **Programming interfaces - Add-Ins** node

## TOOLS AND SUPPORT

You can get tools and support at:

- ▶ Visual Studio developer tools: [marketplace.visualstudio.com](https://marketplace.visualstudio.com)  
(<https://marketplace.visualstudio.com>)
- ▶ COPA-DATA user forum: [forum.copadata.com](https://forum.copadata.com/) (<https://forum.copadata.com/>)

### 3.1 Prior knowledge

As with the creation of other software components, certain prior knowledge is also necessary for the development of Add-Ins for zenon.

This includes

- ▶ Programming knowledge in C# or VB.NET (depending on the requirement / complexity of the Add-Ins to be developed)
- ▶ Good zenon knowledge

The following is also recommended

- ▶ Experience in using professional development environments (such as Microsoft Visual Studio)
- ▶ The use of a version management system for the developments (TFS, SVN, GIT)

### 3.2 Terminology

Specialist terms and their meaning in relation to Add-Ins.

Term	Meaning
<b>Add-In</b>	A collection of <b>Extensions</b> . Is represented by the add-in package. An <b>Add-In</b> can contain several <b>Extensions</b> .
<b>Add-In Assembly</b>	The compiled library (*.dll), that contains <b>Extensions</b> .
<b>Add-In Package</b>	A ZIP file (*.scadaAddIn), that contains metadata and libraries with their dependencies.  An <b>Add-In Package</b> contains an <b>Add-In Assembly</b> .
<b>Extension</b>	<b>Extensions</b> are classes that can be derived from an interface with one or more methods. They are entry points in the <b>Add-In Package</b> .  Each class that represents an <b>Extension</b> is marked with a .NET attribute.  There are two types of <b>Extensions</b> (on page 17), one for the Editor and one for Runtime <ul style="list-style-type: none"> <li>▶ <b>Wizard Extensions</b> (Wizards)</li> <li>▶ <b>Service Extensions</b> (Services)</li> </ul>
<b>IDE</b>	Integrated development environment. Abbreviation for Integrated Development Environment.
<b>Project Add-In Management</b>	Components for zenon Editor and Runtime that manage the <b>Project Add-In Packages</b> .
<b>Service Extension</b>	Implementation of an <b>Extension</b> that represents a <b>Service</b> . <b>Services:</b> <ul style="list-style-type: none"> <li>▶ Runs in the background</li> <li>▶ Is started automatically or manually</li> <li>▶ Remains in the memory until it is stopped</li> </ul>
<b>Wizard Extension</b>	Implementation of an <b>Extension</b> that represents a wizard. <b>Wizards:</b> <ul style="list-style-type: none"> <li>▶ Are for solving a certain task</li> <li>▶ Are removed again after closing</li> </ul>

**Note:** Specialist terms are only used in English in the documentation for Add-Ins.

### 3.3 Limitations

The following limitations are applicable for Add-Ins:

- ▶ The zenon Web Client supports the execution of Add-Ins, however without IDE support.  
The local debugging of Add-Ins is not possible in the zenon Web Client for this reason; debugging using remote debugging tools is supported.
- ▶ The execution of Add-Ins is not supported in the HTML web engine.

### 3.4 Create add-in

#### GENERAL INFORMATION

Add-Ins are identified by means of the Add-In ID. It is used for installation and import.

An Add-In ID consists of an optional Namespace and a local ID, **namespace.localId**. The Local ID identifies an Add-In; the Namespace assigns an Add-In to an organization. Both pieces of information are established by the NET attribute **Mono.AddIn.AddInAttribute**. In order for the name for an Add-In Package to also remain unique beyond organization boundaries, the Namespace of the Add-In project is used as part of the ID.

Example of a complete Add-In ID: **com.organisation.importwizard**.

The issue of Namespaces is optional.

**Recommendation:** Use Namespaces. This is how you ensure the global uniqueness beyond organizational boundaries. It is best to use the URL of your own organization in reverse order. For example: For **www.example.com** use **com.example** as a Namespace.

Namespaces can be further divided within an organization, for example by teams, departments or projects.

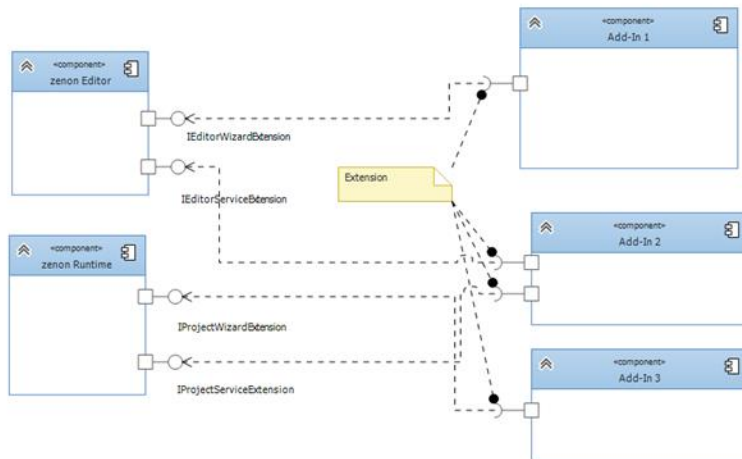
The following are available to you to create Add-Ins:

- ▶ Microsoft Visual Studio (on page 22)
- ▶ SharpDevelop (included in the installation package) (on page 20)

Add-Ins can be created with each .NET programming language. IDE support is available for the programming languages C# and Visual Basic.NET.

### 3.4.1 Basics

There are different Extensions available for the zenon Editor and zenon Runtime. Depending on the application, a corresponding Extension can be selected.



All interfaces required for the development of Add-Ins are defined in the **Scada.AddIn.Contracts** assembly.

An Add-In Package can contain one or more Extensions .

#### TYPES OF EXTENSIONS - INTERFACES

A distinction is made between the following Extensions :

##### **EDITOR WIZARD EXTENSION - IEDITORWIZARDEXTENSION (ON PAGE 17)**

- ▶ Execution is triggered by the user using the **Extras -> Editor Wizards...** dialog in the zenon Editor.
- ▶ Implements the **Run** method in order to execute a Wizard Extension in the zenon Editor.

##### **EDITOR SERVICE EXTENSION - IEDITORSERVICEEXTENSION (ON PAGE 17)**

- ▶ Execution is carried out automatically when the Editor is started, depending on the **DefaultStartMode**.

**Note:** For automatic start, the `DefaultStartMode=DefaultStartupModes.Auto` attribute must be set in the **Extension**. The start mode can also be changed with the **Manage Editor services** (on page 33) dialog.

- ▶ Administration is via the **Extras -> Manage Editor services...** dialog in the zenon Editor.
- ▶ Implements the **Start** and **Stop** methods to execute a Service Extension in the zenon Editor.

#### **PROJECT WIZARD EXTENSION - IPROJECTWIZARDEXTENSION (ON PAGE 18)**

- ▶ Execution is triggered by the user with the zenon **Execute Project Wizard Extension** function.
- ▶ Implements the **Run** method in order to execute a Wizard Extension in zenon Runtime.

#### **PROJECT SERVICE EXTENSION - IPROJECTSERVICEEXTENSION (ON PAGE 18)**

- ▶ Execution takes place automatically when starting the project in Runtime, depending on the **DefaultStartMode**.

**Note:** For automatic start, the `DefaultStartMode=DefaultStartupModes.Auto` attribute must be set in the **Extension**. The start mode can also be changed with the **Manage Editor services** (on page 33) dialog.

- ▶ Administration is carried out using the zenon **Show "Manage Runtime services"** function.
- ▶ Implements the methods **Start** and **Stop** to execute a Service Extension in zenon Runtime.

#### **EXTENSION ATTRIBUTES**

The following metadata is defined using the .NET attribute with the name **AddInExtension**.

Property	Description	IEditorWizardExtension	IEditorServiceExtension	IProjectWizardExtension	IProjectServiceExtension
<b>Name</b>	Display name in dialogs. <b>Info:</b> Corresponds to the name of the wizard or the service in zenon.	<b>X</b>	<b>X</b>	<b>X</b>	<b>X</b>
<b>Description</b>	Description of the display in dialogs. <b>Info:</b> Corresponds to the description of the wizard or the service in zenon.	<b>X</b>	<b>X</b>	<b>X</b>	<b>X</b>
<b>Category</b>	Category for grouping in dialogs. <b>Info:</b> Is used to categorize wizards in the <b>Editor Wizards...</b> dialog in the zenon Editor.	<b>X</b>	--	<b>X</b>	--
<b>DefaultStartMode</b>	Describes the standard starting behavior of service add-in extensions after installation. <ul style="list-style-type: none"> <li>▶ <b>Manual:</b> Extension must be started manually by the user.</li> <li>▶ <b>Auto:</b> Is started if the workspace in the Editor or the project is loaded in Runtime.</li> </ul> N/A: Manual is used. <b>Info:</b> Corresponds to the start type of the service in zenon.	--	<b>X</b>	--	<b>X</b>

Key:

- ▶ **X:** applicable
- ▶ **--:** not applicable

**ADD-IN ATTRIBUTES**

An Add-In Assembly needs several .NET attributes, which are defined in the **AddInInfo.cs** file.

The most important attributes and those necessary for use in zenon are:

Attribute name	Obligatory	Description	Examples
Addin	<b>X</b>	<p>ID and version of the Add-In.</p> <p><b>Info:</b> The version corresponds to the <b>Version</b> column in the Add-In administration dialog in the zenon Editor.</p>	<p><u>Example 1:</u></p> <pre>[assembly: Addin("AddInIdentifier", "1.0", Namespace="com.example ")]</pre> <p><u>Example 2:</u></p> <pre>[assembly: Addin("MultitouchAddIn", "1.0")]</pre> <p><u>Example 3:</u></p> <pre>[assembly: Addin("ProjectCreation", "1.0")]</pre>
AddinDependency	<b>X</b>	<p>Defines a dependency on the zenon system. This value is constant and is needed in order for zenon to use the Add-In</p>	<pre>[assembly: AddinDependency("scada", "1.0")]</pre>
AddinName	--	<p>Name of the Add-In.</p> <p><b>Info:</b> Corresponds to the <b>Name</b> column in the Add-In administration dialog in the zenon Editor.</p>	<pre>[assembly: AddinName("Editor Add-In Demo")]</pre>
AddinDescription	--	<p>Description of the Add-In.</p> <p><b>Info:</b> Corresponds to the <b>Description</b> column in the Add-In administration dialog in the zenon Editor.</p>	<pre>[assembly: AddinDescription("Demos trates an Add-In with a Wizard and a Service extension.")]</pre>

Key:

- ▶ **X:** applicable
- ▶ **--:** not applicable

### 3.4.2 Types of extensions

#### Wizard in the Editor - Editor wizard extension

Editor Wizard Extensions are used to implement wizards or other user-controlled API processes in the Editor.

- ▶ Execution is triggered by the user using the **Extras -> Editor Wizards...** dialog in the zenon Editor.
- ▶ Implements the **Run** method in order to execute a Wizard Extension in the zenon Editor.

#### EXAMPLE CODE (C#)

```
[AddInExtension("Wizard Extension Name", "Description of Wizard Extension", "Category of Wizard Extension")]

public class WizardExtension : IEditorWizardExtension
{
    public void Run(IEditorApplication context, IBehavior behavior)
    {
        // Implement feature here...
    }
}
```

#### Services in the Editor - Editor Service Extension

Editor Service Extensions are used to implement API processes in the Editor automatically.

- ▶ Execution is carried out automatically when the Editor is started, depending on the **DefaultStartMode**.

**Note:** For automatic start, the `DefaultStartMode=DefaultStartupModes.Auto` attribute must be set in the **Extension**. The start mode can also be changed with the **Manage Editor services** (on page 33) dialog.

- ▶ Administration is via the **Extras -> Manage Editor services...** dialog in the zenon Editor.
- ▶ Implements the **Start** and **Stop** methods to execute a Service Extension in the zenon Editor.

#### EXAMPLE CODE (C#)

```
[AddInExtension("Service Extension Name", "Description of Service Extension", "Category of Service Extension", DefaultStartMode=DefaultStartupModes.Auto)]

public class ServiceExtension : IEditorServiceExtension
```

```
{  
    public void Start(IEditorApplication context, IBehavior behavior)  
    {  
        // Startup code here  
    }  
    public void Stop()  
    {  
        // Stop code here  
    }  
}
```

### Wizard in Runtime - project wizard extension

Project Wizard Extensions are used to implement wizards or other user-controlled API processes in Runtime.

- ▶ Execution is triggered by the user with the zenon **Execute Project Wizard Extension** function.
- ▶ Implements the **Run** method in order to execute a Wizard Extension in zenon Runtime.

#### EXAMPLE CODE (C#)

```
[AddInExtension("Wizard Extension Name", "Description of Wizard Extension", "Category of  
Wizard Extension")]  
  
public class WizardExtension : IProjectWizardExtension  
{  
    public void Run(IProject context, IBehavior behavior)  
    {  
        // Implement feature here...  
    }  
}
```

### Service in Runtime - project service extension

Runtime Service Extensions are used to implement background tasks in Runtime.

- ▶ Execution takes place automatically when starting the project in Runtime, depending on the **DefaultStartMode**.

**Note:** For automatic start, the `DefaultStartMode=DefaultStartupModes.Auto` attribute must be set in the **Extension**. The start mode can also be changed with the **Manage Editor services** (on page 33) dialog.

- ▶ Administration is carried out using the zenon **Show "Manage Runtime services"** function.
- ▶ Implements the methods **Start** and **Stop** to execute a Service Extension in zenon Runtime.

#### EXAMPLE CODE (C#)

```
[AddInExtension("Service Extension Name", "Description of Service Extension", "Category
of Service Extension", DefaultStartMode=DefaultStartupModes.Auto)]

public class ServiceExtension : IProjectServiceExtension
{
    public void Start(IProject context, IBehavior behavior)
    {
        // Startup code here
    }

    public void Stop()
    {
        // Stop code here
    }
}
```

### 3.4.3 Content of add-in packages

Add-In Packages are ZIP files with the filename extension **.scadaAddIn**. They contain binary data and metadata. Binary files must be present and can be saved as their own ZIP file in the Add-In Package. Packages can be created with the **Add-In Utility** (on page 24). The SharpDevelop and Microsoft Visual Studio development environments are used to create the Packages automatically when compiling.

**Recommendation:** Do not change the files manually, always use the **Add-In Utility**.

#### CONTENT OF THE ADD-IN PACKAGE

The Add-In Package contains:

- ▶ **Content.xml:** XML metadata file with contents of the package.
- ▶ **Binary.zip:** ZIP file with the binary files.

## XML-FILE

The content of an Add-In Package is listed in its own XML file (**Content.xml**).

## BINARY FILES

Binary files are saved in their own ZIP file. This file must contain all components that are needed on another system to execute the Add-In. Direct and indirect references to .NET Assemblies. In doing so, only external Assemblies are included. Assemblies that are not part of the .NET Framework Class Library or are installed during the zenon installation.

**Attention:** Ensure that you have all the licenses required for this.

**Note:** Dynamically-loaded .NET Assemblies are not included in the metadata of .NET Assemblies.



### Information

COPA-DATA accepts no liability and offers not technical support for external Libraries.

## 3.4.4 SharpDevelop

**SharpDevelop** (also #develop) is an open-source development environment that is based on Microsoft's .NET platform. **SharpDevelop** is supplied with zenon and is also installed during the installation of zenon.

**Note:** You can find the respective current version and the official documentation for **SharpDevelop** online at: [www.icsharpcode.net/opensource/sd/](http://www.icsharpcode.net/opensource/sd/) (<http://www.icsharpcode.net/opensource/sd/>).

### Create add-in package with SharpDevelop

To create an Add-In Package with SharpDevelop:

1. Start **SharpDevelop** by
  - selecting the **Extras -> Open add-in editor ...** menu in the zenon Editor, or
  - selecting, in the **Startup Tool**, under **Tools**, the **SharpDevelop IDE** linking
2. Select **File -> New Solution** and select, under **C# -> SCADA Add-Ins** or **VB -> SCADA Add-Ins**, a corresponding template; create a new project based on this:
  - Editor Service Extension (on page 17)
  - Editor Wizard Extension (on page 17)
  - Project Service Extension (on page 18)

- Project Wizard Extension (on page 18)

**Note:** All examples used in this help chapter are based on C#.

3. Add your code, references, etc.
4. Compile the project.  
The Add-In Package with the file suffix **\*.scadaAddIn** is created in the output window of the project.  
This is called, depending on the configuration `...bin\Debug\` or `...bin\Release\`.
5. Import the add-in package in the Editor, see the following in relation to this:
  - Installing and manging add-ins for the Editor (on page 30)
  - Installing and manging add-ins for Runtime (on page 35)

## Deploying and debugging add-ins

### DEBUGGING IN THE EDITOR

To debug an Add-In in the Editor, proceed as follows:

1. Start the zenon Editor.
2. Start **SharpDevelop** by
  - selecting the **Extras -> Open add-in editor ...** menu in the zenon Editor, or
  - selecting, in the **Startup Tool**, under **Tools**, the **SharpDevelop IDE** linking
3. Open the project.
4. In the project settings in SharpDevelop, go to the **Add-In** tab.
5. Select `Editor` under **Debug Target**.
6. Start the debugging using `F5` or **Debug -> Run**.
7. The Add-In project is compiled and deployed in the Editor.

**Note:** The Add-In is copied to the  
`%ProgramData%\COPA-DATA\zenon760\EditorAddInCache` directory in the  
 process.

To remove it from the Editor again, proceed as outlined in the Installing and managing Add-Ins  
 for the Editor (on page 30) chapter.

If there is already an Add-In with the same ID, this is replaced.

8. Debug the Add-In.

## DEBUGGING IN RUNTIME

To debug an Add-In in Runtime, proceed as follows:

1. Start the zenon Runtime.
2. Start **SharpDevelop** by selecting, in the **Startup Tool** under **Tools**, the **SharpDevelop IDE** link
3. Open the project.
4. In the project settings in SharpDevelop, go to the **Add-In** tab.
5. Select, under **Debug Target**, the corresponding project in Runtime in which you want to debug the Add-In.
6. Start the debugging using **F5** or **Debug -> Run**.
7. The Add-In project is compiled and temporarily deployed in the selected zenon project.

**Attention:** The Add-In is temporarily copied to the  
`... \RT\FILES\zenon\system\AddInCache\...` directory in the process.

It is removed again after Runtime is closed.

To install an Add-In on a permanent basis, import it as described in the Installing and managing Add-Ins for Runtime (on page 35).

8. Debug the Add-In.



### Information

*As an alternative to the methods described, you can also debug your Add-Ins using **Debug -> Attach to Process**. In doing so, connect from the corresponding Add-In project in SharpDevelop to the process in the zenon Editor or Runtime.*

## 3.4.5 Microsoft Visual Studio

### Create add-in package with Visual Studio

To create an Add-In Package with Visual Studio:

1. Start Visual Studio.
2. Install the Visual Studio Developer Tools.  
**Note:** This can be downloaded from the Visual Studio Marketplace  
<https://marketplace.visualstudio.com/>.
3. Select **File -> New Project** and select, under **Templates -> Visual C# -> SCADA Add-Ins** or **Templates -> Visual Basic -> SCADA Add-Ins**, a corresponding template; create a new project based on this:

- Editor Service Extension (on page 17)
- Editor Wizard Extension (on page 17)
- Project Service Extension (on page 18)
- Project Wizard Extension (on page 18)

**Note:** All examples used in this help chapter are based on C#.

4. Add your code, references, etc.
5. Create the project.
6. The Add-In Package with the file suffix **\*.scadaAddIn** is automatically created in the output folder of the project (`...bin\Debug\` or `...bin\Release\` depending on configuration).
7. Import the add-in package in the Editor, see the following in relation to this:
  - Installing and manging add-ins for the Editor (on page 30)
  - Installing and manging add-ins for Runtime (on page 35)

## Deploying and debugging add-ins

### DEBUGGING IN THE EDITOR

To debug an Add-In in the Editor, proceed as follows:

1. Start the zenon Editor.
2. Start the debugging in Visual Studio using **F5** or **Debug -> Start Debugging**.
3. The Add-In project is compiled and deployed in the Editor.

**Note:** The Add-In is copied to the `%ProgramData%\COPA-DATA\zenon760\EditorAddInCache` directory in the process.

To remove it from the Editor again, proceed as described in the Installing and managing Add-Ins for the Editor (on page 30) chapter.

4. Debug the Add-In.



#### Information

*As an alternative to the method described, you can also debug your Add-Ins by means of **Debug -> Attach to Process**. In doing so, connect from the corresponding Add-In project in Visual Studio to the zenon Editor process.*

*Recommended debugger type (**Debugger Type**): Managed mit CLR 4.0*

## DEBUGGING IN RUNTIME

To debug an Add-In in Runtime, proceed as follows:

1. Compile the Add-In and import it as described in Installing and managing Add-Ins for Runtime (on page 35).
2. Start the zenon Runtime.
3. In Visual Studio, select **Debug -> Attach to Process**
4. Select the zenon Runtime process (**Zenrt32.exe**) and click on **Attach**
5. Debug the Add-In.

### 3.4.6 Source code management

To administer the source code, the use of a version control system - such as TFS, GIT or SVN - is recommended.



#### Attention

*In contrast to VSTA and VBA, the Source Code of Add-Ins is not saved and administered in the zenon project.*

### 3.4.7 Add-in analysis and packaging utility (AddInUtility)

The **AddInUtility** tool is automatically called up when compiling Add-Ins and has the following tasks:

- ▶ Analysis of the dependencies
- ▶ Packaging of Add-Ins

A package contains the Add-In Assembly and an XML file with the extension **\*.scadaAddin**, which contains the metadata for the Add-In.

#### Command line tool AddInUtility.exe

The **AddInUtility** packages Add-Ins using the command line.

Syntax: **AddInUtility.exe** [-[shortTerm] or [/ or --][longTerm] [argument value]] ...

- ▶ Arguments are case sensitive
- ▶ Values are case sensitive if marked as such.
- ▶ Arguments can be stated in any desired order.

