zenon Handbuch

Programmierschnittstellen

v.7.60

©2017 Ing. Punzenberger COPA-DATA GmbH

Alle Rechte vorbehalten.

Die Weitergabe und Vervielfältigung dieses Dokuments ist - gleich in welcher Art und Weise - nur mit schriftlicher Genehmigung der Firma COPA-DATA gestattet. Technische Daten dienen nur der Produktbeschreibung und sind keine zugesicherten Eigenschaften im Rechtssinn. Änderungen - auch in technischer Hinsicht - vorbehalten.

Inhaltsverzeichnis

1. Willkommen bei der COPA-DATA Hilfe 6

2. Programmierschnittstellen 6

3. Add-Ins 8

3.1 Vorkenntnisse 10

3.2 Terminologie 10

3.3 Einschränkungen 12

3.4 Add-In erstellen 12

3.4.1 Grundlagen 13

3.4.2 Typen von Extensions 17

3.4.3 Inhalt von Add-In Packages 19

3.4.4 SharpDevelop 20

3.4.5 Microsoft Visual Studio 23

3.4.6 Source Code Verwaltung 24

3.4.7 Add-In Analysis und Packaging Utility (AddInUtility) 25

3.4.8 Verhalten beim Nachladen 26

3.4.9 Isolierung 26

3.5 Verwenden von Add-Ins in Editor und Runtime 26

3.5.1 Add-Ins in zenon aktivieren 28

3.5.2 Verhalten bei der Installation 28

3.5.3 Knoten Add-Ins im Projektmanager 29

3.5.4 Verwendung im zenon Editor 30

3.5.5 Verwendung in der zenon Runtime 36

3.6 Umstieg/Konvertierung von VSTA 38

4. Makroliste 39

4.1 VBA Symbolleiste und Kontextmenü Detailansicht 40

4.2 VBA auf 64-Bit Systemen 44

4.3 Grundlagen 44

4.3.1 Objekteigenschaften - PROPERTIES 44

4.3.2 Objektmethoden - METHODS 45

4.3.3 Objektereignisse - EVENTS 45

4.3.4 VBA Objektmodell in zenon ... 46
4.3.5 Wie setzt man VBA Makros ein? ... 48
4.3.6 Wie bindet man ein ActiveX - Element in zenon ein? ... 50
4.3.7 Zugriff über externes Programm ... 51
4.3.8 Funktionsweise einer OnlineVariable ... 52
4.3.9 Liste Status-Bits ... 55
4.3.10 Lasso für Auswahl Dynamischer Elemente zur Runtime ... 57
4.4 Makros im Editor ... 58
4.4.1 Symbolleiste Makroliste ... 59
4.4.2 Zuordnung der Makros ... 61
4.5 Funktionen in zenon ... 62
4.5.1 VBA-Makro ausführen ... 63
4.6 Entwicklung von Wizards in VBA ... 64
4.6.1 Verwendung eines Wizards ... 65
4.6.2 Aufbau eines Wizards ... 66
4.6.3 Integration in VBA ... 66
4.6.4 Entwicklung eines Wizards ... 66
4.6.5 Aktualisierung von Wizards ... 72
4.7 Frequently asked Questions ... 73
4.7.1 Warum bleibt der Button gedrückt? ... 73
4.7.2 Makro wird nicht beim 1. Klick ausgeführt ... 73
4.7.3 Makros funktionieren in der RT nicht mehr? ... 74
4.7.4 Windows CE und VBA ... 74
4.8 Beispiele ... 74
4.8.1 MouseEvents and ActiveX Control initialization ... 74
4.8.2 Variableninformationen anzeigen ... 75
4.8.3 Variablenwerte lesen und schreiben ... 76
4.8.4 Variablen lesen und schreiben und Online Variablen implementieren ... 77
4.8.5 Dialog mehrfach nutzen ... 79
4.8.6 Alarm – Events and ActiveX Control handling ... 81
4.8.7 Zugriff auf Alarme ... 83
4.8.8 Schalter setzen (Arbeiten mit Prozessvariablen) ... 85

5. VSTA ... 88
5.1 Grundlagen ... 89
5.1.1 Einrichten der VSTA-Umgebung ... 90

5.1.2 Zugriff auf das Objektmodell in zenon 91
5.1.3 Funktionen in zenon 92
5.1.4 Debuggen eines VSTA Add-ins 94
5.1.5 Events in VSTA 94
5.1.6 Erstellen eines Backups von VSTA-Projekten 95
5.2 Erstellung eines VSTA-Projektes 95
5.2.1 VSTA-Projekte im Editor 96
5.2.2 VSTA-Projekte in der Runtime 97
5.2.3 Entwicklung von Wizards in VSTA 98
5.3 Beispiele 99
5.3.1 Erstellen von Variablen im zenon-Editor 99
5.3.2 Schreiben von Projektinformationen in das zenon-Ausgabefenster 102
5.3.3 Auslesen von Variablen in zenon über Regular Expressions 104

6. Process Control Engine (PCE) 107

6.1 Der PCE Editor 108
6.1.1 Der Taskmanager 108
6.1.2 Der Editierbereich 109
6.1.3 Das Ausgabefenster 109
6.1.4 Die Menüs des PCE Editors 109
6.1.5 Die Symbolleiste des PCE Editors 112
6.2 Vorgangsweise 113
6.2.1 Erstellen eines Tasks 113
6.2.2 Eingabe des Codes 115
6.2.3 Funktion PCE anzeigen 117
6.2.4 Ausführen von Tasks 117
6.3 VBScript - Einführung 118
6.3.1 Datentypen 118
6.3.2 Variablen 119
6.3.3 Konstanten 122
6.3.4 Operatoren 123
6.3.5 Steuern der Programmausführung 124
6.3.6 Verwenden von Schleifen zum Wiederholen von Code 127
6.3.7 Arten von Prozeduren 132
6.3.8 Konventionen für die Programmierung 134

1. Willkommen bei der COPA-DATA Hilfe

ZENON VIDEO-TUTORIALS

Praktische Beispiele für die Projektierung mit zenon finden Sie in unserem YouTube-Kanal (https://www.copadata.com/tutorial_menu). Die Tutorials sind nach Themen gruppiert und geben einen ersten Einblick in die Arbeit mit den unterschiedlichen zenon Modulen. Alle Tutorials stehen in englischer Sprache zur Verfügung.

ALLGEMEINE HILFE

Falls Sie in diesem Hilfekapitel Informationen vermissen oder Wünsche für Ergänzungen haben, wenden Sie sich bitte per E-Mail an documentation@copadata.com (mailto:documentation@copadata.com).

PROJEKTUNTERSTÜTZUNG

Unterstützung bei Fragen zu konkreten eigenen Projekten erhalten Sie vom Support-Team, das Sie per E-Mail an support@copadata.com (mailto:support@copadata.com) erreichen.

LIZENZEN UND MODULE

Sollten Sie feststellen, dass Sie weitere Module oder Lizenzen benötigen, sind unsere Mitarbeiter unter sales@copadata.com (mailto:sales@copadata.com) gerne für Sie da.

2. Programmierschnittstellen

In zenon stehen verschiedene Schnittstellen zur Einbindung eigener Programme oder zur Automatisierung der Projektierung zur Verfügung:

- **Makroliste (auf Seite 39) (**VBA**)**
- **VSTA (auf Seite 88)**
- Add-In Framework (auf Seite 26) (Add-Ins**)**

- **Process Control Engine** (PCE)
  PCE wird ab Version 7.20 nicht mehr unterstützt und nicht mehr im Modulbaum von zenon angezeigt. Beim Konvertieren von Projekten aus Versionen vor 7.20, die PCE-Tasks enthalten, wird der Knoten PCE für diese Projekte wieder eingeblendet. PCE wird nicht weiterentwickelt und nicht weiter dokumentiert.

  Empfehlung: Verwenden Sie **zenon Logic** anstelle von PCE.

  > **Lizenzinformation**
  >
  > *In Standardlizenz für Editor und Runtime enthalten.*

KONTEXTMENÜ

Menüpunkt	Aktion
VBA-Editor öffnen	Öffnet den VBA-Editor
VSTA-Editor mit Projectaddin öffnen	Öffnet den VSTA-Editor.
Hilfe	Öffnet die Online-Hilfe.

> **Info**
>
> *Informationen zur Erstellung und Implementierung von Controls (ActiveX, .NET, WPF) finden Sie im Handbuch Controls.*
> *Informationen zur Projektierung und Verwendung der SAP Schnittstelle finden Sie im Kapitel SAP Interface.*

EDITOREN ÖFFNEN

VBA-EDITOR FÜR MAKROLISTE

VBA startet für **Arbeitsbereich** und **Projekt** dieselbe Entwicklungsumgebung.
Um den VBA Editor zu öffnen:

1. Navigieren Sie im zenon Editor in den Knoten **Programmierschnittstellen**.
2. Erweitern Sie die Ansicht dieses Knotens mit Klick auf `[+]`.
   Die Ansicht des Knotens wird erweitert.
3. Führen Sie einen Rechtsklick auf **Makroliste** aus.
4. Wählen Sie im Kontextmenü den Eintrag **VBA-Editor öffnen**.

Alternative: drücken Sie die Tastenkombination Alt+F11

VSTA-EDITOR

VSTA stellt für **Arbeitsbereich** und **Projekt** getrennte Entwicklungsumgebungen zur Verfügung. Es kann jeweils nur eine von beiden genutzt wird. Beim Start wird jede andere bereits geöffnete VSTA Entwicklungsumgebung geschlossen.

Um den VSTA-Editor für den Arbeitsbereich zu öffnen:

1. Drücken Sie die Tastenkombination `Alt+F10`.

   Der Code für den Arbeitsbereich und alle geladenen Projekte wird angezeigt.

Um den VSTA-Editor für das aktuell geladene Projekt zu öffnen:

1. Navigieren Sie zum Knoten **Programmierschnittstellen**.
2. Erweitern Sie die Ansicht dieses Knotens mit Klick auf `[+]`.
   Die Ansicht des Knotens wird erweitert.
3. Führen Sie einen Rechtsklick auf **VSTA** aus.

   Wählen Sie im Kontextmenü **VSTA-Editor mit ProjectAddin öffnen**.
   Der Editor wird für das aktuell geladene Projekt geöffnet.

API AB VERSION 7.10

Für die Verwendung der zenon Programmierschnittstellen gilt ab zenon 7.10:

- VSTA/.NET: **.NET Framework 3.5 Service Pack 1** muss installiert sein
- VBA: Falls im VBA-Code auf Windows-API oder sonstige importierte DLL-Funktionen zugegriffen wird, müssen diese Aufrufe an die Gegebenheiten von 64Bit angepasst werden. Generell gilt: Eine mit 32-Bit erstellte VBA-Datei kann nicht ohne Änderungen in einer 64 Bit-Version von VBA verwendet werden.

> **Achtung**
>
> Fehler in Anwendungen wie ActiveX, PCE, VBA, VSTA, WPF und externen Anwendungen, die über die API auf zenon zugreifen, können auch die Stabilität der Runtime beeinflussen.

3. Add-Ins

Add-Ins stellen Möglichkeiten zur Verfügung, die Funktionalitäten von zenon im Editor und in der Runtime mit unterschiedlichen Entwicklungsumgebungen zu erweitern.
Zur Erstellung von Add-Ins stehen Ihnen zur Verfügung:

- Microsoft Visual Studio (auf Seite 23)
- SharpDevelop (im Installationspaket enthalten) (auf Seite 20)

Add-Ins können mit jeder .NET Programmiersprache erstellt werden. Eine IDE Unterstützung steht für die Programmiersprachen C# und Visual Basic.NET zur Verfügung.

Hinweise:

- Sämtliche Listen und die Schnittstelle unterstützen **IEnumerable**.
  Referenz: https://msdn.microsoft.com/de-de/library/9eekhta0(v=vs.110).aspx
  (https://msdn.microsoft.com/de-de/library/9eekhta0(v=vs.110).aspx)
- **LINQ** (Language Integrated Query) kann verwendet werden.
  Referenz: https://msdn.microsoft.com/en-us/library/mt693024.aspx
  (https://msdn.microsoft.com/en-us/library/mt693024.aspx)

> **Achtung**
>
> *Für Add-Ins werden die .NET Naming Guidelines verwendet.*
>
> *Dafür wurden die bisher für VSTA und VBA verwendeten Objektnamen angepasst. In VSTA und VBA bleiben die bisherigen Namen aus Kompatibilitätsgründen gleich. Für Add-Ins werden Benennungen entsprechend dem .NET Naming Guidelines verwendet.*
>
> Eine Übersicht über die jeweiligen Objektnamen finden Sie in der Excel-Tabelle **API_Naming_Conversion.xlsx**. Diese finden Sie im zenon Installationspfad im Ordner **\HELP\AddIns**.

FUNKTIONALITÄT

Add-Ins ermöglichen Ihnen unter anderem:

- für den Editor Projektierungshilfen (z.B. Wizards) zur Verfügung zu stellen
- das Projekt in der Runtime um zusätzliche Funktionalitäten zu erweitern

Sie finden Add-Ins:

- für den Editor im Menü **Extras** unter den Einträgen:
  - **Editor Add-Ins verwalten...**
  - **Editor Wizards starten...**
  - **Editor Dienste verwalten...**
- für die Runtime im Knoten **Programmierschnittstellen - Add-Ins**

TOOLS UND UNTERSTÜTZUNG

Tools und Unterstützung erhalten Sie bei:

- Visual Studio Developer Tools: marketplace.visualstudio.com (https://marketplace.visualstudio.com)
- COPA-DATA Anwenderforum: forum.copadata.com (https://forum.copadata.com/)

3.1 Vorkenntnisse

Wie bei der Erstellung von anderen Softwarekomponenten, sind auch für die Entwicklung von Add-Ins für zenon gewisse Vorkenntnisse notwendig.

Dazu zählen

- Programmierkenntnisse in C# oder VB.NET (je nach Anforderung/Komplexität der zu entwickelnden Add-Ins)
- Gute zenon Kenntnisse

Empfohlen wird weiters

- Erfahrung im Umgang mit professionellen Entwicklungsumgebungen (z.B. Microsoft Visual Studio)
- Die Verwendung eines Versionsverwaltungssystems für die Entwicklungen (TFS, SVN, GIT)

3.2 Terminologie

Fachbegriffe und ihre Bedeutung im Zusammenhang mit Add-Ins.

Term	Bedeutung
Add-In	Eine Sammlung von **Extensions**. Wird durch das Add-In Package repräsentiert. Ein **Add-In** kann mehrere **Extensions** enthalten.
Add-In Assembly	Die kompilierte Bibliothek (*.dll), die **Extensions** enthält.
Add-In Package	Eine ZIP-Datei (*.scadaAddIn), die Metadaten und Bibliotheken mit ihren Abhängigkeiten enthält. Ein **Add-In Package** enthält eine **Add-In Assembly**.
Extension	**Extensions** sind Klassen, die von einem Interface mit einer oder mehreren Methoden abgeleitet werden. Sie sind Einsprungspunkte im **Add-In Package.** Jede Klasse, die eine **Extension** repräsentiert, wird mit einem .NET Attribut gekennzeichnet. Es gibt zwei Typen von **Extensions** (auf Seite 17), jeweils für den Editor und für die Runtime ▸ **Wizard Extensions** (Wizards) ▸ **Service Extensions** (Dienste)
IDE	Integrierte Entwicklungsumgebung. Abkürzung für Integrated Development Environment.
Project Add-In Management	Komponente für zenon Editor und Runtime, die **Project Add-In Packages** managt.
Service Extension	Implementierung einer **Extension**, die einen Dienst (**Service**) repräsentiert. **Dienste:** ▸ laufen im Hintergrund ▸ werden automatisch oder manuell gestartet ▸ bleiben im Speicher bis sie gestoppt werden
Wizard Extension	Implementierung einer **Extension**, die einen Wizard repräsentiert. **Wizards:** ▸ dienen zur Lösung einer bestimmten Aufgabe ▸ werden beim Schließen wieder entladen

Hinweis: Fachbegriffe werden in der Dokumentation für Add-Ins ausschließlich in Englisch verwendet.

3.3 Einschränkungen

Für Add-Ins gelten folgende Einschränkungen:

- Der zenon Web Client unterstützt die Ausführung von Add-Ins, allerdings ohne IDE-Unterstützung.

  Das lokale Debuggen von Add-Ins ist im zenon Web Client aus diesem Grund nicht möglich, ein Debuggen über Remote Debugging Tools wird unterstützt.
- In der HTML Web Engine wird die Ausführung von Add-Ins nicht unterstützt.

3.4 Add-In erstellen

ALLGEMEINE INFORMATIONEN

Add-Ins werden über die Add-In ID identifiziert. Sie wird für Installation und Import benutzt.

Eine Add-In ID besteht aus einem optionalen Namespace und einer lokalen ID, also **namespace.localId**. Die Local ID identifiziert ein Add-In, der Namespace ordnet ein Add-In einer Organisation zu. Beide Informationen werden durch das .NET Attribut **Mono.AddIn.AddInAttribute** festgelegt. Damit der Name für ein Add-In Package auch über Organisationsgrenzen hinweg eindeutig bleibt, wird der Namespace des Add-In Projekts als Teil der ID genutzt.
Beispiel für eine vollständige Add-In ID: **com.organisation.importwizard**.

Die Vergabe von Namespaces ist optional.
Empfehlung: Verwenden Sie Namespaces. Damit stellen Sie die globale Eindeutigkeit über Organisationsgrenzen hinweg sicher. Verwenden Sie am besten die URL der eigenen Organisation in umgekehrter Reihenfolge.
Zum Beispiel: Für **www.example.com** verwenden Sie **com.example** als Namespace.
Namespaces können innerhalb der Organisation weiter unterteilt werden, zum Beispiel durch Teams, Abteilungen oder Projekte.

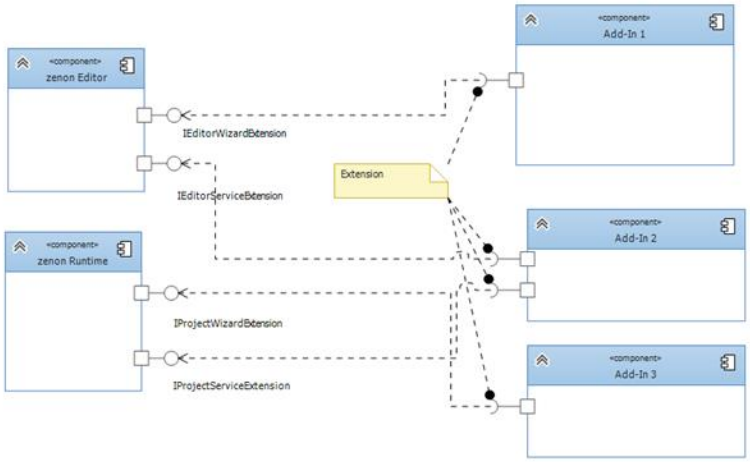
Zur Erstellung von Add-Ins stehen Ihnen zur Verfügung:

- Microsoft Visual Studio (auf Seite 23)
- SharpDevelop (im Installationspaket enthalten) (auf Seite 20)

Add-Ins können mit jeder .NET Programmiersprache erstellt werden. Eine IDE Unterstützung steht für die Programmiersprachen C# und Visual Basic.NET zur Verfügung.

3.4.1 Grundlagen

Für den zenon Editor und die zenon Runtime stehen unterschiedliche Extensions zur Verfügung. Je nach Anwendungsfall kann eine entsprechende Extension gewählt werden.



Sämtliche für die Entwicklung von Add-Ins benötigten Schnittstellen sind im Assembly **Scada.AddIn.Contracts** definiert.

Ein Add-In Package kann eine oder mehrere Extensions beinhalten.

TYPEN VON EXTENSIONS - SCHNITTSTELLEN

Es wird zwischen den folgenden Extensions unterschieden:

EDITOR WIZARD EXTENSION - IEDITORWIZARDEXTENSION (AUF SEITE 17)

- Die Ausführung wird durch den Benutzer über den Dialog **Extras -> Editor Wizards...** im zenon Editor ausgelöst.
- Implementiert die Methode **Run**, um eine Wizard Extension im zenon Editor auszuführen.

EDITOR SERVICE EXTENSION - IEDITORSERVICEEXTENSION (AUF SEITE 17)

- Die Ausführung geschieht automatisch beim Starten des Editors, abhängig vom **DefaultStartMode**.

  Hinweis: Für den automatischen Start muss in der **Extension** das Attribut `DefaultStartMode=DefaultStartupModes.Auto` gesetzt sein. Der Startmodus kann später auch mit dem Dialog **Editor Dienste verwalten** (auf Seite 33) verändert werden.
- Die Verwaltung geschieht über den Dialog **Extras -> Editor Dienste verwalten...** im zenon Editor.

- Implementiert die Methoden **Start** und **Stop** um eine Service Extension im zenon Editor auszuführen.

PROJECT WIZARD EXTENSION - IPROJECTWIZARDEXTENSION (AUF SEITE 18)

- Die Ausführung wird vom Benutzer mit der zenon Funktion **Wizard ausführen** ausgelöst.
- Implementiert die Methode **Run**, um eine Wizard Extension in der zenon Runtime auszuführen.

PROJECT SERVICE EXTENSION - IPROJECTSERVICEEXTENSION (AUF SEITE 19)

- Die Ausführung geschieht automatisch beim Starten des Projekts in der Runtime, abhängig vom **DefaultStartMode**.

  Hinweis: Für den automatischen Start muss in der **Extension** das Attribut `DefaultStartMode=DefaultStartupModes.Auto` gesetzt sein. Der Startmodus kann später auch mit dem Dialog **Editor Dienste verwalten** (auf Seite 33) verändert werden.
- Die Verwaltung geschieht über die zenon Funktion **Dialog "Runtime Dienste verwalten" anzeigen**.
- Implementiert die Methoden **Start** und **Stop** um eine Service Extension in der zenon Runtime auszuführen.

EXTENSION ATTRIBUTE

Folgende Metadaten werden über das .NET Attribut mit dem Namen **AddInExtension** definiert.

Property	Beschreibung	IEditorWizardExtension	IEditorServiceExtension	IProjectWizardExtension	IProjectServiceExtension
Name	Anzeigename in Dialogen. Info: Entspricht dem Namen des Wizards oder des Dienstes in zenon.	**X**	**X**	**X**	**X**
Description	Beschreibung zur Anzeige in Dialogen. Info: Entspricht der Beschreibung des Wizards oder des Dienstes in zenon.	**X**	**X**	**X**	**X**
Category	Kategorie zur Gruppierung in Dialogen. Info: Wird zur Kategorisierung von Wizards im Dialog **Editor Wizards...** im zenon Editor verwendet.	**X**	**--**	**X**	**--**
DefaultStartMode	Beschreibt das Standard-Startverhalten von Service Add-In Extensions nach der Installation. ▸ `Manual`: Extension muss vom Benutzer manuell gestartet werden. ▸ `Auto`: Wird gestartet, wenn der Arbeitsbereich im Editor oder das Projekt in der Runtime geladen wird. Keine Angabe: `Manual` wird benutzt. Info: Entspricht der Startart des Dienstes in zenon.	**--**	**X**	**--**	**X**

Legende:

- **X**: trifft zu
- **--**: trifft nicht zu

ADD-IN ATTRIBUTE

Ein Add-In Assembly benötigt mehrere .NET Attribute, die in der Datei **AddInInfo.cs** definiert werden.

Die wichtigsten und für die Verwendung in zenon benötigten Attribute sind:

Attribute Name	Obligatorisch	Beschreibung	Beispiele
`Addin`	**X**	ID und Version des Add-Ins. **Info:** Die Version entspricht der Spalte **Version** im Add-In Verwaltungsdialog im zenon Editor.	Beispiel 1: [assembly: Addin("AddInIdentifier", "1.0", Namespace="com.example")] Beispiel 2: [assembly: Addin("MultitouchAddIn", "1.0")] Beispiel 3: [assembly: Addin("ProjectCreation", "1.0")]
`AddinDependency`	**X**	Definiert eine Abhängikeit zum zenon System. Dieser Wert ist konstant und wird benötigt, damit zenon das Add-In verwenden kann.	[assembly: AddinDependency("scada", "1.0")]
`AddinName`	--	Name des Add-Ins. **Info:** Entspricht der Spalte **Name** im Add-In Verwaltungsdialog im zenon Editor.	[assembly: AddinName("Editor Add-In Demo")]
`AddinDescription`	--	Beschreibung des Add-Ins. **Info:** Entspricht der Spalte **Beschreibung** im Add-In Verwaltungsdialog im zenon Editor.	[assembly: AddinDescription("Demonstrates an Add-In with a Wizard and a Service extension.")]

Legende:

- **X**: trifft zu
- **--**: trifft nicht zu

3.4.2 Typen von Extensions

Wizard im Editor - Editor Wizard Extension

Editor Wizard Extensions werden dazu verwendet, um Wizards oder andere benutzergesteuerte API-Vorgänge im Editor zu realisieren.

- Die Ausführung wird durch den Benutzer über den Dialog **Extras -> Editor Wizards...** im zenon Editor ausgelöst.
- Implementiert die Methode **Run**, um eine Wizard Extension im zenon Editor auszuführen.

BEISPIELCODE (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...

	}

}
```

Dienste im Editor - Editor Service Extension

Editor Service Extensions werden dazu verwendet, um automatisiert API-Vorgänge im Editor zu realisieren.

- Die Ausführung geschieht automatisch beim Starten des Editors, abhängig vom **DefaultStartMode**.

  Hinweis: Für den automatischen Start muss in der **Extension** das Attribut `DefaultStartMode=DefaultStartupModes.Auto` gesetzt sein. Der Startmodus kann später auch mit dem Dialog **Editor Dienste verwalten** (auf Seite 33) verändert werden.
- Die Verwaltung geschieht über den Dialog **Extras -> Editor Dienste verwalten...** im zenon Editor.
- Implementiert die Methoden **Start** und **Stop** um eine Service Extension im zenon Editor auszuführen.

BEISPIELCODE (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 zur Runtime - Project Wizard Extension

Project Wizard Extensions werden dazu verwendet, um Wizards oder andere benutzergesteuerte API-Vorgänge in der Runtime zu realisieren.

- Die Ausführung wird vom Benutzer mit der zenon Funktion **Wizard ausführen** ausgelöst.
- Implementiert die Methode **Run**, um eine Wizard Extension in der zenon Runtime auszuführen.

BEISPIELCODE (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...

	}

}
```

Dienst zur Runtime - Project Service Extension

Runtime Service Extensions werden dazu verwendet, um Hintergrundaufgaben in der Runtime zu realisieren.

- Die Ausführung geschieht automatisch beim Starten des Projekts in der Runtime, abhängig vom **DefaultStartMode**.

