www.asix.com.pl

Asix.Evo - Techniques of Diagram Creation

Doc. No ENP7E009
Version: 2012-08-30

ASKOM

ASKOM Sp. z o. o., ul. Józefa Sowińskiego 13, 44-121 Gliwice,
tel. +48 32 3018100, fax +48 32 3018101,

http://www.askom.com.pl, e-mail: office@askom.com.pl

1 Universal Object Creation Methods

Easy copying of the objects created as well as easy and fast modification implementation for the entire application should be the goal of diagram creation. This chapter describes the available mechanisms and techniques aiming to achieve this goal. In subsequent chapters, these techniques will be consequently used in the described examples.

1.1. Suffix Notation of Variable Names

A well-designed object for monitoring the process variable status may be copied and switched to monitoring other variable by changing just a single property. The first of the mechanisms facilitating such a structure is the *Main variable* property and suffix notation of variable names.

Case 1

State Properties, primary group		
	Text	=VarStringValue(V1)
	Color	=Variable(V1)>50?Red:Black

Case 2

Basic Properties		
	Main Variable	V1
State Properties, primary group		
	Text	#
	Color	*=Variable()>50?Red:Black*

These fragments of *Text* object parameterization function the same way. In the first example the variable name is used twice. Making a copy and changing variable would require manual edition of the two properties. In the second case, the variable name is used only once. Using the main variable allows using ***Variable*** parameterless function and # short notation, indicating on the reference to the formatted value of the main variable. Copying of the object is very simple, moreover, also the *Group replacement of variables* editor function may be used.

Basic Properties		
	Main Variable	V1
	Control Variable	#Control
State Properties, primary group		
	Text	=VarStringValue(”#Value”)

Another example demonstrates the use of suffix notation based on the main variable name. The *#* symbol used in the context of the variable name means that object main variable name will be used directly. In conjunction with additional text it creates a variable name which is a combination of main variable name and the specified text. In this example, the controlled variable name is *V1Control*, and the text displayed is the *V1Value* variable value. Because there is no direct reference to the main variable, it is even not required from *V1* to be the existing variable.

1.2. Using Variable Attributes

Another mechanism enabling creation of universal objects is provided by process variable attributes.

Basic Properties		
	Main Variable	V1
State Properties, primary group		
	Text	#
	Color	=Variable()>Attribute(LimitHi)?Red:Black

Generally, each variable will have its own critical limit level. Entering its value directly would require the *Color* property do be modified each time. The ***Attribute*** function takes a *LimitHi* attribute of object main variable. After changing main variable name new limit value is retrieved automatically. The additional benefit of using the attributes of a variable is that a subsequent change in the limits of the variable definition database will not require modification of diagrams.

Any attributes from a predefined set may be placed in the variable definition database. It is also possible to add custom attributes of any significance.

1.3. Global Properties

The global properties are defined in the separate operating panel which is opened via the *Application Explorer* panel. The global properties allow creating a set of parameters that define the behaviour and appearance of objects on all diagrams. Change of the global property value is immediately visible in all objects that refer to this property.

State Properties, primary group		
	Text	#
	Color	!NormalColor

In the above example the text is displayed in the colour specified in the *NormalColor* global property. The subsequent colour change will not require change in the object.

The global value may be referenced also via the ***Property*** function.

The global properties are defined directly as usual. There are also other ways, such as the use of the expression.

=Color(Variable(),0,0)

This expression above when used in the global property of a colour type calculates a colour value based on the main variable value of the object, which refers to the property. The calculating algorithm may be changed at any time, without changing objects.

If the global property is not defined directly, the reference to it should be done by the **!** notation.

1.4. Templates and Parameterized Diagrams

Templates and parameterized diagrams enable creating complete, predefined object sets for multiple use. The templates can be embedded on the diagrams, while the diagrams itself make an independent entity. The parameterized diagrams, by using appropriate opening function can be used to display various data (box of "control window" type).

In both of these mechanisms, the designer defines certain set of parameters that control the operation of objects in the embedded template or opened diagram. However, most of the properties is defined directly in the template. Subsequent modification of the template (diagram) is immediately reflected in the place of occurrence.

2 Displaying Process Variable Values in Text Form

This section describes method of displaying the process variable in a text form. The displayed text formatting and display attributes changing depending on the different conditions will be discussed. In particular, multi-state object creation process will be presented.