Argument	Description
<b>--action</b> <b>(-a)</b>	Action that is to be carried out. Possible values: <ul style="list-style-type: none"> <li>▶ BuildPackage</li> <li>▶ GetPackageInfo</li> </ul> Mandatory information.
<b>--path</b> <b>(-p)</b>	Path information. Required information for: <ul style="list-style-type: none"> <li>▶ <b>action</b> = BuildPackage: Path to add-in assembly file.</li> <li>▶ <b>action</b> = GetPackageInfo: Path to Add-In Package file.</li> </ul> Mandatory information.
<b>--targetDir</b> <b>(-t)</b>	Target folder. Required information for: <ul style="list-style-type: none"> <li>▶ <b>action</b> = BuildPackage: Path to the folder in which the add-in package is created.</li> </ul> Default: current folder

#### Examples

- ▶ Creates an Add-In Package (arguments written out in full):  
**AddInUtility.exe --action BuildPackage --path C:\Addin.dll --targetDir C:\Directory\**
- ▶ Creates an Add-In Package (arguments in abbreviated form):  
**AddInUtility.exe -a BuildPackage -p C:\Addin.dll -t C:\Directory\**
- ▶ Returns information about the given Add-In Package:  
**AddInUtility.exe -a GetPackageInfo -p C:\Directory\Addin.scadaAddIn**

### 3.4.8 Action in the event of reloading

zenon offers a reload function for projects that support hot plugging or the updating of amended project files in runtime. According to the functions of the Add-Ins triggered, the sequence of activities for VSTA and VBA is as follows:

1. Pre-reload VSTA
2. Stop of Service Extensions

3. Reload VBA
4. Start of Service Extensions
5. Post-reload VSTA

Add-Ins behave like VSTA and provide a Pre-reload-function and a Post-reload-function. These are used after the VSTA Pre-reload and before reloading of VSTA and as a Wrapper for the VBA `reload` function. Reloading is delayed until the execution of Wizard Extensions has been completed.

**Recommendation:** If you want to use the reload functionality, avoid Wizard-Extensions that block or run for a long time.

When being loaded again, all running service expansions are restarted. Newly-installed service expansions are started after the restart if the **DefaultStartMode** is set to `automatic`.

### 3.4.9 Isolation

All Add-Ins are executed in an isolated memory area (App-Domain). As a result, different versions of the same Assemblies can be loaded at the same time.

## 3.5 Use of add-ins in Editor and Runtime

Add-Ins provide project configuration aids in the zenon Editor and supplement the functionalities of zenon in Runtime.

You can find Add-Ins:

- ▶ For the Editor in the **Extras** menu under the following entries:
  - **Manage Editor Add-Ins**
  - **Start Editor Wizards**
  - **Manage Editor services...**
- ▶ For Runtime in the **Programming interfaces - Add-Ins** node

### ADD-INS FOR EDITOR AND RUNTIME

- ▶ Add-Ins for the Editor (on page 30):  
Add-Ins are imported, installed and can be used in the Editor.  
Administration (on page 30) is carried out by means of entries in the **Extras** menu.
- ▶ Add-Ins for Runtime (on page 35):  
Add-Ins are imported; when starting Runtime, they are activated automatically and can then be used.

The administration (on page 35) is carried out using the detail view of the **programming interface - Add-Ins** node in the project tree.

Add-Ins are available as:

- ▶ Wizard: Is called up by the user and closed again after the task has been completed. For details on the wizards supplied with zenon, read the **Wizards** manual.
- ▶ Service: Can be started automatically or manually and runs in the background.

Add-Ins can:

- ▶ Be supplied with zenon
- ▶ Be developed by integrators and provided to their customers
- ▶ Developed by customers themselves

## FILTER AND SORT THE LIST OF ADD-INS AND SERVICES

To filter a list in the detail view or in a dialog:

1. Click in the filter line of the desired curve.
2. Enter the filter criteria.  
Placeholders can be used.
  - ?: Placeholder for precisely one desired character.
  - \*: Placeholder for as many characters as desired.
3. Several filters can be combined.

**Note:** To remove all filters, click on the corresponding symbol in the tool bar.

To sort the list:

1. Click on the column title of the column according to which sorting is to take place.  
The binoculars symbol shows the column according to which sorting takes place.  
The list is sorted according to the column.  
An arrow shows whether the sorting is ascending or descending.
2. To change the sorting sequence, click on the column title again.

## BACKING UP AND RESTORING

The following is applicable for the backup and restoring of Add-Ins:

- ▶ When securing the workspace, Editor Add-Ins are not saved.

**Note:** Installed Add-Ins for the Editor are in the  
`%ProgramData%\COPA-DATA\zenon760\EditorAddInStore` directory and can be  
 backed up manually if required.

- Add-In Packages of projects are also backed up when creating project and workspace backups and are restored again when read back in.

## HISTORY OF CHANGES

If the change history has been activated for the project, entries for the change history are generated in the project when installing and uninstalling Add-In Packages.

### 3.5.1 Activate add-ins in zenon

The use of **Add-Ins** is activated in **zenon6.ini**. The following entry must be present for this:

**[AddIns]**

**ON=1**

This entry is activated by default after the installation of zenon.

**Note:** If VBA/VSTA for zenon (on page 85) is deactivated, Add-Ins are also deactivated.

### 3.5.2 Action during installation

Add-Ins can be used in the Editor or in Runtime.

The following is applicable when installing Add-Ins:

1. New installation:  
The selected Package is saved in the AddInStore folder.
  - Folder for installation in the Editor (on page 30):  
%ProgramData%\COPA-DATA\zenon760\EditorAddInStore
  - Folder for installation in Runtime (on page 35):  
...\RT\FILES\zenon\system\AddInStore\...The Add-In ID is used as a file name and the **.scadaAddIn** file suffix is added.
2. New installation with the same ID:  
The pre-existing Add-In is replaced.
3. During installation, pre-existing files of other versions (higher or lower) are replaced.



### Information

The Add-In ID is unique for each Add-In and is defined using a .NET attribute in the **AddInInfo.cs** file.

Example (**Add-In ID** = AddInProject):

```
// Declares that this assembly is an add-in
[assembly: Addin("AddInProject", "1.0")]
```

## 3.5.3 Add-ins node in the Project Manager

In the project tree of the Project Manager, it is possible to import add-ins under the **Programming interfaces** node item. You can find details in the **Project tree context menu** (on page 29) chapter.

### Project tree context menu

Right-click in the Add-Ins symbol in the **Programming interfaces** node in the project tree to open a context menu:

Entry	Description
<b>Import Add-In...</b>	Opens the dialog to import an <b>Add-In Package</b> that is to be used in Runtime. <b>Note:</b> The package is only imported, not installed. Installation is carried out in Runtime.
<b>Editor profile</b>	Opens the drop-down list with predefined editor profiles.
<b>Help</b>	Opens online help.

### Detail view of context menu and toolbar

In the detail view of the **Add-In** node in the project tree, Add-Ins that are used in Runtime are displayed and administered by means of a tool bar and a context menu.



Entries in the context menu and meaning of the symbols from left to right:

Symbol/Entry	Description
<b>Import Add-In</b>	Opens the dialog to import an <b>Add-In Package</b> that is to be used in Runtime. <b>Note:</b> The package is only imported, not installed. Installation is carried out in Runtime.
<b>Delete</b>	Uninstalls and deletes the selected Add-In after requesting confirmation.
<b>Remove all filters</b>	Removes all filters that are currently applied in the list of Add-In.
<b>Properties</b>	Opens the <b>Properties</b> window.
<b>Help</b>	Opens online help.

### 3.5.4 Use in the zenon Editor

Add-Ins for use in the zenon Editor are imported, installed and administered in the Editor.

Add-Ins for the Editor can contain the following:

- ▶ Editor Wizard Extensions (on page 17), which are started manually via the **Extras -> Editor Wizards** menu item.
- ▶ Editor service extensions (on page 17) are started if the workspace is loaded and its start mode is set to `automatic`. It is stopped as soon as the workspace is closed.

Installed Add-Ins for the Editor are saved under:

%ProgramData%\COPA-DATA\zenon760\EditorAddInStore

#### Installing and managing add-ins for the Editor

In the zenon Editor, Add-Ins can be imported, installed and administered.

**Note:** Only Add-Ins with Editor Extensions can be imported and installed. If no Editor Extensions are found during import, a corresponding warning is shown.

#### IMPORTING AND INSTALLING ADD-INS

To import and install an Add-In:

1. Open the **Extras** menu.
2. Select the **Manage Editor Add-Ins** entry.  
The dialog to manage Add-Ins is opened.

3. Click on the **Import and Install** symbol, select the entry in the context menu or press the **Insert** key.

The dialog (on page 32) to select an Add-In is opened.

4. Select the desired Add-In.

5. Click on **Open**.

The Add-In is imported and installed.

6. Click on **Close** to close the dialog.

## UNINSTALL AND DELETE ADD-IN

To uninstall and delete an Add-In:

1. Open the menu **Tools**.

2. Select the **Manage Editor add-ins** entry.  
The dialog to manage Add-Ins is opened.

3. Select an Add-In.

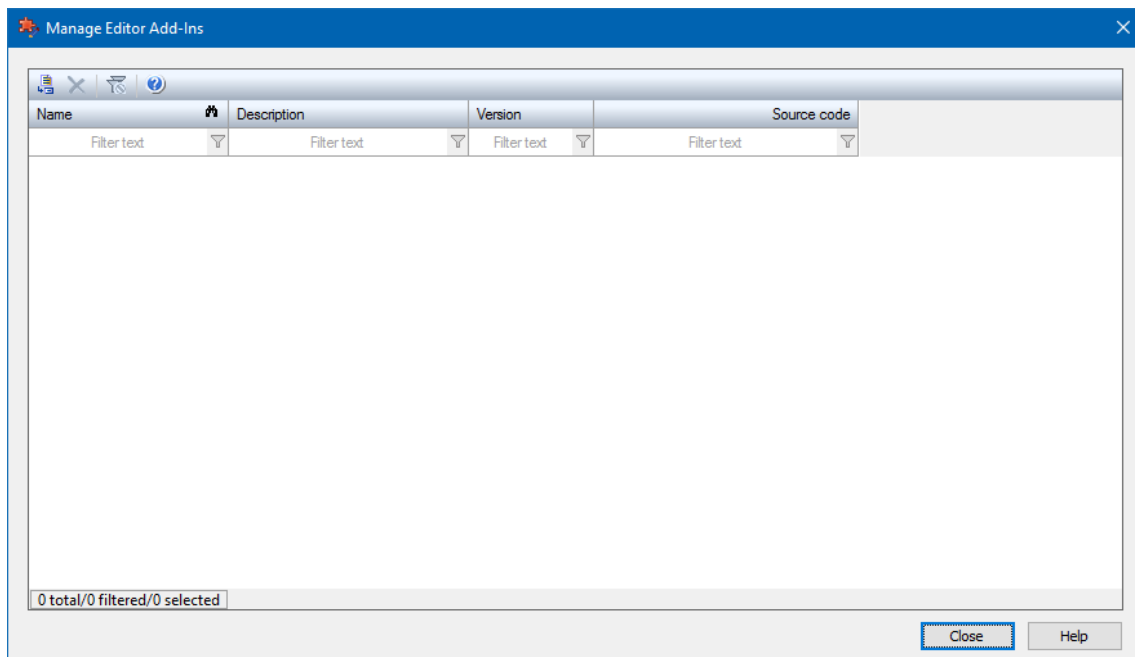
4. Click on the **Uninstall and delete** symbol, select this entry in the context menu or press the **Delete** key.

A dialog requesting confirmation is opened.

5. Confirm this when requested to do so.

The Add-In is uninstalled and deleted.

## Manage Editor Add-Ins dialog



Option	Description
<b>Toolbar</b>	Contains symbols for: <ul style="list-style-type: none"> <li>▶ Importing, installing and deleting Add-Ins</li> <li>▶ Removing the filter</li> <li>▶ Help display</li> </ul>
<b>List of the Add-Ins</b>	Shows all installed Add-Ins. Information on the following is shown: <ul style="list-style-type: none"> <li>▶ Name</li> <li>▶ Description</li> <li>▶ Version</li> <li>▶ Whether source code is included.</li> </ul> Elements can be shown filtered and sorted.
<b>Close</b>	Closes the dialog.
<b>Help</b>	Opens online help.

### TOOLBAR AND CONTEXT MENU



Meaning of the entries in the context menu and the symbols, from left to right:

Symbol	Description
<b>Import and install</b>	Opens the dialog to select an Add-In. This can be imported and installed.
<b>Uninstall and delete</b>	Uninstalls and deletes the selected Add-In after requesting confirmation.
<b>Remove all filters</b>	Removes all filters that are currently applied in the list of Add-Ins.
<b>Help</b>	Opens online help.

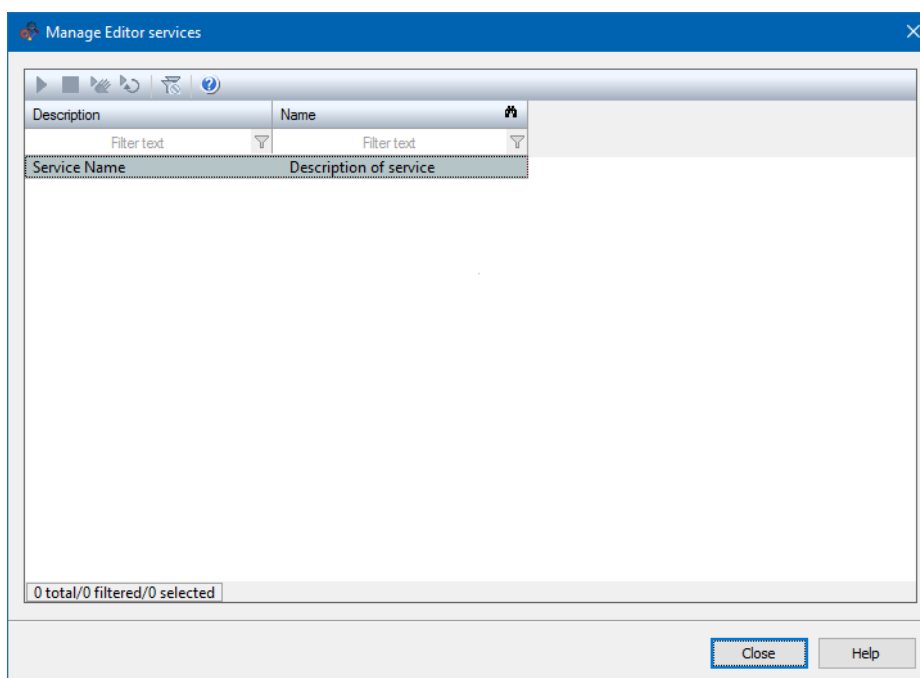
## Manage Editor services

Add-In Extensions that run in the Editor as services are managed using their own dialog.

To manage Editor services

1. Open the menu **Tools**
  2. Select the **Manage Editor services...** entry.
- The dialog to manage the services is opened.

## MANAGE EDITOR SERVICES DIALOG



Option	Description
<b>Toolbar</b>	Contains symbols for: <ul style="list-style-type: none"> <li>▶ Starting and stopping the services</li> <li>▶ Setting the start type</li> <li>▶ Removing the filter</li> <li>▶ Help display</li> </ul>
<b>List of services</b>	Shows all installed Services. Information on the following is shown: <ul style="list-style-type: none"> <li>▶ Name</li> <li>▶ Description</li> <li>▶ Start type</li> <li>▶ Status</li> </ul> Elements can be shown filtered and sorted.
<b>Close</b>	Closes the dialog.
<b>Help</b>	Opens online help.

## TOOLBAR AND CONTEXT MENU



Meaning of the entries in the context menu and the symbols, from left to right:

Symbol	Description
<b>Start service</b>	Starts the selected service.
<b>Stop service</b>	Stops the selected service.
<b>Start type manually</b>	Sets the start type for the selected service to manual.
<b>Start type automatically</b>	Sets the start type for the selected service to automatic.
<b>Remove all filters</b>	Removes all filters that are currently applied in the list of services.
<b>Help</b>	Opens online help.

## BEHAVIOR FOR START AND STOP

If the `automatic` start mode has been selected for a Workspace Service Add-In, it is loaded before the Event **OnWorkspaceStartup** when the Editor is started. When the Editor is closed, it is stopped after the event **OnWorkspaceExit**. The add-in can thus react to the zenon events.

### 3.5.5 Use in zenon Runtime

Add-ins for use in zenon Runtime are imported and administered in the Editor. They are automatically installed when Runtime starts, in order to be able to be executed in Runtime. Each zenon project manages its own add-ins.

- ▶ Add-Ins are copied to the following folder when creating the Runtime files for a project:  
... \RT\FILES\zenon\system\AddInStore\...
- ▶ The Add-Ins are installed in the following folder when starting the Runtime:  
... \RT\FILES\zenon\system\AddInCache\...

Furthermore, in doing so:

- Amended Add-In Packages are updated.  
The sync is carried out using the change date of the Add-In Package.
- Add-Ins that are no longer present are deleted.
- Project Service Extensions (on page 18) with the start mode set to automatic are instanced and started automatically.
- Project Wizard Extensions (on page 18) can be executed using the zenon **Execute Project Wizard Extension** function.



#### Hint

You can get to the Runtime folder most quickly by highlighting the project in the Editor and pressing the key combination **Ctrl+Alt+R**.

### Installing and manging add-ins for Runtime

Add-Ins for Runtime are imported into the Editor and automatically installed when Runtime is started. Import is carried out by means of the context menu or the tool bar in the detail view.

#### IMPORT ADD-IN

To import an Add-in :

1. In the Editor, open the detail view for Add-Ins in the project tree.
2. Click on the **Import Add-In** symbol, select this entry in the context menu or press the **Insert** key.  
The dialog to select an Add-In is opened.
3. Select the desired Add-In.
4. Click on **Open**.

The Add-in is imported.

#### Info:

- Only once the Runtime files have been created is the Add-In saved in the Runtime files.  
Save folder: `... \RT\FILES\zenon\system\AddInStore\...`
- When starting or reloading Runtime, it is installed, modified or updated in the following folder:  
`... \RT\FILES\zenon\system\AddInCache\...`

During modification, the sync is carried out using the time stamp of the Add-In Package.

5. Click on **Close** to close the dialog.

**Note:** Only Add-Ins with Project Extensions can be imported and installed. If no Project Extensions are found during import, a corresponding warning is shown.

## INSTALLING AND UNINSTALLING ADD-INS

Add-Ins are automatically installed or updated when Runtime is started.

Procedure:

- ▶ When Runtime is started, the Packages installed for the project are compared to those in the Editor.
- ▶ Comparison is carried out using the time stamp.
- ▶ With different time stamps or new Packages, the Package is installed. A pre-existing Package with the same name is overwritten.
- ▶ If there is no longer a Package in the Editor that is present in Runtime, it is also removed from Runtime.

## DELETE ADD-IN

To delete an Add-In:

1. In the Editor, open the detail view for Add-Ins in the project tree.
2. Select an Add-In.
3. Click on the **Delete** symbol or press the `Delete` key

A dialog requesting confirmation is opened.

4. Confirm this when requested to do so.

The Add-In is deleted.

After the Runtime files have been compiled the next time, it is uninstalled when Runtime is reloaded or restarted.

## 3.6 Switch/conversion from VSTA

### EDITOR

To convert existing VSTA functionality from the Editor, proceed as follows.

#### FUNCTIONALITY FROM WIZARDS

Functionality that has been implemented using a VSTA wizard can only be implemented by means of Editor Wizard Extensions (on page 17).

Copy your existing code to an Editor Wizard Extension and make changes to the code if necessary.

Use the **Editor wizards...** dialog to start the wizard.

#### FUNCTIONALITY FROM VSTA EVENTS

Functionality that is executed on an event-triggered basis (such as via **.OnElementCreated**, **.OnPreBuild**) can now be implemented using Editor Service Extensions (on page 17).

To do this, copy the code to initialize the event handler to the **Start** method of the Editor Service Extension. You copy the code for release into the **Stop** method.

**Note:** For automatic start, the `DefaultStartMode=DefaultStartupModes.Auto` attribute must be set in the **Extension**. The start mode can also be changed with the **Manage Editor services** (on page 33) dialog.

#### FUNCTIONALITY FROM VSTA MACROS.

Functionality that is executed using the **Execute VBA/VSTA macro** combobox can now be implemented using Editor Wizard Extensions (on page 17).

To do this, copy the code from your existing VSTA macro in to the **Run** method of the Editor Wizard Extension and make changes to the code if necessary.

Use the **Editor wizards...** dialog to execute the code.

### RUNTIME

To convert existing VSTA functionality from Runtime, proceed as follows.

## FUNCTIONALITY FROM VSTA MACROS.

Functionality that was executed by means of the **Execute VSTA macro** function can now be implemented by means of Project Wizard Extensions (on page 18).

To do this, copy the code from your existing VSTA macro in to the **Run** method of the Project Wizard Extension and make changes to the code if necessary.

To start the Project Wizard Extension, use the zenon **Execute Project Wizard Extension** function.

## FUNCTIONALITY FROM VSTA EVENTS

Functionality that is executed on an event-triggered basis (such as via **.DynPictures().Open**, **.Alarm().AlarmComes**) can now be implemented using Project Service Extensions (on page 18).

To do this, copy the code to initialize the event handler to the **Start** method of the Project Service Extension. You copy the code for release into the **Stop** method.

**Note:** For automatic start, the `DefaultStartMode=DefaultStartupModes.Auto` attribute must be set in the **Extension**.

# 4. Macro list

You can use VBA and VSTA in order to extend zenon functionality. The usage of macros with zenon is described.

## CONTEXT MENU

Menu item	Action
<b>Open VBA Editor</b>	Opens the VBA Editor.
<b>Export all VBE</b>	Opens the dialog for selecting the storage directory for the VBE export.
<b>Import VBE</b>	Opens the dialog for selecting the VBE import file.
<b>Editor profiles</b>	Opens the drop-down list with predefined editor profiles.
<b>Help</b>	Opens online help.