  Hinweis: Für den automatischen Start muss in der **Extension** das Attribut `DefaultStartMode=DefaultStartupModes.Auto` gesetzt sein. Der Startmodus kann später auch mit dem Dialog **Editor Dienste verwalten** (auf Seite 33) verändert werden.
- Die Verwaltung geschieht über die zenon Funktion **Dialog "Runtime Dienste verwalten" anzeigen**.
- Implementiert die Methoden **Start** und **Stop** um eine Service Extension in der zenon Runtime auszuführen.

BEISPIELCODE (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 Inhalt von Add-In Packages

Add-In Packages sind ZIP-Dateien mit der Namenserweiterung **.scadaAddIn**. Sie enthalten Binär- und Metadatendateien. Binärdateien müssen vorhanden sein und werden als eigene ZIP-Datei im Add-In Package gespeichert.
Packages können mit dem **Add-In Utility** (auf Seite 25) erstellt werden. Über die Entwicklungsumgebungen SharpDevelop und Microsoft Visual Studio werden die Packages beim Kompilieren automatisch erzeugt.

Empfehlung: Verändern Sie Dateien nicht manuell, sondern benutzen Sie immer das **Add-In Utility**.

INHALT DES ADD-IN PACKAGES

Das Add-In Package enthält:

- **Content.xml**: XML-Metadatendatei mit Inhalten des Packages.
- **Binary.zip**: ZIP-Datei mit den Binärdateien.

XML-DATEI

Der Inhalt eines Add-In Package wird in einer eigenen XML-Datei (**Content.xml**) aufgelistet.

BINÄRDATEIEN

Binärdateien werden in einer eigenen ZIP-Datei gespeichert. Diese Datei muss alle Komponenten beinhalten, die auf einem anderen System benötigt werden, um das Add-In auszuführen. Benötigt werden direkte und indirekte Referenzen auf .NET Assemblies. Dabei müssen nur externe Assemblies inkludiert werden. Als extern gelten Assemblies, die nicht Teil der .NET Framework Class Library sind oder bei der zenon Installation installiert werden.

Achtung: Stellen Sie sicher, dass Sie über alle dafür benötigten Lizenzen verfügen.

Hinweis: Dynamisch geladene .NET Assemblies sind in den Metadaten von .NET Assemblies nicht enthalten.

> **Info**
>
> COPA-DATA übernimmt keine Haftung und bietet keine technische Unterstützung für externe Libraries.

3.4.4 SharpDevelop

SharpDevelop (auch #develop) ist eine Open Source Entwicklungsumgebung, die auf Microsofts .NET Plattform basiert. **SharpDevelop** ist im Lieferumfang von zenon enthalten und wird bei der Installation von zenon mitinstalliert.
Hinweis: Die jeweils aktuelle Version sowie die offizielle Dokumentation zu **SharpDevelop** finden Sie online auf: www.icsharpcode.net/opensource/sd/ (http://www.icsharpcode.net/opensource/sd/).

Add-In Package mit SharpDevelop erstellen

Um ein Add-In Package mit SharpDevelop zu erstellen:

1. Starten Sie **SharpDevelop** indem Sie
   - im zenon Editor das Menü **Extras -> Add-In Editor öffnen...** wählen, oder
   - im **Startup Tool** unter **Tools** die Verknüpfung **SharpDevelop IDE** wählen
2. Wählen Sie **File -> New Solution** und wählen Sie unter **C# -> SCADA Add-Ins** oder **VB -> SCADA Add-Ins** eine entsprechende Vorlage, erstellen Sie basierend auf dieser ein neues Projekt:
   - Editor Service Extension (auf Seite 17)
   - Editor Wizard Extension (auf Seite 17)
   - Project Service Extension (auf Seite 19)
   - Project Wizard Extension (auf Seite 18)

   Hinweis: Sämtliche in diesem Hilfekapitel verwendeten Beispiele basieren auf C#.
3. Fügen Sie Ihren Code, Referenzen, etc. hinzu.
4. Bauen Sie das Projekt.
   Das Add-In Package mit der Dateiendung ***.scadaAddIn** wird im Ausgabeordner des Projektes erstellt.
   Dieser lautet je nach Konfiguration `...bin\Debug\` oder `...bin\Release\`.
5. Importieren Sie das Add-In Package im Editor, siehe dazu:
   - Add-Ins für den Editor installieren und verwalten (auf Seite 31)
   - Add-Ins für die Runtime installieren und verwalten (auf Seite 36)

Deployen und Debuggen von Add-Ins

DEBUGGEN IM EDITOR

Um ein Add-In im Editor zu debuggen, gehen Sie wie folgt vor:

1. Starten Sie den zenon Editor.
2. Starten Sie **SharpDevelop** indem Sie
   - im zenon Editor das Menü **Extras -> Add-In Editor öffnen...** wählen, oder
   - im **Startup Tool** unter **Tools** die Verknüpfung **SharpDevelop IDE** wählen
3. Öffnen Sie das Projekt.
4. Navigieren Sie in den Projekteinstellungen in SharpDevelop zur Registerkarte **Add-In**.
5. Wählen Sie unter **Debug Target** `Editor`.
6. Starten Sie das Debugging über `F5` oder **Debug -> Run**.

7. Das Add-In Projekt wird kompiliert und in den Editor deployed.

   Hinweis: Das Add-In wird hierbei in das Verzeichnis `%ProgramData%\COPA-DATA\zenon760\EditorAddInCache` kopiert.

   Um es wieder aus dem Editor zu entfernen, gehen Sie wie im Kapitel Add-Ins für den Editor installieren und verwalten (auf Seite 31) vor.
   Sollte bereits ein Add-In mit der gleichen ID vorhanden sein, wird dieses ersetzt.

8. Debuggen Sie das Add-In.

DEBUGGEN IN DER RUNTIME

Um ein Add-In in der Runtime zu debuggen, gehen Sie wie folgt vor:

1. Starten Sie die zenon Runtime.
2. Starten Sie **SharpDevelop** indem Sie im **Startup Tool** unter **Tools** die Verknüpfung **SharpDevelop IDE** wählen
3. Öffnen Sie das Projekt.
4. Navigieren Sie in den Projekteinstellungen in SharpDevelop zur Registerkarte **Add-In**.
5. Wählen Sie unter **Debug Target** das entsprechende Projekt in der Runtime aus, in dem Sie das Add-In debuggen wollen.
6. Starten Sie das Debugging über `F5` oder **Debug -> Run**.
7. Das Add-In Projekt wird kompiliert und temporär in das gewählte zenon Projekt deployed.

   Achtung: Das Add-In wird hierbei temporär in das Verzeichnis `...\RT\FILES\zenon\system\AddInCache\...` kopiert.

   Nach dem Beenden der Runtime wird es wieder entfernt.

   Um ein Add-In dauerhaft zu installieren, importieren Sie es wie im Kapitel Add-Ins für die Runtime installieren und verwalten (auf Seite 36) beschrieben.

8. Debuggen Sie das Add-In.

> **Info**
>
> *Alternativ zu den beschriebenen Methoden können Sie Ihre Add-Ins auch mittels **Debug -> Attach to Process** debuggen. Verbinden Sie sich dabei aus dem entsprechenden Add-In Projekt in SharpDevelop auf den Prozess in zenon Editor oder Runtime.*

3.4.5 Microsoft Visual Studio

Add-In Package mit Visual Studio erstellen

Um ein Add-In Package mit Visual Studio zu erstellen:

1. Starten Sie Visual Studio.
2. Installieren Sie die Visual Studio Developer Tools.
   Hinweis: Diese können vom Visual Studio Marketplace https://marketplace.visualstudio.com/ heruntergeladen werden.
3. Wählen Sie **File -> New Project** und wählen Sie unter **Templates -> Visual C# -> SCADA Add-Ins** oder **Templates -> Visual Basic -> SCADA Add-Ins** eine entsprechende Vorlage, erstellen Sie basierend auf dieser ein neues Projekt:
   - Editor Service Extension (auf Seite 17)
   - Editor Wizard Extension (auf Seite 17)
   - Project Service Extension (auf Seite 19)
   - Project Wizard Extension (auf Seite 18)

   Hinweis: Sämtliche in diesem Hilfekapitel verwendeten Beispiele basieren auf C#.
4. Fügen Sie Ihren Code, Referenzen, etc. hinzu.
5. Bauen Sie das Projekt.
6. Das Add-In Package mit der Dateiendung ***.scadaAddIn** wird automatisch erstellt im Ausgabeordner des Projektes (je nach Konfiguration `...bin\Debug\` oder `...bin\Release\`).
7. Importieren Sie das Add-In Package im Editor, siehe dazu:
   - Add-Ins für den Editor installieren und verwalten (auf Seite 31)
   - Add-Ins für die Runtime installieren und verwalten (auf Seite 36)

Deployen und Debuggen von Add-Ins

DEBUGGEN IM EDITOR

Um ein Add-In im Editor zu debuggen, gehen Sie wie folgt vor:

1. Starten Sie den zenon Editor.
2. Starten Sie in Visual Studio das Debugging über `F5` oder **Debug -> Start Debugging**.
3. Das Add-In Projekt wird kompiliert und in den Editor deployed.

Hinweis: Das Add-In wird hierbei in das Verzeichnis `%ProgramData%\COPA-DATA\zenon760\EditorAddInCache` kopiert.

Um es wieder aus dem Editor zu entfernen, gehen Sie wie im Kapitel Add-Ins für den Editor installieren und verwalten (auf Seite 31) vor.

4. Debuggen Sie das Add-In.

> **Info**
>
> *Alternativ zu der beschriebene Methode können Sie Ihre Add-Ins auch mittels **Debug -> Attach to Process** debuggen. Verbinden Sie sich dabei aus dem entsprechenden Add-In Projekt in Visual Studio auf den zenon Editor Prozess.*
>
> *Empfohlener Debugger-Typ (**Debugger Type**): `Managed mit CLR 4.0`*

DEBUGGEN IN DER RUNTIME

Um ein Add-In in der Runtime zu debuggen, gehen Sie wie folgt vor:

1. Kompilieren Sie das Add-In und importieren Sie es wie in Add-Ins für die Runtime installieren und verwalten (auf Seite 36) beschrieben.
2. Starten Sie die zenon Runtime.
3. Wählen Sie in Visual Studio **Debug -> Attach to Process**
4. Wählen Sie den zenon Runtime Prozess (**Zenrt32.exe**) und drücken Sie **Attach**
5. Debuggen Sie das Add-In.

3.4.6 Source Code Verwaltung

Zur Verwaltung des Source Codes wird die Verwendung eines Versionskontrollsystems - wie TFS, GIT oder SVN - empfohlen.

> **Achtung**
>
> *Im Unterschied zu VSTA und VBA wird der Source Code von Add-Ins nicht im zenon Projekt gespeichert und verwaltet.*

3.4.7 Add-In Analysis und Packaging Utility (AddInUtility)

Das Tool **AddInUtility**, wird automatisch beim Kompilieren von Add-Ins aufgerufen und hat folgende Aufgaben:

- Analyse der Abhängigkeiten
- Packen von Add-Ins

Ein Package enthält die Add-In Assembly und eine XML-Datei mit der Erweiterung ***.scadaAddin**, welche die Metadaten für das Add-In beinhaltet.

Kommandozeilen-Tool AddInUtility.exe

Das **AddInUtility** packt Add-Ins über die Kommandozeile.

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

- Argumente sind unabhängig von Groß-/Kleinschreibung.
- Werte sind sind unabhängig von Groß-/Kleinschreibung, wenn so gekennzeichnet.
- Argumente können in beliebiger Reihenfolge angegeben werden.

Argument	Beschreibung
--action **(-a)**	Aktion, die ausgeführt werden soll. Mögliche Werte: ▸ `BuildPackage` ▸ `GetPackageInfo` Pflichtangabe.
--path **(-p)**	Pfadangabe. Benötigte Angaben bei: ▸ **action** = `BuildPackage`: Pfad zur Add-In Assembly Datei. ▸ **action** = `GetPackageInfo`: Pfad zur Add-In Package Datei. Pflichtangabe.
--targetDir **(-t)**	Zielordner. Benötigte Angaben bei: ▸ **action** = `BuildPackage`: Pfad zum Ordner, in dem das Add-In Package erzeugt wird. Default: aktueller Ordner

Beispiele

- Erzeugt ein Add-In Package (Argumente ausgeschrieben):
  AddInUtility.exe --action BuildPackage --path C:\Addin.dll --targetDir C:\Directory

- Erzeugt ein Add-In Package (Argumente in Kurzform):
  AddInUtility.exe -a BuildPackage -p C:\Addin.dll -t C:\Directory
- Gibt Informationen zum angegebenen Add-In Package zurück:
  AddInUtility.exe -a GetPackageInfo -p C:\Directory\Addin.scadaAddIn

3.4.8 Verhalten beim Nachladen

zenon bietet eine Nachlade-Funktionalität für Projekte, die zur Laufzeit Hot-Plugging oder das Aktualisieren veränderter Projektdateien unterstützen. Entsprechend den ausgelösten Funktionen der Add-Ins, ist für VSTA und VBA die Reihenfolge der Aktivitäten wie folgt:

1. Pre-reload VSTA
2. Stop of Service Extensions
3. Reload VBA
4. Start of Service Extensions
5. Post-reload VSTA

Add-Ins verhalten sich wie VSTA und stellen eine Pre-reload-Funktion und eine Post-reload-Funktion zur Verfügung. Diese werden nach dem VSTA Pre-reload und vor dem Nachladen von VSTA ausgelöst und als Wrapper für die VBA-Funktion `reload` benutzt. Ein erneutes Laden wird verzögert, bis das Ausführen von Wizard Extensions abgeschlossen ist.
Empfehlung: Falls Sie die Nachladen-Funktionalität nutzen möchten, vermeiden Sie blockierende oder lange laufende Wizard-Extensions.

Beim erneuten Laden werden alle laufenden Service-Erweiterungen neu gestartet. Neu installierte Service-Erweiterungen werden nach dem Neustart gestartet, wenn der **DefaultStartMode** auf `Automatisch` gesetzt ist.

3.4.9 Isolierung

Sämtliche Add-Ins werden in einem isolierten Speicherbereich ausgeführt (App-Domain). Dadurch können auch verschiedene Versionen derselben Assemblies parallel geladen werden.

3.5 Verwenden von Add-Ins in Editor und Runtime

Add-Ins stellen Projektierungshilfen im zenon Editor zur Verfügung und ergänzen die Funktionalitäten von zenon in der Runtime.

Sie finden Add-Ins:

- für den Editor im Menü **Extras** unter den Einträgen:
  - **Editor Add-Ins verwalten...**
  - **Editor Wizards starten...**
  - **Editor Dienste verwalten...**
- für die Runtime im Knoten **Programmierschnittstellen - Add-Ins**

ADD-INS FÜR EDITOR UND RUNTIME

- Add-Ins für den Editor (auf Seite 30):
  Add-Ins werden importiert, installiert und können dann im Editor verwendet werden.
  Die Verwaltung (auf Seite 31) erfolgt über die Einträge im Menü **Extras**.
- Add-Ins für die Runtime (auf Seite 36):
  Add-Ins werden importiert, beim Starten Runtime werden sie automatisch aktiviert und können dann verwendet werden.
  Die Verwaltung (auf Seite 36) erfolgt über die Detailansicht des Knotens **Programmierschnittstellen** - **Add-Ins** im Projektbaum.

Add-Ins stehen zur Verfügung als:

- Wizard: Wird vom Benutzer aufgerufen und wird nach Lösung der Aufgabe wieder beendet.
  Details zu mit zenon mitgelieferten Wizards lesen Sie im Handbuch **Wizards**.
- Dienst (Service): Kann automatisch oder manuell gestartet werden und läuft im Hintergrund.

Add-Ins können:

- mit zenon ausgeliefert werden
- von Integratoren entwickelt und deren Kunden zur Verfügung gestellt werden
- von Kunden selbst entwickelt werden

LISTE DER ADD-INS UND DIENSTE FILTERN UND SORTIEREN

Um eine Liste in der Detailansicht oder in einem Dialog zu filtern:

1. Klicken Sie in die Filterzeile der gewünschten Spalte.
2. Tragen Sie die Filterkriterien ein.
   Platzhalter können verwendet werden:
   - ?: Platzhalter für genau ein beliebiges Zeichen.
   - *: Platzhalter für beliebig viele Zeichen.
3. Mehrere Filter können kombiniert werden.

Hinweis: Um alle Filter zu entfernen, klicken Sie in der Symbolleiste auf das entsprechende Symbol.

Um die Liste zu sortieren:

1. Klicken Sie auf den Spaltentitel der Spalte, nach der sortiert werden soll.

   Das Symbol Fernglas zeigt an, nach welcher Spalte sortiert wird.

   Die Liste wird entsprechend der Spalte sortiert.

   Ein Pfeil zeigt an, ob absteigend oder aufsteigend sortiert wird.
2. Um die Sortierreihenfolge zu ändern klicken Sie erneut in den Spaltentitel.

SICHERUNG UND WIEDERHERSTELLUNG

Für die Sicherung und Wiederherstellung von Add-Ins gilt:

- Bei der Sicherung des Arbeitsbereichs werden Editor Add-Ins nicht gesichert.

  Hinweis: Installierte Add-Ins für den Editor befinden sich im Verzeichnis `%ProgramData%\COPA-DATA\zenon760\EditorAddInStore` und können bei Bedarf manuell gesichert werden.
- Add-In Packages von Projekten werden bei der Erstellung von Projekt- und Arbeitsbereichsicherungen mitgesichert und werden beim Rücklesen dieser wiederhergestellt.

ÄNDERUNGSHISTORIE

Wurde die Änderungshistorie für das Projekt aktiviert, dann werden Einträge für die Änderungshistorie beim Installieren und Deinstallieren von Add-In Packages im Projekt erstellt.

3.5.1 Add-Ins in zenon aktivieren

Die Verwendung von **Add-Ins** wird in der **zenon6.ini** aktiviert. Dazu muss der folgende Eintrag vorhanden sein:

[AddIns]

ON=1

Nach der Installation von zenon ist dieser Eintrag standardmäßig aktiviert.

Hinweis: Wenn VBA/VSTA für zenon (auf Seite 90) deaktiviert wird, werden auch Add-Ins deaktiviert.

3.5.2 Verhalten bei der Installation

Add-Ins können im Editor oder in der Runtime verwendet werden.
Bei der Installation von Add-Ins gilt:

1. Neuinstallation:
   Das ausgewählte Package wird im AddInStore-Ordner abgelegt.
   - Ordner bei Installation im Editor (auf Seite 31):
     `%ProgramData%\COPA-DATA\zenon760\EditorAddInStore`
   - Ordner bei Installation in der Runtime (auf Seite 36):
     `...\RT\FILES\zenon\system\AddInStore\...`

   Die Add-In ID wird als Dateiname verwendet und mit der Dateinamenerweiterung **.scadaAddIn** ergänzt.
2. Neuinstallation mit gleicher ID:
   Das bereits bestehende Add-In wird ersetzt.
3. Bei der Installation werden bereits vorhandene Dateien anderer Versionen (höher oder niedriger) ersetzt.

Info

Die Add-In ID ist für jedes Add-In eindeutig und wird über ein .NET Attribut in der Datei **AddInInfo.cs** definiert.

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

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

3.5.3 Knoten Add-Ins im Projektmanager

Im Projektbaum des Projektmanager können unter dem Knotenpunkt **Programmierschnittstellen** Add-Ins importiert werden. Details dazu finden Sie im Kapitel **Kontextmenü Projektbaum** (auf Seite 29).

Kontextmenü Projektbaum

Ein Rechtsklick auf das Symbol Add-Ins im Knoten **Programmierschnittstellen** im Projektbaum öffnet ein Kontextmenü:

Eintrag	Beschreibung
Add-In importieren...	Öffnet den Dialog zum Import eines **Add-In Packages**, das zur Runtime eingesetzt werden soll. Hinweis: Das Paket wird nur importiert, aber nicht installiert. Die Installation erfolgt zur Runtime.
Editorprofil	Öffnet die Dropdownliste mit vordefinierten Editorprofilen.
Hilfe	Öffnet die Online-Hilfe.

Kontextmenü und Symbolleiste Detailansicht

In der Detailansicht des Knotens **Add-Ins** im Projektbaum werden Add-Ins, die zur Runtime verwendet werden, angezeigt und über eine Symbolleiste und ein Kontextmenü verwaltet.

Einträge im Kontextmenü und Bedeutung der Symbole von links nach rechts:

Symbol/Eintrag	Beschreibung
Add-In importieren	Öffnet den Dialog zum Import eines **Add-In Packages**, das zur Runtime eingesetzt werden soll. Hinweis: Das Paket wird nur importiert, aber nicht installiert. Die Installation erfolgt zur Runtime.
Löschen	Deinstalliert und löscht das ausgewählte Add-In nach einer Sicherheitsabfrage.
Alle Filter entfernen	Entfernt alle Filter, die in der Liste der Add-Ins aktuell angewendet werden.
Eigenschaften	Öffnet das Fenster **Eigenschaften**.
Hilfe	Öffnet die Online-Hilfe.

3.5.4 Verwendung im zenon Editor

Add-Ins für die Verwendung im zenon Editor, werden im Editor importiert, installiert und verwaltet.

Add-Ins für den Editor können beinhalten:

- Editor Wizard Extensions (auf Seite 17), diese werden manuell gestartet, über den Menüpunkt **Extras -> Editor Wizards**.

- Editor Service Extensions (auf Seite 17), diese werden gestartet wenn der Arbeitsbereich geladen wird und ihr Startmodus auf `Automatisch` steht. Sie werden gestoppt, sobald der Arbeitsbereich geschlossen wird.

Installierte Add-Ins für den Editor werden gespeichert unter:
`%ProgramData%\COPA-DATA\zenon760\EditorAddInStore`

Add-Ins für den Editor installieren und verwalten

Im zenon Editor können Add-Ins importiert, installiert und verwaltet werden.

Hinweis: Es können nur Add-Ins mit Editor Extensions importiert und installiert werden. Werden beim Import keine Editor Extensions gefunden, wird eine entsprechende Warnung angezeigt.

ADD-IN IMPORTIEREN UND INSTALLIEREN

Um ein Add-In zu importieren und zu installieren:

1. Öffnen Sie das Menü **Extras.**
2. Wählen Sie den Eintrag **Editor Add-Ins verwalten**.
   Der Dialog zur Verwaltung von Add-Ins wird geöffnet.
3. Klicken Sie auf das Symbol **Importieren und installieren**, wählen Sie diesen Eintrag im Kontextmenü oder drücken Sie die Taste `Einfg`.

   Der Dialog (auf Seite 32) zur Auswahl eines Add-Ins wird geöffnet.
4. Wählen Sie das gewünschte Add-In aus.
5. Klicken Sie auf **Öffnen**.

   Das Add-In wird importiert und installiert.
6. Klicken Sie auf **Schließen**, um den Dialog zu schließen.

ADD-IN DEINSTALLIEREN UND LÖSCHEN

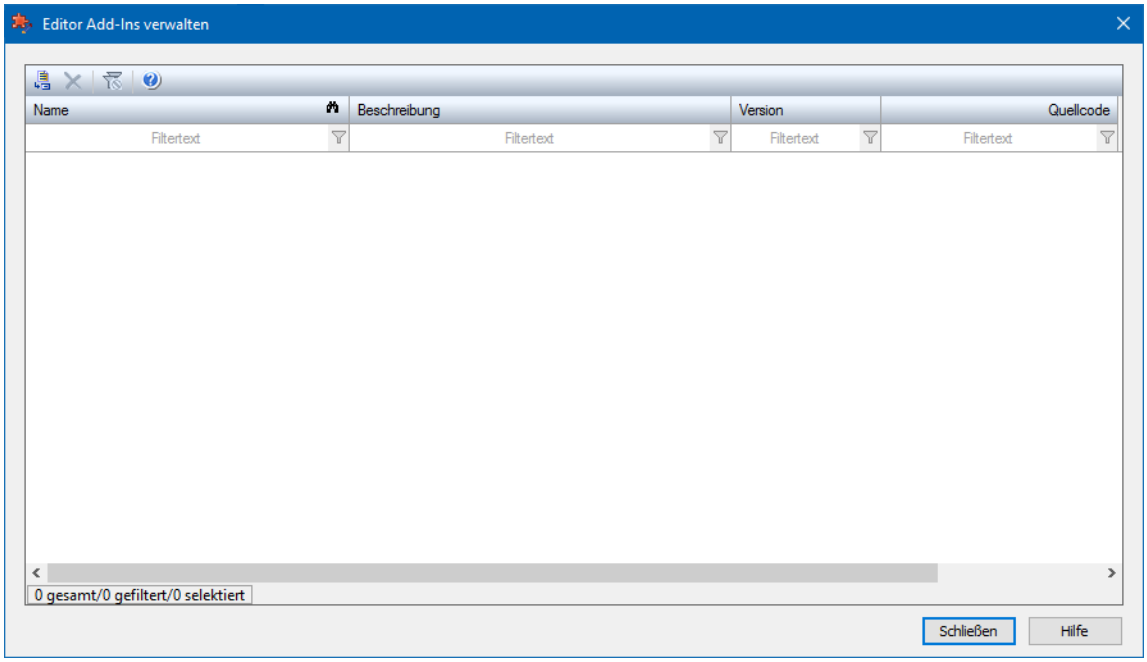
Um ein Add-In zu deinstallieren und löschen:

1. Öffnen Sie das Menü **Extras.**
2. Wählen Sie den Eintrag **Editor Add-Ins verwalten**.
   Der Dialog zur Verwaltung von Add-Ins wird geöffnet.
3. Wählen Sie ein Add-In aus.
4. Klicken Sie auf das Symbol **Deinstallieren und löschen** oder wählen Sie diesen Eintrag im Kontextmenü oder drücken Sie die Taste `Entf`.

   Ein Dialog mit einer Sicherheitsabfrage wird geöffnet.
5. Bestätigen Sie die Sicherheitsabfrage.

Das Add-In wird deinstalliert und gelöscht.

Dialog Editor Add-Ins verwalten



Option	Beschreibung
Symbolleiste	Enthält Symbole für: ▸ Importieren, Installieren und Löschen von Add-Ins ▸ Entfernen der Filter ▸ Anzeige der Hilfe
Liste der Add-Ins	Zeigt alle installierten Add-Ins an. Angezeigt werden Informationen zu: ▸ Name ▸ Beschreibung ▸ Version ▸ ob Quellcode inkludiert ist Die Elemente können gefiltert und sortiert angezeigt werden.
Schließen	Schließt den Dialog.
Hilfe	Öffnet die Online-Hilfe.