The *Text* class object is used in the Asix.Evo applications to display texts of any type, both static and dynamic.

Note:
If displaying the values of multiple variables in tabular form is required the *Variable table* class object may be used.

2.1. Displaying Formatted Value

Variant A

Basic Properties		
	Main Variable	VA001
State Properties, primary group		
	Text	#

Variant B

Basic Properties, primary group		
	Text	&VA001

Both of these parameterization variants cause the VA001 variable displaying. A variable value will be converted into text based on *Format* attribute stored in the variable definition database. For example, if the format is *f2*, the value of the variable will be displayed as a number with two decimal places. If the format is not defined in the variable definition database, the conversion into text is performed in a standard way, depending on the value type.

The variable name in the variant A is specified in the *Main Variable* property, and the text was determined by a short notation **#**, indicating that the object main variable value is downloaded as a text.

In the variant B only short notation ***&variable_name*** was used. It means that the specified variable value will be downloaded as a text. For reasons that will be described later, it is recommended to use the standard procedure of the variant A. The variant B should be used only in the simplest parameterizations.

Variant C

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	=VarStringValue()

This variant functioning is identical to the variant A. Instead of the short notation the ***VarStringValue*** function was used that returns the object main variable value as a text.

Variant D

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	=Variable()

The difference between the variant D and the previous one is the use of the ***Variable*** function. This function returns the main variable value as a number. Conversion into text will be made automatically, the *Format* attribute will be negligible.

Variant E

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	=”Temperature = ” + VarStringValue()

The text displayed will combine *Temperature = text* and formatted value of the *VA001* variable.

Variant F

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	=Format("Temperatura = {0} \u00b0C", VarStringValue())

Formatted *VA001* variable value is inserted into the text *Temperatura = °C,* replacing the *{0}* tag. The ***Format*** function enables free creation of text strings based on the formatting text. The formatting text content shows method of insertion of the sign specified in the Unicode (\u00b0) into the text string.

Variant G

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	=Format("Temperatura = {0:f3} {1}", Variable(), Attribute(Unit))

This variant functioning is similar to the previous one, except the two following differences. The numerical value of the main variable is transferred to the *Format* function, and the conversion method is specified by the *{0:f3}* tag. This allows for formatting other than those specified in the variable definition database. In addition, the unit description is taken from the variable definition database, from the *Unit* attribute by calling the *Attribute* function. This example shows one of the benefits of the *Main Variable* property use. Without this, the analogous expression would look like this:

=Format("Temperatura = {0:f2} {1}", Variable(VA001),Attribute(VA001,Unit))

The variable name would be used twice. Much more difficult is switching the object to monitor the values of another variable.

2.2. Making the Text Dependent on the Value

In many cases, in addition to displaying the variable value, it is necessary to control display attributes. A typical case is the control of limit overreaching states, measurement errors, etc.

2.2.1. Use of Conditional Expressions

Variant A

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	#
	Color	=Variable()>50?Red:Black

The conditional expression has been used in the *Color* property. If the main variable value exceeds 50, then the variable value is displayed in red.

Variant B

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	#
	Color	=Variable()>50?Red:Black
	Font style	=Variable()>50?Bold:””

In the variant B, a text colour and font style is changed when the value of 50 is exceeded. Despite the use of the main variable mechanism, the solution is still inefficient because of the double use of the identical logical condition. Changing of this condition requires a lot of efforts and increases the possibility of errors. This problem may be solved with the object state mechanism.

2.2.2. Using Multi-State Method

Variant C

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	#
	Color	Black
State Properties, state number 1		
	State Condition	=Variable()>50
	Color	Red
	Font Style	Bold

Each additionally created state, in addition to a full set of basic state properties, has an additional *State Condition* property. Its value specifies whether the state is taken into account when determining the property value. In the example, if the value of main variable exceeds 50, state definitions of the *Color and Font Style* properties will have priority over definitions of the primary state. As a result, there will be obtained the effect as in option B, but without unnecessary iteration of condition and conditional expressions.

2.3. Using Warning Limits for the Variable Value

Another common problem is the need to signalling the alarm limit exceeding. The alarm limit values of measurement are sometimes stored in the other process variables. They can also be stored in the variable definition database.