### Information

*If VBA macros are changed in the Editor,*

- ▶ the Runtime files are compiled and transferred to the Runtime
- ▶ the Runtime is reloaded
- ▶ VSTA elements are also reloaded even if no changes were made in VSTA

VBA starts the same development environment for **Workspace** and **Project**.  
To open the VBA Editor:

1. In the zenon Editor, navigate to the **Programming interface** node.
2. Expand the view of this node by clicking on [+].  
The view of the node is expanded.
3. Right-click on **Macro list**
4. Select the entry **Open VBA Editor** in the context menu.

*Alternative: press the short cut **Ctrl+F11***

## 4.1 VBA toolbar and context menu detail view

### TOOLBAR



Menu item	Action
<b>New VBA macro</b>	Creates a new macro and opens the macro Editor.
<b>Open VBA Editor</b>	Opens the VBA Editor.
<b>Save</b>	Saves macros.
<b>Delete</b>	Deletes the selected element.
<b>Export all VBE</b>	Opens the dialog for selecting the storage directory for the VBE export.
<b>Import VBE</b>	Opens the dialog for selecting the VBE import file.
<b>Rename</b>	Makes it possible to rename the selected macro.
<b>Help</b>	Opens online help.

#### CONTEXT MENU MODULE

Menu item	Action
<b>Open VBA Editor</b>	Opens the VBA Editor.
<b>Save</b>	Saves macros.
<b>Export all VBE</b>	Opens the dialog for selecting the storage directory for the VBE export.
<b>Import VBE</b>	Opens the dialog for selecting the VBE import file.
<b>Help</b>	Opens online help.

#### CONTEXT MENU MODULE

Menu item	Action
<b>New VBA macro</b>	Creates a new macro and opens the VBA Editor.
<b>Help</b>	Opens online help.

#### CONTEXT MENU MACRO

Menu item	Action
<b>Edit</b>	Opens macro in the Editor for editing. Alternative: Enter button or double click.
<b>Delete</b>	Deletes macro. Alternative: Del key
<b>Rename</b>	Opens list elements for editing. Alternative: F2 key.
<b>Help</b>	Opens online help.

## TOOLBAR EDITOR

Macros that were created with VBA can be administrated via toolbar-item **Macro list**.



Symbol (from left to right)	Function
<b>Reload list of VBA/VSTA macros</b>	Loads all <b>Public Sub Name ()</b> macros that are included in <b>myWorkspace</b> and in modules to the drop-down list of the toolbar.
<b>Search Macro</b>	Search for macros via combobox input field or selection from drop-down list. The drop-down list is adjusted to the widest element when opened.
<b>Drop-down list Macros</b>	Contains all loaded macros for selection.
<b>Execute selected macro</b>	Executes the macro selected in the drop-down list.
<b>execute allocated macro #&lt;x&gt;</b>	Executes the macro allocated with the symbol.
<b>Allocate macros</b>	Opens the allocation dialog for macros. Up to 5 macros can be allocated with the symbols 1 to 5.
<b>VBA</b>	Filters for VBA-macros. Only VBA-macros are displayed.
<b>VSTA</b>	Filters for VSTA-macros. Only VSTA-macros are displayed.
<b>ALL</b>	Cancels the current filter and all macros are displayed.
<b>AZ</b>	Sorts macros in ascending order from 0 - 9 and A - Z.
<b>ZA</b>	Sorts macros in descending order from Z - A and 9 - 0.
<b>Options for symbol bar</b>	<p>Clicking on the arrow opens the submenu:</p> <p>Active: Toolbar is displayed.</p> <p>If the toolbar is not displayed, it can be activated using the <b>Options -&gt; Toolbar</b> menu.</p> <p><b>Note:</b> For free placed toolbar (undocked from the Editor) options are not displayed. The toolbar can be closed by clicking on button X.</p>



### Information

If the macro assignment dialog does not list all macros from **myWorkspace**, execute the function **Reload list of VBA macros** in the toolbar.

## 4.2 VBA on 64-bit systems

zenon has supported 64-bit operating systems since version 7.10. VBA was thus converted to VBA version 7.1. Therefore VBA is also available in zenon 64-bit. If, in the VBA code, Windows API or other

imported DLL functions are accessed, these calls must be adapted to 64-bit. In general, the following applies: A VBA file created for a 32-bit version cannot be used without changes in a 64-bit version.

There are some defines/functions available in VBA in order to write 32-bit and 64-bit compatible code. For example:

```
#if Win64 then
    Declare PtrSafe Function MyMathFunc Lib "User32" (ByVal N As LongLong) As LongLong
#else
    Declare Function MyMathFunc Lib "User32" (ByVal N As Long) As Long
#endif if
#if VBA7 then
    Declare PtrSafe Sub MessageBeep Lib "User32" (ByVal N AS Long)
#else
    Declare Sub MessageBeep Lib "User32" (ByVal N AS Long)
#endif if
```

You can also obtain some useful notes on the porting of VBA 32-bit code to VBA 64-bit from Microsoft:

- ▶ Microsoft Office 2010, notes on porting:  
<http://msdn.microsoft.com/en-us/library/ee691831.aspx>  
<http://msdn.microsoft.com/en-us/library/ee691831.aspx>
- ▶ 32-bit and 64-bit declares for API calls: <http://www.jkp-ads.com/articles/apideclarations.as>  
<http://www.jkp-ads.com/articles/apideclarations.as>

## 4.3 Basics

Describes the basics of the programming language VBA - Visual Basic for Applications

### 4.3.1 Object PROPERTIES

An object property is a certain attribute of the object. In case of a variable object this e.g. can be the value, the name or the identification. In case of a circle the position or the color of the circle in the screen. Each object has at least one property (usually more), each property has a certain value. While the **property name** is a text, the **property value** is a value between 0 and e.g. 1000.

The special thing with properties is, that with changing the property value in a VBA program you can change the behavior or the appearance of the object. If you e.g. change the **property value** of a variable object, the currently selected variable gets this new value. You cannot change the value of each property. The **property count** of the variable object cannot be changed, because it represents the number of created variables. You cannot add variables by changing the value of Count. So some properties are read only, i.e. their values only can be read.

### 4.3.2 Object METHODS

Beside the properties each object can have methods. A method is not an attribute but a request to the object to do something. So a form has the method `Show`. What does it do? It requests the form to appear on the screen. Accordingly the form disappears when using the method `Unload`.

The advantage of methods is, that the programmer does not have to know anything about the structure of the object and most of all has no opportunity to change the internal data of the object.

Executing the method `Show` or `Unload` works as follows:

```
frmSollwert.Show or Unload frmSollwert
```

If you want to open another form, the method stays the same, only the name of the form (object name) changes.

```
frmChange.Show or Unload frmChange
```

So one and the same method can be used for different object types. But not every object must have methods.

### 4.3.3 Object EVENTS

In 90% of working with objects you will use properties and methods, but there is a third kind of attributes objects can have: Events. Some objects of the control system object hierarchy can react on events. Events take place during the work with zenon on their own.



#### Example

*Whenever a screen is opened, an open event is triggered in the according screen object. As a programmer you can add commands to the event procedure (procedure to be executed, when the event happens), which define, what should happen in this case. One example for this is changing a variable. You can create an event, which reacts on value changes of a variable.*

### 4.3.4 VBA object structure in zenon

Basically there is a object list and objects again and again in the project structure.

Example:

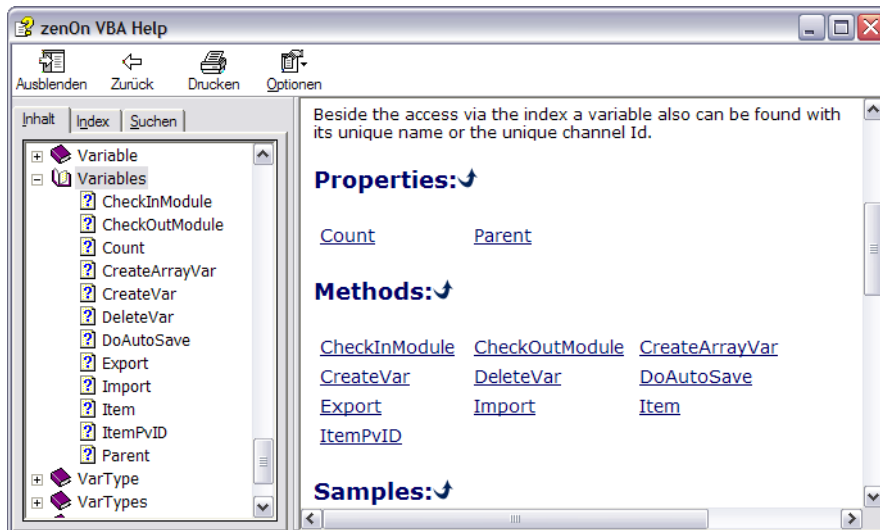
Projects – Project

Variables – Variable

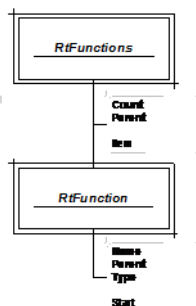
Elements – Element

You can find more about the object model:

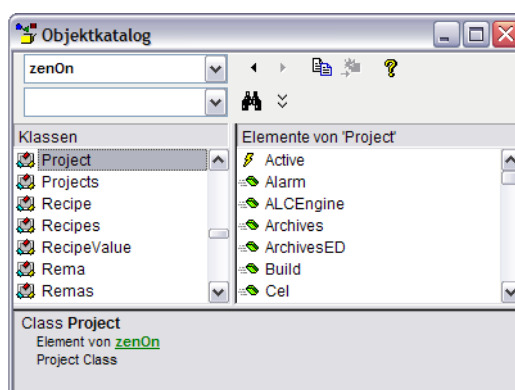
- ▶ in the VBA help



- ▶ in the graphical overview which you can obtain from COPA-DATA complete as printed overview.

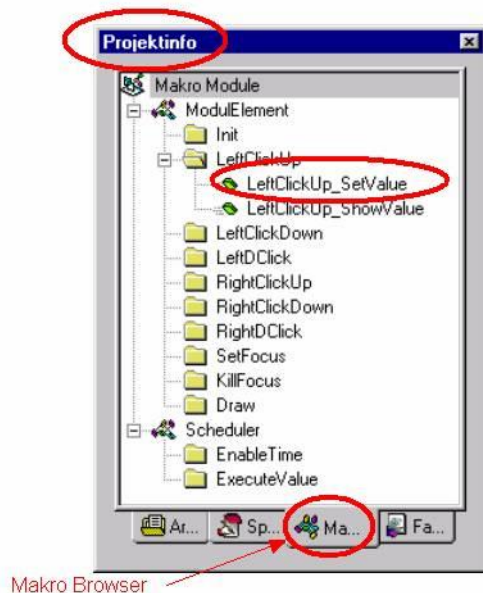


- ▶ in the VBA object browser



### 4.3.5 How to use VBA macros

In order to create a new macro in the window **Project info** on the property page **Macro Browser** select a desired event, when the new macro should be executed.



Clicking on this event with the right mouse button opens a menu.

Select the menu entry "New macro..." Thus zenon generates a procedure:

```
Public Sub LeftClickUp_Sollwert(obElem As Element)
End Sub
```

If a macro already exists, it can be edited, deleted or renamed by clicking it with the right mouse button.



#### Attention

*If you select menu item **Rename macro**, take care that you do not change the name of the event e.g. LeftClickUp\_..., - of the current name. Otherwise renaming will not be executed. Additionally you have to change the name of the sub program to be executed in the VBA Editor by hand, if you rename a macro.*

After you have filled the procedure generated by zenon with the source code to be executed, the created macro has to be linked to an element.

Doubleclicking the element opens the property dialog of the element.

On the property page Events the macro is linked to the element.

Clicking the element with the left mouse button executed the LeftClickDown event of the element and the linked macro.

## Inserting existing macros

In order to insert existing macros into another project do the following:

1. In the VBA Editor export all needed forms and modules and import them in the other project.
2. Event dependent macros, in `ModuleElement.bas`, are not displayed in the macro browser at the moment. So this macros have to be created in the macro browser.

The easiest way is to use the name of the existing macro.

e.g.:

`LeftClickUp_DateSet2`

`LeftClickUp_DateSet4`

`LeftClickUp_TimeSet`

`Draw_Date2`

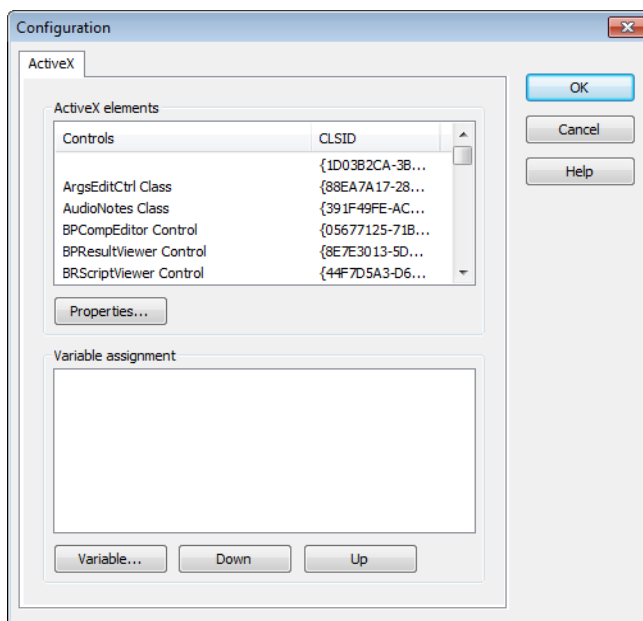
`Draw_Date4`

`Draw_Time`

3. On creating the macros zenon generates procedures with the same name as the existing macros. You have to delete these generated procedures.
4. Connect the macros as usual with a dynamic element.

### 4.3.6 How to insert an ActiveX element in zenon?

An ActiveX element is drawn into the screen like any other dynamic element; a dialog opens, where you must select an ActiveX element.



- ▶ After you have selected the element from the list, you can link variables to it. For this click the button **Variable** and select a variable or create a new one.
- ▶ In the next step we give the ActiveX element an object name, so that we can access it in VBA.
- ▶ In our example we give it the object name Slide6\_DW18, because it is an ActiveX element Slider linked to the variable Doubleword18.
- ▶ Now the Slider element has to be activated and edited in the VBA Editor.
- ▶ For this we create a new macro as described in chapter "How to use VBA macros? (on page 46)".

The macro Init\_Slider passes the element to be initialized to a sub program in the control system object **thisProject**, whereby the allocation to the current project is defined.

```
Public Sub Init_Slider(obElem As Element)
thisProject . Init _ Slider obElem
End Sub
```

Just like in the macro Init\_Slider also Draw\_SliderValue passes the element to the control system object thisProject.

```
Public Sub Draw_SliderValue (obElem As Element, ByVal hdc As OLE _ HANDLE )
thisProject.Draw_Slider obElem
obElem.Draw hdc
End Sub
```

The code below is added in the control system object this Project.

```
Public Declarations
Public WithEvents obSlider As Slider
Public obSliderPV As Variable
Public Sub Init_Slider (obElem As Element)
Set obSlider = obElem.ActiveX
'ActiveX exists
If obSlider Is Nothing Then
Exit Sub
End If
Set obSliderPV = obElem . ItemVariable(0)
'variable exists
If obSliderPV Is Nothing Then
Exit Sub
End If
obSlider.Max = obSliderPV.RangeMax
obSlider.Min = obSliderPV.RangeMin
obSlider.TickFrequency = 1000
obSlider.LargeChange = 25
obSlider.SmallChange = 1
obSlider.Value = obSliderPV.Value
End Sub
```

```

Public Sub Draw _ Slider ( obElem As Element )
Dim vVar As Variant
Dim obDynPic As DynPicture

Set obSliderPV = obElem.ItemVariable ( 0 )
Set obDynPic = thisProject.DynPictures. Item (BILD_1)
'variable exists
If obSliderPV Is Nothing Then
Exit Sub
End If

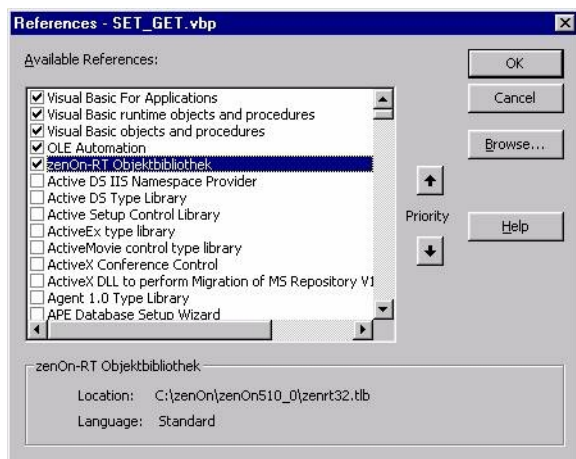
```

#### 4.3.7 Access from an external program

In order to access zenon data from an external program such as e.g. Visual Basic the COM interface is used. This COM interface is also used by VBA. So there are only a few small differences, that should be cared of.

### Visual Basic 6

In order to be able to access the COM interface it has to be implemented:



With this type library you can access the application object of zenon (the Runtime).

As here there is no thisProject object, it has to be created to get access to the data.

```

Dim obProject As zenon.Project
Set obProject = zenon.Application.Projects.Item(PROJEKTNAME)

```

If the VB project should work with all zenon projects - should be project name independent - it can be defined in the following way:

```

Set obProject = zenon.Application.Projects.Item(0)

```

After the project object (thisProject) has been created, e.g. the variables can be accessed for reading and writing.

Read:

```
Value = obProject.Variables.Item(Variablename).Value
```

Write:

```
obProject.Variables.Item(Variablename).Value = Value
```

### 4.3.8 Functionality of online variables

You can imagine a VBA OnlineVariable as a container; this container contains control system variables, which have to be added. If the value of one of the variables of the container changes, this is indicated with an event.

#### Functionality of the event:

If the container is activated ([Container.Define](#)), all variables in the container are forced once, so that the current value of the variables are known. So the procedure [Container\\_VariableChange](#) is executed for each variable in the container. As soon as all variables then have been initialized, this event always occurs, if one of the variables of the container changes its value.

So it is avoided, that a value is read, which is not the current value of the variable.

#### Define and create container

Definition:

```
Public WithEvents Container As OnlineVariable
```

With this line of code the container is defined.

Creating:

```
Set Container = thisProject . OnlineVariables . CreateOnlineVariables ( Container name )
```

#### Put variables in the container

```
Container . Add Variablename1
Container . Add Variablename2
Container . Add Variablename3
Container . Add Variablename4
...
```

Repeat this line, until all needed variables are added to the container.

## Create event

```
Private Sub Container_VariableChange(ByVal obVar As zenon.IVariable)
...
End Sub
```

This event is automatically created, when the container is selected in the left combobox at the top of the VBA Editor. The procedure above then is added to the source code. With **obVar** the variable with the changed value is passed on. When this event occurs, e.g. the current value of the variable (**ob-Var.Value**) can be read. Refer to the object hierarchy in the VBA documentation to see the properties and values of variables, which can be used.

## Activate event

```
Container.Define
```

This command line activates the monitoring of the variables in the container. After executing the command **Define**, the container is active.

## Switching off the event

```
Container.Undefine
```

With this command the surveillance in the container is switched off. The event (VariableChange) is no longer carried out.

## Remove on closing

In order not to leave anything in the memory on closing the Runtime, the container has to be removed at the latest on closing the Runtime.

```
thisProject . OnlineVariables . DeleteOnlineVariables ( Container_name )
```

Not before the container is deleted can another container with the same name be created.

#### 4.3.9 List of status bits

Bit number	Short term	Long name	zenon Logic label
0	M1	User status 1; for Command Processing: Action type "Block"; Service Tracking (Main.chm::/IEC850.chm::/117281.htm) of the IEC 850 driver	_VSB_ST_M1
1	M2	User status 2	_VSB_ST_M2
2	M3	User status 3	_VSB_ST_M3
3	M4	User status 4	_VSB_ST_M4
4	M5	User status 5	_VSB_ST_M5
5	M6	User status 6	_VSB_ST_M6
6	M7	User status 7	_VSB_ST_M7
7	M8	User status 8	_VSB_ST_M8
8	NET_SEL	Select in the network	_VSB_SELEC
9	REVISION	Revision	_VSB_REV
10	PROGRESS	In operation	_VSB_DIRECT
11	TIMEOUT	Command "Timeout exceeded" (command runtime exceeded)	_VSB_RTE
12	MAN_VAL	Manual value	_VSB_MVALUE
13	M14	User status 14	_VSB_ST_14
14	M15	User status 15	_VSB_ST_15
15	M16	User status 16	_VSB_ST_16
16	GI	General query	_VSB_GR
17	SPONT	Spontaneous	_VSB_SPONT
18	INVALID	Invalid	_VSB_I_BIT
19	T_STD_E	External standard time (standard time)  <b>Caution:</b> up to version 7.50, this was the status bit T_CHG_A	_VSB_SUWI
20	OFF	Switched off	_VSB_N_UPD
21	T_EXTERN	Real time - external time stamp	_VSB_RT_E
22	T_INTERN	Internal time stamp	_VSB_RT_I
23	N_SORTAB	Not sortable	_VSB_NSORT
24	FM_TR	Error message transformer value	_VSB_DM_TR

25	RM_TR	Working message transformer value	_VSB_RM_TR
26	INFO	Information for the variable	_VSB_INFO
27	ALT_VAL	Alternate value	_VSB_AVALUE
28	RES28	Reserved for internal use (alarm flashing)	_VSB_RES28
29	N_UPDATE	Not updated (zenon network)	_VSB_ACTUAL
30	T_STD	Internal standard time	_VSB_WINTER
31	RES31	Reserved for internal use (alarm flashing)	_VSB_RES31
32	COT0	Cause of transmission bit 1	_VSB_TCB0
33	COT1	Cause of transmission bit 2	_VSB_TCB1
34	COT2	Cause of transmission bit 3	_VSB_TCB2
35	COT3	Cause of transmission bit 4	_VSB_TCB3
36	COT4	Cause of transmission bit 5	_VSB_TCB4
37	COT5	Cause of transmission bit 6	_VSB_TCB5
38	N_CONF	Negative confirmation of command by device (IEC 60870 [P/N])	_VSB_PN_BIT
39	TEST	Test bit (IEC870 [T])	_VSB_T_BIT
40	WR_ACK	Writing acknowledged	_VSB_WR_ACK
41	WR_SUC	Writing successful	_VSB_WR_SUC
42	NORM	Normal status	_VSB_NORM
43	N_NORM	Deviation normal status	_VSB_ABNORM
44	BL_870	IEC 60870 Status: blocked	_VSB_BL_BIT
45	SB_870	IEC 60870 Status: substituted	_VSB_SP_BIT
46	NT_870	IEC 60870 Status: not topical	_VSB_NT_BIT
47	OV_870	IEC 60870 Status: overflow	_VSB_OV_BIT
48	SE_870	IEC 60870 Status: select	_VSB_SE_BIT
49	T_INVAL	External time stamp invalid	not defined
50	CB_TRIP	Breaker tripping detected	not defined
51	CB_TR_I	Breaker tripping detection inactive	not defined
52	OR_DRV	Value out of the valid range (IEC 61850)	not defined
53	T_UNSYNC	ClockNotSynchronized (IEC 61850)	not defined

54	PR_NR	Not recorded in the Process Recorder	not defined
55	RES55	reserved	not defined
56	RES56	reserved	not defined
57	RES57	reserved	not defined
58	RES58	reserved	not defined
59	RES59	reserved	not defined
60	RES60	reserved	not defined
61	RES61	reserved	not defined
62	RES62	reserved	not defined
63	RES63	reserved	not defined



### Information

*In formulas all status bits are available. For other use the availability can be limited.*

*You can read details on status processing in the Status processing chapter.*

## 4.3.10 Lasso for selecting dynamic elements in the Runtime

Dynamic elements which are linked with a variable or function can be pre-selected with the lasso in the Runtime and can be used for events.