SYMBOLLEISTE UND KONTEXTMENÜ

Bedeutung der Einträge im Kontextmenü und der Symbole von links nach rechts:

Symbol	Beschreibung
Importieren und installieren	Öffnet den Dialog zur Auswahl eines Add-Ins. Dieses kann importiert und installiert werden.
Deinstallieren und löschen	Deinstalliert und löscht das ausgewählte Add-In nach einer Sicherheitsabfrage.
Alle Filter entfernen	Entfernt alle Filter, die in der Liste der Add-Ins aktuell angewendet werden.
Hilfe	Öffnet die Online-Hilfe.

Editor Dienste verwalten

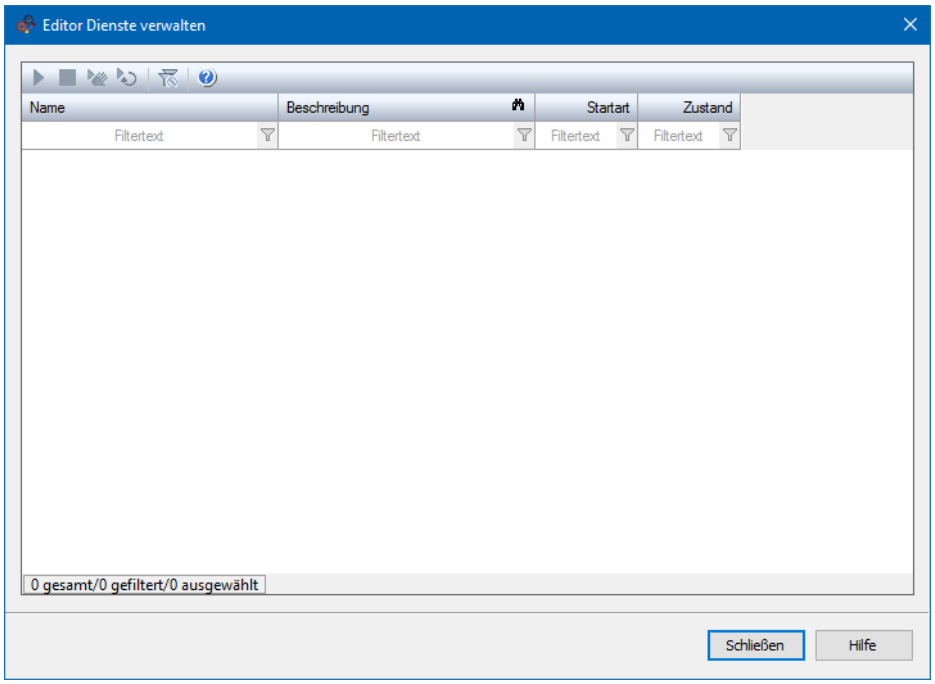
Add-In Extensions, die im Editor als Dienste laufen, werden über einen eigenen Dialog verwaltet.

Um Editor Dienste zu verwalten

1. Öffnen Sie das Menü **Extras**.
2. Wählen Sie den Eintrag **Editor Dienste verwalten...**

Der Dialog zum Verwalten der Dienste wird geöffnet.

DIALOG EDITOR DIENSTE VERWALTEN



Option	Beschreibung
Symbolleiste	Enthält Symbole für: ▸ Starten und Stoppen der Dienste ▸ Einstellen der Startart ▸ Entfernen der Filter ▸ Anzeige der Hilfe
Liste der Dienste	Zeigt alle installierten Dienste (Services) an. Angezeigt werden Informationen zu: ▸ Name ▸ Beschreibung ▸ Startart ▸ Zustand Die Elemente können gefiltert und sortiert angezeigt werden.
Schließen	Schließt den Dialog.
Hilfe	Öffnet die Online-Hilfe.

SYMBOLLEISTE UND KONTEXTMENÜ

Bedeutung der Einträge im Kontextmenü und der Symbole von links nach rechts:

Symbol	Beschreibung
Dienst starten	Startet den ausgewählten Dienst.
Dienst stoppen	Stoppt den ausgewählten Dienst.
Startart manuell	Stellt die Startart für den ausgewählten Dienst auf manuell.
Startart automatisch	Stellt die Startart für den ausgewählten Dienst auf automatisch.
Alle Filter entfernen	Entfernt alle Filter, die in der Liste der Dienste aktuell angewendet werden.
Hilfe	Öffnet die Online-Hilfe.

VERHALTEN FÜR START UND STOPP

Wurde für ein Workspace Service Add-In der Startmodus `Automatisch` gewählt, dann wird es beim Starten des Editors noch vor dem Event **OnWorkspaceStartup** geladen. Beim Beenden des Editors wird es nach dem Event **OnWorkspaceExit** gestoppt. Das Add-in kann damit auf die zenon Events reagieren.

3.5.5 Verwendung in der zenon Runtime

Add-Ins für die Verwendung in der zenon Runtime, werden im Editor importiert und verwaltet. Beim Starten der Runtime werden sie automatisch installiert, um in der Runtime ausgeführt werden zu können. Jedes zenon Projekt verwaltet seine eigenen Add-Ins.

- Add-Ins werden beim Erstellen der Runtime-Dateien für ein Projekt in den folgenden Ordner Verzeichnis: `...\RT\FILES\zenon\system\AddInStore\...`
- Beim Starten der Runtime werden die Add-Ins im folgenden Ordner installiert: `...\RT\FILES\zenon\system\AddInCache\...`

  Dabei werden weiters:

  - Geänderte Add-In Packages upgedatet.
    Der Abgleich erfolgt dabei über das Änderungsdatum des Add-In Packages.
  - Nicht mehr vorhandene Add-Ins gelöscht.
  - Project Service Extensions (auf Seite 19) mit Startmodus Automatisch werden instanziert und automatisch gestartet.
  - Project Wizard Extensions (auf Seite 18) können über die zenon Funktion **Wizard ausführen** ausgeführt werden.

> **Tipp**
>
> Den Runtime-Ordner öffnen Sie am schnellsten, indem Sie das Projekt im Editor markieren und die Tastenkombination `Strg+Alt+R` drücken.

Add-Ins für die Runtime installieren und verwalten

Add-Ins für die Runtime werden im Editor importiert und beim Starten der Runtime automatisch installiert. Der Import erfolgt über das Kontextmenü oder die Symbolleiste der Detailansicht.

ADD-IN IMPORTIEREN

Um ein Add-in zu importieren:

1. Öffnen Sie im Editor die Detailansicht für Add-Ins im Projektbaum.
2. Klicken Sie auf das Symbol **Add-In importieren**, wählen Sie diesen Eintrag im Kontextmenü oder drücken Sie die Taste `Einfg`.

   Der Dialog zur Auswahl eines Add-Ins wird geöffnet.
3. Wählen Sie das gewünschte Add-In aus.
4. Klicken Sie auf **Öffnen**.

Das Add-in wird importiert.

Info:

- Erst beim Erzeugen der Runtime-Dateien wird das Add-In in den Runtime-Dateien abgelegt. Ablage-Ordner: `...\RT\FILES\zenon\system\AddInStore\...`
- Beim Starten oder beim Nachladen der Runtime wird es im folgenden Ordner installiert, modifiziert oder aktualisiert:

  `...\RT\FILES\zenon\system\AddInCache\...`

Beim Modifizieren erfolgt der Abgleich über den Zeitstempel des Add-In Packages.

5. Klicken Sie auf **Schließen**, um den Dialog zu schließen.

Hinweis: Es können nur Add-Ins mit Project Extensions importiert und installiert werden. Werden beim Import keine Project Extensions gefunden, wird eine entsprechende Warnung angezeigt.

ADD-IN INSTALLIEREN UND DEINSTALLIEREN

Add-Ins werden automatisch beim Start der Runtime oder beim Nachladen installiert oder aktualisiert.

Ablauf:

- Beim Starten der Runtime werden die für das Projekt installierten Packages mit den im Editor importierten abgeglichen.
- Der Abgleich erfolgt über den Zeitstempel.
- Bei unterschiedlichen Zeitstempeln oder neuen Packages wird das Package installiert. Ein bereits vorhandenes Package mit gleichem Namen wird überschrieben.
- Ist ein in der Runtime vorhandenes Package im Editor nicht mehr vorhanden, wird es auch aus der Runtime entfernt.

ADD-IN LÖSCHEN

Um ein Add-In zu löschen:

1. Öffnen Sie im Editor die Detailansicht für Add-Ins im Projektbaum.
2. Wählen Sie ein Add-In aus.
3. Klicken Sie auf das Symbol **Löschen** oder drücken Sie die Taste `Entf`.

   Ein Dialog mit einer Sicherheitsabfrage wird geöffnet.
4. Bestätigen Sie die Sicherheitsabfrage.

   Das Add-In wird gelöscht.
   Nach dem nächsten Kompilieren der Runtime-Dateien wird es beim Nachladen oder Neustart der Runtime deinstalliert.

3.6 Umstieg/Konvertierung von VSTA

EDITOR

Um bestehende VSTA-Funktionalität aus dem Editor zu konvertieren, gehen Sie wie folgt vor.

FUNKTIONALITÄT AUS WIZARDS

Funktionalität die mittels VSTA-Wizards umgesetzt wurde, kann nun über Editor Wizard Extensions (auf Seite 17) umgesetzt werden.

Kopieren Sie Ihren bestehenden Code in eine Editor Wizard Extension und nehmen Sie gegebenenfalls Anpassungen im Code vor.

Zum Starten des Wizards verwenden Sie den Dialog **Editor Wizards...**

FUNKTIONALITÄT AUS VSTA EVENTS

Funktionalität die eventgetriggert (z.B. über **.OnElementCreated**, **.OnPreBuild**) ausgeführt wurde, kann nun über Editor Service Extensions (auf Seite 17) umgesetzt werden.

Kopieren Sie dazu den Code zum Initialisieren der Eventhandler in die **Start** Methode der Editor Service Extension. Den Code zum Freigeben kopieren Sie in die **Stop** Methode.

Hinweis: Für den automatischen Start muss in der **Extension** das Attribut `DefaultStartMode=DefaultStartupModes.Auto` gesetzt sein. Der Startmodus kann später auch mit dem Dialog **Editor Dienste verwalten** (auf Seite 33) verändert werden.

FUNKTIONALITÄT AUS VSTA MAKROS

Funktionalität aus Makros, die über die Combobox **VBA/VSTA Makro ausführen** ausgeführt wurde, kann nun über Editor Wizard Extensions (auf Seite 17) umgesetzt werden.

Kopieren Sie dazu den Code aus Ihrem bestehenden VSTA Makro in die **Run** Methode der Editor Wizard Extension und nehmen Sie gegebenenfalls Anpassungen im Code vor.

Zum Ausführen des Codes verwenden Sie den Dialog **Editor Wizards...**

RUNTIME

Um bestehende VSTA-Funktionalität aus der Runtime zu konvertieren, gehen Sie wie folgt vor.

FUNKTIONALITÄT AUS VSTA MAKROS

Funktionalität die über die Funktion **VSTA-Makro ausführen** ausgeführt wurde, kann nun über Project Wizard Extensions (auf Seite 18) umgesetzt werden.

Kopieren Sie dazu den Code aus Ihrem bestehenden VSTA Makro in die **Run** Methode der Project Wizard Extension und nehmen Sie gegebenenfalls Anpassungen im Code vor.

Zum Starten der Project Wizard Extension verwenden Sie die zenon Funktion **Wizard ausführen**.

FUNKTIONALITÄT AUS VSTA EVENTS

Funktionalität die eventgetriggert (z.B. über **.DynPictures().Open**, **.Alarm().AlarmComes**) ausgeführt wurde, kann nun über Project Service Extensions (auf Seite 19) umgesetzt werden.

Kopieren Sie dazu den Code zum Initialisieren der Eventhandler in die **Start** Methode der Project Service Extension. Den Code zum Freigeben kopieren Sie in die **Stop** Methode.

Hinweis: Für den automatischen Start muss in der **Extension** das Attribut `DefaultStartMode=DefaultStartupModes.Auto` gesetzt sein.

4. Makroliste

Mit VBA und VSTA kann die Funktionalität von zenon selbstständig erweitert werden. Die Verwendung von Makros mit zenon wird beschrieben.

KONTEXTMENÜ

Menüpunkt	Aktion
VBA-Editor öffnen	Öffnet den VBA-Editor.
VBE exportieren alle	Öffnet den Dialog zur Auswahl des Speicherorts für den VBE-Export.
VBE importieren	Öffnet den Dialog zur Auswahl der VBE-Importdatei.
Editorprofile	Öffnet die Dropdownliste mit vordefinierten Editorprofilen.
Hilfe	Öffnet die Online-Hilfe.

Info

Wenn VBA-Makros im Editor geändert werden:

- werden die Runtime-Dateien kompiliert und zur Runtime übertragen
- wird die Runtime neu geladen
- werden VSTA-Elemente ebenfalls neu geladen, selbst wenn keine Änderungen in VSTA erfolgt sind

VBA startet für **Arbeitsbereich** und **Projekt** dieselbe Entwicklungsumgebung.
Um den VBA Editor zu öffnen:

1. Navigieren Sie im zenon Editor in den Knoten **Programmierschnittstellen**.
2. Erweitern Sie die Ansicht dieses Knotens mit Klick auf `[+]`.
   Die Ansicht des Knotens wird erweitert.
3. Führen Sie einen Rechtsklick auf **Makroliste** aus.
4. Wählen Sie im Kontextmenü den Eintrag **VBA-Editor öffnen**.

Alternative: drücken Sie die Tastenkombination Alt+F11

4.1 VBA Symbolleiste und Kontextmenü Detailansicht

SYMBOLLEISTE

Menüpunkt	Aktion
VBA-Makro neu	Legt ein neues Makro an und öffnet den Makro-Editor.
VBA-Editor öffnen	Öffnet den VBA-Editor.
Speichern	Speichert Makros.
Entfernen	Löscht das ausgewählte Element.
VBE exportieren alle	Öffnet den Dialog zur Auswahl des Speicherorts für den VBE-Export.
VBE importieren	Öffnet den Dialog zur Auswahl der VBE-Importdatei.
Umbenennen	Ermöglicht umbenennen des ausgewählten Markos.
Hilfe	Öffnet die Online-Hilfe.

KONTEXTMENÜ MAKRO MODULE

Menüpunkt	Aktion
VBA-Editor öffnen	Öffnet den VBA-Editor.
Speichern	Speichert Makros.
VBE exportieren alle	Öffnet den Dialog zur Auswahl des Speicherorts für den VBE-Export.
VBE importieren	Öffnet den Dialog zur Auswahl der VBE-Importdatei.
Hilfe	Öffnet die Online-Hilfe.

KONTEXTMENÜ MODULE

Menüpunkt	Aktion
VBA-Makro neu	Legt ein neues Makro an und öffnet den VBA-Editor.
Hilfe	Öffnet die Online-Hilfe.

KONTEXTMENÜ MAKRO

Menüpunkt	Aktion
Bearbeiten	Öffnet Makro im Editor zum Bearbeiten. Alternativ: Taste `Eingabe` oder Doppelklick.
Entfernen	Löscht Makro. Alternativ: Taste `Entf`.
Umbenennen	Öffnet Namensfeld zum Bearbeiten. Alternativ: Taste `F2`.
Hilfe	Öffnet die Online-Hilfe.

SYMBOLLEISTE EDITOR

Makros, die mit VBA erstellt wurden, können über die Symbolleiste **Makroliste** verwaltet werden.

Symbol (von links nach rechts)	Funktion
Liste der VBA/VSTA-Makros neu laden	Lädt alle in **myWorkspace** und in Modulen enthaltenen `Public Sub Name ()` Makros in die Dropdownliste der Symbolleiste.
Makro suchen	Suche nach Makros über Eingabefeld der Combobox oder Auswahl aus Dropdownliste. Die Dropdownliste wird beim Ausklappen an das breiteste Element angepasst.
Combobox Makros	Enthält alle geladenen Makros zur Auswahl.
Ausgewähltes Makro ausführen	Führt das in der Dropdownliste ausgewählte Makro aus.
Zugeordnetes Makro #<x> ausführen	Führt das dem Symbol zugeordnete Makro aus.
Makros zuordnen	Öffnet den Dialog für die Zuordnung von Makros. Bis zu 5 Makros können mit den Symbolen 1 bis 5 verknüpft werden.
VBA	Filtert nach VBA-Makros. Es werden nur noch VBA-Markos angezeigt.
VSTA	Filtert nach VSTA-Makros. Es werden nur noch VSTA-Markos angezeigt.
ALL	Aktuell angewandter Filter wird aufgehoben und alle Makros werden angezeigt.
AZ	Sortiert Makros aufsteigend von 0 - 9 und A - Z.
ZA	Sortiert Makros absteigend von Z - A und 9 - 0.
Symbolleistenoptionen	Klick auf Pfeil öffnet Submenü: `Aktiv`: Symbolleiste wird angezeigt. Wird die Symbolleiste nicht angezeigt, kann sie über das Menü **Ansicht -> Symbolleisten** aktiviert werden. **Hinweis:** Bei freier Platzierung der Symbolleiste (vom Editor abgedockt) werden Optionen nicht angezeigt. Die Symbolleiste kann dann per Klick auf die Schaltfläche X geschlossen werden.

Info

Werden im Makro-Zuordnungsdialog nicht alle in `myWorkspace` enthaltenen Makros zur Auswahl angeboten, dann führen Sie in der Symbolleiste die Funktion **Liste der VBA-Makros neu laden** aus.

4.2 VBA auf 64-Bit Systemen

zenon unterstützt seit Version 7.10 64-Bit Betriebssysteme. VBA wurde daher auf Version VBA 7.1 umgestellt. Somit steht VBA auch in zenon 64-Bit zur Verfügung. Falls im VBA-Code auf Windows-API oder sonstige importierte DLL-Funktionen zugegriffen wird, müssen diese Aufrufe an 64-Bit angepasst werden. Generell gilt: Eine für 32-Bit erstellte VBA-Datei kann nicht ohne Anpassungen in einer 64-Bit-Version von VBA verwendet werden.

Um 32-Bit und 64-Bit kompatiblen Code zu schreiben, stehen in VBA einige Defines/Funktionen zur Verfügung. Zum Beispiel:

```
#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
#end if
#if VBA7 then
   Declare PtrSafe Sub MessageBeep Lib "User32" (ByVal N AS Long)
#else
   Declare Sub MessageBeep Lib "User32" (ByVal N AS Long)
#end if
```

Nützliche Hinweise zur Portierung von VBA 32-Bit Code auf VBA 64-Bit finden Sie auch bei Microsoft:

- Microsoft Office 2010, Hinweise zur Portierung:
  http://msdn.microsoft.com/en-us/library/ee691831.aspx
  (http://msdn.microsoft.com/en-us/library/ee691831.aspx)
- 32-Bit und 64-Bit Declares für API-Calls: http://www.jkp-ads.com/articles/apideclarations.as
  (http://www.jkp-ads.com/articles/apideclarations.as)p

4.3 Grundlagen

Beschreibt die Grundlagen der Programmiersprache VBA - Visual Basic for Applications

4.3.1 Objekteigenschaften - PROPERTIES

Eine Objekteigenschaft (Property) ist ein bestimmtes Attribut des Objekts. Im Falle eines Variablen-Objekts kann das z. B. der Wert, der Name oder die Kennung sein. Im Falle eines Kreis-Objekts etwa die Position oder die Farbe des Kreises im Bild. Jedes Objekt besitzt mindestens ein Property (in der Regel sind es mehrere), jedes Property besitzt einen bestimmten Wert. Während der **Property Name** ein Text ist, besitzt der **Property Value** einen Wert zwischen 0 und z. B. 1000.

Das Besondere an Properties ist, dass sie durch Ändern der Propertywerte in einem VBA-Programm das Verhalten oder Aussehen des Objekts verändern können. Geben Sie etwa dem **Property Value** (Wert) des Variablenobjekts einen anderen Wert, erreichen Sie dadurch, dass die aktuell ausgewählte Variable den von Ihnen gewünschten Wert erhält. Nicht jedem Property können Sie einen neuen Wert geben. So ist **Property Count** (Anzahl) des Variablenobjekts fix, denn die Anzahl der Variablen wird durch die angelegten Variablen bestimmt. Indem Sie Count einen höheren Wert geben, fügen Sie keine neuen Variablen hinzu. Einige Properties sind daher Nur-Lese-Eigenschaften, d.h. ihr Wert kann nur gelesen werden.

4.3.2 Objektmethoden - METHODS

Neben den Properties kann jedes Objekt Methoden besitzen. Eine Methode ist kein Attribut, sondern eine Aufforderung an das Objekt, etwas zu tun. So besitzt eine Form die Methode `Show`. Was bewirkt diese Methode? Sie fordert die Form auf, sich am Bildschirm darzustellen. Entsprechend verschwindet die Form, unter Verwendung des `Unload`-Befehls wieder.

Der Vorteil einer Methode ist, dass der Programmierer nichts über den Aufbau des Objekts wissen muss und vor allem keine Gelegenheit erhält, an den internen Daten des Objekts zu drehen.

Der Aufruf der Show-Methode oder des Unload-Befehls sieht wie folgt aus:

`frmSollwert.Show` oder `Unload frmSollwert`

Wenn Sie zum Beispiel eine andere Form aufschalten wollen, bleibt die Methode gleich, es ändert sich lediglich der Name der Form (Objektname).

`frmChange.Show` oder `Unload frmChange`

Ein und dieselbe Methode kann daher auf verschiedene Objekttypen angewendet werden. Jedoch nicht alle Objekte müssen zwangsläufig Methoden besitzen.

4.3.3 Objektereignisse - EVENTS

Auch wenn Sie sich beim Umgang mit Objekten zu 90% mit Properties und Methoden beschäftigen, gibt es noch eine dritte Kategorie von Merkmalen, über die ein Objekt verfügen kann: Events. Einige Objekte in der Objekthierarchie können auf Events reagieren. Events passieren beim Arbeiten mit zenon von alleine.

Beispiel

Wann immer ein Bild aufgeschalten wird, wird in dem dahinterstehenden Picture-Objekt ein Open-Event ausgelöst. Als Programmierer können Sie innerhalb der sogenannten Ereignissprozedur (Prozedur, die bei Eintreten eines Ereignisses aufgerufen wird) Anweisungen festlegen, die bestimmen, was in diesem Fall passieren soll. Ein Beispiel dafür ist die Änderung einer Variablen. Man könnte ein Event, welches auf die Wertänderung einer Variablen reagiert, erzeugen.

4.3.4 VBA Objektmodell in zenon

Grundsätzlich gibt es in der Projektstruktur immer wieder eine Objektliste und Objekte.

Beispiel:

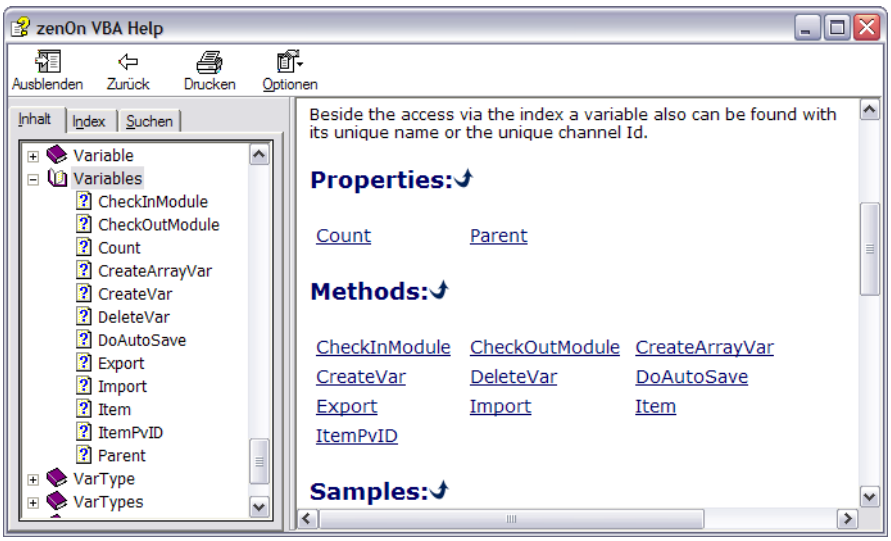
Projects – Project

Variables – Variable

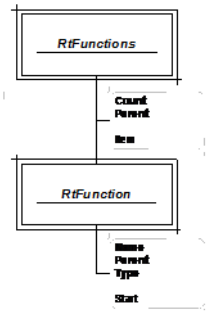
Elements – Element

Mehr zum Objektmodells finden Sie:

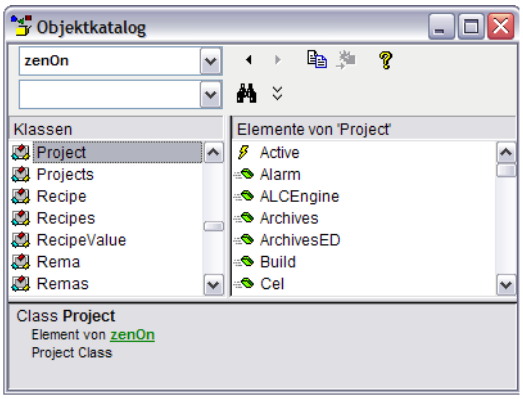
- in der VBA-Hilfe



- in der grafischen Übersicht, die Sie bei COPA-DATA komplett als gedruckte Übersicht erhalten

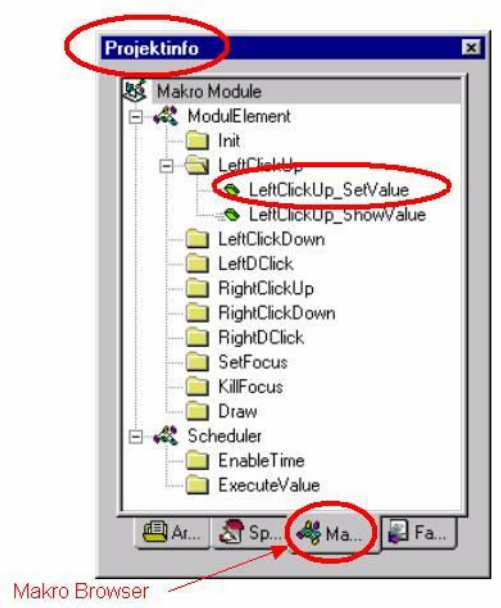


- im Object-Browser von VBA



4.3.5 Wie setzt man VBA Makros ein?

Um ein neues Makro zu erstellen, wählt man im Fenster **Projektinfo** in der Registerkarte **Makro Browser** ein gewünschtes Ereignis, bei dem das neue Makro ausgeführt werden soll.



Durch Klicken der rechten Maustaste auf dieses Ereignis, öffnet sich ein Menü.

Wählen Sie den Menüpunkt Makro neu. Dadurch wird von zenon eine Prozedur generiert:

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

Wenn bereits ein Makro besteht, kann dieses durch Anklicken mit der rechte Maustaste, bearbeitet, gelöscht oder umbenannt werden.