2.3.1. The Use of Suffix Notation

In the following example, it is assumed that the limits are stored in variables. Where, the following naming convention is maintained: variable names of the limits are created by adding suffixes *_lolo, _lo, _hi, _hihi* to the name of the measurement variable.

Variant A

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	#
	Color	Black
State Properties, state number 1		
	State Condition	=Variable()<Variable("#_lo")
	Color	Aqua
State Properties, state number 2		
	State Condition	=Variable()<Variable("#_lolo")
	Color	Blue
State Properties, state no. 3		
	State Condition	=Variable()>Variable("#_hi")
	Color	Red
State Properties, state no. 4		
	State Condition	=Variable()<Variable("#_hihi")
	Color	Yellow

There were created 4 states which controls the exceeding of 4 limits. In the state expressions, the main variable value is compared with the limit variable value. Variable names as *#_lolo* are the suffix notation in which the *#* represents the main variable name.

Object dependency of variable names is minimal, switching the object to monitoring other variable requires only change of the main variable name.

Each of the states has its own definition of the *Color* property. The value defined for active state will be used to display. If there is more than one active state, then the value from the state with higher number will be taken. Therefore, the sequence of defining the states is essential. The state *_lolo* must be defined after the state *_lo,* and the state *_hihi* must be defined after the state *_hi*.

2.3.2. Application of Variable Attributes

The following example assumes that the limit values are stored in the attributes of variable definition database.

Variant B

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	#
	Color	Black
State Properties, state no. 1		
	State Condition	=Variable()<Attribute(LimitLo)
	Color	Aqua
State Properties, state no. 2		
	State Condition	=Variable()<Attribute(LimitLoLo)
	Color	Blue
State Properties, state no. 3		
	State Condition	=Variable()>Attribute(LimitHi)
	Color	Red
State Properties, state no. 4		
	State Condition	=Variable()<Attribute(LimitHiHi)
	Color	Yellow

The state conditions compare the variable value with the value of the specific limit loaded from a database. The limit values must be stored in a database as a numerical values.

Variant C

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	#
	Color	Black
State Properties, state number 1		
	State Condition	=Variable()<Variable(Attribute(LimitLo))
	Color	Aqua
State Properties, state number 2		
	State Condition	=Variable()<Variable(Attribute(LimitLoLo))
	Color	Blue
State Properties, state number 3		
	State Condition	=Variable()>Variable(Attribute(LimitHi))
	Color	Red
State Properties, state number 4		
	State Condition	=Variable()<Variable(Attribute(LimitHiHi))
	Color	Yellow

The state conditions compare the variable value with the value of the specific limit loaded from a database. However, in this case, the limit values stored in the database are the variable names (limit attribute value is passed to the *Variable* function).

Variant D

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	#
	Color	Black
State Properties, state number 1		
	State Condition	=Variable()<Asix6Attribute(LimitLo)
	Color	Aqua
State Properties, state number 2		
	State Condition	=Variable()<Asix6Attribute(LimitLoLo)
	Color	Blue
State Properties, state number 3		
	State Condition	=Variable()>Asix6Attribute(LimitHi)
	Color	Red
State Properties, state number 4		
	State Condition	=Variable()<Asix6Attribute(LimitHiHi)
	Color	Yellow

This example is a combination of options B and C. The *Asix6Atrribute* function accept both numerical limits (returned directly) and limits in a form of the variable names (returned value of the variable). In more complex cases, the *Asix6Limit* function can also be used.

2.4. Handling Monitored Variable Status

In addition to limit signalling, the indication of problems with the reading out of process variables is also often required.

Variant A

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Text	#
	Color	Black
State Properties, state number 1		
	State Condition	=Variable()<Asix6Attribute(LimitLo)
	Color	Aqua
State Properties, state no. 2		
	State Condition	=Variable()<Asix6Attribute(LimitLoLo)
	Color	Blue
State Properties, state number 3		
	State Condition	=Variable()>Asix6Attribute(LimitHi)
	Color	Red
State Properties, state number 4		
	State Condition	=Variable()<Asix6Attribute(LimitHiHi)
	Color	Yellow
State Properties, state number 5		
	State Condition	=VarIsNotGood()
	Text	=VarStringValue() + " ?"
	Cross out color	Violet

State no. 5 controls the status of the main variable. When it is incorrect, a question mark is added to the value and displayed text is crossed out. Regardless of variable state, the overreaching control is done in a normal way.