With method **SelElements** the user can identify selected dynamic elements as selected in the Runtime. These **DynPicture.SelElements** can then be used for events such as drag&drop.

### SELECTION VIA LASSO

To select elements with the lasso in the Runtime, you must:

- ▶ activate property **Runtime lasso** in the project settings
- ▶ activate property **Runtime/selectable with lasso** in the property of the dynamic element
- ▶ property **Move Frame via mouse** must be activated

There are several methods for selecting elements available in Runtime, depending on the settings for touch operation:

- ▶ Touch operation deactivated or Windows 7 (**Recognition** property deactivated or Windows 7):
  - Left mouse click + movement: New selection is created.

- Left mouse click + Ctrl key + movement: Selection is expanded.
- ▶ Native Windows 8 touch operation active (**Recognition** property on Windows 8):
  - Left mouse click + movement: Screen is moved
  - Left mouse click + Ctrl key + movement: Screen is moved.
  - Left mouse click + Shift key + movement: New selection is created.
  - Left mouse click + Ctrl key + Shift key + movement: Selection is expanded.

Rules:

- ▶ Only elements that are visible can be selected.
- ▶ If a button is selected and it is clicked on, the respective function is executed.
- ▶ If a button is clicked on with a mouse button and the mouse is moved before the mouse button is released, the respective function is not executed.
- ▶ Cancel selection: Spanning a lasso which does not contain elements.

## 4.4 Macros in the Editor

Macros can be carried out with the help of a configurable Toolbar (on page 56) in the Editor. For this macros are linked (on page 57) with buttons in toolbar **VBA**.

In addition macros can be run manually using the VBA Editor.

With the help of Wizards repeating engineering tasks can be run or whole projects can be created with the click on a button. As examples a few wizards are already included in the shipped version of zenon. These wizards can be enhanced and completed at will. They help when creating a project, at the import and export, at creating variables and so on. You can find details in chapter Wizards.

### EDITOR EVENTS

Editor events are part of the VBA workspace and make it possible to react to Events in the workspace programming, e.g. for wizards or Remote Transport. For example:

- ▶ OnElementCreated
- ▶ OnElementDeleted
- ▶ OnElementDoubleClicked
- ▶ OnObjectCreated
- ▶ ...

All Events and information about them can be found in the help in chapter Object Model.

#### 4.4.1 Tool bar macro list

Macros that were created with VBA can be administrated via toolbar-item **Macro list**.



Symbol (from left to right)	Function
<b>Reload list of VBA/VSTA macros</b>	Loads all <b>Public Sub Name ()</b> macros that are included in <b>myWorkspace</b> and in modules to the drop-down list of the toolbar.
<b>Search Macro</b>	Search for macros via combobox input field or selection from drop-down list. The drop-down list is adjusted to the widest element when opened.
<b>Drop-down list Macros</b>	Contains all loaded macros for selection.
<b>Execute selected macro</b>	Executes the macro selected in the drop-down list.
<b>execute allocated macro #&lt;x&gt;</b>	Executes the macro allocated with the symbol.
<b>Allocate macros</b>	Opens the allocation dialog for macros. Up to 5 macros can be allocated with the symbols 1 to 5.
<b>VBA</b>	Filters for VBA-macros. Only VBA-macros are displayed.
<b>VSTA</b>	Filters for VSTA-macros. Only VSTA-macros are displayed.
<b>ALL</b>	Cancels the current filter and all macros are displayed.
<b>AZ</b>	Sorts macros in ascending order from 0 - 9 and A - Z.
<b>ZA</b>	Sorts macros in descending order from Z - A and 9 - 0.
<b>Options for symbol bar</b>	<p>Clicking on the arrow opens the submenu:</p> <p>Active: Toolbar is displayed.</p> <p>If the toolbar is not displayed, it can be activated using the <b>Options -&gt; Toolbar</b> menu.</p> <p><b>Note:</b> For free placed toolbar (undocked from the Editor) options are not displayed. The toolbar can be closed by clicking on button X.</p>

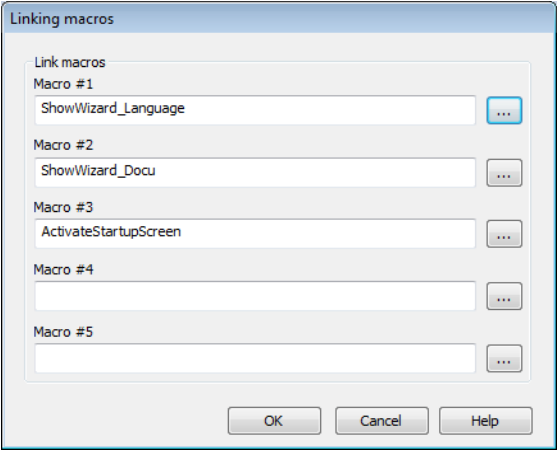


#### Information

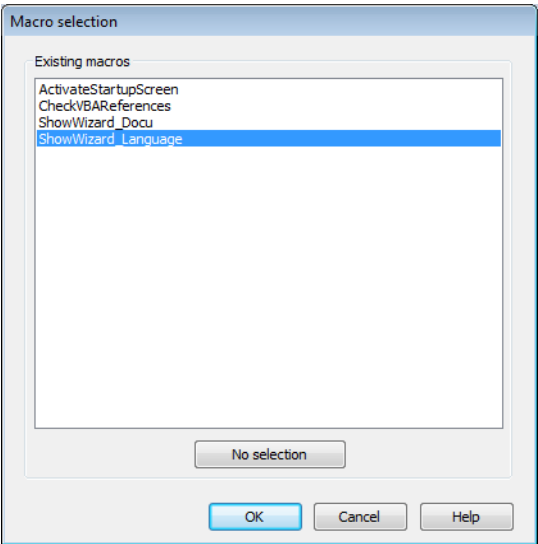
If the macro assignment dialog does not list all macros from **myWorkspace**, execute the function **Reload list of VBA macros** in the toolbar.

4.4.2    Linking macros

Macros can be called via a button in the toolbar. A maximum of five macros can be linked this way. Via button **Assign macros** the dialog for assigning macros is opened.



Parameters	Description
<b>Macro #</b>	Macro number matches the number of the button in the toolbar.  A click on button ... opens the dialog for selecting the macro.
<b>OK</b>	Creates links to the buttons and closes the dialog.
<b>Cancel</b>	Discards all changes and closes the dialog.
<b>Help</b>	Opens online help.



Parameters	Description
<b>Existing selection</b>	List of macros which can be linked.
<b>No selection</b>	Deletes existing assignment for the button.
<b>OK</b>	Assigns the selected macro to the button.
<b>Cancel</b>	Discards all changes and closes the dialog.
<b>Help</b>	Opens online help.

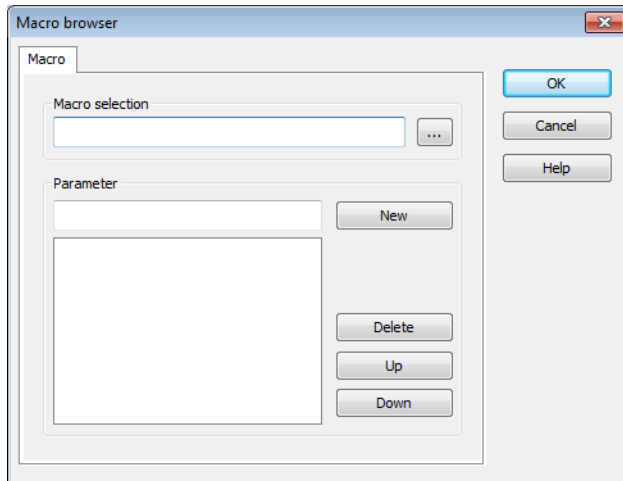
## 4.5 Functions in zenon

In dialog **Function selection** you can find the following functions under element **VBA**.

Function	Description
Open PCE editor	Opens the editor of the optional module Process Control Engine (PCE).
Open VBA Editor	Opens the VBA editor
Execute VBA Macro	Executes a selected VBA macro. <b>Attention:</b> The VBA Event <b>project inactive</b> is carried out by script <b>AUTO_END_XXX</b> . Therefore zenon function <b>Execute VBA</b> macro is no longer executed in scripts as VBA is not running at this time.
Show VBA macro dialog	Opens the VBA macro dialog.

### 4.5.1 Execute VBA macro

If you select function Execute VBA macro, the following dialog is displayed.



These settings are available.

Parameters	Description
<b>Macro selection</b>	Opens the dialog for selecting the macros (see also Macro selection (on page 60)) Hint: Only lists VBA macros that match the number of parameters defined at the function Parameter (below).
<b>Parameters</b>	Enter the desired value (string) for a parameter.
<b>New</b>	Click on this button in order to apply the value at parameter in the list of available parameter.
<b>Delete</b>	Click on this button in order to delete the selected entry from the list of available parameter. You can always only delete one entry at a time.
<b>Up</b>	Click on this button in order to move the selected entry up one place. In the parameter order the entry is moved one place to the front.
<b>Down</b>	Click on this button in order to move the selected entry down one place. In the parameter order the entry is moved one place to the back.

It is possible to add strings to macros which were created with parameters. These strings are transferred in the Runtime as individual parameters when the macro is carried out.

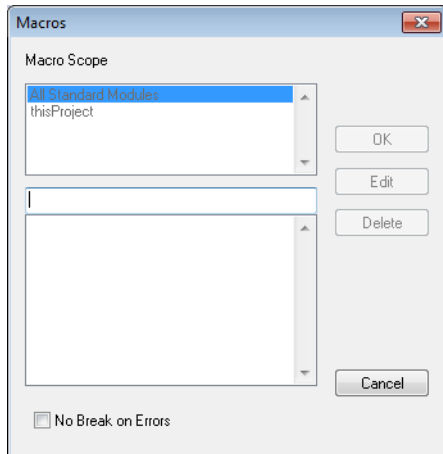


#### Information

*You must make sure that the number of parameters of the linked macro matches the number of the created parameters.*

## Macro selection

After clicking button ..., the following dialog is displayed.



Select the desired macro from the available macros and then click **OK**.

## 4.6 Developing wizard in VBA

Since version 6 it is possible to automate engineering projects with wizards. So frequently recurring tasks can be sourced out to a wizard which executes the desired actions, e.g. creating a project, creating frames and screens in a pre-defined standard.

Another field of application for wizards are automated changes in existing projects, e.g. changing properties of dynamic elements in all screens of an existing project.

The basis for the wizards is Microsoft Visual Basic for Applications (VBA) and the object model of zenon.

At the moment the following wizards are available:

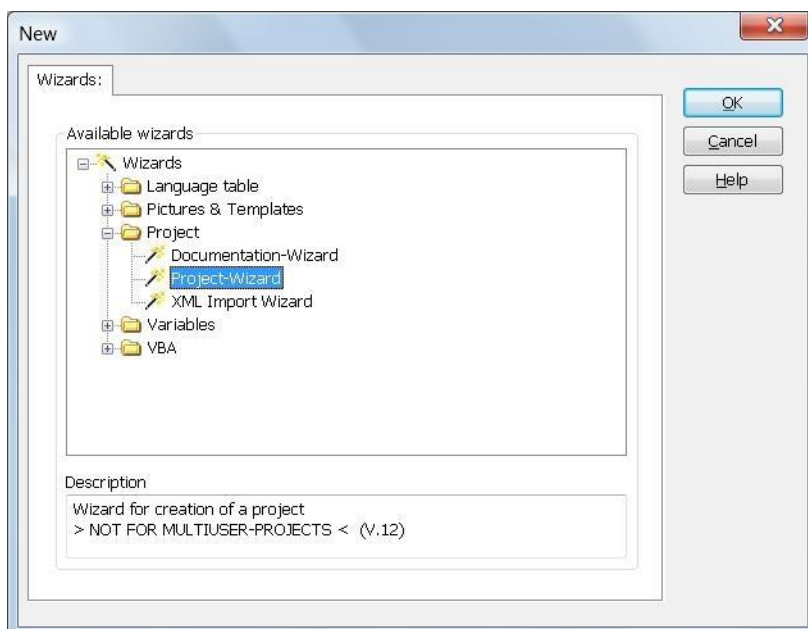
- ▶ Project Wizard
- ▶ Import Wizard
- ▶ World View Wizard
- ▶ Find VBA-Text Wizard
- ▶ Wizard for keyword creation
- ▶ Wizard for keyword translation
- ▶ Wizard for creating variables

The wizards are available as VBA source code files on the installation medium. New wizards can be implemented with the VBA environment.

### 4.6.1 Using a wizard

The menu entry **Editor Wizards** in the menu **Tools** opens the wizard selection. In this dialog, all available wizards are shown according to their category.

If wizards do not contain a category, a **Not linked** entry is created automatically. In this category all not linked wizards are displayed.



By selecting a wizard and pressing the button **OK** the selected wizard is executed.



#### Information

*Wizards do not support multi-user projects.*

### 4.6.2 Structure of a wizard

A wizard is a UserForm stored in the application specific node of the application. Usually the UserForm consists of a multi-page element displaying the single steps of the wizard.

With a button **Next** the next page of the multi-page element is displayed. All entries have to be stored temporarily - the creation of objects, e.g. frames, screens, ... has to be done with **Finishing** the wizards.



### Information

*UserForms to be used as wizards have to contain some public methods, which provide the control system with information about the wizard. If this routines are missing in a UserForm, it is not treated as a wizard.*

## 4.6.3 Integration in VBA

The wizards are stored in the application specific node **ZWorkspace**. This object represents the currently loaded workspace in the Editor and is only available in the zenon Editor.

All objects in this VBA project can access the current workspace with the object **MyWorkspace**. It is always linked to the currently active project, which can be accessed with the property **ActiveDocument**.

The contents of the object ZWorkspace are stored in the file `ZenWorkspace.vba`. It is copied to the installation directory with the first installation of version 6. This file is not overwritten by later updates. You will find more information on updating wizards at the end of this tutorial.

## 4.6.4 Developing a wizard

This tutorial develops a wizard creating variables for a defined driver.

Start the VBA environment from the zenon Editor and change to folder **ZWorkspace/Forms**. This file contains the basics for developing a wizard. Change the name of the UserForm.

If the folder mentioned above is not available, you can import it via Import the **Import file** command.



### Information

*For developing a wizard knowledge about the object model of zenon and VBA are required. These topics are not part of this tutorial.*

Variable creation wizard

 Wizard

zenon

Welcome

Driver

Variable

### Select the driver and enter the variable name

Please choose from the combo-box the driver for which the variables should be created and enter a name for the variables. The names of the variables will be enhanced by an incrementing number.



Please select a driver:

Interne Variablen

Please enter the name for the variables:

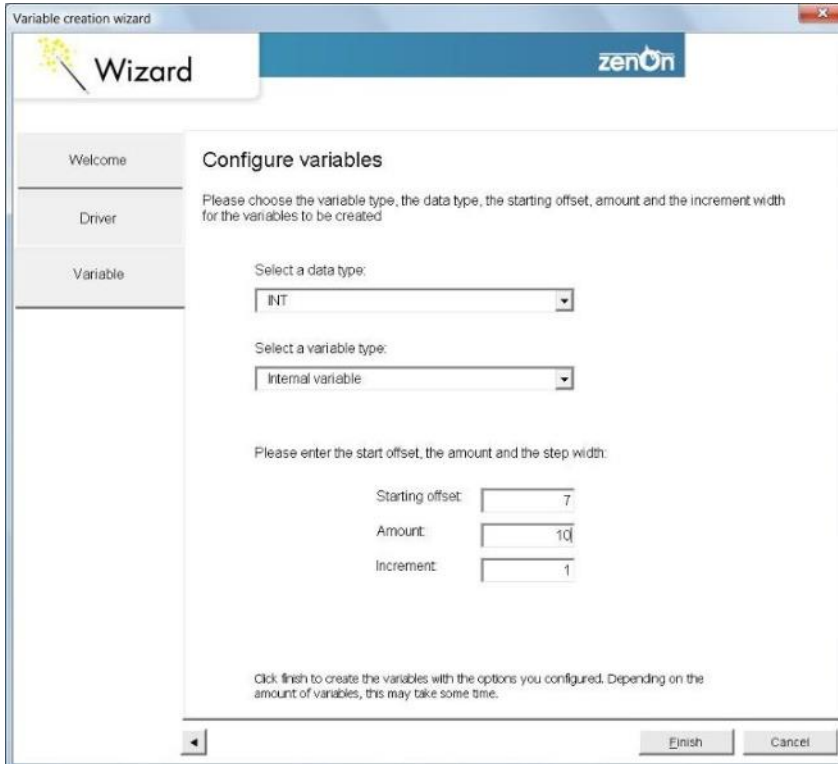
TestVar

Click the right arrow to continue

Finish

Cancel



Create the surface displayed above. Then switch to the code module of the UserForm and scroll to the end of the file. There you will find the following methods.

- ▶ **Public Function GetWizardName () As String**  
Returns the unique name of the wizard. Change the contents to Wizard for creating variables
- ▶ **Public Function GetWizardInfo () As String**  
Returns a short description displayed in the wizard selection. Change the contents to Wizard for creating variables to a selected driver
- ▶ **Public Function GetWizardCategory () As String**  
Returns the category of the wizard. In the wizard selection the wizards are displayed in a tree structure of the categories. Change the contents to Variables.
- ▶ **Public Function IsZenOnWizard () As Boolean**  
Displaying the wizard in the wizard selection. If this method returns False, the wizard is not displayed, e.g. because it is not yet finished. Change the return type to True.

These methods provide the information about the wizard, which is requested by the control system. Keep in mind that the wizard is only displayed in the wizard selection if the method **IsZenOnWizard** returns **True**.

Switch to the event **Initialize** of the UserForm and change the contents of the string array **m\_strCaption**. As our wizard only consists of two steps, you can delete the other allocations.

Add the following definitions to the top area of the code module:

```
Private m_obDriver As Driver
Private m_obVarType As VarType
Private m_nChannelType As Integer
```

Create a method for initializing the driver combobox. The task of this routine is to display all the loaded drivers of the current project in a combobox.

```
cbDriver.Clear
Dim nIndex As Long
For nIndex = 0 To MyWorkspace.ActiveDocument.Drivers.Count - 1
    Dim obDriver As Driver
    Set obDriver = MyWorkspace.ActiveDocument.Drivers.Item(nIndex)
    If (Not obDriver Is Nothing) Then
        cbDriver.AddItem
        obDriver.Name
    End If
Next nIndex
If (cbDriver.ListCount > 0) Then
    cbDriver.ListIndex = 0
End If
```

Additionally we need a routine displaying all defined variable types of the project in a combobox.

```
If (Not m_obDriver Is Nothing) Then
cbVarType.Clear
Dim nIndex As Long , nSelect As Integer
For nIndex = 0 To MyWorkspace.ActiveDocument.VarTypes.Count - 1
    Dim obVarType As VarType
    Set obVarType = MyWorkspace.ActiveDocument.VarTypes.Item(nIndex)
    If ( Not obVarType Is Nothing And obVarType.IsSimple = True) Then
        cbVarType.AddItem
        obVarType.Name
    End If
    If (obVarType.Name = INT) Then
        nSelect = nIndex
    End If
```

```

End If
Next nIndex
cbVarType.ListIndex = nSelect
End If

```

On opening the wizard all existing variables are checked to find a free start offset for the the new variables to be created. This is done with the following method.

```

Private Function FindHighestOffsetVar() As Long
On Error GoTo Error
Dim nIndex As Long , nOffset As Long
For nIndex = 0 To MyWorkspace.ActiveDocument.Variables.Count - 1
Dim obVar As Variable
Set obVar = MyWorkspace.ActiveDocument.Variables.Item(nIndex)
If ( Not obVar Is Nothing) Then
If (obVar.Offset > nOffset) Then
nOffset = obVar.Offset
End If
End If
Next nIndex
FindHighestOffsetVar = nOffset
Exit Function
Error : MsgBox
Error occurs: + Err.Description + Source + Err.Source
End Function

```

Switch to the event Initialize of the UserForm and add the following lines to this method:

```

txtStart.Value = CStr(FindHighestOffsetVar + 1)
InitializeDriver

```

The allocation to **txtStart** sets the proposed start offset for the variables to be created. The method **InitializeDriver** fills the combobox with the existing drivers.