> **Achtung**
>
> *Wenn Sie den Menüpunkt **Makro umbenennen** wählen, achten Sie darauf, dass Sie nicht den Namen des Ereignisses - z. B.: LeftClickUp_..., - des derzeitigen Namens ändern. Ansonsten wird die Namensänderung nicht durchgeführt. Außerdem müssen Sie, wenn Sie den Makronamen ändern, auch den Namen des aufzurufenden Unterprogramms per Hand im VBA-Editor ändern.*

Nachdem man die von zenon generierte Prozedur mit dem auszuführenden Quellcode gefüllt hat, muss man das erzeugte Makro mit einem Element verbinden.

Durch einen Doppelklick auf ein gewünschtes Element, erscheinen die Element-Eigenschaften auf dem Bildschirm.

Auf der Registerkarte Ereignisse, verbindet man das Makro mit dem Element.

Durch einen Klick mit der linken Maustaste, wird das Ereignis LeftClickDown des Elements ausgelöst und das verbundene Makro ausgeführt.

Einbinden von bereits bestehenden Makros

Um bereits bestehende Makros in ein anderes Projekt einzubinden, gehen Sie so vor:

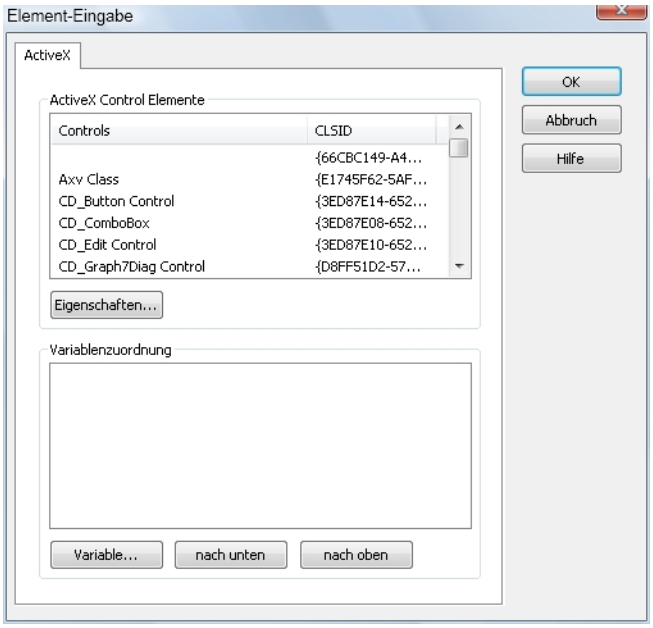
1. Exportieren Sie im VBA-Editor alle benötigten Formulare und Module und importieren diese in das gewünschte Projekt.
2. Ereignisbezogene Makros, in `ModulElement.bas`, werden jetzt noch nicht im Makro Browser angezeigt. Darum müssen Sie jetzt die Makros im Makro Browser neu anlegen.

   Am einfachsten ist es, wenn Sie den Namen des bereits bestehenden Makros verwenden.

   z. B.:
   LeftClickUp_ `DateSet2`
   LeftClickUp_ `DateSet4`
   LeftClickUp_ `TimeSet`
   Draw_ `Date2`
   Draw_ `Date4`
   Draw_ `Time`
3. Beim Anlegen von Makros, generiert zenon Prozeduren, die den gleichen Namen haben wie die bereits bestehenden Makros. Diese neu generierten Prozeduren müssen Sie löschen.
4. Verbinden Sie die Makros wie gewohnt mit einem Dynamischen Element.

4.3.6 Wie bindet man ein ActiveX - Element in zenon ein?

Ein ActiveX-Element wird wie ein Dynamisches Element einfach in einem Bild auf die gewünschte Größe aufgezogen, es öffnet sich ein Fenster, in dem Sie ein ActiveX-Element auswählen müssen.



- Wenn in der oben dargestellten Liste das Element ausgewählt wurde, können Sie diesem eine Variable zuordnen. Dazu klicken Sie auf die Schaltfläche **Variable...** und wählen Sie eine Variable aus oder legen eine neue an.
- Als nächsten Schritt geben wir unserem ActiveX-Element einen Objektnamen, über diesen wir später in VBA darauf zugreifen können.
- In unserem Beispiel vergeben wir den Objektnamen Slider6_DW18, da es sich um ein Slider-ActiveX-Element handelt, das mit der Variable Datenwort 18 verknüpft ist.
- Nun muss das Slider-Element über den VBA-Editor aktiviert und weiterbearbeitet werden.
- Dazu legen wir ein neues Makro an, wie bereits im Kapitel "Wie setzt man VBA Makros ein? (auf Seite 48)" beschrieben.

Das Makro Init_slider übergibt das zu initialisierende Element an ein Unterprogramm im Objekt **thisProject**, wodurch ein Projektbezug auf das aktuelle Projekt hergestellt wird:

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

Wie im Makro Init_Slider wird auch im Draw_SliderValue, das Element dem Objekt thisProject übergeben.

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

```
obElem.Draw hdc
End Sub
```

Im Objekt thisProject wird der unten angeführte Quellcode eingetragen.

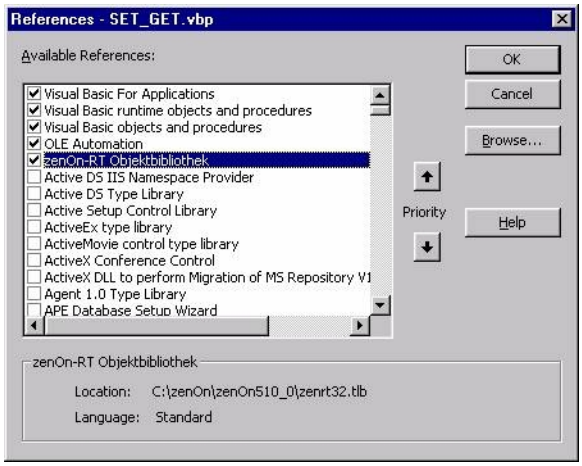
```
Public Declarations
Public WithEvents obSlider As Slider
Public obSliderPV As Variable
Public Sub Init_Slider (obElem As Element)
Set obSlider = obElem.ActiveX
'ActiveX exists
If obSchieberegler Is Nothing Then
Exit Sub
End If
Set obSliderPV = obElem . ItemVariable(0)
'variable exists
If obSliderPV 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 _ Schieberegler ( 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 Zugriff über externes Programm

Um über ein externes Programm wie zum Beispiel Visual Basic auf zenon Daten zugreifen zu können wird die COM-Schnittstelle verwendet. Diese COM-Schnittstelle ist jene, die auch von VBA verwendet wird. Es gibt also nur ein paar kleine Unterschiede, die aber bedacht werden müssen.

Visual Basic 6

Um auf die COM-Schnittstelle zugreifen zu können, muss diese eingebunden weden:



Mittels dieser Typelibrary kommt man auf das zenon Applicationsobjekt (die Runtime).

Da es hier kein thisProject - Objekt gibt, muss man dieses erstellen um an die Daten zu kommen.

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

Wenn das VB-Projekt bei allen zenon Projekten funktionieren und sozusagen unabhängig vom Projektnamen sein soll, kann es auch so gesetzt werden:

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

Wenn das Projektobjekt (thisProject) angelegt wurde, kann zum Beispiel auf die Variablen lesend und schreibend zugegriffen werden.

Lesen:

```
Wert = obProject.Variables.Item(Variablenname).Value
```

Schreiben:

```
obProject.Variables.Item(Variablenname).Value = Wert
```

4.3.8 Funktionsweise einer OnlineVariable

Eine VBA - OnlineVariable kann man sich wie einen Container vorstellen, dieser Container beinhaltet Variablen, die natürlich hinein gegeben werden müssen. Ändert sich nun der Wert einer im Container befindlichen Variablen, wird dies durch ein Ereignis bekanntgegeben.

Funktionsweise des Ereignisses

Wenn der Container aktiviert wird (`Container.Define`), werden alle im Container befindlichen Variablen einmal angereizt, damit der aktuelle Wert der Variablen bekannt wird. Es wird also die Prozedur „`Container_VariableChange`“ für jede im Container befindliche Variable einmal ausgeführt. Wenn alle Variablen dann initialisiert wurden, tritt das Ereignis immer dann in Kraft, wenn sich im Container der Wert einer Variablen ändert.

Es ist somit ausgeschlossen, dass ein Wert ausgelesen wird, der nicht dem aktuellen Wert der Variable entspricht.

Container definieren und anlegen

Definition:

```
Public WithEvents Container As OnlineVariable
```

Mit dieser Codezeile wird der Container definiert.

Anlegen:

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

Variablen in den Container geben

```
Container . Add Variablenname1
Container . Add Variablenname2
Container . Add Variablenname3
Container . Add Variablenname4
...
```

Dieser Vorgang wird solange wiederholt, bis sich alle gewünschten Variablen im Container befinden.

Ereignis anlegen

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

Dieses Ereignis wird automatisch angelegt, wenn in der linken Combobox am oberen Rand des BA-Editors der Container ausgewählt wird. Es wird dann die obige Prozedur in den Sourcecode eingefügt. Es wird mittels „`obVar`“ die Variable mitgegeben, deren Wert sich geändert hat. Es kann beim Eintreten des Ereignisses dann z. B. der aktuelle Wert abgefragt werden (`ob-Var.Value`). Welche Eigenschaften und Werte von der Variablen abgefragt werden können, ist aus der Objekthierarchie in der VBA-Dokumentation ersichtlich.

Ereignis aktivieren

```
Container.Define
```

Diese Befehlszeile aktiviert die Überwachung der Variablen im Container. Erst wenn der Befehl „**Define**“ ausgeführt wird, ist der Container aktiv.

Ausschalten des Ereignisses

```
Container.Undefine
```

Mit diesem Befehl wird die Überwachung im Container wieder abgeschaltet, das Ereignis (VariableChange) wird ab diesem Zeitpunkt nicht mehr ausgeführt.

Entfernen beim Beenden

Um beim Beenden der Runtime wirklich nichts mehr im Speicher zu lassen, muss der Container spätestens beim Beenden der Runtime wieder entfernt werden.

```
thisProject . OnlineVariables . DeleteOnlineVariables ( Containername)
```

Erst wenn dieserer Container gelöscht wurde, kann ein anderer Container mit dem gleichen Namen neu angelegt werden.

4.3.9 Liste Status-Bits

Bitnummer	Kurzbezeichner	Langbezeichnung	zenon Logic Bezeichner
0	M1	Benutzerstatus 1, oder Aktionstyp Sperre bei Befehlsgabe, oder Service Tracking (Main.chm::/IEC850.chm::/117281.htm) des IEC 850 Treibers.	_VSB_ST_M1
1	M2	Benutzerstatus 2	_VSB_ST_M2
2	M3	Benutzerstatus 3	_VSB_ST_M3
3	M4	Benutzerstatus 4	_VSB_ST_M4
4	M5	Benutzerstatus 5	_VSB_ST_M5
5	M6	Benutzerstatus 6	_VSB_ST_M6
6	M7	Benutzerstatus 7	_VSB_ST_M7
7	M8	Benutzerstatus 8	_VSB_ST_M8
8	NET_SEL	Select im Netzwerk	_VSB_SELEC
9	REVISION	Revision	_VSB_REV
10	PROGRESS	In Bedienung	_VSB_DIREC
11	TIMEOUT	Befehl Laufzeit-Überschreitung (command runtime exceeded)	_VSB_RTE
12	MAN_VAL	Handwert	_VSB_MVALUE
13	M14	Benutzerstatus 14	_VSB_ST_14
14	M15	Benutzerstatus 15	_VSB_ST_15
15	M16	Benutzerstatus 16	_VSB_ST_16
16	GI	Generalabfrage	_VSB_GR
17	SPONT	Spontan	_VSB_SPONT
18	INVALID	Ungültig	_VSB_I_BIT
19	T_STD_E	Standardzeit (Winterzeit) extern **Achtung:** bis Version 7.50 war dies das Statusbit T_CHG_A	_VSB_SUWI
20	OFF	Abgeschaltet	_VSB_N_UPD
21	T_EXTERN	Echtzeit - externer Zeitstempel	_VSB_RT_E
22	T_INTERN	Interner Zeitstempel	_VSB_RT_I
23	N_SORTAB	Nicht sortierfähig	_VSB_NSORT
24	FM_TR	Störmeldung Trafowert	_VSB_DM_TR
25	RM_TR	Laufmeldung Trafowert	_VSB_RM_TR

26	INFO	Info der Variable	_VSB_INFO
27	ALT_VAL	Ersatzwert	_VSB_AVALUE
28	RES28	Reserviert für interne Verwendung (Alarmblinken)	_VSB_RES28
29	N_UPDATE	Nicht aktualisiert (zenon Netzwerk)	_VSB_ACTUAL
30	T_STD	Standardzeit intern	_VSB_WINTER
31	RES31	Reserviert für interne Verwendung (Alarmblinken)	_VSB_RES31
32	COT0	Übertragungsursache Bit 1	_VSB_TCB0
33	COT1	Übertragungsursache Bit 2	_VSB_TCB1
34	COT2	Übertragungsursache Bit 3	_VSB_TCB2
35	COT3	Übertragungsursache Bit 4	_VSB_TCB3
36	COT4	Übertragungsursache Bit 5	_VSB_TCB4
37	COT5	Übertragungsursache Bit 6	_VSB_TCB5
38	N_CONF	Negative Bestätigung des Kommandos durch Gerät (IEC 60870 [P/N])	_VSB_PN_BIT
39	TEST	Test-Bit (IEC 60870 [T])	_VSB_T_BIT
40	WR_ACK	Schreiben bestätigt	_VSB_WR_ACK
41	WR_SUC	Schreiben erfolgreich	_VSB_WR_SUC
42	NORM	Normalzustand	_VSB_NORM
43	N_NORM	Abweichung Normalzustand	_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	Externer Zeitstempel ungültig	nicht definiert
50	CB_TRIP	Schalterfall erkannt	nicht definiert
51	CB_TR_I	Schalterfallerkennung inaktiv	nicht definiert
52	OR_DRV	Wert außerhalb des gültigen Bereiches (IEC 61850)	nicht definiert
53	T_UNSYNC	ClockNotSynchronized (IEC 61850)	nicht definiert
54	PR_NR	In Prozess-Rekorder nicht aufgezeichnet	nicht definiert

55	RES55	reserviert	nicht definiert
56	RES56	reserviert	nicht definiert
57	RES57	reserviert	nicht definiert
58	RES58	reserviert	nicht definiert
59	RES59	reserviert	nicht definiert
60	RES60	reserviert	nicht definiert
61	RES61	reserviert	nicht definiert
62	RES62	reserviert	nicht definiert
63	RES63	reserviert	nicht definiert

Info

In Formeln stehen alle Statusbits zur Verfügung. Für andere Verwendungen kann die Verfügbarkeit eingeschränkt sein.

Details zur Statusverarbeitung lesen Sie im Kapitel Statusverarbeitung.

4.3.10 Lasso für Auswahl Dynamischer Elemente zur Runtime

Dynamische Elemente, die mit einer Variablen oder Funktion verknüpft sind, können zur Runtime mit dem Lasso vorausgewählt und damit für Events genutzt werden.

Mit der Methode **SelElements** können zur Runtime vom Bediener ausgewählte Dynamische Elemente als ausgewählt identifiziert werden. Diese **DynPicture.SelElements** können dann für Events wir Drag&Drop benutzt werden.

AUSWAHL PER LASSO

Um zur Runtime Elemente mit dem Lasso auszuwählen, muss:

- in den Projekteigenschaften die Eigenschaft **Runtime Lasso** aktiv sein
- in den Eigenschaften des Dynamischen Elements die Eigenschaft **Runtime/mit Lasso auswählbar** aktiv sein
- die Eigenschaft **Schablone per Maus verschieben** deaktiviert sein

Zur Runtime stehen mehrere Methoden zur Auswahl von Elementen zur Verfügung, je nach Einstellungen für Touchbedienung:

- Touch-Bedienung deaktiviert oder Windows 7 (Eigenschaft **Auswertung** `Deaktiviert` oder `Windows 7`):

- `Linker Mausklick + Bewegung`: Neue Auswahl wird erstellt.
- `Linker Mausklick + Taste Ctrl + Bewegung`: Auswahl wird erweitert.

- Native Windows 8 Touch-Bedienung aktiv (Eigenschaft **Auswertung** auf `Windows 8`):
  - `Linker Mausklick + Bewegung`: Bild wird bewegt
  - `Linker Mausklick + Taste Ctrl + Bewegung`: Bild wird bewegt.
  - `Linker Mausklick + Taste Umschalt + Bewegung`: Neue Auswahl wird erstellt.
  - `Linker Mausklick + Taste Ctrl + Taste Umschalt + Bewegung`: Auswahl wird erweitert.

Regeln:

- Es können nur Elemente ausgewählt werden, die sichtbar sind.
- Wird ein Button ausgewählt und ein Klick darauf ausgeführt, wird die hinterlegte Funktion ausgeführt.
- Wird ein Mausklick auf einen Button ausgeführt und die Maus vor dem Loslassen der Maustaste bewegt, dann wird die hinterlegte Funktion nicht ausgeführt.
- Auswahl aufheben: Ein Lasso aufspannen, das keine Elemente enthält.

4.4 Makros im Editor

Makros können im Editor über eine konfigurierbare Symbolleiste (auf Seite 59) ausgeführt werden. Dazu werden Makros mit Schaltflächen in der Symbolleiste **VBA** verknüpft (auf Seite 61).

Weiters können Makros natürlich manuell über den VBA Editor ausgeführt werden.

Mit Wizards können immer wiederkehrende Projektierarbeiten automatisch ausgeführt oder ganze Projekte auf Knopfdruck erstellt werden. Exemplarisch werden schon einige Wizards mit zenon ausgeliefert, die nach Belieben erweitert und ergänzt werden können. Sie helfen bei der Erstellung von Projekten, beim Import und Export, bei der Variablenerstellung und vielem mehr. Details dazu finden Sie im Kapitel Wizards.

EDITOR-EVENTS

Editor-Events sind Bestandteil des VBA-Workspace und ermöglichen es, auf Events in der Workspace-Programmierung zu reagieren, z. B. für Wizards oder Remote-Transport. Zum Beispiel:

- OnElementCreated
- OnElementDeleted
- OnElementDoubleClicked
- OnObjectCreated

- ...

Alle Events und Informationen dazu finden Sie in der Hilfe zum Object Model.

4.4.1 Symbolleiste Makroliste

Makros, die mit VBA erstellt wurden, können über die Symbolleiste **Makroliste** verwaltet werden.

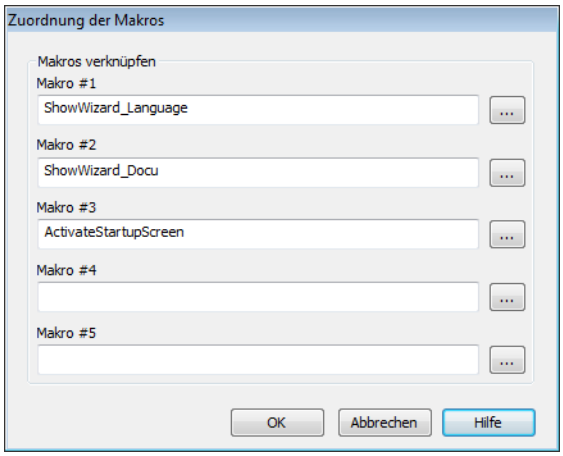
Symbol (von links nach rechts)	Funktion
Liste der VBA/VSTA-Makros neu laden	Lädt alle in **myWorkspace** und in Modulen enthaltenen `Public Sub Name ()` Makros in die Dropdownliste der Symbolleiste.
Makro suchen	Suche nach Makros über Eingabefeld der Combobox oder Auswahl aus Dropdownliste. Die Dropdownliste wird beim Ausklappen an das breiteste Element angepasst.
Combobox Makros	Enthält alle geladenen Makros zur Auswahl.
Ausgewähltes Makro ausführen	Führt das in der Dropdownliste ausgewählte Makro aus.
Zugeordnetes Makro #<x> ausführen	Führt das dem Symbol zugeordnete Makro aus.
Makros zuordnen	Öffnet den Dialog für die Zuordnung von Makros. Bis zu 5 Makros können mit den Symbolen 1 bis 5 verknüpft werden.
VBA	Filtert nach VBA-Makros. Es werden nur noch VBA-Markos angezeigt.
VSTA	Filtert nach VSTA-Makros. Es werden nur noch VSTA-Markos angezeigt.
ALL	Aktuell angewandter Filter wird aufgehoben und alle Makros werden angezeigt.
AZ	Sortiert Makros aufsteigend von 0 - 9 und A - Z.
ZA	Sortiert Makros absteigend von Z - A und 9 - 0.
Symbolleistenoptionen	Klick auf Pfeil öffnet Submenü: `Aktiv`: Symbolleiste wird angezeigt. Wird die Symbolleiste nicht angezeigt, kann sie über das Menü **Ansicht -> Symbolleisten** aktiviert werden. **Hinweis:** Bei freier Platzierung der Symbolleiste (vom Editor abgedockt) werden Optionen nicht angezeigt. Die Symbolleiste kann dann per Klick auf die Schaltfläche X geschlossen werden.

Info

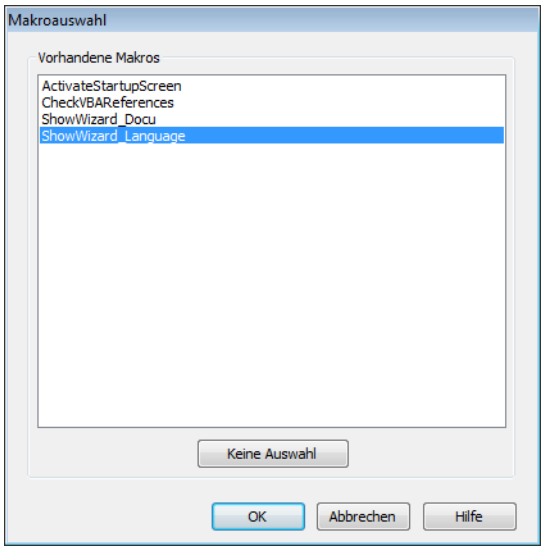
Werden im Makro-Zuordnungsdialog nicht alle in `myWorkspace` enthaltenen Makros zur Auswahl angeboten, dann führen Sie in der Symbolleiste die Funktion **Liste der VBA-Makros neu laden** aus.

4.4.2 Zuordnung der Makros

Makros können über eine Schaltfläche in der Symbolleiste aufgerufen werden. Maximal fünf Makros können so verknüpft werden. Über die Schaltfläche Makros **zuordnen** wird der Dialog zur Makrozuordnung geöffnet.



Parameter	Beschreibung
Makro #	Makro-Nummer entspricht Nummer der Schaltfläche in der Symbolleiste. Klick auf **...** öffnet Dialog zur Auswahl des Makros.
OK	Erstellt Verknüpfungen zu den Schaltflächen und schließt den Dialog.
Abbrechen	Verwirft alle Änderungen und schließt den Dialog.
Hilfe	Öffnet die Online-Hilfe.



Parameter	Beschreibung
Vorhandene Auswahl	Liste der Makros, die verknüpft werden können.
Keine Auswahl	Löscht vorhandene Zuordnung für diese Schaltfläche.
OK	Ordnet gewähltes Makro der Schaltfläche zu.
Abbrechen	Verwirft alle Änderungen und schließt den Dialog.
Hilfe	Öffnet die Online-Hilfe.

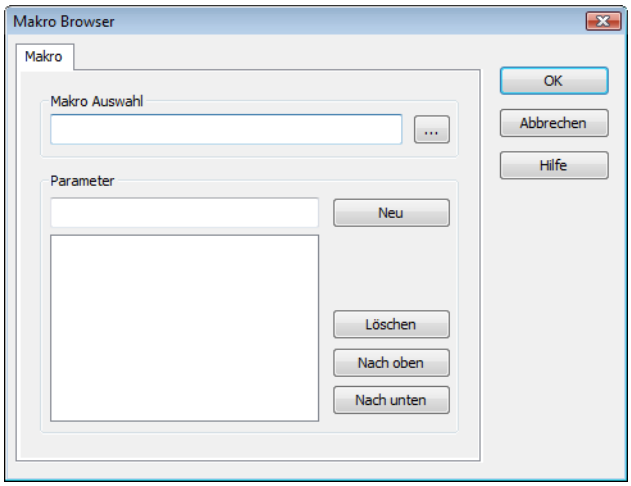
4.5 Funktionen in zenon

Im Dialog **Funktionsauswahl** finden Sie unter dem Element **VBA** folgende Funktionen.

Funktion	Beschreibung
PCE-Editor öffnen	Öffnet den Editor des optionalen Moduls Process Control Engine (PCE).
VBA-Editor öffnen	Öffnet den VBA-Editor
VBA-Makro ausführen	Führt ein ausgewähltes VBA-Makro aus. Achtung: Der VBA Event `project inactive` wird vor `AUTO_END_xxx Skripte` ausgeführt. Daher wird in Skripten die zenon Funktion **VBA-Makro ausführen** dann nicht mehr ausgeführt, da VBA zu diesem Zeitpunkt nicht mehr läuft.
VBA-Makros Dialog anzeigen	Öffnet den VBA-Makro Dialog.

4.5.1 VBA-Makro ausführen

Wenn Sie die Funktion VBA-Makro ausführen auswählen, erscheint folgender Dialog.



Diese Einstellungen stehen Ihnen zur Verfügung.

Parameter	Beschreibung
Makro Auswahl	Öffnet den Dialog zur Auswahl des Makros (siehe Makro Auswahl (auf Seite 64)) Hint: Only lists VBA macros that match the number of parameters defined at the function Parameter (below).
Parameter	Tragen Sie den gewünschten Wert (String) für einen Parameter ein.
Neu	Klicken Sie auf diese Schaltfläche, um den Wert bei Parameter in die Liste der verfügbaren Parameter zu übernehmen.
Löschen	Klicken Sie auf diese Schaltfläche, um den ausgewählten Eintrag aus der Liste der verfügbaren Parameter zu löschen. Sie können immer jeweils nur einen Eintrag löschen.
Nach oben	Klicken Sie auf diese Schaltfläche, um den ausgewählten Eintrag eine Position nach oben zu verschieben. In der Parameter-Reihenfolge wird der Eintrag damit eine Position nach vorne verschoben.
Nach untern	Klicken Sie auf diese Schaltfläche, um den ausgewählten Eintrag eine Position nach unten zu verschieben. In der Parameter-Reihenfolge wird der Eintrag damit eine Position nach hinten verschoben.

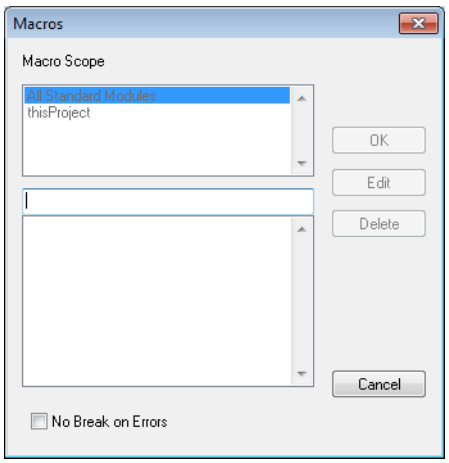
Sie haben die Möglichkeit für Makros, die mit Parameter erstellt wurden, Strings zu hinterlegen. Diese Strings werden in der Runtime als einzelne Parameter übertragen, wenn das Makro ausgeführt wird.

Info

Sie müssen sicher stellen, dass die Anzahl der Parameter des verknüpften Makros mit der Anzahl der hier angelegten Parameter übereinstimmt.

Makro Auswahl

Nach Klick auf Schaltfläche **...** erscheint folgender Dialog.



Wählen Sie hier aus den vorhandenen Makros das gewünschte aus und klicken Sie anschließend auf **OK**.

4.6 Entwicklung von Wizards in VBA

Ab Version 6 ist es möglich, die Projektierung von Projekten über Wizards zu automatisieren. Dadurch können oft zu wiederholende Tätigkeiten in einen Wizard ausgelagert werden, der die gewünschten Aktionen durchführt, z. B. Anlegen eines Projekts, Erstellen der Schablonen und Bilder nach einem definierten Standard.

Eine andere Einsatzmöglichkeit für Wizards sind automatisierte Änderungen an einem bestehenden Projekt, z. B. Änderung von Eigenschaften dynamischer Elemente in allen Bildern eines bestehenden Projekts.

Als Basis für die Wizards dienen Microsoft Visual Basic for Applications (VBA) und das zenon Objektmodell.

Derzeit stehen folgende Wizards zur Verfügung:

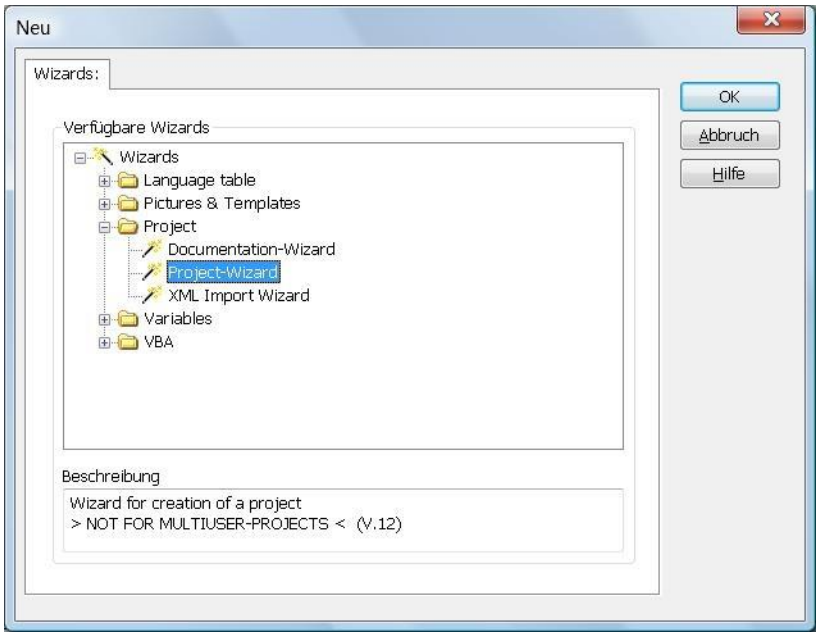
- Project-Wizard
- Import-Wizard
- World view wizard
- Find VBA-Text wizard
- Wizard for keyword-creation
- Wizard for keyword-translation

- Wizard for creating variables

Die Wizards stehen als VBA-Quellcode Dateien auf dem Installationsmedium zur Verfügung. Neue Wizards können über die VBA-Umgebung eingebunden werden.

4.6.1 Verwendung eines Wizards

Der Menübefehl **Editor Wizards** im Menü **Extras** öffnet die Wizard-Auswahl. In diesem Dialog werden alle verfügbaren Wizards nach ihrer Kategorie angezeigt.
Sollten Wizards keine Kategorie beinhalten, so wird automatisch ein Eintrag **Nicht verknüpft** erstellt. Unter diesem Eintrag werden alle nicht-verknüpften Assistenten aufgelistet.



Durch Auswahl eines Wizards und Betätigen der **OK**-Schaltfläche wird der ausgewählte Wizard aktiviert.

Info

Wizards unterstützen keine Mehrplatzfähigen Projekte.

4.6.2 Aufbau eines Wizards

Ein Wizard ist eine UserForm, die im applikationsspezifischen Knoten der Applikation abgelegt wird. Diese UserForm besteht in der Regel aus einem Multiseiten-Element, in dem die einzelnen Schritte des Wizards dargestellt werden.

Durch eine **Weiter** Schaltfäche wird die nächste Seite des Multiseiten-Elements angezeigt. Alle Eingaben müssen temporär abgelegt werden - die Erstellung von Objekten, z. B. Schablonen, Bilder, ... darf erst beim **Fertig stellen** des Wizards erfolgen.

Info

UserForms, die als Wizard verwendet werden sollten, müssen einige public Methoden enthalten, die die Informationen über den Wizard an das Leitsystem liefern. Fehlen diese Routinen in einer UserForm, so wird diese nicht als Wizard behandelt.

4.6.3 Integration in VBA

Die Wizards sind im applikationsspezifischen Knoten **ZWorkspace** untergebracht. Dieses Objekt stellt den geladenen Workspace im Editor dar und steht nur im zenon Editor zur Verfügung.

Alle Objekte in diesem VBA-Projekt können über das **MyWorkspace**-Objekt auf den aktuellen Workspace zugreifen. Dieser ist immer mit dem derzeit aktiven Projekt verknüpft, auf dass Sie über die **ActiveDocument**-Eigenschaft zugreifen können.

Die Inhalte des ZWorkspace-Objekts sind in der Datei `ZenWorkspace.vba` enthalten. Diese wird mit der erstmaligen Installation der Version 6 ins Installationsverzeichnis kopiert. Durch später folgende Updates wird diese Datei nicht überschrieben. Zur Aktualisierung von Wizards erhalten Sie am Ende des Tutorials weitergehende Informationen.

4.6.4 Entwicklung eines Wizards

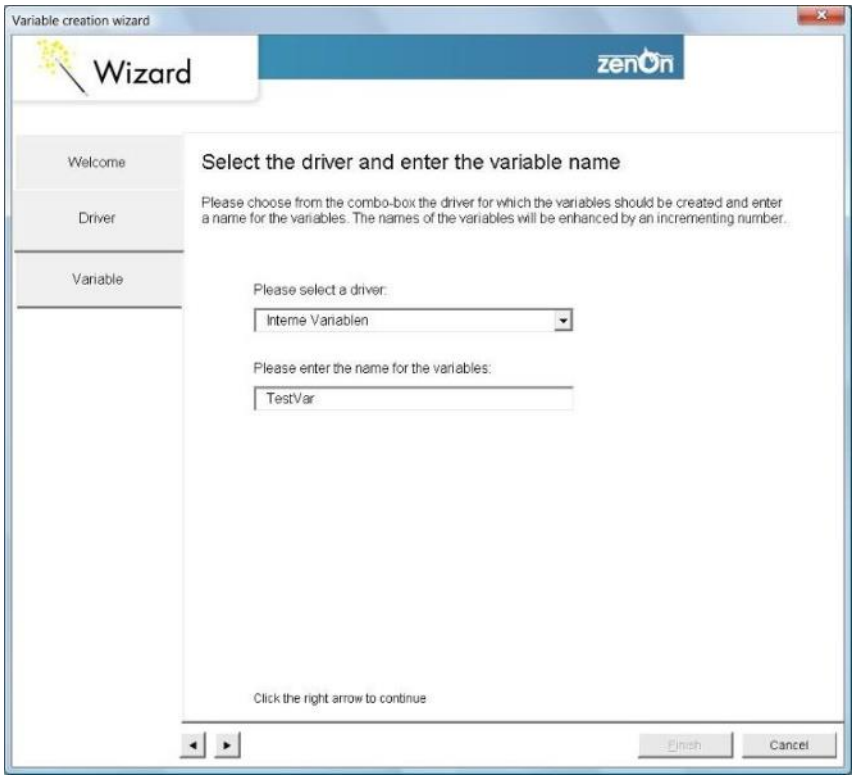
Dieses Tutorial erstellt einen Wizard zum Anlegen von Variablen zu einem projektierten Treiber.

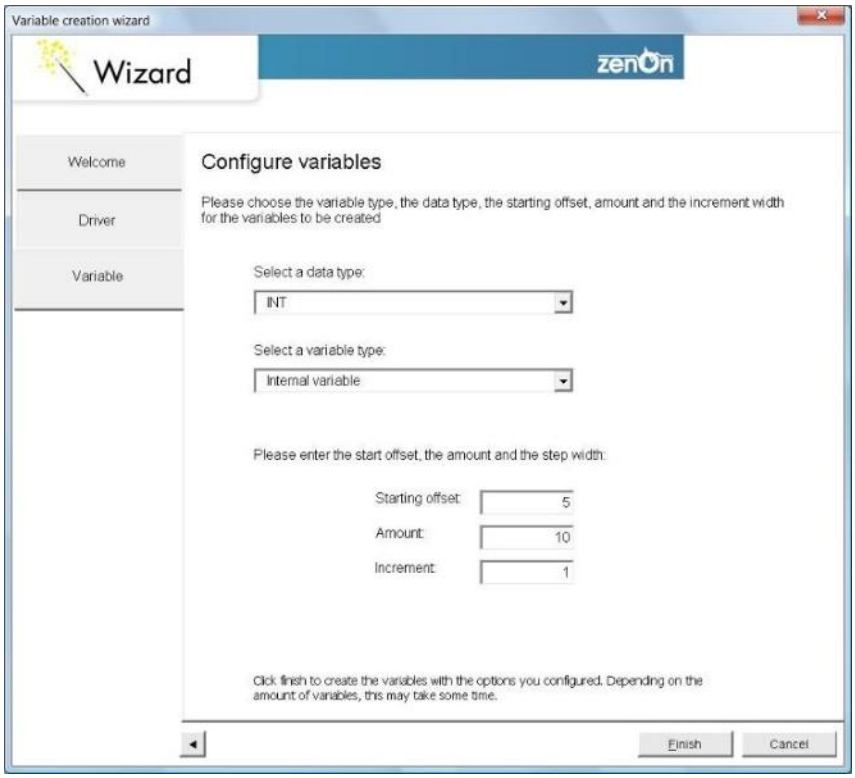
Starten Sie die VBA-Umgebung aus dem zenon Editor und wechseln Sie in den Ordner **ZWorkspace/Formulare**. Diese Datei enthält die Grundlagen für die Entwicklung eines Wizards. Ändern Sie den Namen der UserForm.

Sollte der Wizard in dem oben genannten Ordner nicht zur Verfügung stehen, können Sie diesen über den Befehl **Datei importieren** importieren.

Info

Zur Entwicklung eines Wizards sind Kenntnisse über das zenon Objektmodell und VBA erforderlich. Diese Themane werden in diesem Tutorial nicht näher erläutert.





Erstellen Sie die oben dargestellte Oberfläche. Wechseln Sie anschließend ins Code-Modul der UserForm und scrollen Sie bis ans Ende der Datei. Dort befinden sich folgende Methoden:

- **`Public Function`** GetWizardName () **`As String`**

  Liefert den eindeutigen Namen des Wizards zurück. Ändern Sie den Inhalt in `Wizard for creating variables`

- **`Public Function`** GetWizardInfo () **`As String`**

  Liefert eine kurze Beschreibung, die in der Wizard-Auswahl angezeigt wird, zurück. Ändern Sie den Inhalt in `Wizard for creating variables to a selected driver`

- **`Public Function`** GetWizardCategory () **`As String`**

  Liefert die Kategorie des Wizards zurück. In der Wizardauswahl werden die Wizards kategorisiert in einer Baumstruktur dargestellt. Ändern Sie den Inhalt in `Variables`.

- **`Public Function`** IsZenOnWizard () **`As Boolean`**

  Anzeigen des Wizards in der Wizardauswahl. Liefert diese Methode False zurück, so wird der Wizard nicht angezeigt, z. B. weil dieser noch nicht fertig ist. Ändern Sie den den Returntypen auf `True`.

Diese Methoden lieferen die Informationen über den Wizard, die vom Leitsystem abgefragt werden. Zu beachten ist, dass ein Wizard nur in der Wizard Auswahl angezeigt wird, wenn die Methode **IsZenOnWizard** `True` zurückliefert.

Wechseln Sie in das **Initialize**-Ereignis der UserForm und ändern Sie die Inhalte des String-Arrays **m_strCaption**. Da unser Wizard nur aus zwei Schritten besteht, können Sie die anderen Zuweisungen löschen.

Fügen Sie folgende Definitionen im oberen Bereich des Code-Moduls ein:

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

Erstellen Sie eine Methode zum Initialisieren der Treiber-Combobox. Aufgabe der Routine ist es, alle projektierten Treiber des aktuellen Projekts in der ComboBox darzustellen.