3 Formatting Numerical Values

Section *2.1. Displaying formatted value* is an overview of basic rules for conversion numerical values into texts. This section describes in detail a form of the formatting strings that control a method of conversion.

The formatting strings are used in two positions. First of them is the *Format* attribute of the variable definition database. It is used in the case of default conversion into text (use of the references #, &name or the *VarStringValue* function). The second position is a formatting parameter of the *Format* function.

The form of the formatting strings is compatible with the format used in the .NET Framework libraries. Full description of the format can be found in the *.NET Framework* documentation available in the Internet.

The format string used to call the *Format* function has the following structure:

{index, field_lenght: field_format}

The *field_format* element is to be used only in the *Format* attribute of variable definition database.

The *index* element specifies the parameter number of the *Format* function call which relates to the given format string. The first parameter has an index of 0.

The *field_length* element specifies the minimum length of text being created. If the length of formatted text will be shorter than declared, then it will be filled up with suitable number of spaces. Blank spaces are added to the left side, when the field assumes a positive value or otherwise to the right side.

The *field_format* element defines the way of converting a number into a text. The method of using it will be demonstrated on examples.

3.1. Conversion of Numerical Values

In a case of the numerical value conversion, the field format consists of the one-character type specifier and the numerical precision field. An interpretation of the precision field depends on type used.

Type specifier	Description	Example
D , d	Formatting integers. It can be used for floating point values, but in this case rounding to integer will be done. The precision field specifies the minimum number of digits. Leading zeros will be added if necessary.	”D” 1234 -> 1234 ”D” 1234,56 -> 1235 ”D6” 1234 -> 001234
E , e	Formatting floating point value in exponential form. The precision field specifies the number of fractional digits.	”E” 1234,56 -> 1,234560E+003 ”e2” 1234,56 -> 1,23e+003
F , f	Formatting floating point value in the total and fractional parts separated by the system separator. The precision field specifies the number of fractional digits.	”F” 1234,567 ->1234,57 ”F4” 1234,567 ->1234,5670 ”F1” 1234,567 ->1234,6
N , n	Formatting floating point value in the total and fractional parts separated by the system separator. In addition, a group of digits are separated by separator. The precision field specifies the number of fractional digits.	”N” 1234,567 ->1 234,57 ”N0” 1234,567 ->1 235
P , p	Formatting integers and floating point values in percentage form with a percent sign (%) added. The value will be multiplied by 100 before formatting. The precision field	”P” 0,567 ->56,70% ”P1” 0,567 ->56,7%

	specifies the number of fractional digits.	
X , x	Formatting integers in hexadecimal form. It can be used for floating point values, but in this case rounding to integer will be done. The precision field specifies the minimum number of digits. Leading zeros will be added if necessary.	”X” 1234 ->4D3 ”x4” 1234 ->04d3

The *Custom Numeric Format Strings* can be used in addition to the formatting described above. The detailed description can be found in the .NET Framework documentation.

3.2. Conversion of Values of the DateTime Type

The standard predefined formats or freely created custom formats may be used in case of conversion of the *DateTime* value.

The following table shows an example of standard formatting.

Type specifier	Description	Example
d	Short date	2012-02-16
D	Long date	February 16, 2012
f	Long date + short time	February 16, 2012 16: 33
F	Long date + short time	February 16, 2012 16:33:48
g	Full date + short time	2012-02-06 16:33
G	Full date + long time	2012-02-06 16:33:48
M, m	Month + day	February 16
t	Short time	16:33
T	Long time	16:33:48
Y, y	Year + month	February 2012

If these methods of formatting are insufficient, custom formats may be used. This format is defined by a sequence of specifiers that describe each field of the text being created. For example *„dddd dd MMMM yyyy HH:mm:ss.fff"* format creates a text *"Thursday 16 February 2012 16:33:48.240"*.

The following table describes the basic component specifiers of custom formats.