Create an event Change for the driver combobox and add the following code. After having selected a driver the variable types are acquired. The selected driver object is stored in the variable **m\_obDriver** for later use.

```

Private Sub cbDriver_Change()
cmdNext.Enabled = True
Set m_obDriver = MyWorkspace.ActiveDocument.Drivers.Item(cbDriver.Value)
If ( Not m_obDriver Is Nothing) Then
InitializeVarType
End If
End Sub

```

Create an event **Change** for the variable type combobox and add the following code. The selected variable type is stored in the variable `m_obVarType` for later use.

```
Private Sub cbVarType_Change()  
Set m_obVarType = MyWorkspace.ActiveDocument.VarTypes.Item(cbVarType.Value)  
End Sub
```

Now the only thing left is to create the event routine for creating the variables with the defined settings. This is done with the button **Finish**.

```
Private Sub cmdFinish_Click()  
On Error GoTo Error  
If (cbVarType.ListIndex = -1) Then  
MsgBox 'Please select a variable type'  
cbVarType.SetFocus  
Exit Sub  
End If  
If (txtStart.Value = Or txtCount.Value = Or txtStep.Value = ) Then  
MsgBox 'Please enter Start-Offset', 'count of creating variables and the step'  
txtStart.SetFocus  
End If  
If (m_obVarType Is Nothing) Then  
MsgBox 'Variable type + cbVarType.Name + doesnt exist!'  
Exit Sub  
End If  
Dim nPrvMousePtr As Integer  
nPrvMousePtr = MousePointer  
MousePointer = fmMousePointerHourGlass  
DoEvents  
Dim strName As String  
Dim nIndex As Long , nVarIndex As Integer  
Dim nStartOff As Long , nStep As Integer  
nVarIndex = 1  
nStartOff = CLng (txtStart.Value)  
nStep = CLng (txtStep.Value)  
For nIndex = 0 To CLng (txtCount.Value - 1)  
Dim obVar As Variable  
strName = txtName.Value + _ + CStr (nIndex + 1)  
'*** Guaranteeing uniqueness of the variable name  
Dim bResult As Boolean  
bResult = False  
Do  
Set obVar = MyWorkspace.ActiveDocument.Variables.Item(strName)  
If (obVar Is Nothing) Then  
bResult = True
```

```

Else
nVarIndex = nVarIndex + 1
strName = txtName.Value + _ + CStr (nVarIndex)
End If
Loop While
bResult = False
'*** Create variable
Set obVar = MyWorkspace.ActiveDocument.Variables.CreateVar (strName, m_obDriver,
tpSPSMerker, m_obVarType)
If ( Not obVar Is Nothing) Then
obVar.Offset = nStartOff
nStartOff = nStartOff + nStep
End If
Next nIndex
MousePointer = nPrvMousePtr
Unload Me
Exit Sub
Error :
MousePointer = nPrvMousePtr
MsgBox Error occurs: + Err.Description + Source + Err.Source
End Sub

```

On finishing the wizard it is checked, if the defined settings are valid. If this is not the case, a messages is displayed and the user is demanded to correct the entries.

If the defined settings are valid, the variables are created. The variables are named with a name and an index. If a variables with the same name already exists in the project, the next free index is acquired. In our code example always a variable with the channel type PLC marker is created. With each cycle the offset of the variable is increased.

#### 4.6.5 Updating wizards

To update the wizards:

1. In the **Extras** menu, select the **Update Editor Add-Ins** entry.  
A dialog for updating available wizards is opened
2. Select the desired wizards
3. Start the update by clicking **Start update**

If a wizard or a class already exists in the workspace, a warning is displayed.



### Attention

*Already existing wizards are overwritten during the update. Individual changes made at the wizard are lost.*

## 4.7 Frequently asked questions

In this chapter a few frequently asked questions are answered. You can find additional solutions online in the COPA-DATA User forum (<http://www.copadata.com/forums/>).

### 4.7.1 Why does the button stay pressed?

If a button is linked e.g. to a LeftClickUp event, in the end of the procedure the LeftClickUp has to be executed.

```
Public Sub LeftClickUp_Schalter(obElem As Element)
frmSchalter.Show
obElem.LeftClickUp
End Sub
```

### 4.7.2 Macro is not performed with the first click

The solution matches the one from the question: Why does the button stay pressed (on page 69):

If a button is linked e.g. to a LeftClickUp event, in the end of the procedure the LeftClickUp has to be executed.

```
Public Sub LeftClickUp_Schalter(obElem As Element)
frmSchalter.Show

obElem.LeftClickUp
End Sub
```

### 4.7.3 Macros no longer work in the Runtime?

This effect can occur, if the VBA Editor is opened in the Runtime and then Stop/Start is pressed to stop/start VBA. In this case the objects (OnlineVariables, ScreenObjects, ...) become invalid, because they lose the link in case of a new initialization.

#### 4.7.4 Windows CE and VBA

In the Editor VBA can be used for wizards. It cannot be used in the Runtime. For detailed information about the Editor refer to chapter How to create projects in CE.

### 4.8 Examples

Here you can find a few examples for VBA

#### 4.8.1 MouseEvents and ActiveX Control initialization

**Option Explicit**

```
Public Sub Init_ActiveX(obElem As Element)
    'Initializing ActiveX...
    thisProject.Init_MSChart_AX obElem
End Sub
```

```
Public Sub LeftClickUp_Sample1(obElem As Element)
    'Initializing Userform...
    frmSample1.InitForm obElem
    'Show Userform
    frmSample1.Show
End Sub
```

```
Public Sub LeftClickUp_Sample2(obElem As Element)
    'Initializing Userform...
    frmSample2.InitForm obElem
    'Show Userform
    frmSample2.Show
End Sub
```

```
Public Sub LeftClickUp_Sample3(obElem As Element)
    'Initializing Userform...
```

```

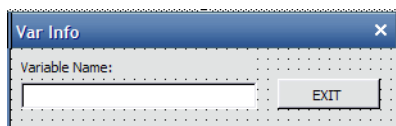
frmSample3.InitForm obElem
'Show Userform
frmSample3.Show
End Sub

Public Sub LeftClickUp_Sample4(obElem As Element)
    Dim NewForm As New frmSample4
    'Initializing NEW Userform...
    NewForm.InitForm obElem
    'Show NEW Userform
    NewForm.Show (0)
End Sub

```

## 4.8.2 Display variable information

Show variable name for clicked element:



**Option Explicit**

```

Dim obVar As Variable

'User defined Public Procedure for initializing Objects

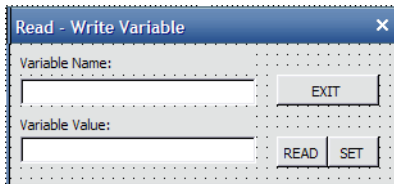
Public Sub InitForm(obElem As Element)
    'set the variable object like the linked variable of the element
    Set obVar = obElem.ItemVariable(0)
    'write variable name into the textbox
    txtVarName.Text = obVar.Name
End Sub

Private Sub cmdExit_Click()
    'close Userform
    Unload Me
End Sub

```

### 4.8.3 Read and write variable values

Read value from variable and write it back:



**Option Explicit**

```
Dim obVar As Variable
```

```
'User defined Public Procedure for initializing Objects
```

```
Public Sub InitForm(obElem As Element)
```

```
    'set the variable object like the linked variable of the element
```

```
    Set obVar = obElem.ItemVariable(0)
```

```
    'write variable name into the textbox
```

```
    txtVarName.Text = obVar.Name
```

```
End Sub
```

```
Private Sub cmdExit_Click()
```

```
    'close Userform
```

```
    Unload Me
```

```
End Sub
```

```
Private Sub cmdRead_Click()
```

```
    'read value from variable and write into textbox
```

```
    txtValue.Text = obVar.Value
```

```
End Sub
```

```
Private Sub cmdWrite_Click()
```

```
    'write text as value to variable
```

```
    obVar.Value = txtValue.Text
```

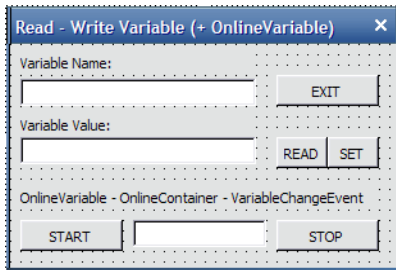
```
    'or changing text to value before writing...
```

```
    'obVar.Value = Val(txtValue.Text)
```

```
End Sub
```

#### 4.8.4 Read and write variables and implement online variables

Read variable information, write values and implement online variables:



**Option Explicit**

```
Dim obVar As Variable
```

```
Dim WithEvents zOnlineVariable As OnlineVariable
```

```
'User defined Public Procedure for initializing Objects
```

```
Public Sub InitForm(obElem As Element)
```

```
    'set the variable object like the linked variable of the element
```

```
    Set obVar = obElem.ItemVariable(0)
```

```
    'write variable name into the textbox
```

```
    txtVarName.Text = obVar.Name
```

```
    'create an OnlineVariable container
```

```
    Set zOnlineVariable = thisProject.OnlineVariables.CreateOnlineVariables("OIV")
```

```
    'add variables to the container (by name of the variable)
```

```
    zOnlineVariable.Add obVar.Name
```

```
End Sub
```

```
Private Sub cmdExit_Click()
```

```
    'close Userform
```

```
    Unload Me
```

```
End Sub
```

```
Private Sub cmdRead_Click()
```

```
    'read value from variable and write into textbox
```

```
    txtValue.Text = obVar.Value
```

```
End Sub
```

```
Private Sub cmdWrite_Click()
```

```
'write text as value to variable
obVar.Value = txtValue.Text
'or changing text to value before writing...
'obVar.Value = Val(txtValue.Text)
End Sub

Private Sub cmdOLV_Start_Click()
    'start the OnlineVariable - Define
    'the VariableChange Event will be executed
    zOnlineVariable.Define
End Sub

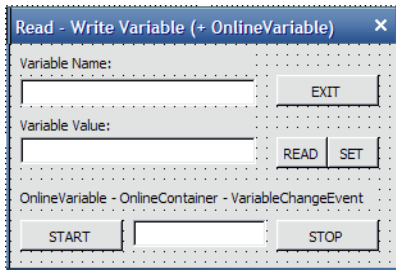
Private Sub cmdOLV_Stop_Click()
    'stop the OnlineVariable - UnDefine
    'the VariableChange Event will be stopped
    zOnlineVariable.Undefine
End Sub

Private Sub zOnlineVariable_VariableChange(ByVal obVar As IVariable)
    'write actual value into textbox
    txtOLV.Text = obVar.Value
End Sub

Private Sub UserForm_Terminate()
    'the VariableChange Event will be stopped if running
    zOnlineVariable.Undefine
    'delete OnlineVariable container
    thisProject.OnlineVariables.DeleteOnlineVariables ("OLV")
End Sub
```

## 4.8.5 Use dialog multiple times

Userforms can be used multiple times.



### Option Explicit

```

Dim obVar As Variable
Dim WithEvents zOnlineVariable As OnlineVariable
Dim strOLVName As String

Public Sub InitForm(obElem As Element)
    'set the variable object like the linked variable of the element
    Set obVar = obElem.ItemVariable(0)
    'write variable name into the textbox
    txtVarName.Text = obVar.Name
    'create name for Online Container
    strOLVName = "OLV_" & obElem.Name
    'get existing online container
    Set zOnlineVariable = thisProject.OnlineVariables.Item(strOLVName)
    'check if online container exists
    If zOnlineVariable Is Nothing Then
        'create an OnlineVariable container
        Set zOnlineVariable =
thisProject.OnlineVariables.CreateOnlineVariables(strOLVName)
        'add variables to the container (by name of the variable)
        zOnlineVariable.Add obVar.Name
    End If
End Sub

Private Sub cmdExit_Click()
    Unload Me 'close Userform
End Sub

Private Sub cmdRead_Click()

```

```
'read value from variable and write into textbox
txtValue.Text = obVar.Value
End Sub

Private Sub cmdWrite_Click()
    'write text as value to variable
    obVar.Value = txtValue.Text
    'or changing text to value before writing...
    'obVar.Value = Val(txtValue.Text)
End Sub

Private Sub cmdOLV_Start_Click()
    'the VariableChange Event will be executed
    zOnlineVariable.Define
End Sub

Private Sub cmdOLV_Stop_Click()
    'the VariableChange Event will be stopped
    zOnlineVariable.Undefine
End Sub

Private Sub zOnlineVariable_VariableChange(ByVal obVar As IVariable)
    'write actual value into textbox
    txtOLV.Text = obVar.Value
End Sub

Private Sub UserForm_Terminate()
    'the VariableChange Event will be stopped if running
    zOnlineVariable.Undefine
    'delete OnlineVariable container
    thisProject.OnlineVariables.DeleteOnlineVariables (strOLVName)
End Sub
```

## 4.8.6 Alarm – Events and ActiveX Control handling

### Option Explicit

```
Dim WithEvents obChart As MSChart
Dim WithEvents zOLV As OnlineVariable
Dim WithEvents zAlarm As Alarm
```

'procedure is executed on startup of the zenon Runtime

```
Private Sub Project_Active()
    'init the alarm object for events
    Set zAlarm = thisProject.Alarm
End Sub
```

'procedure is executed when an Alarm comes

```
Private Sub zAlarm_AlarmComes(ByVal obItem As IAlarmItem)
    Dim strInfo As String
    'write specific information about the alarm into a StringVariable
    strInfo = obItem.Text & " - " & obItem.Name
    thisProject.Variables.Item("Var_Comes").Value = strInfo
End Sub
```

'procedure is executed when an Alarm has gone

```
Private Sub zAlarm_AlarmGoes(ByVal obItem As IAlarmItem)
    Dim strInfo As String
    'write specific information about the alarm into a StringVariable
    strInfo = obItem.Text & " - " & obItem.Name
    thisProject.Variables.Item("Var_Goes").Value = strInfo
End Sub
```

'procedure is executed when an Alarm was acknowledged by a user

```
Private Sub zAlarm_AlarmAcknowledged(ByVal obItem As IAlarmItem)
    Dim strInfo As String
    'write specific information about the alarm into a StringVariable
    strInfo = obItem.Text & " - " & obItem.Name
    thisProject.Variables.Item("Var_Acknowledged").Value = strInfo
End Sub
```

```

'procedure is executed on terminating the zenon Runtime
Private Sub Project_Inactive()
    'free the alarm object
    Set zAlarm = Nothing
    'delete OnlineVariable for Chart actualization...
    thisProject.OnlineVariables.DeleteOnlineVariables "CHART"
End Sub

'procedure for MSChart ActiveX initialization...
Public Sub Init_MSChart_AX(YourAX As Element)
    Set obChart = YourAX.AktiveX
    obChart.RowCount = 3
    obChart.ColumnCount = 1
    Set zOLV = thisProject.OnlineVariables.Item("CHART")
    'does existing OnlineVariable exist?
    If zOLV Is Nothing Then
        'if not, create it...
        Set zOLV = thisProject.OnlineVariables.CreateOnlineVariables("CHART")
        zOLV.Add "Internal_UINT_001"
        zOLV.Add "Internal_UINT_002"
        zOLV.Add "Internal_UINT_003"
    End If
    zOLV.Undefine 'if not stopped, refreshing not possible
    'START watching variables...
    zOLV.Define
End Sub

'event on Variable change - refresh chart...
Private Sub zOLV_VariableChange(ByVal obVar As IVariable)
    'setting values to display in chart control
    Select Case obVar.Name
        Case "Internal_UINT_001"
            obChart.Row = 1
            obChart.RowLabel = "Var1"
            obChart.Data = obVar.Value
        Case "Internal_UINT_002"
            obChart.Row = 2
            obChart.RowLabel = "Var2"
            obChart.Data = obVar.Value
    End Select
End Sub

```

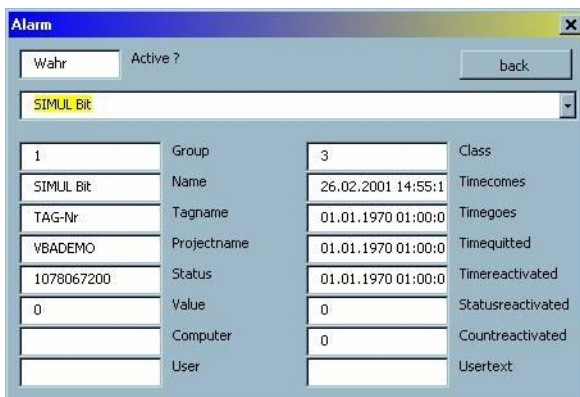
```

    Case "Internal_UINT_003"
        obChart.Row = 3
        obChart.RowLabel = "Var3"
        obChart.Data = obVar.Value
    End Select
End Sub

'event of the Chart AX...
Private Sub obChart_DblClick()
    MsgBox "You have DoubleClicked the ActiveX!"
End Sub

```

#### 4.8.7 Access to alarms



Group	Class
1	3
SIMUL Bit	26.02.2001 14:55:1
TAG-Nr	01.01.1970 01:00:0
VBADEMO	01.01.1970 01:00:0
1078067200	01.01.1970 01:00:0
0	0
Computer	0
User	

##### DESCRIPTION:

In the form frmAlarm an alarm from the memory can be selected in a combobox. After the selection all data of the alarm are written to the textboxes below (group, class, variable, ...).

We use an event independent macro to display frmAlarm, because we do not link it to an element.

```

Sub Alarm ()
    frmAlarm.Show
End Sub

```

'The macro is executed with the function Execute macro.

'On opening the form it is initialized and so the following procedure is executed. This procedure cares, that all alarms in the memory are written to the combobox in the form.

```

Private Sub UserForm _ Initialize ()

```

```

'fill combobox with all AlarmItems
Dim i As Integer
Dim obAlarmItems As AlarmItems
Dim obAlarm As Alarm

Set obAlarm = thisProject.Alarm
Set obAlarmItems = obAlarm.AlarmItems (*)

If obAlarmItems.Count = 0 Then
MsgBox (# Alarms = 0 )
Exit Sub
End If

For i = 0 To obAlarmItems.Count - 1
cmbAlarmItems.AddItem obAlarmItems.Item ( i ). Name
Next i

txtAktiv.Text = obAlarm.Aktiv
cmbAlarmItems.Text = cmbAlarmItems.List ( 0 )
End Sub

'Wenn nun ein Alarm aus der Combobox ausgewählt wird reagiert das Change - Ereigniss der
Combobox.
Private Sub cmbAlarmItems _ Change ()

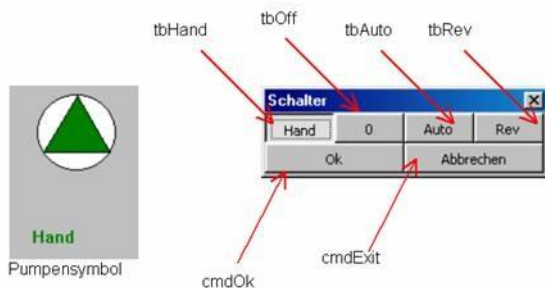
'put actual properties from AlarmItem in textboxes
Dim obAlarmItems As AlarmItems
Dim obAlar As Alarm

Set obAlarm = thisProject.Alarm
Set obAlarmItems = obAlarm.AlarmItems (*)
txtComputer.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Computer
txtCountreactivated.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Countreactivated
txtName.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ). Name
txtProjectname.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Projectname
txtStatus.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Status
txtStatusreactivated.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex
).Statusreactivated
txtTagname.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Tagname
txtTimecomes.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Timecomes
txtTimegoes.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Timegoes
txtTimequitted.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Timequitted
txtTimereactivated.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Timereactivated
txtUser.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).User
txtUsertext.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Usertext
txtValue.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).Value

```

```
tbGroup.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).AlarmGroup
tbClass.Text = obAlarmItems.Item ( cmbAlarmItems.ListIndex ).AlarmClass
End Sub
```

#### 4.8.8 Set switch (working with process variables)



In this example we draw a pump consisting of a circle and a triangle. Define the triangle as a symbol. On top draw a multibinary element and link it to three bit marker variables.

Additionally define, which color the triangle should get, if the values of the variables change.

First we combine the multibinary element with a macro, which opens a form frmSwitch.

In the form frmSwitch we will be able to change the values of the three bit marker variables.



#### Information

*Only one of the three variables may have the value 1. (i.e. if one variable is set to 1, the other two have to be set to 0)*

To be able to use this macro several times in project with different variables, you only may link bit marker variables to the multibinary element, which contain in their names, which status of the pump they control.

for example:

Variable\_Auto

Variable\_Hand

Variable\_Revi



### Information

*The suffixes `_Auto`, `_Hand` and `_Revi` are fixly defined in the source code of the example.*

With this five characters suffix of the variable names it is defined, which variable is set to 1 and which is set to 0 on clicking a toggle button.

In the macro `LeftClickUp_Switch` a sub program `FindVariable` is called in the form `frmSwitch`, which gets the clicked element `obElem`.

```
Public Sub LeftClickUp_Schalter (obElem As Element)
frmSchalter.FindVariable obElem
position (pixel to points = (pixel * 0.75))
frmSchalter.Top = obElem.Bottom * 0.75
frmSchalter.Left = obElem.Left * 0.75
frmSchalter.Show
obElem.LeftClickUp
End Sub
```

Module global variable declaration:

```
Dim cmdLast As ToggleSchaltfläche
Dim strHand As String
Dim strAuto As String
Dim strRevi As String
```

In the sub program `FindVariable` all variables linked to the passed element are checked.

Depending on the suffix (`_Auto`, `_Hand` or `_Revi`) the variable names are assigned to the string variables declared above.

Additionally the status of the variables is determined and depending on the value (1 or 0) the according toggle button is pressed or not.