```
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
```

Weiters benötigen wir eine Routine, die alle projektierten Variablentypen des Projekts in einer ComboBox darstellt.

```
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
If (obVarType.Name = INT) Then
   nSelect = nIndex
End If
End If
Next nIndex
cbVarType.ListIndex = nSelect
End If
```

Beim Aufschalten des Wizards werden die bereits projektierten Variablen durchlaufen, um nach einem freien Start-Offset für die neu zu erstellenden Variablen zu suchen. Dies geschieht über die folgende Methode.

```
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
```

Wechseln Sie ins Intialize-Ereignis der UserForm und erweitern Sie diese Methode um folgende Zeilen:

```
txtStart.Value = CStr(FindHighestOffsetVar + 1)
InitializeDriver
```

Die Zuweisung auf **txtStart** setzt den vorgeschlagenen Start-Offset für die zu erstellenden Variablen. Über die Methode **InitializeDriver** wird die ComboBox mit den projektierten Treibern gefüllt.

Erstellen Sie ein Change-Ereignis für die Treiber-Combobox und fügen Sie folgenden Code ein. Nach Auswahl eines Treibers werden die Variablentypen ermittelt. Das ausgewählte Treiber-Objekt wird in der Variable **m_obDriver** für die spätere Verwendung abgelegt.

```
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
```

Erstellen Sie eine Ereignisbehandlung **Change** für die VarType-Combobox und fügen Sie folgenden Code ein. Der ausgewählte Variablentyp wird in der Variable m_obVarType für die spätere Verwendung abgelegt.

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

Nun müssen wir nur mehr die Ereignisroutine zum Erstellen der Variablen mit den getroffenen Einstellungen erstellen. Dies geschieht über die Finish-Schaltfläche.

```
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
```

Beim Fertig stellen des Wizards wird überprüft, ob die getroffenen Einstellungen gültig sind. Ist dies nicht der Fall, so wird eine Fehlermeldung ausgegeben und der User aufgefordert, die Fehleingabe zu korrigieren.

Sind die getroffenen Einstellungen in Ordnung, so werden die Variablen erstellt. Die Variablen werden mit einem Namen und dem Index benannt. Existiert bereits eine gleichnamige Variable im Projekt, so wird der nächste freie Index ermittelt. In unserem Code-Beispiel wird immer eine Variable vom Kanaltyp SPS-Merker erstellt. Mit jedem Schleifendurchlauf wird der Offset für die Variable erhöht.

4.6.5 Aktualisierung von Wizards

Um die Wizards zu aktualisieren,:

1. Wählen Sie im Menü **Extras** den Eintrag **Editor Add-Ins aktualisieren**.

   Ein Dialog zum Update vorhandener Wizards wird geöffnet.

2. Wählen Sie die gewünschten Wizards aus.
3. Schalten Sie mit Klick auf **Aktualisierung starten** das Update

Ist ein Wizard oder eine Klasse bereits im Arbeitsbereich vorhanden, wird eine Warnungs anzeigt.

> **Achtung**
>
> *Bereits vorhandene Wizards werden beim Update überschrieben. Individuelle Änderungen am Wizard gehen damit verloren.*

4.7 Frequently asked Questions

In diesem Kapitel werden einige häufig gestellte Fragen beantwortet. Weiter Lösungen finden Sie auch online im COPA-DATA Anwender-Forum (http://www.copadata.com/forums/).

4.7.1 Warum bleibt der Button gedrückt?

Wenn ein Button z. B. mit einem LeftClickUp - Ereignis verknüpft ist, muss am Ende der Prozedur das LeftClickUp ausgeführt werden:

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

4.7.2 Makro wird nicht beim 1. Klick ausgeführt

Die Lösung entspricht der aus der Frage: Warum bleibt der Schaltbutton gedrückt (auf Seite 73):

Wenn ein Button z. B. mit einem LeftClickUp - Ereignis verknüpft ist, muss am Ende der Prozedur das LeftClickUp ausgeführt werden:

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

4.7.3 Makros funktionieren in der RT nicht mehr?

Dieser Effekt kann auftreten, wenn in der Runtime der VBA-Editor geöffnet ist und dann auf Stop/Start gedrückt wird um VBA zu stoppen/starten. Es werden in diesem Fall die Objekte (OnlineVariablen, Bildobjekte,...) ungültig, da diese den Bezug verlieren wenn eine erneute Initialisierung stattfindet.

4.7.4 Windows CE und VBA

VBA steht im CE-Editor für Wizards zur Verfügung. In der Runtime wird VBA nicht unterstützt. Details zum Editor finden Sie im Kapitel Projekte für CE erstellen.

4.8 Beispiele

Hier finden Sie einige Beispiele für VBA

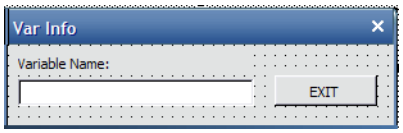
4.8.1 MouseEvents and ActiveX Control initialization

```
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 Variableninformationen anzeigen

Variablenname für geklicktes Element anzeigen:

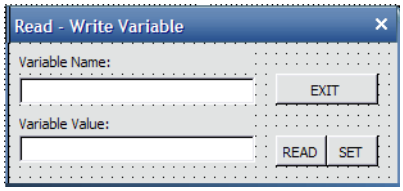



```
Option Explicit

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 Variablenwerte lesen und schreiben

Werte aus einer Variablen auslesen und zurückschreiben:

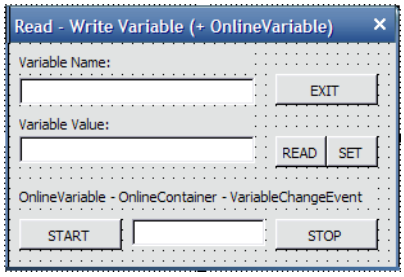



```
Option Explicit

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 Variablen lesen und schreiben und Online Variablen implementieren

Variableninformationen auslesen, Werte schreiben und Online Variablen implementieren:

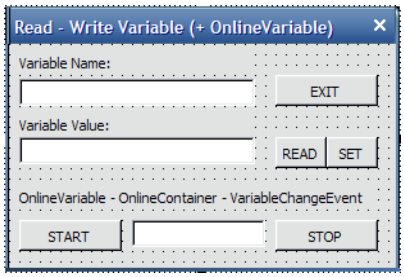



```
Option Explicit

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("OLV")
    '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 Dialog mehrfach nutzen

Userforms können mehrfach genutzt werden:




```
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
```

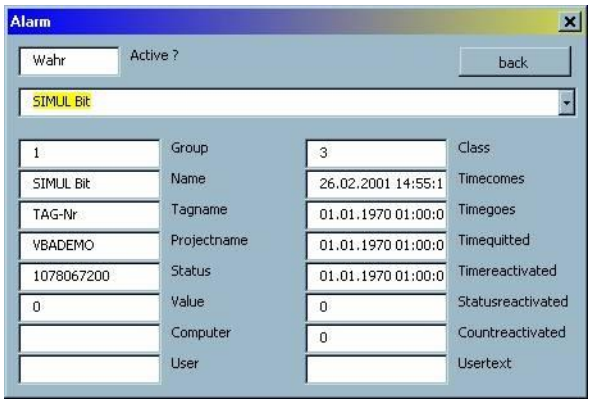
4.8.6 Alarm – Events and ActiveX Control handling

```
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
```

```
        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 Zugriff auf Alarme



BESCHREIBUNG:

Im Formular frmAlarm kann ein Alarm der sich im Speicher befindet in einer Combobox ausgewählt werden. Nach der getroffenen Auswahl werden alle Daten den Alarm betreffend in die darunter angeordneten Textboxen eingetragen (Gruppe, Klasse, Variable, ...).

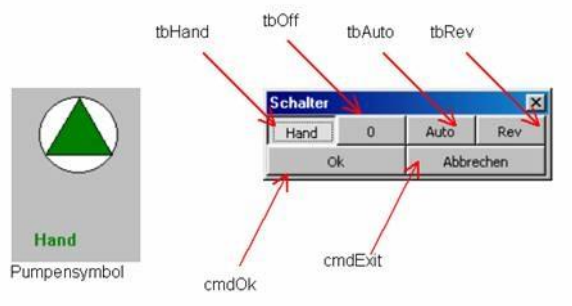
Um frmAlarm anzuzeigen, verwenden wir ein Ereignis unabhängiges Makro, da wir bei diesem Beispiel keinen Bezug zu einem Element benötigen.

```
Sub Alarm ()
frmAlarm.Show
End Sub
'Das Makro wird über eine Funktion Makro ausführen...ausgeführt.
'Bei Aufschalten der Form wird diese initialisiert und somit folgende Prozedur
ausgeführt.Diese Prozedur ist dafür zuständig, dass die in der Form angelegte Combobox alle
```

```
augenblicklich im Speicher befindlichen Alarme erhält.

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 Schalter setzen (Arbeiten mit Prozessvariablen)



In diesem Beispiel zeichnet man eine Pumpe, die aus einem Kreis und einem Dreieck besteht. Das Dreieck definieren Sie als eigenes Symbol. Darüber wird ein Multibin-Element gelegt und mit drei Bitmerker-Variablen verbunden.

Außerdem legt man fest, welche Farbe das Dreieck in der Mitte bei einer Wertänderung der Variablen bekommt.

Zunächst verbindet man das Multibin-Element mit einem Makro, das eine Form frmSchalter erscheinen lässt.

Über die Form frmSchalter kann man den Wert der drei Bitmerker-Variablen ändern.

Info

Nur eine der drei Variablen darf den Wert 1 enthalten. (d.h. wird eine Variable auf 1 gesetzt, müssen die zwei anderen auf 0 gesetzt werden)

Um dieses Makro mehrere Male in einem Projekt mit Bezug auf verschiedene Variablen zu verwenden, dürfen Sie mit dem Multibin-Element nur Bitmerker-Variablen verbinden, die bereits im Variablennamen enthalten welchen Zustand der Pumpe sie steuern.

z. B.:
Variable_Auto

Variable_Hand
Variable_Revi

Info

Die Endungen _Auto, _Hand und _Revi sind im Quellcode des Beispiels fix vorgegeben.

Durch diese 5-stellige Endungen der Variablennamen kann nun festgestellt werden, welche Variable beim Klick auf einen der ToggleButtons auf 1 gesetzt wird und welche auf 0.

Im Makro LeftClickUp_Schalter wird ein Unterprogramm FindVariable in der From frmSchalter aufgerufen, diesem wird das angeklickte Element obElem mit übergeben.

```
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
```

Modulglobale Variablen Deklaration:

```
Dim cmdLast As ToggleSchaltfläche
Dim strHand As String
Dim strAuto As String
Dim strRevi As String
Im Unterprogramm FindVariable werden zuerst alle Variablen, die das mitgegebene Element
anbietet, überprüft.
```

Je nach Endung (_Auto, _Hand oder _Revi) werden die Variablennamen den oben deklarierten String-Variablen zugewiesen.

Außerdem wird ermittelt, welchen Zustand die Variable hat, und je nach Wert (1 oder 0) wird auch der jeweilige Toggle-Button gedrückt oder nicht.

Beim öffnen der Form frmSchalter wird der Name des derzeit gedrückten ToggleButtons in eine String Variable geschrieben. Für den Fall dass sich der Benutzer entscheidet, seine Wertänderungen abzubrechen, werden die ursprünglichen Werte wieder hergestellt.

```
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
```

Die selbst erzeugte Function VarExists überprüft lediglich, ob die verbundenen Variablen auch existieren, falls dies nicht zutrifft, wird eine Fehlermeldung ausgegeben: 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
```

Wenn der Benutzer auf Abbrechen klickt, wird die Wertänderung zurück gesetzt und der ursprüngliche Zustand wieder hergestellt.

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

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

Wenn eine Toggleschaltfläche gedrückt ist, darf keine anderer Toggleschaltfläche gedrückt sein.

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

Im Click-Erreignis eines jeden ToggleButtons wird überprüft, ob dieser nicht gedrückt ist und ob die Variable existiert. Treffen beide Zustände zu, werden die Werte den verbunden Variablen übergeben.

5. VSTA

Mit **Visual Studio Tools for Applications (VSTA)** kann die Funktionalität der zenon Runtime und des Editors selbstständig um Funktionalität mittels .NET-Programmierung erweitert werden.

Info

Beachten Sie das in VSTA nur Assemblies (DLLs) bis maximal der .NET Framework Version 3.5 eingebunden werden können.

VSTA steht mit wenigen Einschränkungen auch am zenon Web Client zur Verfügung.

VSTA stellt für **Arbeitsbereich** und **Projekt** getrennte Entwicklungsumgebungen zur Verfügung. Es kann jeweils nur eine von beiden genutzt wird. Beim Start wird jede andere bereits geöffnete VSTA Entwicklungsumgebung geschlossen.

Um den VSTA-Editor für den Arbeitsbereich zu öffnen:

1. Drücken Sie die Tastenkombination `Alt+F10`.

   Der Code für den Arbeitsbereich und alle geladenen Projekte wird angezeigt.

Um den VSTA-Editor für das aktuell geladene Projekt zu öffnen:

1. Navigieren Sie zum Knoten **Programmierschnittstellen**.
2. Erweitern Sie die Ansicht dieses Knotens mit Klick auf `[+]`.
   Die Ansicht des Knotens wird erweitert.
3. Führen Sie einen Rechtsklick auf **VSTA** aus.