Type specifier	Description
d	Day of the month in one- or two-digit format
dd	Day of the month in two-digit format
ddd	Abbreviated weekday name

dddd	Full weekday name
f	Fractional part of the second, depending on the number of specifiers *f* a various time accuracy can be obtained.
H	Hour in one- or two-digit format
HH	Hour, two-digit (upper-case H - 24-hour format, lower-case h - 12-hour format)
m	Minute in one- or two-digit format
mm	Minute, two-digit
M	Month number in one- or two-digit format
MM	Month number, two-digit
MMM	Abbreviated name of month
MMMM	Full name of month
s	Second in one- or two-digit format
ss	Second, two-digit
yy	Last two digits of the year
yyyy	Number of year
Neutral characters	Copied directly into the output text
Characters inside the apostrophes ‘ ’	Copied directly into the output text

4 Object Appearance Animation

In many applications, there is a need to present the impression of movement of selected elements or dynamic signalling of certain states on the diagrams. The following section describes how to implement these functionality.

4.1. Animated Images

The simplest method is to use an animated GIF images. The only thing needed to be done is to use an animated image in the *Picture Name* property of the *Picture* class object.

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Image Name	NoFire
State Properties, state number 1		
	State Condition	=Variable()&1
	Picture Name	Fire

If the least significant bit of the main variable is set, then the state no. 1 will become active and as a result the *Fire* animated image will be displayed.

4.2. Animation of Pipes, Conveyors and Lines

Some of the object classes are initially prepared for animation of their movement. Motion parameterization consist of appropriate setting of properties corresponding to appearance of object and dynamic change of property modifying the move of the displayed shape. In general, the *Counter* internal variable is used for animation, this variable is automatically incremented on the basis of regular cycle.

4.2.1. Pipe

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Stripes Color	Blue
	Fill Offset	0
State Properties, state number 1		
	State Condition	=Variable()&1
	Fill Offset	=Variable(Counter)
State Properties, state number 2		
	State Condition	=Variable()&2
	Fill Offset	=-Variable(Counter)*2
State Properties, state number 3		
	State Condition	=Variable()&4
	Fill Offset	=-Variable(Counter)
State Properties, state number 4		
	State Condition	=Variable()&8
	Fill Offset	=-Variable(Counter)*2

Setting the *Stripes Color* property causes that the pipe will be displayed in the striped form. Changing the *Fill Offset* property causes shift of the stripes which allows to achieve the flow effect in the pipe. Depending on the bit set for the main variable, the *Fill Offset* property is changed in different ways: multiplication (or alternatively division) of the *Counter* variable value controls the speed of motion, while the change of the expression sign controls the motion direction.

4.2.2. Conveyor

Basic Properties		
	Main Variable	VA001
	Begin roller style	Drive
	End roller style	Drive
	Harrow Conveyor	True
State Properties, primary state		
	Conveyor offset	0
	Begin drive rotation	0
	End drive rotation	0
State Properties, state number 1		
	State Condition	=Variable()&1
	Conveyor offset	=Variable(Counter)
	Begin drive rotation	=Variable(Counter)
	End drive rotation	=Variable(Counter)

Setting the *Roller style …* and *Harrow conveyor* properties causes that the conveyor is displayed in a way which allows to animate the rotation of end rollers and the conveyor belt movement. If the least significant bit of the main variable is set, then the state no. 1 in which the motion animation is executed becomes active. Changing the *Conveyor offset* property gives a conveyor movement effect. Changing the *drive rotation …* property allows for the rotation animation of appropriate end roller.

4.2.3. Line

Basic Properties		
	Main Variable	Speed
	Dash style	Dash
State Properties, primary state		
	Stripes Color	Blue
	Fill Offset	=Variable()

Setting the *Dash style* and *Stripes Color* properties results in displaying the line in the striped form. *Fill Offset* is taken from the *Speed* variable value. Shift speed of strips depends directly on the rate of change of the variable - the controller programme can smoothly control the speed of motion. The absence of the *Speed* variable stops the motion.

4.3. Implementation of Blinking Effect by Changing Object Properties

The most universal method of animation is the cyclic change of selected property of object. This method can be applied to the object of any class and any property responsible for object appearance may be used. Typically, the change of the property is synchronised by value of the *IsBlinkOff* function. This function returns cyclically the *true* and *false value.* The standard transition period is 500 ms, but it can be changed in the application settings. Application of the *IsBlinkOff* function provides the same blinking rate for all object.