On opening the form `frmSwitch` the name of the currently pressed toggle button is written to a string variable. For the case, that the user decides to cancel his action, the original values are reset.

```
Public Sub FindVariable (obElem As Element)
Dim i As Integer
Dim obVariable As Variable

For i = 0 To obElem . CountVariable - 1
Select Case Right $( obElem . ItemVariable ( i ). Name , 5 )
Case _ Auto
strAuto = obElem . ItemVariable ( i ). Name

Case _ Hand
strHand = obElem . ItemVariable ( i ). Name
```

```

Case _ Revi
strRevi = obElem . ItemVariable ( i ). Name
End Select
Next i

Set obVariable = thisProject . Variables . Item ( strHand )
If obVariable . Value = 1 Then
tbHand . Value = True
Set cmdLast = tbHand
End If

Set obVariable = thisProject . Variables . Item ( strAuto )
If obVariable . Value = 1 Then
tbAuto . Value = True
Set cmdLast = tbAuto
End If

Set obVariable = thisProject . Variables . Item ( strRevi )
If obVariable . Value = 1 Then
tbRev . Value = True
Set cmdLast = tbRev
End If

If tbHand . Value = False And tbAuto . Value = False And tbRev . Value = False Then
tbOff . Value = True
Set cmdLast = tbOff
End If
End Sub

```

The self-created function VarExists only checks, whether the linked variables really exist. If this is not the case, an error message is displayed. Variable doesn't exist.

```

Function VarExists ()

Dim obVariable As Variable
Set obVariable = thisProject . Variables . Item (strHand)

If obVariable Is Nothing Then
MsgBox (Variable doesnt extist)
VarExitsts = False
Exit Function
End If

Set obVariable = thisProject . Variables . Item (strAuto)
If obVariable Is Nothing Then
MsgBox ( Variable doesnt extist )
VarExitsts = False
Exit Function

```

```
End If
```

```
Set obVariable = thisProject . Variables . Item (strRev)
If obVariable Is Nothing Then
MsgBox ( Variable doesnt extist )
VarExitsts = False
Exit Function
End If
```

```
VarExists = True
End Function
```

If the user clicks Cancel, the value change is undone and the original status is reset.

```
Private Sub cmdExit _ Click ()
cmdLast.Value = True
Unload Me
End Sub
```

```
Private Sub cmdOk _ Click ()
Unload Me
End Sub
```

If one toggle button is pressed, no other toggle button may be pressed.

```
Private Sub tbAuto_Change ()
If tbAuto . Value = False And tbHand.Value = False And tbRev . Value = False Then
tbOff . Value = True
End Sub
```

In the click event of every toggle button it is checked, whether it is pressed and whether the variable exists. If both conditions are true, the values are sent to the linked variables.

## 5. VSTA

With **Visual Studio Tools for Applications (VSTA)**, the functionality of zenon Runtime and the Editor can be enhanced independently by means of .NET programming.



### Information

*Note that in VSTA, only assemblies (DLLs) up to a maximum of .NET framework version 3.5 can be included.*

VSTA is also, with a few restrictions, available on the zenon Web Client.

VSTA provides separate development environments for **Workspace** and **project**. You can only use one of them at a time. At the start every other VSTA development environment which is open will be close.

To open the VSTA Editor for the workspace:

1. Press the short cut `Alt+F10`.

The code for the workspace and all loaded projects is displayed.

To open the VSTA Editor for the currently loaded project:

1. Navigate to the **Programming interfaces** node
2. Expand the view of this node by clicking on `[+]`.  
The view of the node is expanded.
3. Right-click on **VSTA**

In the context menu, select **Open VSTA editor with ProjectAddin**.  
The editor is opened for the currently-loaded project.



#### Information

*If VBA macros are changed in the Editor,*

- ▶ the Runtime files are compiled and transferred to the Runtime
- ▶ the Runtime is reloaded
- ▶ VSTA elements are also reloaded even if no changes were made in VSTA

## 5.1 Basics

VSTA is a Microsoft tool set that is based on .NET technology. It is necessary to have basic knowledge of object-orientated programming, .NET and C#/Visual Basic.NET to understand it.



#### Attention

*Note in Runtime: Referenced dlls cannot be replaced if they are loaded. Runtime must be ended in order for a reference file to be updated.*

### 5.1.1 Setting up the VSTA environment

Support for VSTA is already activated as standard in zenon. When deactivating VBA support, the VSTA environment is also not available any more.

The VSTA environment can be activated or deactivated manually with the following entry in **zenon6.ini**:

Activate VSTA	Deactivate VSTA
[VSTA] ON=1	[VSTA] ON=0

Support for VBA is activated or deactivated as follows:

Activate VBA	Deactivate VBA
[VBA] EIN=1	[VBA] EIN=0

After this, the development environment for VSTA in zenon is available.



### Information

*VSTA allows the development of projects in the programming languages C# and Visual Basic.NET. C# is envisaged as the standard language for VSTA projects in the Editor. The language can be changed to Visual Basic.NET with the following entry:*

**[VSTA]**

**CSHARP=0**

## 5.1.2 Access to the object model in zenon

The zenon that is also used in VBA can be accessed in VSTA. The object model offers the same functionality in both development environments.



### Attention

*Some changes to the object model have been made due to limitations in naming VSTA objects. You can find these in the table below*

Old property	New property
IDriver.Name	IDriver.Identification
IDriver.Driver	IDriver.Name
Old event	New event
IApplication.Close	IApplication.OnClose
IZenWorkspace.Startup	IZenWorkspace.OnStartup
IZenWorkspace.Exit	IZenWorkspace.OnExit

Access to VSTA is enabled via the `this` object and it replaces the `MyWorkspace` object in VBA. The following methods and objects are identical. In the following method, a template with the name "TemplateName" is created in zenon.

```
public void Macro1()
{
    this.ActiveDocument.Templates().Create("TemplateName", true);
}
```



### Information

*In contrast to VBA, capitalization and brackets after function names are important in VSTA.*

To access the methods in zenon, the project must be saved and compiled using the following steps:

1. Click on *File -> Save MyWorkspace.cs* to save the project.
2. Click on *File -> Build WorkspaceAddin* to compile the project.

After this, the method is available as a macro in the VBA macro toolbar in the zenon editor. If the macro assignment dialog does not list all macros from MyWorkspace, the function 'Reload list of VBA macros' has to be executed from the toolbar.

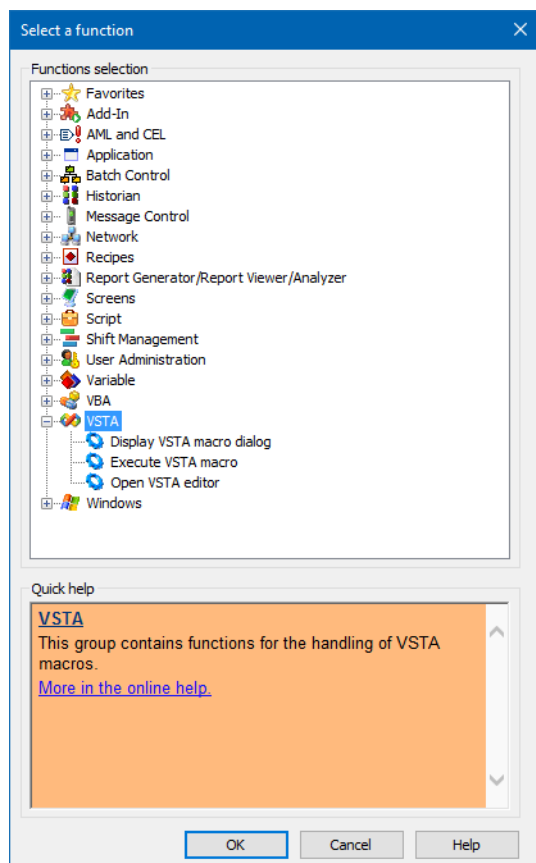


### Information

*VSTA macros with parameters, e.g. `Public void MacroWithParam(string mString)`, are not supported and also not made available in the macro toolbar.*

### 5.1.3 Functions in zenon

For VSTA, functions were created in zenon. These are in the **VSTA** node.

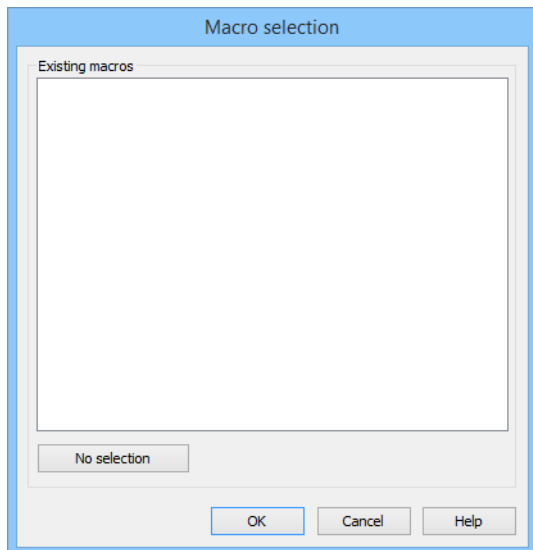


At the same time as existing VBA functions, similar functions were implemented for VSTA:

Function name	Description
<b>Open VSTA editor</b>	opens the VSTA editor in Runtime
<b>Execute VSTA macro</b>	<p>A VSTA macro can be selected in the editor, which is started when executing the function in Runtime.</p> <p><b>Note:</b> VSTA macros with parameters, e.g. <code>Public void MacroWithParam(string mString)</code>, are not supported. They are neither offered at the engineering in the Editor nor at the start of the function in the Runtime.</p>
<b>Show VSTA macro dialog</b>	A dialog is shown in Runtime, in which existing VSTA macros are shown and can be selected and executed

## Execute VSTA macro

Dialog for configuration of the **Execute VSTA macro** function.



Parameters	Description
<b>Existing macros</b>	
<b>List of VSTA macros</b>	Lists all existing VSTA macro. Selection of a macro from the list by clicking on it. <b>Note:</b> A double click executes the selected macro.
<b>None selected</b>	Discards the selection and closes the dialog.
<b>OK</b>	Executes selected macro and closes the dialog.
<b>Cancel</b>	Discards the selection and closes the dialog.
<b>Help</b>	Opens online help.

### 5.1.4 Debugging a VSTA add-in

It is possible to debug add-ins you have written yourself with the VSTA Editor. In doing so, note that project add ins can only be debugged in zenon Runtime and workspace add-ins can only be debugged in zenon Editor.

A debug session is started via the *Debug - Start Debugging* menu. You can place breakpoints in the same way as the VBA editor, by left clicking in the gray breakpoint toolbar at the left margin next to the respective cell.



### Information

*When debugging Runtime add-ins consider:*

*The Runtime files changed in zenon must be newly created before debugging.*

## 5.1.5 Events in VSTA

Because an add-in is terminated when compiling amended code, starting a debug session or ending a debug session, corresponding events were implemented in VSTA. These enable, for example, an object reference to be evaluated and approved and existing data to be saved.

Two events exist for each termination. The first event is started shortly before termination, the second after the start of a new add-in session.

Event	Description
<a href="#">OnPreVSTADebugStart</a>	Is triggered shortly before a debug session is started. When starting, an active add-in is removed, references must be approved and existing data must be saved if necessary.
<a href="#">OnVSTADebugStart</a>	Is triggered shortly after a debug session is started.
<a href="#">OnPreVSTADebugStop</a>	Is triggered shortly before a debug session is stopped. When stopping a debug session, an active add-in is removed, references must be approved and existing data must be saved if necessary.
<a href="#">OnVSTADebugStopped</a>	Is triggered shortly after a debug session is stopped.
<a href="#">OnPreVSTAUpdate</a>	Is triggered before the add-in is removed if a new version of the add-in was successfully created.
<a href="#">OnPostVSTAUpdate</a>	Is triggered when a new version of the add-in is loaded.

## 5.1.6 Creating a backup of VSTA projects

VSTA projects in Runtime are automatically zipped when creating the Runtime file and included in workspace saves.

VSTA projects in the editor must be saved manually however. You can find the VSTA Editor projects in the folder  
C:\ProgramData\COPA-DATA\\*version\*\VSTAWorkspace\.

## 5.2 Creating a VSTA project

Similar to VBA, there is the possibility in VSTA to create projects (enhancements) for both the Editor and Runtime. In principle, projects in the editor are implemented in the C# programming language. For Runtime and Editor, both C# and Visual Basic.NET are available.



### Information

*Note that in VSTA, only assemblies (DLLs) up to a maximum of .NET framework version 3.5 can be included.*



### Information

*Only one project can be displayed at a time in the VSTA editor. In addition, only one instance of the VSTA editors can be active. When starting the VSTA editor, any instance that may already be running is closed.*

### 5.2.1 VSTA projects in the editor

When creating a project for the zenon editor, a VSTA add-in for the workspace is loaded. To edit the add-in, the VSTA editor must be opened via *File - Open VSTA editor...* The user interface of the VSTA editor is identical to Microsoft's Visual Studio development environment.



### Information

*VSTA editor help can be accessed via the *Help / Contents* menu. This help gives an overview of the editor's functions, the features of the .NET framework and programming in Visual Basic.NET and C#.*

The VSTA add-in basically consists of the **MyWorkspace** class. This class can now be expanded with your own methods. The class accommodates the following two methods by default:

Function	Description
<b>MyWorkspace_Startup</b>	Is executed automatically when starting zenon, after a build has been created and when a debug session is

	started.
<b>MyWorkspace_Shutdown</b>	Is executed automatically when starting zenon, after a build has been created and when a debug session is started.



### Attention

*The method names may only start with **Macro** (for example Macro1, MacroVSTA) may not contain parameters and must be defined as **Public**. In addition, the class names and other methods and events created by VSTA may not be changed.*

To access the methods in zenon, the project must be saved and compiled using via the following steps:

1. Click on *File -> Save MyWorkspace.cs* to save the project.
2. Click on *File -> Build WorkspaceAddin* to compile the project.

After this, the method is available as a macro in the VBA macro toolbar in the zenon editor. If the macro assignment dialog does not list all macros from MyWorkspace, the function 'Reload list of VBA macros' has to be executed from the toolbar.

## 5.2.2 VSTA projects in Runtime

To create a VSTA project for Runtime, the VSTA environment must be started.

Proceed in the following way:

1. Open the node **Programming interface** in the project manager.
2. Open the **VSTA** context menu.
3. Click on **Open VSTA editor...**

Note: A selection dialog is shown when it is opened for the first time. You select the programming language in this.

A project is created in the desired programming language.



### Information

*The programming language cannot be subsequently changed. This dialog is therefore only shown when being opened for the first time.*

In this project, a class named **ThisProject** is created by zenon, which accommodates the following two methods:

Function	Description
<code>ThisProject_Startup</code>	Is executed automatically when Runtime is started
<code>ThisProject_Shutdown</code>	Is executed automatically when Runtime is ended

The class can now be expanded with your own methods.



### Attention

*The method names may only start with **Macro** (for example `Macro1`, `MacroVSTA`) may not contain parameters and must be defined as **Public**. In addition, the class names and other methods and events created by VSTA may not be changed.*

There is access to all Runtime functionalities via the zenon object model. Editor-specific functions cannot be used, as in VBA.

zenon Runtime is automatically started when the debugger is started. Further information can be found in the chapter on debugging a VSTA add-in (on page 89).



### Information

*VSTA editor help can be accessed via the `Help / Contents` menu. This help gives an overview of the editor's functions, the features of the .NET framework and programming in Visual Basic.NET and C#.*

## 5.2.3 Developing wizards in VSTA

The VSTA environment, like VBA (on page 60), offers the possibility to develop your own wizards.

To be able to access a form in the zenon object model, a reference to this must be copied to the form. To do this, a method is created in the `MyWorkspace` class. In the following example example, a form is instantiated with the name `wizard` and the method `ZenonInstance` with a reference to the zenon object model is called as a parameter. The wizard form is shown by selecting `ShowDialog()`.

```
public void Macro1()
{
    Form1 Wizard = new Form1();
    Wizard.ZenonInstance(this.Application);
    Wizard.ShowDialog();
}
```

A member variable must be created in the form code, which recognizes the zenon object model.

```
public zenOn.IApplication m_Zenon=null;
```

Lastly, the `ZenonInstance` method is created. This method takes the object model reference and places it in the `m_Zenon` object.

```
public void ZenonInstance(zenOn.IApplication app)
{
    m_Zenon = app;
}
```

Now, your own classes and methods can be developed in the form, which make use of the object model. All methods, objects and attributes are available via the `m_Zenon` object.

## 5.3 Examples

Here you find some examples of VSTA being used, both in Runtime and in the editor.

### 5.3.1 Creating variables in the zenon editor

In this example, a text file is opened and the contents of this are used to create variables in the zenon editor. The text file contains any desired number of lines. Each line includes the name and data type of a variable; these are separated by a comma (example: Variable1,BOOL).

The `Macro1` method first looks for the internal driver in the zenon editor. After this, the user is shown a file selection dialog in which he must select the text file. The method then reads the text file and creates the variables. The `GetDataType` method is then required to determine and assign the attendant data type when creating the variables.

#### C# code

```
using System;
using System.Windows.Forms;
using System.IO;
using zenOn;

namespace WorkspaceAddin
{
    [System.AddIn.AddIn("MyWorkspace", Version = "1.0", Publisher = "", Description = "")]
    public partial class MyWorkspace
    {
        private void MyWorkspace_Startup(object sender, EventArgs e)
        {

```

```

    }

    private void MyWorkspace_Shutdown(object sender, EventArgs e)
    {
    }

    public IVarType GetDataType(string vType)
    {
        //gets the corresponding vartypes for bool, int, real and strings
        IVarType retType;
        switch (vType)
        {
            case "BOOL":
                retType = this.ActiveDocument.VarTypes().Item("BOOL");
                break;
            case "INT":
                retType = this.ActiveDocument.VarTypes().Item("INT");
                break;
            case "REAL":
                retType = this.ActiveDocument.VarTypes().Item("REAL");
                break;
            case "STRING":
                retType = this.ActiveDocument.VarTypes().Item("STRING");
                break;
            default:
                retType = this.ActiveDocument.VarTypes().Item("INT");
                break;
        }
        return retType;
    }

    //Reads a defined text file and creates corresponding variables on the ze
non internal driver
    public void MacroCreateVariablesFromFile()
    {
        //create objects that will take the internal driver and the variable
type
        IDriver zenonInternDriver = null;
        //search for the Internal driver and throw exception if no driver was
found

```

```

        try
        {
            for (int driverCount = 0; driverCount < this.ActiveDocument.Drivers().Count; driverCount++)
            {
                if (this.ActiveDocument.Drivers().Item(driverCount).Name == "
Intern")
                {
                    zenonInternDriver = this.ActiveDocument.Drivers().Item(driverCount);

                    break;
                }
            }
        }
        catch (Exception driverEx)
        {
            MessageBox.Show("Unable to find zenon 'Intern' driver. Error: " + driverEx.Message);
            throw;
        }
        this.ActiveDocument.Variables().DoAutoSave(false);
        try
        {
            OpenFileDialog varFileSelect = new OpenFileDialog();
            String[] varLine = new String[2];

            //show file dialog
            if (varFileSelect.ShowDialog() == DialogResult.OK)
            {
                string line = string.Empty;
                //open new stream reader with selected file
                StreamReader importReader = new StreamReader(varFileSelect.FileName, System.Text.Encoding.Default);

                //read in line by line, split the lines when a ',' occurs and create variables
                while ((line = importReader.ReadLine()) != null)
                {
                    varLine = line.Split(new Char[] { ',', ' ' });
                }
            }
        }
    }
}

```

```

        this.ActiveDocument.Variables().CreateVar(varLine[0], zenonInternDriver, tpKanaltypes.tpSystemVariable, GetDataType(varLine[1]));
    }
    importReader.Close();
}
}
catch (Exception fileEx)
{
    MessageBox.Show("An error occurred while opening the file: " + fileEx.Message);
    throw;
}
this.ActiveDocument.Variables().DoAutoSave(true);
}
#region VSTA generated code
private void InternalStartup()
{
    this.Startup += new System.EventHandler(MyWorkspace_Startup);
    this.Shutdown += new System.EventHandler(MyWorkspace_Shutdown);
}
#endregion
}
}

```

### 5.3.2 Writing project information in the zenon output window

In this example, it is demonstrated how the output window of the zenon editors can be accessed using VSTA. The method named `Macro1` reads out the process screens created in the project for this, identifies the respective template and identifies all drivers available as well as their labels.

#### C# code

```

using System;

namespace WorkspaceAddin
{
    [System.AddIn.AddIn("MyWorkspace", Version = "1.0", Publisher = "", Description = "")]

```

```

public partial class MyWorkspace
{
    private void MyWorkspace_Startup(object sender, EventArgs e)
    {

    }

    private void MyWorkspace_Shutdown(object sender, EventArgs e)
    {

    }

    public void MacroPrintDebugInformation()
    {
        string picName = string.Empty;
        string corTemp = string.Empty;
        string driverName = string.Empty;
        string driverDescription = string.Empty;

        //print start string into output window

        this.Application.DebugPrint(" -----START-----", zen
On.tpDebugPrintStyle.tpMsg);

        //go through all pictures and print name and used template into output window

        for (int i = 0; i < this.ActiveDocument.DynPictures().Count; i++)
        {
            picName = this.ActiveDocument.DynPictures().Item(i).Name;
            corTemp = this.ActiveDocument.DynPictures().Item(i).get_DynProperties("Template").ToString();
            this.Application.DebugPrint(" Picture '" + picName + "' uses Template '" + corTemp + "'", zenOn.tpDebugPrintStyle.tpMsg);
        }

        //print separator string into output window

        this.Application.DebugPrint(" -----", zenOn.

```

```

tpDebugPrintStyle.tpMsg);

        //go through all drivers and print name and description into output w
indow

        for (int i = 0; i < this.ActiveDocument.Drivers().Count; i++)
        {
            driverName = this.ActiveDocument.Drivers().Item(i).Name;
            driverDescription = this.ActiveDocument.Drivers().Item(i).Identif
ication;

            this.Application.DebugPrint(" Driver '" + driverName + "' has des
cription '" + driverDescription + "'", zenOn.tpDebugPrintStyle.tpMsg);
        }
        //print end string into output window
        this.Application.DebugPrint(" -----END-----", zen
On.tpDebugPrintStyle.tpMsg);
    }

    #region VSTA generated code

    private void InternalStartup()
    {
        this.Startup += new System.EventHandler(MyWorkspace_Startup);
        this.Shutdown += new System.EventHandler(MyWorkspace_Shutdown);
    }
    #endregion
}
}

```

### 5.3.3 Reading in of variables in zenon via regular expressions

In the following example, zenon variables are read out in a Runtime project and saved in a local text file.