   Wählen Sie im Kontextmenü **VSTA-Editor mit ProjectAddin öffnen**.
   Der Editor wird für das aktuell geladene Projekt geöffnet.

   Info

   Wenn VBA-Makros im Editor geändert werden:

   - werden die Runtime-Dateien kompiliert und zur Runtime übertragen
   - wird die Runtime neu geladen
   - werden VSTA-Elemente ebenfalls neu geladen, selbst wenn keine Änderungen in VSTA erfolgt sind

5.1 Grundlagen

VSTA ist ein Toolset der Firma Microsoft, das auf der .NET-Technologie basiert. Es ist zum Verständnis notwendig, grundlegende Kenntnisse zur objektorientierten Programmierung, zu .NET und zu C#/Visual Basic.NET zu besitzen.

Achtung

Beachten Sie zur Runtime: Referenzierte Dlls können nicht ersetzt werden, wenn sie geladen sind. Damit eine referenzierte Datei aktualisiert werden kann, muss die Runtime beendet werden

5.1.1 Einrichten der VSTA-Umgebung

Standardmäßig ist die Unterstützung für VSTA in zenon bereits aktiviert. Beim Deaktivieren der VBA-Unterstützung ist auch die VSTA-Umgebung nicht mehr verfügbar.

Durch folgenden Eintrag in der **zenon6.ini** kann die VSTA-Umgebung manuell aktiviert oder deaktiviert werden:

VSTA aktivieren	VSTA deaktivieren
[VSTA] **ON**=1	**[VSTA]** **ON**=0

Die Unterstützung für VBA wird folgendermaßen aktiviert oder deaktiviert:

VBA aktivieren	VBA deaktivieren
[VBA] **EIN**=1	**[VBA]** **EIN**=0

Danach steht die Entwicklungsumgebung für VSTA in zenon zur Verfügung.

Info

VSTA ermöglicht die Entwicklung von Projekten in den Programmiersprachen C# und Visual Basic.NET. Als Standardsprache für VSTA-Projekte im Editor ist C# vorgesehen. Über folgenden Eintrag kann die Sprache auf Visual Basic.NET geändert werden:

[VSTA]

***CSHARP**=0*

5.1.2 Zugriff auf das Objektmodell in zenon

In VSTA kann auf das zenon-Objektmodell zugegriffen werden, welches auch in VBA verwendet wird. Das Objektmodell bietet hierbei in beiden Entwicklungsumgebungen die selbe Funktionalität.

Achtung

Wegen Einschränkungen in der Benennung von VSTA-Objekten wurden einige Änderungen am Objektmodell durchgeführt. Diese finden Sie in der untenstehenden Tabelle

Property alt	Property neu
IDriver.Name	IDriver.Identification
IDriver.Driver	IDriver.Name
Event alt	**Event neu**
IApplication.Close	IApplication.OnClose
IZenWorkspace.Startup	IZenWorkspace.OnStartup
IZenWorkspace.Exit	IZenWorkspace.OnExit

Der Zugriff geschieht in VSTA über das Objekt `this` und ersetzt somit das VBA-Objekt `MyWorkspace`. Die darauffolgenden Methoden und Objekte sind identisch. In der folgenden Methode wird eine Schablone mit dem Namen "TemplateName" in zenon erzeugt.

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

Info

Anders als in VBA wird in VSTA Wert auf Groß- und Kleinschreibung sowie auf Klammern nach Funktionsnamen gelegt.

Um die Methode in zenon aufzurufen, muss das Projekt mittels folgender Schritte gespeichert und kompiliert werden:

1. Klick auf *File -> Save MyWorkspace.cs* speichert das Projekt.
2. Klick auf *Build -> Build WorkspaceAddin* kompiliert das Projekt.

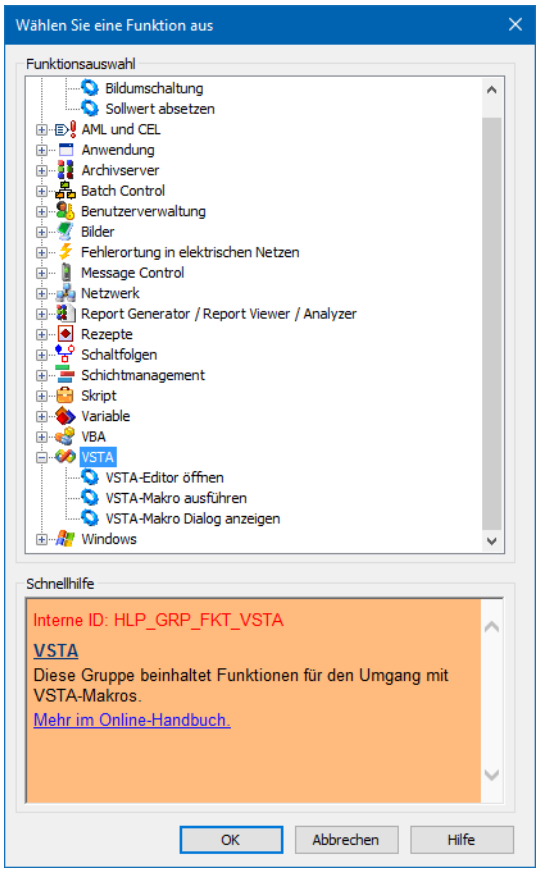
Danach steht die Methode als Makro in der VBA-Makro-Symbolleiste im zenon Editor zur Verfügung. Sollten im Makro-Zuordnungsdialog nicht alle im MyWorkspace enthaltenen Makros zur Auswahl

angeboten werden, sollte die Funktion „Liste der VBA-Makros neu laden“ in der Symbolleiste ausgeführt werden.

> **Info**
>
> *VSTA-Makros mit Parameter, z. B.* `Public void MacroWithParam(string mString)`*, werden nicht unterstützt und auch nicht in der Makro Symbolleiste angeboten.*

5.1.3 Funktionen in zenon

Für VSTA wurden in zenon Funktionen entwickelt. Diese befinden sich im Knoten **VSTA**.

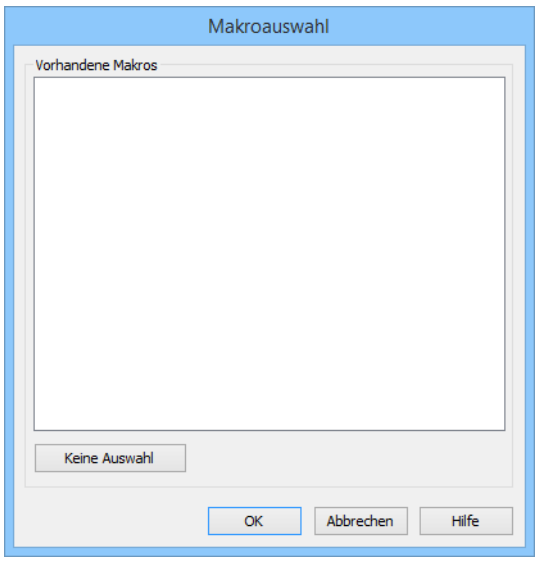


Parallel zu bestehenden VBA-Funktionen wurden ähnliche Funktionen für VSTA implementiert:

Funktionsname	Beschreibung
VSTA-Editor öffnen	öffnet den VSTA-Editor in der Runtime
VSTA-Makro ausführen	Im Editor kann ein VSTA-Makro ausgewählt werden, das beim Ausführen der Funktion in der Runtime gestartet wird. Hinweis: VSTA Makros mit Parameter, z. B. `Public void MacroWithParam(string mString)`,werden nicht unterstützt. Sie werden weder bei der Projektierung im Editor noch beim Start der Funktion zur Runtime angeboten.
VSTA-Makro Dialog anzeigen	In der Runtime wird ein Dialog angezeigt, in welchem die vorhandenen VSTA-Makros angezeigt, ausgewählt und ausgeführt werden können

VSTA-Makro ausführen

Dialog zur Konfiguration der Funktion **VSTA-Makro ausführen**.



Parameter	Beschreibung
Vorhandene Makros	
Liste der VSTA-Makros	Listet alle vorhanden VSTA-Makros auf. Auswahl eines Makros aus der Liste via Klick. **Hinweis:** Doppelklick führt das ausgewählte Makro aus.
Keine Auswahl	Verwirft die Auswahl und schließt den Dialog.
OK	Führt ausgewähltes Makro aus und schließt den Dialog.
Abbrechen	Verwirft die Auswahl und schließt den Dialog.
Hilfe	Öffnet die Online-Hilfe.

5.1.4 Debuggen eines VSTA Add-ins

Über den VSTA-Editor ist das Debuggen von selbst geschriebenen Add-ins möglich. Dabei gilt zu beachten, dass Projekt Add-ins nur in der zenon Runtime, Workspace Add-ins nur im zenon Editor debuggt werden können.

Eine Debug-Session wird über das Menü *Debug - Start Debugging* gestartet. Breakpoints können wie im VBA-Editor über einen Linksklick in die graue Breakpoint-Leiste am linken Rand neben der betreffenden Zeile gesetzt werden.

Info

Beachten Sie beim Debuggen von Runtime Add-ins:

Die in zenon geänderten Runtime-Dateien müssen vor dem Start des Debuggers erneut erzeugt werden.

5.1.5 Events in VSTA

Da beim Kompilieren von geändertem Code, dem Starten einer Debug-Session oder dem Beenden einer Debug-Session eine Terminierung eines Add-ins geschieht, wurden in VSTA entsprechende Events implementiert. Diese ermöglichen beispielsweise das Auswerten und Freigeben einer Objekt-Referenz und das Speichern von vorhandenen Daten.

Für jede Terminierung existieren zwei Events. Das erste Event wird kurz vor der Terminierung gestartet, das zweite nach dem Start einer neuen Add-in-Session.

Event	Beschreibung
`OnPreVSTADebugStart`	Wird getriggert kurz bevor eine Debug-Session gestartet wird. Beim Start wird ein aktives Add-in entladen, Referenzen müssen freigegeben und vorhandene Daten gegebenenfalls gespeichert werden.
`OnVSTADebugStart`	Wird getriggert nachdem eine Debug-Session gestartet wurde.
`OnPreVSTADebugStop`	Wird getriggert kurz bevor eine Debug-Session gestoppt wird. Beim Stoppen einer Debug-Session wird das Add-in entladen, Referenzen müssen freigegeben werden und vorhandene Daten gegebenenfalls gespeichert werden.
`OnVSTADebugStopped`	Wird getriggert nachdem eine Debug-Session gestoppt wurde.
`OnPreVSTAUpdate`	Wird vorm Entladen des Add-ins getriggert, wenn eine neue Version des Add-ins erfolgreich erzeugt wurde.
`OnPostVSTAUpdate`	Wird beim Laden einer neuen Version des Add-ins getriggert.

5.1.6 Erstellen eines Backups von VSTA-Projekten

VSTA-Projekte in der Runtime werden automatisch beim Erstellen der Runtime-Dateien gezippt und sind in Sicherungen des Arbeitsbereiches enthalten.

VSTA-Projekte im Editor müssen jedoch manuell gesichert werden.
Die VSTA-Editorprojekte finden Sie Im Ordner
C:\ProgramData\COPA-DATA*version*\VSTAWorkspace\.

5.2 Erstellung eines VSTA-Projektes

Ähnlich zu VBA existiert in VSTA die Möglichkeit, sowohl Projekte (Erweiterungen) für den Editor als auch für die Runtime zu erstellen. Prinzipiell sind Projekte im Editor in der Programmiersprache C# realisiert. Für die Runtime und Editor stehen sowohl C# als auch Visual Basic.NET zur Verfügung.

Info

Beachten Sie das in VSTA nur Assemblies (DLLs) bis maximal der .NET Framework Version 3.5 eingebunden werden können.

Info

Innerhalb des VSTA-Editors kann immer nur ein Projekt gleichzeitig angezeigt werden. Zusätzlich kann immer nur eine Instanz des VSTA-Editors aktiv sein. Beim Start des VSTA-Editors wird eine eventuell bereits vorhandene Instanz geschlossen.

5.2.1 VSTA-Projekte im Editor

Beim Erstellen eines Projekts für den zenon-Editor wird ein VSTA Add-in für den Workspace geladen. Um das Add-in zu bearbeiten, muss der VSTA-Editor über *Datei - VSTA-Editor öffnen...* geöffnet werden. Die Benutzeroberfläche des VSTA-Editors ist identisch mit der Visual Studio-Entwicklungsumgebung der Firma Microsoft.

Info

Über das Menü Help / Contents kann die Hilfe des VSTA-Editors aufgerufen werden. Diese Hilfe gibt einen Überblick über die Funktionen des Editors, die Features des .NET Frameworks und der Programmierung in Visual Basic.NET und C#.

Das VSTA-Add-in besteht grundsätzlich aus der Klasse **`MyWorkspace`**. Diese Klasse kann nun um eigene Methoden erweitert werden. Die Klasse beherbergt per Default welche folgende zwei Methoden:

Funktion	Beschreibung
`MyWorkspace_Startup`	wird automatisch beim Start von zenon, nach dem Erstellen eines Builds und beim Start einer Debug-Session ausgeführt.
`MyWorkspace_Shutdown`	wird automatisch beim Beenden von zenon, vor dem Erstellen eines neuen Builds und beim Beenden einer Debug-Session ausgeführt.

Achtung

*Die Methodennamen dürfen ausschließlich mit **`Macro`** (beispielsweise Macro1, MacroVSTA) beginnen, keine Parameter beinhalten und müssen als **`Public`** definiert sein. Zudem dürfen der Klassenname und andere von VSTA erstellte Methoden und Events nicht verändert werden.*

Um die Methode in zenon aufzurufen, muss das Projekt mittels folgender Schritte gespeichert und kompiliert werden:

1. Klick auf *File -> Save MyWorkspace.cs* speichert das Projekt.

2. Klick auf *Build -> Build WorkspaceAddin* kompiliert das Projekt.

Danach steht die Methode als Makro in der VBA-Makro-Symbolleiste im zenon Editor zur Verfügung. Sollten im Makro-Zuordnungsdialog nicht alle im MyWorkspace enthaltenen Makros zur Auswahl angeboten werden, sollte die Funktion „Liste der VBA-Makros neu laden“ in der Symbolleiste ausgeführt werden.

5.2.2 VSTA-Projekte in der Runtime

Um ein VSTA-Projekt für die Runtime zu erstellen, muss die VSTA-Umgebung aufgerufen werden. Dazu gehen Sie wie folgt vor:

1. Öffnen Sie im Projektmanager den Knoten **Programmierschnittstellen**.
2. Öffnen Sie das Kontextmenü von **VSTA**.
3. Klicken Sie auf **VSTA-Editor öffnen...**

   Hinweis: Beim ersten Öffnen wird ein Auswahldialog angezeigt. Darin wählen Sie die Programmiersprache.

   Ein Projekt mit der gewünschten Programmiersprache wird angelegt.

> **Info**
>
> *Die Programmiersprache kann später nicht mehr geändert werden. Dieser Dialog wird daher nur beim ersten Öffnen angezeigt.*

In diesem Projekt wird von zenon eine Klasse namens `ThisProject` angelegt, welche folgende zwei Methoden beherbergt:

Funktion	Beschreibung
`ThisProject_Startup`	wird automatisch beim Start der Runtime ausgeführt
`ThisProject_Shutdown`	wird automatisch beim Beenden der Runtime ausgeführt

Die Klasse kann nun um eigene Methoden erweitert werden.

> **Achtung**
>
> *Die Methodennamen dürfen ausschließlich mit `Macro` (beispielsweise Macro1, MacroVSTA) beginnen, keine Parameter beinhalten und müssen als `Public` definiert sein. Zudem dürfen der Klassenname und andere von VSTA erstellte Methoden und Events nicht verändert werden.*

Es besteht über das zenon-Objektmodell Zugriff auf alle Runtime-Funktionalitäten. Editorspezifische Funktionen können wie in VBA nicht verwendet werden.

Die zenon-Runtime wird beim Starten des Debuggers automatisch gestartet. Weitere Informationen sind im Kapitel Debuggen eines VSTA-Add-ins (auf Seite 94) zu finden.

Info

Über das Menü Help / Contents kann die Hilfe des VSTA-Editors aufgerufen werden. Diese Hilfe gibt einen Überblick über die Funktionen des Editors, die Features des .NET Frameworks und der Programmierung in Visual Basic.NET und C#.

5.2.3 Entwicklung von Wizards in VSTA

Die VSTA-Umgebung bietet wie VBA (auf Seite 64) die Möglichkeit, eigene Wizards zu entwickeln.

Um in einem Formular auf das zenon-Objektmodell zugreifen zu können, muss eine Referenz darauf an das Formular übergeben werden. Dazu wird eine Methode in der Klasse `MyWorkspace` erstellt. Im folgenden Beispiel wird ein Formular mit dem Namen `Wizard` instanziert und die Methode `ZenonInstance` mit einer Referenz auf das zenon-Objektmodell als Parameter aufgerufen. Über den Aufruf `ShowDialog()` wird das Wizard-Formular angezeigt.

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

Im Code des Formulars muss eine Member-Variable angelegt werden, die das zenon-Objektmodell übernimmt.

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

Zuletzt wird die Methode `ZenonInstance` erstellt. Diese Methode nimmt die Referenz auf das Objektmodell entgegen und setzt es auf das Objekt `m_Zenon`.

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

Nun können eigene Klassen und Methoden im Formular entwickelt werden, die auf das Objektmodell zugreifen. Alle Methoden, Objekte und Attribute stehen über das Objekt `m_Zenon` zur Verfügung.

5.3 Beispiele

Hier finden Sie einige Beispiele für die Verwendung von VSTA sowohl in der Runtime als auch im Editor.

5.3.1 Erstellen von Variablen im zenon-Editor

In diesem Beispiel wird eine Textdatei geöffnet und der Inhalt dazu benutzt, um im zenon-Editor Variablen anzulegen. Die Textdatei beinhaltet eine beliebige Anzahl an Zeilen. Jede Zeile beinhaltet dabei den Namen und den Datentyp einer Variable, diese sind durch ein Komma getrennt (Beispiel: Variable1,BOOL).

Die Methode `Macro1` sucht zuerst nach dem Intern Treiber im zenon-Editor. Danach wird dem Benutzer ein Dateiauswahl-Dialog angezeigt, in welchem er die Textdatei auswählen muss. Die Methode liest daraufhin die Textdatei ein und erstellt die Variablen. Die Methode `GetDataType` wird dazu benötigt, um bei der Erstellung der Variablen den zugehörigen Datentyp zu ermitteln und zuweisen zu können.

C#-Code

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

namespace WorkspaceAddin
{
    [System.AddIn.AddIn("MyWorkspace", Version = "1.0", Publisher = "", Descripti
on = "")]
    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 intern driver
        public void MacroCreateVariablesFromFile()
        {

            //create objects that will take the intern driver and the variable ty
pe
            IDriver zenonInternDriver = null;
            //search for the Intern driver and throw exception if no driver was f
ound
            try
            {
                for (int driverCount = 0; driverCount < this.ActiveDocument.Drive
rs().Count; driverCount++)
                {
                    if (this.ActiveDocument.Drivers().Item(driverCount).Name == "
Intern")
                    {
                        zenonInternDriver = this.ActiveDocument.Drivers().Item(dr
```

```
iverCount);
                        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.Fi
leName, 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], zen
onInternDriver, tpKanaltypes.tpSystemVariable, GetDataType(varLine[1]));
                    }
                    importReader.Close();
                }
            }
            catch (Exception fileEx)
            {
                MessageBox.Show("An error occured while opening the file: " + fil
```

```
eEx.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 Schreiben von Projektinformationen in das zenon-Ausgabefenster

In diesem Beispiel wird veranschaulicht, wie über VSTA auf das Ausgabefenster des zenon-Editors zugegriffen werden kann. Die Methode namens `Macro1` liest dazu die im Projekt angelegten Prozessbilder aus, ermittelt die jeweils dazugehörige Schablone und ermittelt alle verfügbaren Treiber sowie deren Bezeichnung.

C#-Code

```
using System;

namespace WorkspaceAddin
{
    [System.AddIn.AddIn("MyWorkspace", Version = "1.0", Publisher = "", Descripti
on = "")]
    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 outpu
t 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_DynProper
ties("Template").ToString();
                this.Application.DebugPrint(" Picture '" + picName + "' uses Temp
late '" + 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 Auslesen von Variablen in zenon über Regular Expressions

Im folgenden Beispiel werden zenon-Variablen in einem Runtime-Projekt ausgelesen und in eine lokale Textdatei gespeichert.

Mittels Regular Expressions werden nur Variablen gelesen, deren Name mit 3 Zahlen und einem darauffolgenden Unterstrich beginnen (beispielsweise "001_var" oder "234_xyz"). Der Benutzer wird danach aufgefordert, einen Ordner auszuwählen. In diesem Ordner wird eine Textdatei mit einem zeitabhängigen Dateinamen angelegt. In diese Datei werden Name, Kennung und aktueller Wert aller zutreffenden Variablen mit einem Strichpunkt getrennt gespeichert.

Info

*Möglicherweise müssen manuell Referenzen hinzugefügt werden, um das Beispiel in der zenon-Runtime auszuführen. Dazu öffnen Sie im Project Explorer das Kontextmenü und klicken auf **Add Reference...**.*
Die in diesem Beispiel benötigten Referenzen sind:

- **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 = "", Descripti
on = "")]
    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 fil
```

```
e date format
                    //then create a txt file with the name "zenonVar" and the cor
responding date
                    DateTime currentTime = DateTime.Now;
                    selectedPath = folderSelect.SelectedPath + "\\zenonVar" + cur
rentTime.ToFileTime() + ".txt";
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show("An error occured: " + 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)

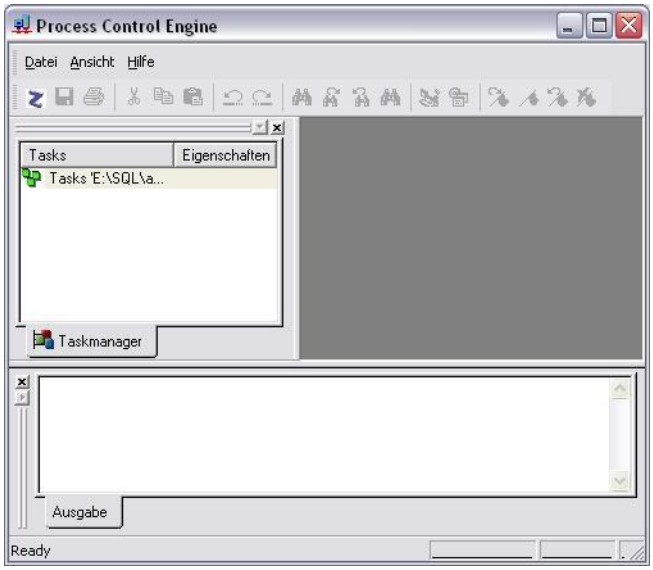
Achtung

PCE wird ab Version 7.20 nicht mehr unterstützt und nicht mehr im Modulbaum von zenon angezeigt. Beim Konvertieren von Projekten aus Versionen vor 7.20, die PCE-Tasks enthalten, wird der Knoten PCE für diese Projekte wieder eingeblendet. PCE wird nicht weiterentwickelt und dokumentiert.

*Empfehlung: Verwenden Sie **zenon Logic** anstelle von PCE.*

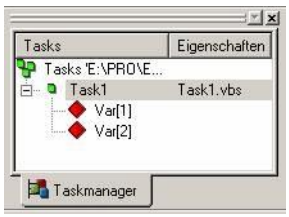
6.1 Der PCE Editor

Die PCE befindet sich im Projektmanager unter dem Eintrag **Programmierschnittstellen**. Der PCE Editor wird über das Kontextmenü **PCE Editor öffnen** geöffnet**.**

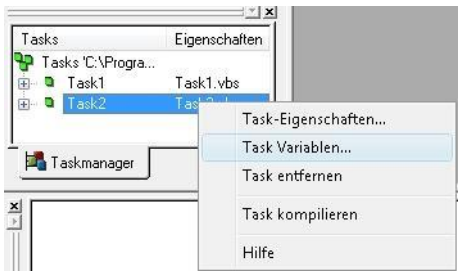


6.1.1 Der Taskmanager

Der Taskmanager des PCE Editors zeigt die angelegten Tasks und die damit verbundenen Variablen an.



Ein Doppelklick auf einen Task öffnet diesen im Editierbereich. Mit der rechten Maustaste lässt sich das Kontextmenü eines Tasks öffnen.



Das Kontextmenü eines Tasks bietet vier Einträge:

Parameter	Beschreibung
Task-Eigenschaften...	Öffnet den Eigenschaften-Dialog des Tasks.
Task Variablen...	Öffnet den Variablenauswahl-Dialog. So können weitere Variablen zum Task hinzugefügt werden.
Task entfernen	Löscht den Task ohne weitere Abfrage.
Task kompilieren	Kompiliert den Task.

6.1.2 Der Editierbereich

Im Editierbereich des PCE Editors wird der Code für die Tasks in VB Script oder Javascript eingegeben.

6.1.3 Das Ausgabefenster

6.1.4 Die Menüs des PCE Editors

Menü Datei

Das Menü **Datei** beinhaltet folgende Befehle:

Parameter	Beschreibung
Speichern	Speichert neue oder geänderte Tasks.
Drucken	Druckt den aktiven Task.
Schließen	Schließt den PCE Editor.

Menü Bearbeiten

Das Menü **Bearbeiten** beinhaltet folgende Befehle:

Parameter	Beschreibung
Rückgängig	Macht die zuletzt ausgeführte Aktion rückgängig.
Wiederholen	Wiederholt die zuletzt ausgeführte Aktion.
Ausschneiden	Verschiebt einen Text in die Windows-Zwischenablage.
Kopieren	Kopiert einen Text in die Windows-Zwischenablage.
Einfügen	Fügt einen Text aus der Windows-Zwischenablage ein.
Löschen	
Alles auswählen	Markiert den gesamten Text im Task.
Suchen	Sucht einen Text im geöffneten Task.
Nächsten suchen	Geht zur nächsten Fundstelle im Task.
Vorherigen suchen	Geht zur vorherigen Fundstelle im Task.
Ersetzen...	Ersetzt einen Text im Task durch einen anderen.
Bookmarks	Verwaltung von Lesezeichen im Code des Tasks.
- setze Bookmark	Setzt im Code an der markierten Stelle ein Lesezeichen.
- nächste Bookmark	Springt im Code zum nächsten Lesezeichen.
- vorherige Bookmark	Springt im Code zum vorherigen Lesezeichen.
- lösche alle Bookmarks	Löscht alle Lesezeichen aus dem Code.

Menü Ausführen

Das Menü **Ausführen** beinhaltet folgende Befehle:

Alle Tasks speichern u. neustarten	
Task kompilieren	Kompiliert den Task.

Menü Ansicht

Das Menü **Ansicht** beinhaltet folgende Befehle:

Parameter	Beschreibung
Einstellungen	Öffnet den Einstellungen-Dialog des PCE Editors.
Taskmanager	Öffnet/schließt das Taskmanagerfenster.
Ausgabefenster	Öffnet/schließt das Ausgabefenster.
Status Bar	Öffnet/schließt die Statusleiste.

Menü Fenster

Das Menü **Fenster** beinhaltet folgende Befehle:

Schließen
Anordnen
Teilen
Symbole anordnen
Liste der geöffneten Fenster

Menü Hilfe

Das Menü **Hilfe** beinhaltet folgende Befehle:

Befehl	Aktion
Hilfe	Öffnet die Online-Hilfe.
Info über...	Öffnet ein Fenster mit Informationen über zenon: ▸ Seriennummer ▸ Aktivierungsnummer ▸ lizenzierte TAGs/IOs ▸ lizenzierte Module Im Info-Fenster kann über einen Schieberegler navigiert werden. Klick in das Fenster oder Druck auf die Taste **Esc** schließt das Info-Fenster.

6.1.5 Die Symbolleiste des PCE Editors

Die wichtigsten Befehle des PCE Editors lassen sich auch über die Icons in der Symbolleiste ausführen.

Folgende Icons stehen zur Verfügung:

Parameter	Beschreibung
Schließen	Schließt den PCE Editor.
Alles speichern	Speichert neue oder geänderte Tasks.
Aktives Bild drucken	Druckt den aktiven Task.
Ausschneiden	Verschiebt einen Text in die Windows-Zwischenablage.
Kopieren	Kopiert einen Text in die Windows-Zwischenablage.
Einfügen	Fügt einen Text aus der Windows-Zwischenablage ein.
Rückgängig	Macht die zuletzt ausgeführte Aktion rückgängig.
Wiederholen	Wiederholt die zuzletzt ausgeführte Aktion.
Suchen	Sucht einen Text im geöffneten Task.
Nächsten suchen	Geht zur nächsten Fundstelle im Task.
Vorherigen suchen	Geht zur vorherigen Fundstelle im Task.
Ersetzen...	Ersetzt einen Text im Task durch einen anderen.
Alle Tasks speichern und neustarten	
Debugger starten	
Nächste Bookmark	Springt im Code zum nächsten Lesezeichen.
Setze Bookmark	Setzt im Code an der markierten Stelle ein Lesezeichen.
Vorherige Bookmark	Springt im Code zum vorherigen Lesezeichen.
Lösche alle Bookmarks	Löscht alle Lesezeichen aus dem Code.

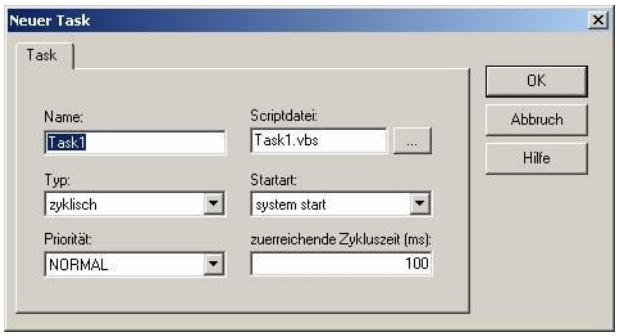
6.2 Vorgangsweise

6.2.1 Erstellen eines Tasks

Über das Kontextmenü des Taskmanagers kann ein neuer Task erstellt werden.

Eigenschaften des Tasks

Nach dem Anlegen des Tasks öffnet sich automatisch der Eigenschaften-Dialog des Tasks.



Folgende Eigenschaften können eingestellt werden:

Parameter	Beschreibung
Name	Eindeutige Bezeichnung des Tasks.
Typ	Tasks können **zyklisch** oder **einmalig** ausgeführt werden. `Zyklisch` : der Task wird zyklisch in dem unter **Zuerreichende Zykluszeit** angegebenen Intervall ausgeführt. `Einmalig` : der Task wird ein einziges Mal ausgeführt.
Priorität	Prozesspriorität für das Betriebssystem-Multithreading (**Leerlauf** , **Niedrig** , **Normal** , **Hoch** , **Höchste** , **Zeitkritisch**). Default: Normal `Hauptprozess` : der Task läuft im gleichen Thread wie die Runtime. Wenn der Task in eine Warteschleife gerät oder gar abstürzt, beeinflusst das auch die Runtime.
Scriptdatei	Auswahl der Scriptdatei: VB-Dateien (*.vbs) für VB Script oder JS-Dateien (*.js) für Javascript. Beim erstmaligen Öffnen des Tasks im Editierbereich wird die entsprechende Datei angelegt.
Startart	`System start` : der Task wird automatisch mit der Runtime gestartet. (Dies ist die einzige Möglichkeit, wie die PCE unter Windows CE verwendet werden kann, da Windows CE kein VBA unterstützt.) `Ereignisgesteuert` : der Task wird über ein VBA-Makro und die Anweisung thisProject.Tasks.Item(Taskname).Run gestartet.
Zu erreichende Zykluszeit	Bei zyklischen Tasks die Zykluszeit in Millisekunden, die erreicht werden soll. Wird diese Zykluszeit nicht erreicht, so wird der Task so schnell wie möglich ausgeführt.

Zum nachträglichen Ändern der Eigenschaften lässt sich der Dialog auch über das Kontextmenü des Tasks und den Befehl **Task-Eigenschaften...** öffnen.

Variablen des Tasks

Nach der Eingabe der Eigenschaften öffnet sich automatisch der Variablenauswahl-Dialog. Hier werden die Variablen ausgewählt, die im Task verarbeitet werden sollen.

Es sollten hier alle Variablen verknüpft werden, die im Task gelesen oder geschrieben werden sollen. Es besteht zwar auch die Möglichkeit über das Variablen-Objekt auf Variablen zuzugreifen, aber nur die direkt mit dem Task verknüpften Variablen werden automatisch beim Initialisieren des Tasks vor der Ausführung eingelesen.

Die Variablen müssen die folgende Syntax haben:

```
Task.Value("Variablenname")=123
```

Zum nachträglichen Ändern der Variablenauswahl lässt sich der Dialog auch über das Kontextmenü des Tasks und den Befehl **Task Variablen...** öffnen.

6.2.2 Eingabe des Codes

Durch einen Doppelklick auf einen Task im Taskmanager wird der Task im Editierbereich geöffnet. Wird der Task das erste Mal geöffnet, so wird nun die entsprechende VBS- oder JS-Datei angelegt.

Vier Prozeduren werden automatisch angelegt:

Parameter	Beschreibung
Task_Init()	Diese Prozedur wird automatisch beim Starten des Tasks ausgeführt.
Task_Main()	Diese Prozedur wird entweder einmal (Typ einmalig) oder zyklisch (Typ zyklisch) ausgeführt.
Task_Exit()	Diese Prozedur wird automatisch vor dem Beenden des Tasks ausgeführt.
Task_Timer(lTimerId)	Diese Prozedur wird, solange der entsprechende Timer läuft, zyklisch ausgeführt. Die Zykluszeit wird mit dem Starten des entsprechenden Timers als Parameter übergeben.

Die PCE verwendet grundsätzlich dasselbe Objekt-Modell wie VBA (siehe VBA Tutorials). Bei der Verwendung von VBA Objekten (ausser dem Objekt Task) geht allerdings die Multithreading-Fähigkeit verloren, da auf diese Objekte nur aus dem Hauptthread heraus zugegriffen werden kann.

Achtung

Nicht alle Funktionen des COM Interfaces sind Multithreadingfähig und dürfen daher nur im Hauptthread Kontext verwendet werden. Diese bedeudet, wenn im PCE Task eine andere Eigenschaft als Im Hauptprozess eingestellt ist, dann darf aus der PCE kein Zugriff auf den Hauptthread erfolgen. Erfolgt trotzdem ein Zugriff auf das COM Interface, kann es zu undefinierten Systemzuständen wie einem einfrieren der Runtime kommen.

Von besonderer Wichtigkeit ist allerdings die Collection Tasks und das Objekt **Task**.

Die Collection Tasks

Count
Item
Parent

Das Objekt Task

ActualCycleTime	Eigenschaft	Tatsächlich erreichte Zykluszeit des Tasks
CountVariable	Eigenschaft	Anzahl der mit dem Task verknüpften Variablen
CycleTime	Eigenschaft	Eingestellte Zykluszeit des Tasks
DynProperties	Eigenschaft	
ErrorNumber	Eigenschaft	
ErrorString	Eigenschaft	
Exit	Ereignis	
Init	Ereignis	
ItemVariable	Methode	
Main	Ereignis	
MemValue	Eigenschaft	Mit Task.MemValue(Name)=Wert wird eine interne Variable angelegt und ein Wert wird ihr zugewiesen. Die Variable muss vorher nicht deklariert werden. Diese Variable kann auch von anderen Tasks aus gelesen werden. Somit erlaubt sie die Übergabe eines Wertes von einem Task an einen anderen.
Name	Eigenschaft	Name des aktuellen Tasks
Parent	Eigenschaft	Die Collection **Tasks**
Priority	Eigenschaft	Priorität des aktuellen Tasks
Run	Methode	Startet einen Task
Sleep	Methode	Hält einen Task an
StartTimer	Methode	Die Methode StartTimer startet einen Timer des Tasks.
Status	Eigenschaft	
Stop	Methode	Stoppt einen Task
StopTimer	Methode	Die Methode StopTimer stoppt einen Timer des Tasks.
Timer	Ereignis	
Type	Eigenschaft	
Value	Eigenschaft	Mit Task.Value(Name einer verknüpften Variable)=Wert kann aus einem Task einer Variable des zugehörigen Projekts ein Wert zugewiesen werden.

6.2.3 Funktion PCE anzeigen

Über die zenon Funktion **PCE anzeigen** kann der PCE Editor aus der Runtime geöffnet werden.

6.2.4 Ausführen von Tasks

Tasks können mit dem Systemstart ausgeführt werden oder ereignisgesteuert.

Tasks mit Systemstart ausführen

Wenn in der Konfiguration des Tasks die **Startart** auf `System start` gesetzt wurde, wird der Task automatisch mit dem Starten der Runtime gestartet.

Dies ist die einzige Möglichkeit, wie die PCE unter Windows CE verwendet werden kann, da Windows CE kein VBA unterstützt.

Tasks ereignisgesteuert ausführen

Am PC

Ein Task kann auch ereignisgesteuert gestartet werden. In diesem Fall muss die **Startart** auf `ereignisgesteuert` gesetzt werden. Nun wird der Task nicht mehr automatisch mit der Runtime gestartet.

Um den Task durch das Drücken eines Buttons, durch eine Grenzwertverletzung oder durch ein anderes Ereignis starten zu können muss ein VBA-Makro erstellt werden. Die VBA Anweisung zum Starten des Tasks lautet: **`thisProject.Tasks.Item(Taskname).Run`**
Der Task wird automatisch in einem eigenen Thread gestartet, wenn in der Konfiguration nicht die **Priorität Hauptprozess** angegeben wurde.

Mit der folgenden VBA Anweisung kann der Task jederzeit wieder beendet werden:
`thisProject.Tasks.Item(Taskname).Stop`

Hinweis: Um einen Task mehrmals auszuführen, muss er nach dem Beenden auch explizit gestoppt werden, damit er neu gestartet werden kann.
Anweisung: **`thisProject.Tasks.Item(Taskname).Stop`**
Nach korrektem Stop kann er neu gestartet werden.

Am CE Terminal

Da Windows CE kein VBA unterstützt, ist die Vorgehensweise für die Ausführung auf einem PC auf einem CE Terminal nicht möglich. Es gibt jedoch eine Möglichkeit, auch hier Tasks ereignisgesteuert aufzurufen.

Dazu wird ein Task mit der **Startart** `System start` erstellt. Dieser Task wird also automatisch mit der Runtime gestartet. Zudem erhält dieser Task die **Priorität** `Hauptprozess`, sodass er im selben Thread wie die Runtime läuft. Mit diesem Task werden nun Bit-Variablen verknüpft, die das ereignisgesteuerte Ausführen der anderen Tasks übernehmen. Die Anweisung zum Starten eines anderen Tasks lautet:
`Parent.Item(Taskname).Run`
Dieser Task wird nun automatisch in einem eigenen Thread gestartet, wenn in der Konfiguration nicht die **Priorität Hauptprozess** angegeben wurde.

Mit der folgenden Anweisung kann der Task jederzeit wieder beendet werden:
`Parent.Item(Taskname).Stop`

6.3 VBScript - Einführung

6.3.1 Datentypen

Variant

VBScript kennt nur einen Datentyp namens **Variant**. **Variant** ist eine besondere Art von Datentyp, der je nach Verwendung verschiedene Arten von Informationen aufnehmen kann. Weil **Variant** der einzige Datentyp in VBScript ist, geben alle Funktionen in VBScript diesen Datentyp zurück.

Im einfachsten Fall kann **Variant** numerische Werte oder Zeichenfolgen enthalten. Ein **Variant** verhält sich wie eine Zahl, wenn Sie ihn in einem numerischen Zusammenhang verwenden, und wie eine Zeichenfolge, wenn Sie ihn wie eine Zeichenfolge verwenden. Wenn Sie also mit Daten arbeiten, die wie Zahlen aussehen, interpretiert VBScript diese als Zahlen und verfährt entsprechend. Natürlich können Sie immer dafür sorgen, dass sich Zahlen wie Zeichenfolgen verhalten, indem Sie diese in Anführungszeichen ( ) einschließen. Wenn Sie mit Daten arbeiten, die nur als Zeichenfolgen interpretiert werden können, behandelt VBScript diese als Zeichenfolgen.

Untertypen des Datentyps Variant

Über die einfache Einteilung in numerische Werte und Zeichenfolgen hinaus kann bei **Variant**-Werten zwischen weiteren Arten von numerischen Informationen unterschieden werden. Numerische Informationen können beispielsweise ein Datum oder eine Uhrzeit darstellen. In Verbindung mit anderen Datums- oder Uhrzeitinformationen wird das Ergebnis immer als Datum oder Uhrzeit

ausgedrückt. Natürlich stehen auch viele verschiedene Arten numerischer Informationen zur Verfügung, angefangen von booleschen Werten bis hin zu großen Fließkommazahlen. Diese unterschiedlichen Informationskategorien, die ein **Variant** aufnehmen kann, werden als **Untertypen** bezeichnet. Meistens genügt es, einem **Variant** einfach die gewünschten Daten zuzuweisen, damit er sich entsprechend verhält.

Die folgende Aufstellung zeigt die verschiedenen Untertypen von Daten, die ein **Variant** enthalten kann.

Untertyp	**Bedeutung**
Empty	Variant ist nicht initialisiert. Der Wert ist 0 bei numerischen Variablen und eine Null-Zeichenfolge () bei Zeichenfolgenvariablen.
Null	Variant enthält absichtlich keine gültigen Daten.
Boolean	Enthält entweder TRUE oder FALSE.
Byte	Enthält eine ganze Zahl von 0 bis 255.
Integer	Enthält eine ganze Zahl von -32.768 bis 32.767.
Currency	-922,337,203,685,477.5808 bis 922,337,203,685,477.5807.
Long	Enthält eine ganze Zahl von -2.147.483.648 bis 2.147.483.647.
Single	Enthält eine Fließkommazahl einfacher Genauigkeit von -3,402823E38 bis -1,401298E-45 für negative Werte und von 1,401298E-45 bis 3,402823E38 für positive Werte.
Double	Enthält eine Fließkommazahl doppelter Genauigkeit von -1,79769313486232E308 bis -4,94065645841247E-324 für negative Werte und von 4,94065645841247E-324 bis 1.79769313486232E308 für positive Werte.
Date (Time)	Enthält eine Zahl, die ein Datum vom 1. Januar 100 bis zum 31. Dezember 9999 darstellt.
String	Enthält eine Zeichenfolge variabler Länge, die bis zu etwa 2 Milliarden Zeichen lang sein kann.
Object	Enthält ein Objekt.
Error	Enthält eine Fehlernummer.

6.3.2 Variablen

Eine Variable ist ein praktischer Platzhalter, der auf eine Stelle im Speicher des Computers verweist, an der Sie Programmdaten speichern können, die sich während der Ausführung des Skripts ändern kann. Sie können beispielsweise eine Variable namens KlickZaehler einrichten, um zu speichern, wie oft ein Benutzer auf ein Objekt in einer bestimmten Web-Seite klickt. Es ist unwichtig, wo sich die Variable im Computerspeicher befindet. Wichtig ist, dass Sie nur ihren Namen angeben müssen, um den Wert der Variablen abzurufen oder zu ändern. In VBScript haben Variablen immer den grundlegenden Datentyp Variant.

Deklarieren von Variablen

Sie deklarieren Variablen in Ihrem Skript explizit mit den Anweisungen Dim, Public und Private. Beispiel:

```
Dim GradFahrenheit
```

Sie deklarieren mehrere Variablen, indem Sie diese jeweils durch Kommas voneinander trennen. Beispiel:

```
Dim Oben, Unten, Links, Rechts
```

Einschränkungen für Namen

Für Variablennamen gelten die Standardregeln zum Benennen von Sprachelementen in VBScript. Für Variablennamen gilt:

- Sie müssen mit einem Zeichen aus dem Alphabet beginnen.
- Sie dürfen keinen Punkt enthalten.
- Sie dürfen nicht länger als 255 Zeichen sein.
- Sie müssen innerhalb des Gültigkeitsbereichs, für den sie deklariert werden, eindeutig sein.

Gültigkeitsbereich und Lebensdauer von Variablen

Wenn Sie eine Variable in einer Prozedur deklarieren, kann normalerweise nur Code innerhalb dieser Prozedur auf den Variablenwert zugreifen oder ihn ändern, d.h. die Variable hat einen lokalen Gültigkeitsbereich und wird als Variable auf Prozedurebene bezeichnet.

Wenn Sie eine Variable außerhalb einer Prozedur deklarieren, ist sie für alle Prozeduren in Ihrem Skript sichtbar. Diese Art von Variable auf Skriptebene hat die Skriptebene als Gültigkeitsbereich.

Die Zeitspanne, in der eine Variable existiert, wird als ihre Lebensdauer bezeichnet. Die Lebensdauer einer Variablen auf Skriptebene reicht vom Zeitpunkt der Deklaration bis zur Beendigung des Skripts. Die Lebensdauer einer lokalen Variablen beginnt mit der Deklarationsanweisung am Anfang der Prozedur und endet mit Beendigung der Prozedur.

Lokale Variablen eignen sich gut als temporärer Speicherplatz während der Ausführung einer Prozedur. Sie können lokale Variablen mit demselben Namen in mehreren Prozeduren verwenden, da jede nur von der Prozedur erkannt wird, in der sie deklariert wurde.

Zuweisen von Werten zu Variablen

Werte werden einer Variablen durch einen Ausdruck zugewiesen, bei dem die Variable auf der linken Seite des Ausdrucks und der zuzuweisende Wert auf der rechten Seite des Ausdrucks steht. Beispiel:

```
B = 200
```

Skalare Daten und Datenfelder

Meistens weisen Sie einer deklarierten Variablen nur einen einzelnen Wert zu. Solche Variablen werden skalare Variablen genannt. In anderen Fällen ist es sinnvoll, einer Variablen mehrere zusammenhängende Werte zuzuweisen. In diesen Fällen können Sie eine Variable erstellen, die eine Folge von Werten enthalten kann. Solche Variablen nennt man Datenfeldvariablen. Datenfeldvariablen werden fast genauso deklariert wie skalare Variablen. Der einzige Unterschied besteht darin, dass bei der Deklaration von Datenfeldvariablen auf den Namen Klammern folgen. Im folgenden Beispiel wird ein eindimensionales Datenfeld mit 11 Elementen deklariert:

```
Dim A(10)
```

Obwohl in Klammern die Zahl 10 angegeben wird, enthält dieses Datenfeld 11 Elemente, weil alle Datenfelder in VBScript nullbasiert sind, d.h. mit dem Index Null beginnen. In einem nullbasierten Datenfeld ist die Anzahl der Elemente immer um Eins höher als der in Klammern angegebene Wert. Diese Art von Datenfeld nennt man Datenfeld fester Größe.

Zur Zuweisung von Daten an die Elemente des Datenfelds verwenden Sie einen Index im Datenfeld. Mit Indizes von Null bis 10 können den Elementen eines Datenfelds auf folgende Weise Werte zugewiesen werden:

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

Auf ähnliche Weise (mit einem Index im Datenfeld) kann der Wert eines beliebigen Datenfeldelements wieder abgerufen werden. Beispiel:

```
. . .
EineVariable = A(8)
. . .
```

Datenfelder sind nicht auf eine einzige Dimension beschränkt. Sie können bis zu 60 Dimensionen angeben, obwohl sich die meisten Menschen nur drei oder vier Dimensionen vorstellen können. Mehrere Dimensionen werden durch Angabe von mehreren Datenfeldgrößen zwischen den Klammern deklariert und jeweils durch ein Komma voneinander getrennt. Im folgenden Beispiel ist die Variable Tabelle1 ein zweidimensionales Datenfeld mit 6 Zeilen und 11 Spalten:

```
Dim Tabelle1(5, 10)
```

In einem zweidimensionalen Datenfeld gibt die erste Zahl immer die Anzahl der Zeilen und die zweite Zahl die Anzahl der Spalten an.

Sie können auch Datenfelder deklarieren, deren Größe sich während des Ablaufs Ihres Skripts ändern kann. Solche Datenfelder werden als dynamische Datenfelder bezeichnet. Das Datenfeld wird in einer Prozedur anfänglich mit einer `Dim` -Anweisung (wie jedes andere Datenfeld auch) oder mit einer `ReDim` -Anweisung deklariert. Der Unterschied besteht darin, dass zwischen den Klammern keine Größe oder Anzahl von Dimensionen angegeben wird. Beispiel:

```
(Dim EinDatenfeld()
ReDim AnderesDatenfeld)
```

Um ein dynamisches Datenfeld verwenden zu können, müssen Sie anschließend die Zahl der Dimensionen und deren Größen mit `ReDim` festlegen. Im folgenden Beispiel setzt `ReDim` die anfängliche Größe des dynamischen Datenfelds auf 25. Eine nachfolgende `ReDim` -Anweisung ändert die Größe auf 30, verwendet aber das Schlüsselwort `Preserve`, so dass der Inhalt des Datenfelds bei der Größenänderung erhalten bleibt.

```
ReDim EinDatenfeld(25)
. . .
ReDim Preserve EinDatenfeld(30)
```

6.3.3 Konstanten

Eine Konstante ist ein aussagekräftiger Name, der an Stelle einer Zahl oder einer Zeichenfolge verwendet wird und dessen Wert sich nie ändert. VBScript legt eine Anzahl integrierter Konstanten fest. Weitere Informationen zu diesen integrierten Konstanten finden Sie im VBScript-Sprachverzeichnis.

Sie können benutzerdefinierte Konstanten in VBScript mit der Const-Anweisung erstellen. Damit können Sie Zeichenfolgenkonstanten oder numerische Konstanten einen sinnvollen Namen zuweisen. Anschließend können Sie ihnen Literalwerte zuweisen und sie in Ihrem Skript verwenden. Beispiel:

```
Const Zeichenfolge1 = Dies ist eine Zeichenfolge.
Const MeinAlter = 49
```

Beachten Sie, dass das Zeichenfolgenliteral in Anführungszeichen () eingeschlossen ist. Anführungszeichen eignen sich am besten, um Zeichenfolgenwerte von numerischen Werten zu unterscheiden. Datums-/Uhrzeitliterale werden in Nummernzeichen (#) eingeschlossen. Beispiel:

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

Weil es zwischen den auf diese Weise erstellten Konstanten und normalen Variablen keinen funktionellen Unterschied gibt, sollten Sie ein Namensschema verwenden, um Konstanten von Variablen zu unterscheiden. Dies schützt Sie davor, einer Konstanten während der Skriptausführung versehentlich einen neuen Wert zuzuweisen. Sie können z. B. den Präfix vb oder con für Ihre Konstantennamen verwenden, oder Sie können Ihre Konstantennamen groß schreiben. Sorgen Sie auf jeden Fall dafür, dass eine Unterscheidung zwischen Konstanten und Variablen möglich ist. Dadurch vermeiden Sie Probleme, wenn Sie komplexere Skripten erstellen.

6.3.4 Operatoren

VBScript verfügt über eine Vielzahl von Operatoren, u.a. für arithmetische, Vergleichs-, Verkettungs- und logische Operationen.

Operatorvorrang

Wenn in einem Ausdruck mehrere Operatoren vorkommen, wird jeder Bestandteil in einer vordefinierten Reihenfolge ausgewertet und aufgelöst. Diese Reihenfolge wird als Operatorvorrang bezeichnet. Sie können den Operatorvorrang mit Klammern übergehen und die Auswertung von Bestandteilen eines Ausdrucks vor anderen Bestandteilen erzwingen. Operationen innerhalb von Klammern werden immer vor den Operationen außerhalb von Klammern ausgeführt. Innerhalb der Klammern gilt jedoch der normale Operatorvorrang.

Wenn ein Ausdruck Operatoren verschiedener Kategorien enthält, werden zuerst arithmetische, dann Vergleichs- und schließlich logische Operatoren ausgewertet. Alle Vergleichsoperatoren haben denselben Rang, d.h. sie werden in der Reihenfolge ihres Auftretens von links nach rechts ausgewertet. Arithmetische und logische Operatoren werden in folgender Reihenfolge ausgewertet.

Arithmetische Operatoren

Beschreibung	Symbol
Potenzierung	^
Unäre Negation	-
Multiplikation	*
Division	/
Ganzzahlige Division	/
Modulo-Arithmetik	Mod
Addition	+
Subtraktion	-
Zeichenfolgenverkettung	

Vergleichs-Operatoren

Beschreibung	Symbol
Gleichheit	=
Ungleichheit	<>
Kleiner als	<
Größer als	>
Kleiner oder gleich	<=
Größer oder gleich	>=
Objektäquivalenz	Is

Logische Operatoren

Wenn in einem Ausdruck mehrere Operatoren vorkommen, wird jeder Bestandteil in einer vordefinierten Reihenfolge ausgewertet und aufgelöst. Diese Reihenfolge wird als Operatorvorrang bezeichnet. Sie können den Operatorvorrang mit Klammern übergehen und die Auswertung von Bestandteilen eines Ausdrucks vor anderen Bestandteilen erzwingen. Operationen innerhalb von Klammern werden immer vor den Operationen außerhalb von Klammern ausgeführt. Innerhalb der Klammern gilt jedoch der normale Operatorvorrang.

Wenn ein Ausdruck Operatoren verschiedener Kategorien enthält, werden zuerst arithmetische, dann Vergleichs- und schließlich logische Operatoren ausgewertet. Alle Vergleichsoperatoren haben denselben Rang, d.h. sie werden in der Reihenfolge ihres Auftretens von links nach rechts ausgewertet. Arithmetische und logische Operatoren werden in folgender Reihenfolge ausgewertet.

6.3.5 Steuern der Programmausführung

Sie können den Ablauf eines Skripts mit Bedingungs- und Schleifenanweisungen steuern. Mit Hilfe von Bedingungsanweisungen können Sie VBScript-Code schreiben, mit dem Entscheidungen getroffen und Aktionen wiederholt werden können.

Treffen von Entscheidungen mit Hilfe von If...Then...Else

Die Anweisung `If...Then...Else` wird verwendet, um zu bestimmen, ob eine Bedingung `True` oder `False` ist, und um anzugeben, ob je nach Ergebnis eine oder weitere Anweisungen ausgeführt werden. Im allgemeinen ist die Bedingung ein Ausdruck, der einen Vergleichsoperator zum Vergleichen eines Wertes oder einer Variablen mit einem/r anderen verwendet. Informationen zu Vergleichsoperatoren

finden Sie unter Vergleichsoperatoren. `If...Then...Else` -Anweisungen können nach Bedarf auf beliebig vielen Ebenen verschachtelt werden.

Ausführen von Anweisungen, wenn eine Bedingung True ist

Um nur eine Anweisung auszuführen, wenn eine Bedingung `True` ist, verwenden Sie die einzeilige Syntax für die `If...Then...Else` -Anweisung. Das folgende Beispiel zeigt die einzeilige Syntax. Beachten Sie, dass in diesem Beispiel das Schlüsselwort `Else` weggelassen wird.

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

Zum Ausführen von mehr als einer Codezeile müssen Sie die mehrzeilige oder Blocksyntax verwenden. Diese Syntax enthält die Anweisung `End If` , wie im folgenden Beispiel gezeigt:

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

Ausführen, wenn eine Bedingung `True` ist, verwenden Sie die einzeilige Syntax für die `If...Then...Else` -Anweisung. Das folgende Beispiel zeigt die einzeilige Syntax. Beachten Sie, dass in diesem Beispiel das Schlüsselwort `Else` weggelassen wird.

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

Zum Ausführen von mehr als einer Codezeile müssen Sie die mehrzeilige oder Blocksyntax verwenden. Diese Syntax enthält die Anweisung `End If` , wie im folgenden Beispiel gezeigt:

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

Ausführen bestimmter Anweisungen, wenn eine Bedingung True ist, und Ausführen anderer Anweisungen, wenn eine Bedingung False ist

Mit der Anweisung `If...Then...Else` können Sie zwei Blöcke mit ausführbaren Anweisungen definieren: ein Block wird ausgeführt, wenn die Bedingung `True` ist, der andere wird ausgeführt, wenn die Bedingung `False` ist.