Using regular expressions, variables are only read if their names start with 3 figures and a subsequent underscore (for example "001\_var" or "234\_xyz"). The user is then requested to select a folder. A text file with a time-dependent file name is created in this folder. In this file, name, labeling and current value of all applicable variables is saved separately with a semi colon.



## Information

*It is possible that manual references may have to be added to execute the example in zenon Runtime. To do this, open the context menu in the Project Explorer and click on **Add Reference..***

*The references required in this example are:*

- ▶ **Microsoft.VisualStudio.Tools.Applications.Runtime.v9.0**
- ▶ **System**
- ▶ **System.AddIn**
- ▶ **System.Data**
- ▶ **System.Windows.Forms**
- ▶ **System.Xml**
- ▶ **zenonVSTAProxy6500**

### C# code

```
using System;
using System.Text.RegularExpressions;
using System.IO;
using System.Windows.Forms;

namespace ProjectAddin
{
    [System.AddIn.AddIn("ThisProject", Version = "1.0", Publisher = "", Description = "")]
    public partial class ThisProject
    {
        private void ThisProject_Startup(object sender, EventArgs e)
        {
        }

        private void ThisProject_Shutdown(object sender, EventArgs e)
        {
        }

        public void MacroGetFilteredVariables()
        {
            string filename = string.Empty;
            string name = string.Empty;
            string description = string.Empty;
            string value = string.Empty;
        }
    }
}
```

```

        //define regular expression pattern
        Regex match = new Regex("^([0-9]){3}[_]");
        try
        {
            filename = FolderSelection("Select place to store the variable in
formation");

            //create stream writer to the .txt file
            StreamWriter matchedVariables = new StreamWriter(filename, true);
            //run through all variables in zenon
            for (int i = 0; i < this.Variables().Count; i++)
            {
                //if name of the variable matches the pattern, get name, tag
name and current value
                if (match.IsMatch(this.Variables().Item(i).Name))
                {
                    name = this.Variables().Item(i).Name;
                    description = this.Variables().Item(i).Tagname;
                    value = this.Variables().Item(i).get_Value(0).ToString();
                    //write information to the .txt file
                    matchedVariables.WriteLine(name + ";" + description + ";"
+ value);
                }
            }
            //close stream
            matchedVariables.Close();
        }
        catch (Exception ex)
        {

            MessageBox.Show("An error occured -> " + ex.Message);
            throw;
        }
    }

    private string FolderSelection(String caption)
    {
        string selectedPath = string.Empty;
        //create a dialog for selecting the output folder

```

```

        FolderBrowserDialog folderSelect = new FolderBrowserDialog();
        folderSelect.Description = caption;
        try
        {
            if (folderSelect.ShowDialog() == DialogResult.OK)
            {
                //if selection was valid, get the current date, put it to file
                //date format
                //then create a txt file with the name "zenonVar" and the corresponding date

                DateTime currentTime = DateTime.Now;
                selectedPath = folderSelect.SelectedPath + "\\zenonVar" + currentTime.ToFileTime() + ".txt";
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("An error occurred: " + ex.Message);
            throw;
        }
        return selectedPath;
    }

    #region VSTA generated code
    private void InternalStartup()
    {
        this.Startup += new System.EventHandler(ThisProject_Startup);
        this.Shutdown += new System.EventHandler(ThisProject_Shutdown);
    }
    #endregion
}

```

## 6. Process Control Engine (PCE)



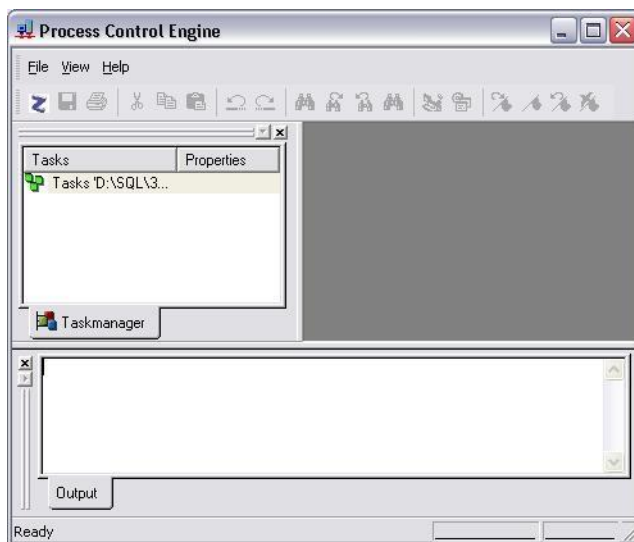
### Attention

*Starting from version 7.20, PCE will not be supported anymore and it will not be shown in the module tree of zenon anymore. While converting projects from versions lower than 7.20, which contain PCE tasks, the node PCE will be shown for these projects again. PCE will not further be developed and documented.*

*Recommendation: Please use **zenon Logic** instead of PCE*

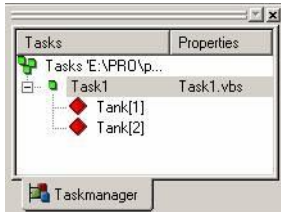
### 6.1 The PCE Editor

The PCE can be found in the Project Manager in the entry **Programming interfaces**. The PCE Editor is opened with the entry **Open PCE Editor** in the context menu.

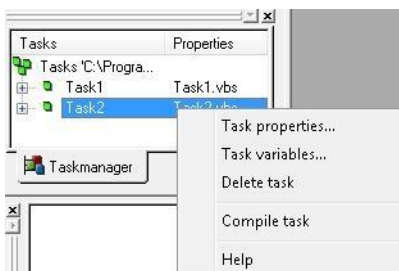


### 6.1.1 The Taskmanager

The Taskmanager of the PCE Editor lists the existing tasks and the linked variables.



A doubleclick on a task opens it in the editing area. With the right mouse button the context menu of a task can be opened.



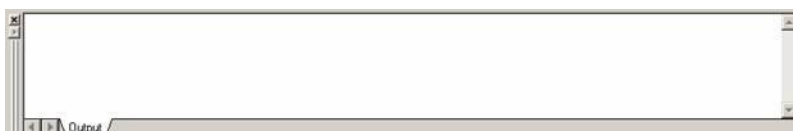
The context menu of a task has four entries:

Parameters	Description
<b>Task properties...</b>	Opens the properties dialog of the task.
<b>Task variables...</b>	Opens the variable selection. So you can add new variables to the task.
<b>Delete task</b>	Deletes the task without any further query.
<b>Compile tasks</b>	Compiles the task.

### 6.1.2 The editing area

In the editing area of the PCE Editor the code of the tasks is entered in VB Script or Java Script.

### 6.1.3 The output window



## 6.1.4 The menus of the PCE Editor

### Menu File

The menu **File** includes the following commands:

Parameters	Description
<b>Save</b>	Saves new or changed tasks.
<b>Print</b>	Prints the current task.
<b>Close</b>	Closes the PCE Editor.

### Menu Edit

The menu **Edit** includes the following commands:

Parameter	Description
<b>Undo</b>	Undoes the last executed action.
<b>Redo</b>	Repeats the last executed action.
<b>Cut</b>	Moves a text to the Windows Clipboard.
<b>Copy</b>	Copies a text to the Windows Clipboard.
<b>Paste</b>	Pastes a text from the Windows Clipboard.
<b>Delete</b>	
<b>Select all</b>	Selects the entire text of the task.
<b>Find</b>	Searches for a text in the current task.
<b>Find next</b>	Goes to the next place of finding.
<b>Find previous</b>	Goes to the previous place of finding.
<b>Replace...</b>	Replaces a text in the task by another.
<b>Bookmarks</b>	Administration of bookmarks in the code of the task.
<b>- set bookmark</b>	Sets a bookmark at the selected line in the code.
<b>- next bookmark</b>	Goes to the next bookmark in the code.
<b>- previous bookmark</b>	Goes to the previous bookmark in the code.
<b>- delete all bookmarks</b>	Deletes all bookmarks in the code.

## Menu Run

The menu **Run** includes the following commands:

<b>Save and restart all tasks</b>	
<b>Compile tasks</b>	Compiles the task.

## Menu View

The menu **View** includes the following commands:

Parameters	Description
<b>Options</b>	Opens the settings dialog of the PCE Editor.
<b>Task manager</b>	Opens/closes the Taskmanager window.
<b>Output</b>	Opens/closes the Output window.
<b>Status Bar</b>	Opens/closes the status bar.

## Menu Window

The menu **Window** includes the following commands:

<b>Close</b>
<b>Arrange</b>
<b>Divide</b>
<b>Align symbols</b>
<b>List of the last open windows</b>

## Menu help

The menu **Help** includes the following commands:

Command	Action
<b>Help</b>	Opens online help.
<b>Info about...</b>	<p>Opens a window with information on zenon:</p> <ul style="list-style-type: none"> <li>▶ Serial Number</li> <li>▶ Activation number</li> <li>▶ Licensed tags/IOs</li> <li>▶ Licensed module</li> </ul> <p>A slider can be used for navigation in the information window. Clicking in the window or pressing the <b>Esc</b> key closes the info window.</p>

### 6.1.5 The icon bar of the PCE Editor

The most important commands of the PCE Editor can also be executed with the icons of the icon bar.



The following icons are available:

Parameters	Description
<b>Close</b>	Closes the PCE Editor.
<b>Save all</b>	Saves new or changed tasks.
<b>Print active screen</b>	Prints the current task.
<b>Cut</b>	Moves a text to the Windows Clipboard.
<b>Copy</b>	Copies a text to the Windows Clipboard.
<b>Paste</b>	Pastes a text from the Windows Clipboard.
<b>Undo</b>	Undoes the last executed action.
<b>Redo</b>	Repeats the last executed action.
<b>Find</b>	Searches for a text in the current task.
<b>Find next</b>	Goes to the next place of finding.
<b>Find previous</b>	Goes to the previous place of finding.
<b>Replace...</b>	Replaces a text in the task by another.
<b>Save and restart</b>	
<b>Start debugger</b>	
<b>Next bookmark</b>	Goes to the next bookmark in the code.
<b>Set bookmark</b>	Sets a bookmark at the selected line in the code.
<b>Previous bookmark</b>	Goes to the previous bookmark in the code.
<b>Delete all bookmarks</b>	Deletes all bookmarks in the code.

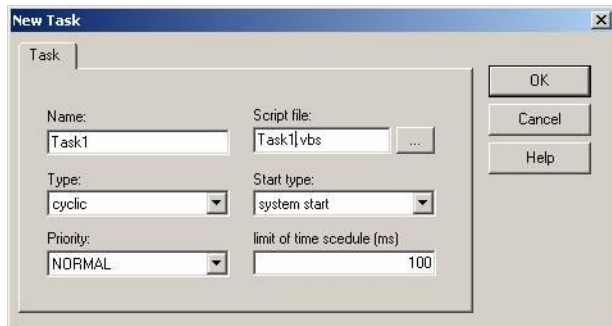
## 6.2 Course of actions

### 6.2.1 Creating a task

With the context menu of the Taskmanager a new task can be created.

## Properties of the task

After creating the task the properties dialog of the new task opens automatically.



The following properties can be defined:

Parameters	Description
<b>Name</b>	Unique name of the task.
<b>Type</b>	Tasks can be executed <b>cyclic</b> or <b>once</b> . Cyclic: the task is executed cyclically in the interval defined under <b>limit of time schedule</b> . Once: the task is executed one single time.
<b>Priority</b>	Process priorities for operating system multithreading ( <b>idle, low, normal, high, highest, time critical</b> ). Default: Normal main process: the task runs in the same thread as Runtime. If the task gets into a waiting loop or crashes, that also influences the Runtime.
<b>Script file</b>	Selection of the script file: VB-Files (*.vbs) for VB Script or JS-Files (*.js) for Java Script. The according file is created, when the task is opened in the editing area for the first time.
<b>Start type</b>	System start : automatically started with Runtime. (This is the only way to use the PCE under Windows CE, as Windows CE does not support VBA.) event triggered: the task is started in a VBA macro with the statement "thisProject.Tasks.Item("Taskname").Run".
<b>Cycle time to reach</b>	For cyclic tasks the interval in milliseconds that should be achieved. If this cycle time is not achieved, the task is executed as fast as possible.

For a later change of the properties this dialog can also be opened with the context menu of the task and the entry **Task Properties...**

## Variables of the task

After defining the properties the variable selection dialog is automatically opened. Here the variables that should be processed in the task are selected.

All variables that are read or written in the task should be linked here. There is also the possibility to access the variables via the variables object, but only the variables directly linked to the task are automatically updated when initializing the task before execution.

The variables must have the following syntax:

```
Task.Value('Variable name')=123
```

For a later change of the variable selection this dialog can also be opened with the context menu of the task and the entry **Task Variables....**

### 6.2.2 Entering code

Double-clicking the task in the Taskmanager opens it in the editing area. If the task is opened for the first time, the according VBS or JS file is created now.

Four procedures are automatically created:

Parameters	Description
<b>Task_Init()</b>	This procedure is automatically executed when starting the task.
<b>Task_Main()</b>	This procedure is either executed once (type once) or cyclically (type cyclic).
<b>Task_Exit()</b>	This procedure is automatically executed when stopping the task.
<b>Task_Timer(ITimerId)</b>	This procedure is executed cyclically, as long as the according time is running. The cycle time is defined as a parameter with the starting of the timer.

Generally speaking the PCE uses the same object model as VBA (see VBA Tutorials). When using VBA objects (except the object Task) multithreading is lost, because these objects only can be accessed from the main thread.



#### Attention

*Not all functions of the COM interface are multithread-able and therefore can only be used in a main thread context. If a different property than "in the main process" is set as PCE task, there must not be any access from the PCE to the main thread. In case there is an access to the COM interface nevertheless, this can lead to undefined system states, e.g. a Runtime freeze.*

Of special importance are the collection Tasks and the object **Task**.

## The collection Tasks

Count
Item
Parent

## The object Task

ActualCycleTime	Property	Currently achieved cycle time of the task
CountVariable	Property	Number of variables linked to the task
CycleTime	Property	Defined cycle time of the task
DynProperties	Property	
ErrorNumber	Property	
ErrorString	Property	
Exit	Event	
On init	Event	
ItemVariable	Method	
Main	Event	
MemValue	Property	<p>With "Task.MemValue("Name")=value" an internal variable is created and a value is assigned to it. There is no need to declare the variable before.</p> <p>This variable can also be accessed from other tasks. So it allows the exchange of values between tasks.</p>
Name	Property	Name of the current task
Parent	Property	The collection <b>Tasks</b>
Priority	Property	Priority of the current task
Run	Method	Starts a task
Sleep	Method	Holds a task
StartTimer	Method	The method "StartTimer" starts a timer of the task.
Status	Property	
Stop	Method	Stops a task
StopTimer	Method	The method "StopTimer" stops a timer of the task.
Timer	Event	
Type	Property	
Value	Property	With "Task.Value("name of linked variable")=value" a variable of the project can get a new value.

### 6.2.3 Function Show PCE

With the zenon function **Show PCE** the PCE Editor can be opened from the Runtime.

## 6.2.4 Executing tasks

Tasks can be executed when the system is started or can be event-triggered.

### Executing tasks with system start

If in the configuration of the task the **Start type** is set to `System start`, the task is automatically started with the Runtime.

This is the only way to use the PCE under Windows CE, as Windows CE does not support VBA.

### Executing tasks event triggered

#### On a PC

A task can also be started event triggered. In this case the **Start type** has to be set to `Event driven`. Now the task is no longer automatically started with the Runtime.

A VBA macro has to be created in order to execute a task by pressing a button, by a limit value violation or any other event. With the following VBA statement the task can be started:

```
thisProject.Tasks.Item(Taskname).Run
```

The task is automatically started in an own thread if in the configuration **Priority Main process** has not been set.

With the following VBA statement the task can be stopped at any time:

```
thisProject.Tasks.Item("Taskname").Stop
```

**Note:** To execute a task more than once, it must also be explicitly stopped after it has ended, so that it can be restarted.

Instruction: `thisProject.Tasks.Item(Taskname).Stop`

It can be restarted after being stopped properly.

#### On a CE terminal

As Windows CE does not support VBA, the procedure for execution on a PC is not possible on a CE terminal. But there is a possibility to execute tasks event triggered also here.

A task with the **Start type** `System start` is created. This task is automatically started with the Runtime. And this task gets the **Priority Main process** so that it runs in the same thread as the Runtime. Now bit variables are linked to that task, then will execute other tasks event triggered. With the following statement the task can be started: `Parent.Item(Taskname).Run`

Now the task is automatically started in an own Thread if in the configuration the **Priority Main process** has not been set.

With the following statement the task can be stopped at any time: `Parent.Item("Taskname").Stop`

## 6.3 VB Script - Introduction

### 6.3.1 Data types

#### Variant

VBScript has only one data type called a **Variant**. A **Variant** is a special kind of data type that can contain different kinds of information, depending on how it is used. Because **Variant** is the only data type in VBScript, it is also the data type returned by all functions in VBScript.

At its simplest, a **Variant** can contain either numeric or string information. A **Variant** behaves as a number when you use it in a numeric context and as a string when you use it in a string context. That is, if you are working with data that looks like numbers, VBScript assumes that it is numbers and does what is most appropriate for numbers. You can always make numbers behave as strings by enclosing them in quotation marks (" "). If you work with data that only can be interpreted as strings, VBScript will interpret them as strings.

#### Variant Subtypes

Beyond the simple numeric or string classifications, a **Variant** can make further distinctions about the specific nature of numeric information. For example, you can have numeric information that represents a date or a time. When used with other date or time data, the result is always expressed as a date or a time. You can also have a rich variety of numeric information ranging in size from Boolean values to huge floating-point numbers. These different categories of information which can be contained in a **Variant** are called **subtypes**. Most of the time, you can just put the kind of data you want in a **Variant**, and the Variant behaves in a way that is most appropriate for the data it contains.

The following summary shows subtypes of data that a **Variant** can contain.

Subtype	Meaning
<b>Empty</b>	Variant is uninitialized. Value is 0 for numeric variables or a zero-length string ("" ) for string variables.
<b>Null</b>	Variant intentionally contains no valid data.
<b>Boolean</b>	Contains either TRUE or FALSE.
<b>Byte</b>	Contains integer in the range 0 to 255.
<b>Integer</b>	Contains integer in the range -32,768 to 32,767.
<b>Currency</b>	-922,337,203,685,477.5808 bis 922,337,203,685,477.5807.
<b>Long</b>	Contains integer in the range -2,147,483,648 to 2,147,483,647.
<b>Single</b>	Contains a single-precision, floating-point number in the range -3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.
<b>Double</b>	Contains a double-precision, floating-point number in the range -1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.
<b>Date (Time)</b>	Contains a number that represents a date between January 1, 100 to December 31, 9999.
<b>String</b>	Contains a variable-length string that can be up to approximately 2 billion characters in length.
<b>Object</b>	Contains an object.
<b>Error</b>	Contains an error number.

### 6.3.2 Variables

A variable is a convenient placeholder that refers to a computer memory location where you can store program information that may change during the time your script is running. For example, you might create a variable called ClickCount to store the number of times a user clicks an object on a particular Web page. Where the variable is stored in computer memory is unimportant. What is important is that you only have to refer to a variable by name to see or change its value. In VBScript, variables are always of one fundamental data type, Variant.

#### Declaring Variables

You declare variables explicitly in your script using the Dim statement, the Public statement, and the Private statement. Example:

```
Dim DegreesFahrenheit
```

You declare multiple variables by separating each variable name with a comma. Example:

```
Dim Top, Bottom, Left, Right
```

## Limitations for names

Variable names follow the standard rules for naming anything in VBScript. A variable name:

- ▶ Must begin with an alphabetic character.
- ▶ Cannot contain an embedded period.
- ▶ Must not exceed 255 characters.
- ▶ Must be unique in the scope in which it is declared.

## Scope and Lifetime of Variables

When you declare a variable within a procedure, only code within that procedure can access or change the value of that variable. It has local scope and is a procedure-level variable.

If you declare a variable outside a procedure, you make it recognizable to all the procedures in your script. This is a script-level variable, and it has script-level scope.

The lifetime of a variable depends on how long it exists. The lifetime of a script-level variable extends from the time it is declared until the time the script is finished running. At procedure level, a variable exists only as long as you are in the procedure. When the procedure exits, the variable is destroyed.

Local variables are ideal as temporary storage space when a procedure is executing. You can have local variables of the same name in several different procedures because each is recognized only by the procedure in which it is declared.

## Assigning Values to Variables

Values are assigned to variables creating an expression as follows: the variable is on the left side of the expression and the value you want to assign to the variable is on the right. Example:

```
B = 200
```

## Scalar Variables and Array Variables

Much of the time, you only want to assign a single value to a variable you have declared. A variable containing a single value is a scalar variable. Other times, it is convenient to assign more than one related value to a single variable. Then you can create a variable that can contain a series of values. This

is called an array variable. Array variables are declared nearly like scalar variables. The only difference is, that in the declaration brackets follow the names of array variables. In the following example, a single-dimension array containing 11 elements is declared:

```
Dim A(10)
```

Although the number shown in the parentheses is 10, all arrays in VBScript are zero-based, so this array actually contains 11 elements. In a zero-based array, the number of array elements is always the number shown in parentheses plus one. This kind of array is called a fixed-size array.

You assign data to each of the elements of the array using an index into the array. Beginning at zero and ending at 10, data can be assigned to the elements of an array as follows:

```
A(0) = 256
A(1) = 324
A(2) = 100
. . .
A(10) = 55
```

Similarly, the data can be retrieved from any element using an index into the particular array element you want. Example:

```
. . .
SomeVariable = A(8)
. . .
```

Arrays aren't limited to a single dimension. You can have as many as 60 dimensions, although most people can't comprehend more than three or four dimensions. You can declare multiple dimensions by separating an array's size numbers in the parentheses with commas. In the following example, the `MyTable` variable is a two-dimensional array consisting of 6 rows and 11 columns:

```
Dim MyTable(5, 10)
```

In a two-dimensional array, the first number is always the number of rows; the second number is the number of columns.

You can also declare an array whose size changes during the time your script is running. This is called a dynamic array. The array is initially declared within a procedure using either the `Dim` statement or using the `ReDim` statement. However, for a dynamic array, no size or number of dimensions is placed inside the parentheses. Example:

```
(Dim AnArray()
ReDim AnotherArray()
```

To use a dynamic array, you must subsequently use `ReDim` to determine the number of dimensions and the size of each dimension. In the following example, `ReDim` sets the initial size of the dynamic array to 25. A subsequent `ReDim` statement resizes the array to 30, but uses the `Preserve` keyword to preserve the contents of the array as the resizing takes place.

```
ReDim MyArray(25)
. . .
```

```
ReDim Preserve MyArray(30)
```

### 6.3.3 Constants

A constant is a meaningful name that takes the place of a number or string and never changes. VBScript defines a number of intrinsic constants. You can get information about these intrinsic constants from the VBScript Language Reference.

You create user-defined constants in VBScript using the Const statement. So you can assign a meaningful name to string or numerical constants. Then you can assign them literal values and use them in a script. Example:

```
Const MyString = "This is a string."
```

```
Const MyAge = 49
```

Note that the string literal is enclosed in quotation marks (" "). Quotation marks are the most obvious way to differentiate string values from numeric values. You represent Date literals and time literals by enclosing them in number signs (#). Example:

```
Const CutoffDate = #6-1-97#
```

You may want to adopt a naming scheme to differentiate constants from variables. This will prevent you from trying to reassign constant values while your script is running. For example, you might want to use a "vb" or "con" prefix on your constant names, or you might name your constants in all capital letters. Care that constants and variables can be distinguished. So you avoid problems when creating complex scripts.

### 6.3.4 Operators

VBScript has a full range of operators, including arithmetic operators, comparison operators, concatenation operators, and logical operators.

#### Operator Precedence

If several operators appear in a statement, each part is evaluated and resolved in a pre-defined sequence. This sequence is called operator precedence. You can use parentheses to override the order of precedence and force some parts of an expression to be evaluated before others. Operations within parentheses are always performed before those outside. Within parentheses, however, standard operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence.

## Arithmetic Operators

Description	Symbol
Exponentiation	$\wedge$
Unary negation	-
Multiplication	*
Division	/
Integer division	/
Modulus arithmetic	Mod
Addition	+
Subtraction	-
String concatenation	

## Comparison Operators

Description	Symbol
Equality	=
Inequality	<>
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Object equivalence	Is

## Logical Operators

If several operators appear in a statement, each part is evaluated and resolved in a pre-defined sequence. This sequence is called operator precedence. You can use parentheses to override the order of precedence and force some parts of an expression to be evaluated before others. Operations within

parentheses are always performed before those outside. Within parentheses, however, standard operator precedence is maintained.

When expressions contain operators from more than one category, arithmetic operators are evaluated first, comparison operators are evaluated next, and logical operators are evaluated last. Comparison operators all have equal precedence; that is, they are evaluated in the left-to-right order in which they appear. Arithmetic and logical operators are evaluated in the following order of precedence.

### 6.3.5 Conditional Statements

You can control the flow of your script with conditional statements and looping statements. Using conditional statements, you can write VBScript code that makes decisions and repeats actions.

#### Making Decisions Using If...Then...Else

The `If...Then...Else` statement is used to evaluate whether a condition is `True` or `False` and, depending on the result, to specify one or more statements to run. Usually the condition is an expression that uses a comparison operator to compare one value or variable with another. For information about comparison operators, see [Comparison Operators](#). `If...Then...Else` statements can be nested to as many levels as you need.

#### Running Statements if a Condition is True

To run only one statement when a condition is `True`, use the single-line syntax for the `If...Then...Else` statement. The following example shows the single-line syntax. Notice that this example omits the `Else` keyword.

```
Sub FixDate()  
Dim myDate  
myDate = #2/13/95#  
If myDate < Now Then myDate = Now  
End Sub
```

To run more than one line of code, you must use the multiple-line (or block) syntax. This syntax includes the `End If` statement, as shown in the following example:

```
Sub AlertUser(value)  
If value = 0 Then  
AlertLabel.ForeColor = vbRed  
AlertLabel.Font.Bold = True  
AlertLabel.Font.Italic = True  
End If  
End Sub
```

To run only one statement when a condition is `True`, use the single-line syntax for the `If...Then...Else` statement. The following example shows the single-line syntax. Notice that this example omits the `Else` keyword.

```
Sub FixDate()
Dim myDate
myDate = #2/13/95#
If myDate < Now Then myDate = Now
End Sub
```

To run more than one line of code, you must use the multiple-line (or block) syntax. This syntax includes the `End If` statement, as shown in the following example:

```
Sub AlertUser(value)
If value = 0 Then
AlertLabel.ForeColor = vbRed
AlertLabel.Font.Bold = True
AlertLabel.Font.Italic = True
End If
End Sub
```

## Running Certain Statements if a Condition is True and Running Others if a Condition is False

You can use an `If...Then...Else` statement to define two blocks of executable statements: one block to run if the condition is `True`, the other block to run if the condition is `False`.

```
Sub AlertUser(value)
If value = 0 Then
AlertLabel.ForeColor = vbRed
AlertLabel.Font.Bold = True
AlertLabel.Font.Italic = True
Else
AlertLabel.ForeColor = vbBlack
AlertLabel.Font.Bold = False
AlertLabel.Font.Italic = False
End If
End Sub
```

## Deciding Between Several Alternatives

A variation on the `If...Then...Else` statement allows you to choose from several alternatives. Adding `ElseIf` clauses expands the functionality of the `If...Then...Else` statement so you can control program flow based on different possibilities.

Example:

```
Sub ReportValue(value)
If value = 0 Then
MsgBox value
```

```

ElseIf value = 1 Then
MsgBox value
ElseIf value = 2 then
Msgbox value
Else
Msgbox Walue out of range!
End If

```

## Making Decisions with Select Case

The `Select Case` structure provides an alternative to `If...Then...ElseIf` for selectively executing one block of statements from among multiple blocks of statements. A `Select Case` statement provides capability similar to the `If...Then...Else` statement, but it makes code more efficient and readable.

A `Select Case` structure works with a single test expression that is evaluated once, at the top of the structure. The result of the expression is then compared with the values for each `Case` in the structure. If there is a match, the block of statements associated with that `Case` is executed, as in the following example.

```

Select Case Document.Form1.CardType.Options(SelectedIndex).Text
Case MasterCard
DisplayMCLogo
ValidateMCAccount
Case Visa
DisplayVisaLogo
ValidateVisaAccount
Case American Express
DisplayAMEXCOLogo
ValidateAMEXCOAccount
Case Else
DisplayUnknownImage
PromptAgain
End Select

```

### 6.3.6 Looping Through Code

Looping allows you to run a group of statements repeatedly. Some loops repeat statements until a condition is `False`; others repeat statements until a condition is `True`. There are also loops that repeat statements a specific number of times.

The following looping statements are available in VBScript:

Parameters	Description
Using <code>Do</code> loops (on page 123):	Loops while or until a condition is <code>True</code> .

Using While...Wend (on page 125):	Loops while a condition is True.
Using For...Next (on page 125):	Uses a counter to run statements a specified number of times.
Using For Each...Next (on page 126):	Repeats a group of statements for each item in a collection or each element of a

## Using Do Loops

You can use `Do...Loop` statements to run a block of statements an indefinite number of times. The statements are repeated either while a condition is `True` or until a condition becomes `True`.

### Repeating Statements While a Condition is True

Use the `While` keyword to check a condition in a `Do...Loop` statement. You can check the condition before you enter the loop (as shown in the following `ChkFirstWhile` example), or you can check it after the loop has run at least once (as shown in the `ChkLastWhile` example). In the `ChkFirstWhile` procedure, if `myNum` is set to 9 instead of 20, the statements inside the loop will never run. In the `ChkLastWhile` procedure, the statements inside the loop run only once because the condition is already `False`.

```
Sub ChkFirstWhile()
    Dim counter, myNum
    counter = 0
    myNum = 20
    Do While myNum > 10
        myNum = myNum - 1
        counter = counter + 1
    Loop
    MsgBox 'The loop made ' & counter & ' repetitions.'
End Sub
```

```
Sub ChkLastWhile()
    Dim counter, myNum
    counter = 0
    myNum = 9
    Do
        myNum = myNum - 1
        counter = counter + 1
    Loop While myNum > 10
```

```

    MsgBox 'The loop made ' & counter & ' repetitions.'
End Sub

```

## Repeating a Statement Until a Condition Becomes True

There are two ways to use the `Until` keyword to check a condition in a `Do...Loop` statement. You can check the condition before you enter the loop (as shown in the following `ChkFirstUntil` example), or you can check it after the loop has run at least once (as shown in the `ChkLastUntil` example). As long as the condition is `False`, the looping occurs.

```

Sub ChkFirstUntil()
    Dim counter, myNum
    counter = 0
    myNum = 20
    Do Until myNum = 10
        myNum = myNum - 1
        counter = counter + 1
    Loop
    MsgBox 'The loop made ' & counter & ' repetitions.'
End Sub

```

```

Sub ChkLastUntil()
    Dim counter, myNum
    counter = 0
    myNum = 1
    Th
        myNum = myNum - 1
        counter = counter + 1
    Loop Until myNum = 10
    MsgBox 'The loop made ' & counter & ' repetitions.'
End Sub

```

## Exiting a Do...Loop Statement from Inside the Loop

You can exit a `Do...Loop` by using the `Exit Do` statement. Because you usually want to exit only in certain situations, such as to avoid an endless loop, you should use the `Exit Do` statement in the `True` statement block of an `If...Then...Else` statement. If the condition is `False`, the loop runs as usual.

In the following example, myNum is assigned a value that creates an endless loop. The If...Then...Else statement checks for this condition, preventing the endless repetition.

```
Sub ExitExample()
    Dim counter, myNum
    counter = 0
    myNum = 9
    Do Until myNum = 10
        myNum = myNum - 1
        counter = counter + 1
        If myNum < 10 Then Exit Do
    Loop
    MsgBox 'The loop made ' & counter & ' repetitions.'
End Sub
```

## Using While...Wend

The While...Wend statement is provided in VBScript for those who are familiar with its usage. However, because of the lack of flexibility in While...Wend, it is recommended that you use Do...Loop instead.

## Using For...Next

You can use For...Next statements to run a block of statements a specific number of times. For loops, use a counter variable whose value increases or decreases with each repetition of the loop.

The following example causes a procedure called MyProc to execute 50 times. The For statement specifies the counter variable x and its start and end values. The Next statement increments the counter variable by 1.

```
Sub DoMyProc50Times()
    Dim x
    For x = 1 To 50
        MyProc
    Next
End Sub
```

Using the Step keyword, you can increase or decrease the counter variable by the value you specify. In the following example, the counter variable j is incremented by 2 each time the loop repeats. When the loop is finished, the total is the sum of 2, 4, 6, 8, and 10.

```
Sub DoMyProc50Times()
    Dim x
    For x = 1 To 50
```

```

        MyProc
    Next
End Sub

```

To decrease the counter variable, use a negative `Step` value. You must specify an end value that is less than the start value. In the following example, the counter variable `myNum` is decreased by 2 each time the loop repeats. When the loop is finished, `total` is the sum of 16, 14, 12, 10, 8, 6, 4, and 2.

```

Sub NewTotal ()
    Dim myNum, total
    For myNum = 16 To 2 Step -2
        total = total + myNum
    Next
    MsgBox 'The total is ' & total
End Sub

```

## Using For Each...Next

A `For Each...Next` loop is similar to a `For...Next` loop. Instead of repeating the statements a specified number of times, a `For Each...Next` loop repeats a group of statements for each item in a collection of objects or for each element of an array. This is especially helpful if you don't know how many elements are in a collection.

In the following HTML code example, the contents of a `Dictionary` object is used to place text in several text boxes.

```

<HTML>
<HEAD><TITLE>Formulare und Elemente</TITLE></HEAD>
<SCRIPT LANGUAGE='VBScript'>
<!--
Sub cmdChange_OnClick
    Dim d                'Create a variable
    Set d = CreateObject('Scripting.Dictionary')
    d.Add '0', 'Athen'    'Add some keys and items
    d.Add '1', 'Belgrad'
    d.Add '2', 'Kairo'

    For Each I in d
        Document.frmForm.Elements(I).Value = D.Item(I)
    Next
End Sub

```

```
-->
</SCRIPT>
<BODY>
<CENTER>
<FORM NAME='frmForm'

<Input Type = 'Text'><p>
<Input Type = 'Text'><p>
<Input Type = 'Text'><p>
<Input Type = 'Text'><p>
<Input Type = 'Button' NAME='cmdChange' VALUE='Hierauf klicken'><p>
</FORM>
</CENTER>
</BODY>
</HTML>
```

### 6.3.7 Types of procedures

#### Sub Procedures

A **Sub** procedure is a series of VBScript statements (enclosed by **Sub** and **End Sub** statements) that perform actions but don't return a value. A **Sub** procedure can take arguments (constants, variables, or expressions that are passed to it by a calling procedure). If a **Sub** procedure has no arguments, its **Sub** statement must include an empty set of parentheses ().

The following **Sub** procedure uses two intrinsic, or built-in, VBScript functions, **MsgBox** and **InputBox**, to prompt a user for information. It then displays the results of a calculation based on that information. The calculation is performed in a **Function** procedure created using VBScript. The **Function** procedure is shown after the following discussion.

```
Sub ConvertTemp()
    temp = InputBox('Please enter the temperature in degrees F.', 1)
    MsgBox 'The temperature is ' & Celsius(temp) & ' degrees C.'
End Sub
```

#### Function Procedures

A `Function` procedure is a series of VBScript statements enclosed by the `Function` and `End Function` statements. A `Function` procedure is similar to a `Sub` procedure, but can also return a value. A `Function` procedure can take arguments (constants, variables, or expressions that are passed to it by a calling procedure). If a `Function` procedure has no arguments, its `Function` statement must include an empty set of parentheses. A `Function` returns a value by assigning a value to its name in one or more statements of the procedure. The return type of a `Function` is always a `Variant`.

In the following example, the `Celsius` function calculates degrees Celsius from degrees Fahrenheit. When the function is called from the `ConvertTemp` `Sub` procedure, a variable containing the argument value is **passed to** the function. The result of the calculation is returned to the calling procedure and displayed in a message box.

```
Sub ConvertTemp()
    temp = InputBox('Please enter the temperature in degrees F.', 1)
    MsgBox 'The temperature is ' & Celsius(temp) & ' degrees C.'
End Sub

Function Celsius(fDegrees)
    Celsius = (fDegrees - 32) * 5 / 9
End Function
```

## Getting data into or out of procedures

Each piece of data is passed into your procedures using an argument . Arguments serve as placeholders for the data you want to pass into your procedure. When you create a procedure using either the `Sub` statement or the `Function` statement, parentheses must be included after the name of the procedure. Any arguments are placed inside these parentheses, separated by commas. For example, in the following example, **fDegrees** is a placeholder for the value being passed into the `Celsius` function for conversion.

```
Function Celsius(fDegrees)
    Celsius = (fDegrees - 32) * 5 / 9
End Function
```

## Using Sub and Function Procedures in Code

A `Function` in your code must always be used on the right side of a variable assignment or in an expression.

Examples:

```
Temp = Celsius(fDegrees)
```

or

```
MsgBox 'The temperature is ' & Celsius(temp) & ' degrees C.'
```

To call a `Sub` procedure from another procedure, type the name of the procedure along with values for any required arguments, each separated by a comma. The `Call` statement is not required, but if you do use it, you must enclose any arguments in parentheses.

The following example shows two calls to the `MyProc` procedure. In the one case the `Call` statement is used in the code, in the other one it is not. Both calls have the same result.

```
Call MyProc(firstarg, secondarg)
MyProc firstarg, secondarg
```

### 6.3.8 Coding Conventions

Coding conventions are suggestions are designed to help you write code using Microsoft Visual Basic Scripting Edition.

Coding conventions can include the following:

Naming conventions for objects, variables, and procedures
Commenting conventions
Text formatting and indenting guidelines

The main reason for using a consistent set of coding conventions is to standardize the structure and coding style of a script or set of scripts so that you and others can easily read and understand the code. Using good coding conventions results in clear, precise, and readable source code that is consistent with other language conventions and is intuitive.

#### Constant Naming Conventions

Earlier versions of VBScript had no mechanism for creating user-defined constants. Constants, if used, were implemented as variables and distinguished from other variables using all uppercase characters. Multiple words were separated using the underscore (`_`) character.

Examples:

```
USER_LIST_MAX
NEW_LINE
```

Although this way of naming constants still works, you can use a different way of naming. You can create real constants with the statement `Const`. This convention uses a mixed-case format in which constant names have a "con" prefix.

For example:

```
conYourOwnConstant
```

## Variable Naming Conventions

To enhance readability and consistency, use the following summary with descriptive names for variables in your VBScript code.

Subtype	Prefix	Example
Boolean	bln	blnFound
Byte	byt	bytRasterData
Date (Time)	dtm	dtmStart
Double	dbl	dblTolerance
Error	err	errOrderNum
Integer	int	intQuantity
Long	lng	lngDistance
Object	obj	objCurrent
Single	sng	sngAverage
String	str	strFirstName

## Variable Scope

Variables should always be defined with the smallest scope possible. VBScript variables can have the following scope.

Valid range	Declaration	Visibility
Procedure-level	Event, Function, or Sub procedure.	Visible in the procedure in which it is declared.
Script-level	HEAD section of an HTML page, outside any procedure.	Visible in every procedure in the script.

## Variable Scope Prefixes

As script size grows, so does the value of being able to quickly differentiate the scope of variables. A one-letter scope prefix preceding the type prefix provides this, without unduly increasing the size of variable names.

Valid range	Prefix	Example
Procedure-level	None	dblVelocity
Script-level	sec	sbInCalcInProgress

## Descriptive Variable and Procedure Names

In the core of a variable or procedure name also capitals should be used. The name should be long enough to describe the use of the variable. In addition, procedure names should begin with a verb, such as InitNameArray or CloseDialog.

For frequently used or long terms, standard abbreviations are recommended to help keep name length reasonable. In general, variable names greater than 32 characters can be difficult to read. When using abbreviations, make sure they are consistent throughout the entire script. For example, randomly switching between Cnt and Count within a script or set of scripts may lead to confusion.

## Object Naming Conventions

The following table lists recommended conventions for objects you may encounter while programming VBScript.

Object type	Prefix	Example
3D Panel	pnl	pnlGroup
Animated button	ani	aniMailBox
Check box	chk	chkReadOnly
Combo box, drop-down list box	cbo	cboEnglish
Command button	cmd	cmdExit
Common dialog	dlg	dlgFileOpen
Frame	fra	fraLanguage
Horizontal scroll bar	hsb	hsbVolume
Image	img	imgIcon
Label	lbl	lblHelpMessage
Line	lin	linVertical
List Box	lst	lstPolicyCodes
Spin	spn	spnPages
Text box	txt	txtLastName
Vertical scroll bar	vsb	vsbRate
Slider	sld	sldScale

## Code Commenting Conventions

Each procedure should start with a short comment describing the purpose of the procedure. This description should not go into implementation details (how operations are executed), because these might change with the time. This could result in maintenance effort for the comments and - even worse - wrong comments. The code itself and any necessary inline comments describe the implementation.

Arguments passed to a procedure should be described when their purpose is not obvious and when the procedure expects the arguments to be in a specific range. Return values for functions and variables that are changed by a procedure, especially through reference arguments, should also be described at the beginning of each procedure.

Procedure header comments should include the following section headings. For examples, see the "Formatting Your Code" section that follows.

Section Heading	Comment
Purpose	What the procedure does (not how).
Assumptions	List of the procedure's effect on each external variable, control, or other element.
Effects	List of the procedure's effect on each external variable, control, or other element.
Inputs	Explanation of each argument that is not obvious. Each argument should be on a separate line with inline comments.
Return Values	Explanation of the value returned.

Remember the following points:

Every important variable declaration should include an inline comment describing the use of the variable being declared.

Variables, controls, and procedures should be named clearly to ensure that inline comments are only needed for complex implementation details.

At the beginning of your script, you should include an overview that describes the script, enumerating objects, procedures, algorithms, dialog boxes, and other system dependencies. Sometimes a piece of pseudocode describing the algorithm can be helpful.

## Code formatting

Screen space should be conserved as much as possible, while still allowing code formatting to reflect logic structure and nesting. Here are a few suggestions:

- ▶ Indent standard nested blocks four spaces.
- ▶ Indent the overview comments of a procedure one space.
- ▶ The statements on the highest level, directly following the overview comment, should be indented with four blanks. Each nested block should again be indented by four blanks.

Example:

The following code adheres to VB Script coding conventions.

- ▶ ' Purpose: Searches for the first appearance of the stated user in the data field UserList.
- ▶ Inputs: strUserList(): the list of users to be searched.
- ▶ strZielUser: the name of the user to search for.
- ▶ Return values: Index of the first appearance of strTargetUser in the data field strUserList. If the target user is not found, return -1. -1.

```
Function intFindUser (strUserList(), strTargetUser)
    Dim i                ' Loop counter.
```

```
Dim blnFound           ' 'Target found' flag.
intFindUser = -1
i = 0                  ' Initialize loop counter
Do While i <= Ubound(strUserList) and Not blnFound
    If strUserList(i) = strTargetUser Then
        blnFound = True    ' Set flag to True
        intFindUser = i    ' Set return value to loop count
    End If
    i = i + 1            ' Increment loop counter
Loop
End Function
```