```
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
```

Entscheiden zwischen mehreren Alternativen

Mit Hilfe einer Variation der `If...Then...Else` -Anweisung können Sie aus mehreren Alternativen auswählen. Durch das Hinzufügen der Klausel `ElseIf` können Sie die Funktionalität der `If...Then...Else` -Anweisung erweitern, so dass der Programmablauf auf mehreren Möglichkeiten basiert.

```
Beispiel:
Sub ReportValue(value)
If value = 0 Then
MsgBox value
ElseIf value = 1 Then
MsgBox value
ElseIf value = 2 then
Msgbox value
Else
Msgbox Wert ausserhalb des Bereichs!
End If
```

Treffen von Entscheidungen mit Select Case

Die `Select Case` -Struktur bietet eine Alternative zu `If...Then...ElseIf` , um gezielt einen Anweisungsblock aus mehreren Anweisungsblöcken auszuführen. Eine `Select Case` -Anweisung bietet dieselbe Funktionalität wie die `If...Then...Else` -Anweisung, macht den Code jedoch effizienter und leichter lesbar.

Eine `Select Case` -Struktur arbeitet mit einem einzelnen Testausdruck, der einmalig auf der obersten Ebene der Struktur ausgewertet wird. Das Ergebnis der Struktur wird dann mit den Werten für jeden `Case` in der Struktur verglichen. Gibt es eine übereinstimmung, wird der mit diesem `Case` verknüpfte Anweisungsblock ausgeführt:

```
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 Verwenden von Schleifen zum Wiederholen von Code

Das Durchlaufen von Schleifen ermöglicht Ihnen, eine Anweisungsgruppe wiederholt auszuführen. Einige Schleifen wiederholen Anweisungen, bis eine Bedingung falsch (`False)` wird, andere wiederholen Anweisungen, bis eine Bedingung wahr (`True)` wird. Es gibt auch Schleifen, die Anweisungen in einer bestimmten Anzahl wiederholen.

Die folgenden Schleifenanweisungen sind in VBScript verfügbar:

Parameter	Beschreibung
Verwenden von Do-Schleifen (auf Seite 128):	Schleife durchlaufen, solange oder bis eine Bedingung True ist.
Verwenden von While...Wend (auf Seite 130):	Schleife durchlaufen, solange eine Bedingung True ist.
Verwenden von For...Next (auf Seite 130):	Verwendet einen Zähler, der angibt, wie oft Anweisungen ausgeführt werden.
Verwenden von For Each...Next (auf Seite 131):	Wiederholt eine Anweisungsgruppe für jedes Element in einer Auflistung oder je Datenfeld.

Verwenden von Do-Schleifen

Sie können `Do...Loop` -Anweisungen verwenden, um einen Anweisungblock mehrmals auszuführen, ohne dass die Anzahl der Durchläufe festgelegt ist. Die Anweisungen werden entweder wiederholt, solange eine Bedingung `True` ist oder bis eine Bedingung `True` wird.

Wiederholung von Anweisungen, während eine Bedingung True ist

Verwenden Sie das Schlüsselwort `While` , um eine Bedingung in einer `Do...Loop` -Anweisung zu überprüfen. Sie können die Bedingung überprüfen, bevor die Schleife ausgeführt wird (wie im folgenden Beispiel ChkFirstWhile gezeigt), oder Sie können sie überprüfen, nachdem die Schleife mindestens einmal ausgeführt wurde (wie im Beispiel ChkLastWhile gezeigt). Wenn in der ChkFirstWhile-Prozedur myNum auf 9 anstatt auf 20 festgelegt wird, werden die Anweisungen in der Schleife niemals ausgeführt. In der ChkLastWhile-Prozedur werden die Anweisungen in der Schleife nur einmal ausgeführt, weil die Bedingung bereits `False` ist.

```
Sub ChkFirstWhile()
   Dim counter, myNum
   counter = 0
   myNum = 20
   Do While myNum > 10
       myNum = myNum - 1
       counter = counter + 1
   Loop
   MsgBox 'Die Schleife wurde ' & counter & ' mal wiederholt.'
End Sub

Sub ChkLastWhile()
   Dim counter, myNum
   counter = 0
   myNum = 9
   Do
       myNum = myNum - 1
       counter = counter + 1
   Loop While myNum > 10
   MsgBox 'Die Schleife wurde ' & counter & ' mal wiederholt.'
End Sub
```

Wiederholen einer Anweisung, bis eine Bedingung True wird

Sie können das Schlüsselwort `Until` auf zwei Arten verwenden, um eine Bedingung in einer `Do...Loop` -Anweisung zu überprüfen. Sie können die Bedingung überprüfen, bevor die Schleife ausgeführt wird (wie im folgenden Beispiel ChkFirstUntil gezeigt), oder Sie können sie überprüfen, nachdem die Schleife mindestens einmal ausgeführt wurde (wie im Beispiel ChkLastUntil gezeigt). Solange die Bedingung `False` ist, wird die Schleife durchlaufen.

```
Sub ChkFirstUntil()
   Dim counter, myNum
   counter = 0
   myNum = 20
   Do Until myNum = 10
       myNum = myNum - 1
       counter = counter + 1
   Loop
   MsgBox 'Die Schleife wurde ' & counter & ' mal wiederholt.'
End Sub

Sub ChkLastUntil()
   Dim counter, myNum
   counter = 0
   myNum = 1
   Do
       myNum = myNum + 1
       counter = counter + 1
   Loop Until myNum = 10
   MsgBox 'Die Schleife wurde ' & counter & ' mal wiederholt.'
End Sub
```

Beenden einer Do...Loop-Anweisung aus der Schleife heraus

Sie können eine `Do...Loop` -Schleife beenden, indem Sie die `Exit Do` -Anweisung verwenden. Da die Schleifenausführung gewöhnlich nur in bestimmten Situationen beendet werden soll, z. B. um eine Endlosschleife zu vermeiden, sollten Sie die `Exit Do` -Anweisung im `True` -Anweisungsblock einer `If...Then...Else` -Anweisung verwenden. Wenn die Bedingung `False` ist, wird die Schleife normal ausgeführt.

Im folgenden Beispiel wird myNum ein Wert zugewiesen, der eine Endlosschleife verursacht. Die `If...Then...Else` Anweisung überprüft, ob dieser Zustand eintritt, und verhindert die endlose Wiederholung.

```
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 'Die Schleife wurde ' & counter & ' mal wiederholt.'
End Sub
```

Verwenden von While...Wend

Die `While...Wend` -Anweisung wird in VBScript für Entwickler zur Verfügung gestellt, die mit ihrer Verwendung vertraut sind. Aufgrund der eingeschränkten Flexibilität von `While...Wend` wird jedoch empfohlen, stattdessen `Do...Loop` zu verwenden.

Verwenden von For...Next

Sie können `For...Next` -Anweisungen verwenden, um genau festzulegen, wie oft ein Anweisungblock ausgeführt werden soll. Für Schleifen verwenden Sie eine Zählervariable, deren Wert mit jeder Wiederholung der Schleife vergrößert oder verkleinert wird.

Beispielsweise hat die folgende Prozedur zur Folge, dass eine Prozedur mit dem Namen MyProc 50 mal ausgeführt wird. Die `For` -Anweisung gibt als Zählervariable x an und legt den zugehörigen Start- und Endwert fest. Die `Next` -Anweisung zählt die **Zählervariable jeweils** um 1 hoch.

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

Mit dem Schlüsselwort `Step` können Sie die Zählervariable um den darin angegebenen Wert vergrößern oder verkleinern. Im folgenden Beispiel wird die Zählervariable j bei jedem Schleifendurchlauf um 2 hochgezählt. Wenn die Schleife beendet ist, enthält total die Summe von 2, 4, 6, 8 und 10.

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

```
End Sub
```

Zum Verkleinern der Zählervariable verwenden Sie einen negativen Wert für `Step` . Sie müssen einen Endwert angeben, der kleiner als der Startwert ist. Im folgenden Beispiel wird die Zählervariable myNum bei jedem Schleifendurchlauf um 2 verkleinert. Wenn die Schleife beendet ist, enthält total die Summe von 16, 14, 12, 10, 8, 6, 4 und 2.

```
Sub NewTotal()
   Dim myNum, total
   For myNum = 16 To 2 Step -2
       total = total + myNum
   Next
   MsgBox 'Die Summe ist ' & total
End Sub
```

Verwenden von For Each...Next

Eine `For Each...Next` -Schleife ist einer `For...Next` -Schleife ähnlich. Anstatt die Anzahl der Schleifendurchläufe zum Wiederholen der Anweisungen anzugeben, wird in einer `For Each...Next` -Schleife eine Anweisungsgruppe für jedes Element in einer Auflistung von Objekten oder für jedes Element in einem Datenfeld ausgeführt. Dies ist besonders dann hilfreich, wenn nicht bekannt ist, wieviele Elemente in einer Gruppe enthalten sind.

Im folgenden HTML-Codebeispiel wird der Inhalt eines `Dictionary` -Objekts verwendet, um Text in mehrere Textfelder einzusetzen:

```
<HTML>
<HEAD><TITLE>Formulare und Elemente</TITLE></HEAD>
<SCRIPT LANGUAGE='VBScript'>
<!--
Sub cmdChange_OnClick
  Dim d                    'Erstellen einer Variablen
   Set d = CreateObject('Scripting.Dictionary')
  d.Add '0', 'Athen'       'Einige Schlüssel und Elemente hinzufügen
  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 Arten von Prozeduren

Sub-Prozeduren

Eine `Sub` -Prozedur ist eine Folge von VBScript-Anweisungen, die zwischen den Anweisungen `Sub` und `End Sub` eingeschlossen ist, und die Aktionen ausführt, aber keine Werte zurückgibt. Eine `Sub` -Prozedur kann Argumente (Konstanten, Variablen oder Ausdrücke, die von einer aufrufenden Prozedur übergeben wurden) übernehmen. Wenn eine `Sub` -Prozedur keine Argumente hat, muss ihre `Sub` -Anweisung ein leeres Klammernpaar enthalten ().

Die folgende `Sub` -Prozedur verwendet zwei integrierte (oder eingebaute) VBScript-Funktionen, `MsgBox` und `InputBox` , um einen Benutzer zur Eingabe von Informationen aufzufordern. Dann zeigt sie das Ergebnis einer Berechnung an, die auf diesen Informationen basiert. Die Berechnung wird in einer mit VBScript erstellten `Function` -Prozedur ausgeführt. Diese wird in der Behandlung der nachfolgenden `Function` -Prozedur ebenfalls gezeigt.

```
Sub KonvertTemp()
  temp = InputBox('Geben Sie die Temperatur in Fahrenheit ein.', 1)
  MsgBox 'Die Temperatur ist ' & Celsius(temp) & ' Grad Celsius.'
End Sub
```

Function-Prozeduren

Eine `Function` -Prozedur ist eine Folge von VBScript-Anweisungen, die zwischen den Anweisungen `Function` und `End Function` eingeschlossen ist. Eine `Function` -Prozedur ist einer `Sub` -Prozedur ähnlich, kann aber auch einen Wert zurückgeben. Eine `Function` -Prozedur kann Argumente (Konstanten, Variablen oder Ausdrücke, die von einer aufrufenden Prozedur übergeben werden) übernehmen. Wenn eine `Function` -Prozedur keine Argumente hat, muss ihre `Function` -Anweisung ein leeres Klammernpaar enthalten. Eine `Function` -Prozedur gibt Werte zurück, indem ihrem Namen in einer oder mehreren Anweisung(en) der Prozedur ein Wert zugewiesen wird. Der Typ des Rückgabewerts einer `Function` -Prozedur ist immer ein `Variant` .

Die Funktion Celsius im folgenden Beispiel berechnet aus Gradwerten in Fahrenheit die entsprechenden Werte in Grad Celsius. Beim Aufruf der Funktion aus der `Sub` -Prozedur KonvertTemp wird eine Variable mit dem Argumentwert an die Funktion **übergeben.** Das Ergebnis der Berechnung wird an die aufrufende Prozedur zurückgegeben und in einem Meldungsfeld angezeigt.

```
Sub KonvertTemp()
   temp = InputBox('Geben Sie die Temperatur in Fahrenheit ein.', 1)
   MsgBox 'Die Temperatur ist ' & Celsius(temp) & ' Grad Celsius.'
End Sub

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

Übergeben von Daten an oder aus Prozeduren

Jedes Datenelement wird Ihrer Prozedur über ein Argument übergeben. Argumente dienen als Platzhalter für die Daten, die Sie Ihrer Prozedur übergeben möchten. Wenn Sie eine Prozedur mit der `Sub` - oder der `Function` -Anweisung erstellen, müssen nach dem Namen der Prozedur Klammern angegeben werden. Alle Argumente werden zwischen diesen Klammern angegeben und durch Kommas getrennt. In der folgenden Prozedur ist beispielsweise **GradF** ein Platzhalter für den Wert, der an die Funktion Celsius zur Konvertierung übergeben wird:

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

Verwenden von Sub- und Function-Prozeduren im Code

Um eine `Function` -Prozedur in Ihrem Code zu verwenden, müssen Sie diese auf der rechten Seite einer Variablenzuweisung oder in einem Ausdruck verwenden.

Beispiele:

```
Temp = Celsius(GradF)
```

oder

```
MsgBox 'Die Temperatur beträgt ' & Celsius(GradF) & ' Grad Celsius.'
```

Um eine `Sub` -Prozedur aus einer anderen Prozedur aufzurufen, müssen Sie lediglich den Namen der Prozedur zusammen mit den erforderlichen Argumenten (durch Kommas getrennt) eingeben. Die `Call` -Anweisung ist nicht erforderlich, wenn Sie sie jedoch verwenden, müssen Sie die Argumente in Klammern einschließen.

Das folgende Beispiel zeigt zwei Aufrufe der Prozedur MeineProz. In einem Fall wird die `Call` -Anweisung im Code verwendet, im anderen nicht. Beide Aufrufe haben genau dieselbe Wirkung.

```
Call MeineProz(erstesArg, zweitesArg)
MeineProz erstesArg, zweitesArg
```

6.3.8 Konventionen für die Programmierung

Konventionen für die Programmierung sind Vorschläge, die Ihnen das Schreiben von Code mit der Microsoft Visual Basic Scripting Edition erleichtern sollen.

Konventionen für die Programmierung umfassen unter anderem folgende Richtlinien:

Namenskonventionen für Objekte, Variablen und Prozeduren
Konventionen für Kommentare
Richtlinien für die Formatierung und das Einrücken von Text

Der wichtigste Grund für einheitliche Konventionen für die Programmierung ist die Standardisierung der Code-Struktur und des Programmierstils für ein oder mehrere Skripten, damit der Code für Sie und andere leichter lesbar und verständlicher ist. Mit sinnvollen Konventionen für die Programmierung erhalten Sie präzisen, lesbaren und eindeutigen Code, der den für andere Programmiersprachen verwendeten Konventionen entspricht und so intuitiv wie möglich ist.

Namenskonventionen für Konstanten

Frühere Versionen von VBScript besaßen keine Mechanismen für die Erstellung benutzerdefinierter Konstanten. Konstanten wurden, wenn notwendig, wie Variablen implementiert und von anderen Variablen dadurch unterschieden, dass nur Großbuchstaben verwendet wurden. Mehrere Wörter wurden durch den Unterstrich (_) getrennt.

Beispiele:

```
BENUTZER_LISTE_MAX
NEUE_ZEILE
```

Obwohl dieser Weg, Konstanten zu kennzeichnen, noch funktioniert, können Sie eine andere Art der Namensgebung verwenden. Sie können echte Konstanten mit Hilfe der `Const` -Anweisung anlegen. Diese Konvention verwendet ein Format mit gemischter Groß- und Kleinschreibung, in dem Konstanten das Präfix con vorangestellt wird.

Zum Beispiel:

```
conIhreEigeneKonstante
```

Namenskonventionen für Variablen

Um die Lesbarkeit und die Konsistenz zu verbessern, verwenden Sie in VBScript die in der folgenden Übersicht aufgeführten Präfixe zusammen mit aussagekräftigen Variablennamen.

Untertyp	Präfix	Beispiel
Boolean	bln	blnGefunden
Byte	byt	bytRasterDaten
Date (Time)	dtm	dtmStart
Double	dbl	dblToleranz
Error	err	errBestellNr
Integer	int	intAnzahl
Long	lng	lngEntfernung
Object	obj	objAktuell
Single	sng	sngDurchschnitt
String	str	strVorname

Gültigkeitsbereich von Variablen

Variablen sollten immer mit dem kleinstmöglichen Gültigkeitsbereich definiert werden. VBScript-Variablen können einen der folgenden Gültigkeitsbereiche haben.

Gültigkeitsbereich	Variablendeklaration in	Sichtbarkeit
Prozedurebene	Ereignis-, Function- oder Sub-Prozedur	Sichtbar in der Prozedur, in der sie deklariert wird.
Skriptebene	HEAD-Abschnitt einer HTML-Seite, außerhalb jeder Prozedur	Sichtbar in jeder Prozedur des Skripts.

Präfixe für den Gültigkeitsbereich von Variablen

Mit zunehmender Skriptgröße wird es immer wichtiger, die Gültigkeitsbereiche der Variablen schnell unterscheiden zu können. Dies wird durch einen Gültigkeitsbereichs-Präfix aus einem Buchstaben vor dem Typ-Präfix erreicht, ohne die Variablennamen übermäßig zu verlängern.

Gültigkeitsbereich	Präfix	Beispiel
Prozedurebene	keiner	dblGeschwindigkeit
Skriptebene	s	sblnBerechnungInArbeit

Aussagekräftige Variablen- und Prozedurnamen

Im Rumpf eines Variablen- oder Prozedurnamens sollte Groß- und Kleinschreibung verwendet werden. Der Name sollte so ausführlich sein wie nötig, um den Zweck der Variablen zu beschreiben. Darüber hinaus sollten Prozedurnamen mit einem Verb beginnen, wie z. B. InitialisiereFeldName oder SchliesseDialogfeld.

Für häufig verwendete oder lange Ausdrücke sind Standardabkürzungen zu empfehlen, um die Namen kurz zu halten. Im allgemeinen sind Variablennamen mit mehr als 32 Zeichen schwer lesbar. Stellen Sie sicher, dass die verwendeten Abkürzungen im gesamten Skript einheitlich sind. Der grundlose Wechsel zwischen Zhl und Zaehl innerhalb eines oder mehrerer zusammenhängender Skripten kann Probleme bereiten.

Namenskonventionen für Objekte

Die folgende Tabelle führt die empfohlenen Konventionen für die verschiedenen Objekte auf, die beim Programmieren in VBScript verwendet werden.

Objekttyp	Präfix	Beispiel
3D-Grundfläche (3D panel)	pnl	pnlGruppe
Animierte Schaltfläche (animated button)	ani	aniMailBox
Check box	chk	chkReadOnly
Kombinationsfeld (combo box), Dropdownlistenfeld (drop-down list box)	cbo	cboDeutsch
Befehlsschaltfläche (command button)	cmd	cmdExit
Standarddialogfeld (common dialog)	dlg	dlgDateiOeffnen
Rahmen (frame)	fra	fraSprache
Horizontale Bildlaufleiste (horizontal scroll bar)	hsb	hsbLautstaerke
Abbildung (image)	img	imgSymbol
Bezeichnungsfeld (label)	lbl	lblHilfeMeldung
Linie (line)	lin	linVertikal
Listenfeld (list box)	lst	lstVersicherungsnummern
Drehfeld (Spin)	spn	spnSeiten
Textfeld (text box)	txt	txtNachname
Vertikale Bildlaufleiste (vertical scroll bar)	vsb	vsbRate
Bildlauffeld (Slider)	sld	sldSkala

Konventionen für Code-Kommentare

Jede Prozedur sollte mit einem kurzen Kommentar beginnen, der den Zweck der Prozedur beschreibt. Diese Beschreibung sollte nicht auf die Implementierungsdetails (wie die Operationen durchgeführt werden) eingehen, weil sich diese im Laufe der Zeit ändern können, was zu unnötigen Wartungsarbeiten bei den Kommentaren oder, noch schlimmer, zu falschen Kommentaren führen kann. Der Code selbst und ggf. nötige Kommentare im Code beschreiben die Implementierung.

Die an eine Prozedur übergebenen Argumente sollten beschrieben werden, wenn ihr Zweck nicht offensichtlich ist und wenn die Prozedur die Argumente in einem bestimmten Wertebereich erwartet. Rückgabewerte von Funktionen und andere Variablen, die von der Prozedur geändert werden, vor allem über Argumente, die als Referenz übergeben werden, sollten ebenfalls am Anfang jeder Prozedur beschrieben werden.

Die Kommentare im Prozedurkopf sollten folgende Abschnittsüberschriften enthalten. Beispiele finden Sie im nachfolgenden Abschnitt Code-Formatierung.

Abschnittsüberschrift	Kommentar
`Zweck`	Was die Prozedur tut (nicht, wie sie es tut).
`Annahmen`	Liste der Auswirkungen dieser Prozedur auf externe Variablen, Steuerelemente oder andere Elemente.
`Auswirkungen`	Liste der Auswirkungen dieser Prozedur auf externe Variablen, Steuerelemente oder andere Elemente.
`Eingehende Werte`	Erläuterung jedes Arguments, das nicht offensichtlich ist. Jedes Argument sollte in einer separaten Zeile mit Kommentaren im Code beschrieben werden.
`Rückgabewerte`	Erklärung des zurückgegebenen Wertes.

Berücksichtigen Sie folgende Punkte:

Die Deklaration jeder wichtigen Variablen sollte einen Kommentar im Code enthalten, der die Verwendung der deklarierten Variablen beschreibt.
Variablen, Steuerelemente und Prozeduren sollten so klar benannt werden, dass Kommentare im Code nur noch für komplexe Implementierungsdetails nötig sind.
Sie sollten am Anfang Ihres Skripts eine übersicht einfügen, die das Skript beschreibt und die Objekte, Prozeduren, Algorithmen, Dialogfelder und andere Systemabhängigkeiten auflistet. Manchmal kann ein Stück Pseudo-Code helfen, einen Algorithmus zu beschreiben.

Code-Formatierung

Sie sollten mit dem Platz auf dem Bildschirm möglichst sparsam umgehen und trotzdem den Code so formatieren, dass die logische Struktur und Verschachtelung deutlich wird. Hier sind einige Hinweise:

- Normal verschachtelte Blöcke sollten um vier Leerzeichen eingerückt sein.
- Der übersichtskommentar einer Prozedur sollte um ein Leerzeichen eingerückt sein.
- Die Anweisungen auf höchster Ebene, die auf den übersichtskommentar folgen, sollten um vier Leerzeichen eingerückt sein. Jeder verschachtelte Block sollte jeweils um weitere vier Leerzeichen eingerückt werden.

Beispiel:

Der folgende Code entspricht den VB Script Code Konventionen:

- Zweck: Sucht das erste Vorkommen des angegebenen Benutzers im Datenfeld BenutzerListe.
- Eingaben: strBenutzerListe(): Zu durchsuchende Benutzerliste.
- strZielBenutzer: Name des zu suchenden Benutzers.

- Rückgabewerte: Index des ersten Vorkommens von strZielBenutzer im Datenfeld strBenutzerListe.
  Wenn der gesuchte Benutzer nicht gefunden wird: -1.

```
Function intSucheBenutzer (strBenutzerListe(), strZielBenutzer)
    Dim i                           ' Schleifenzähler.
    Dim blnGefunden                 ' 'Ziel gefunden'-Attribut.
    intSucheBenutzer = -1
    i = 0                           ' Schleifenzähler initialisieren
    Do While i <= Ubound(strBenutzerListe) and Not blnGefunden
        If strBenutzerListe(i) = strZielBenutzer Then
            blnGefunden = True       ' Flag auf True setzen
            intSuche Benutzer = i    ' Rückgabewert für Schleifenzähler setzen
        End If
        i = i + 1              ' Schleifenzähler erhöhen
    Loop
End Function
```