Some of the blinking function implementation variants are shown below.

Variant A

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Visible	True
State Properties, state number 1		
	State Condition	=Variable()==3
	Visible	=IsBlinkOff()

This scheme may be applied to the object of any class. If the main variable assumes value of 3, then the active state no. 1 by changing the *Visible* property will cyclically display and hide the entire object.

Variant B

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Color	Blue
	Text	#
State Properties, state number 1		
	State Condition	=IsAlarm("A_"+Attribute(Name))
	Color	=IsBlinkOff()?Red:Green

In the example B, the colour of the text is modified in the *Text* object. In the normal state, the value of main variable is displayed in a blue. If the alarm of *A_VA001 ID is active (*created on the basis of the name of the main variable), then the variable value is displayed alternately in a red and green. The conditional expression which uses the *IsBlinkOff* function, controls the colour choice.

Variant C

Basic Properties		
	Main Variable	VA001
State Properties, primary state		
	Picture Name	wizard
	Brightness	0
State Properties, state number 1		
	State Condition	=Variable()==3 && IsBlinkOff()
	Brightness	-0.5

In the example C, the blinking is performed by changing the brightness of the image in the *Picture* class object. In this example the *IsBlinkOff* function is placed in the state condition expression along with verification of main variable value. If the value of main variable is equal to 3, then the brightness of the picture is taken from the primary state or state no. 1, depending on the current value of the *IsBlinkOff* function.

5 Rules for Opening and Closing Synoptic Windows

One of the key elements of each application is the structure of the navigation system enabling switching between individual diagrams of the application. The *OpenWindow*, *OpenDiagram* and *CloseWindow* operator actions are used to open and close windows and diagrams. Selected windows may also be automatically open during the start-up of the application.

The operator actions can be associated with the events related to mouse handling, e.g. the *Button On* event of the *Button* class object or the *Left Button Click* events etc. for the other classes. The window and diagram switching actions may also be used in the definitions of the application menus and global keys.

5.1. Opening Synoptic Windows

5.1.1. Opening Window at the Application Start-up.

Windows selected can be launched at the start-up of the application. To do this:

a. Using the *Startup options* tab of the *Stations Settings* edit panel, select the windows which are to be opened at the application start-up. The start-up window setting is specific for the workstation. The window selection should be performed individually for each workstation or should be set on the area level.

b. In the settings of all panels constituting the start-up windows, set the *Default Diagram* property accordingly.

c. If the default diagrams have parameters, then their initial values should be specified as the default values. It is not possible to explicitly declare the parameters for the start-up window.

In a typical case, a single start-up window is opened on each available monitor at the Application start-up.

5.1.2. Opening Window and Diagram with *OpenWindow* Action.

The *OpenWindow* action is used to open a new predefined synoptic window or to swap the diagram in the selected panel of already opened window. It can also be used to activate previously opened window without changing the diagrams displayed. For detailed algorithm of the action functioning, see description of the action (Asix.Evo_Operator_Actions.CHM/PDF).

Example 1

OpenWindow(window1,null,null,null)

This action opens the *window1* window with default diagrams in the panels. If *window1* is already open, then it is activated without change of the displayed diagrams. The operation of this action is similar to running the start-up windows.

Example 2

OpenWindow(window1,info,machine,"mid=8")

This action is supposed to open the *window1* window with the *machine* diagram in the *Info* panel and the *mid* parameter of 8 or to swap the diagram in the previously opened *window1* window. If there are several opened *window1* windows, first it is checked if there is any window opened with the *machine* diagram in the *Info* panel and if the window have compatible parameters. If so, then the action will be limited only to activate the already opened window.

The action is used to swap the diagrams in the main windows of the application or to open the windows, the so-called stations, when there is a need to open only one station.

Example 3

OpenWindow(window1,info,machine,"mid=8",true)

The operation of the action is similar as in the previous example. The difference is that the previously opened window will be used (activated) only if there is a full compliance of the name of window, diagram and parameters.

The action used to open windows of so-called stations, when there is a need to open any number of the station (but each with a different diagram or parameters).

Example 4

OpenWindow(null,info,machine,"mid=8")

This action is supposed to exchange the diagram and parameters in the *info* panel in the context window of the operator action. The context window results from the location where the action being executed has been defined, for example, location of the *Button* object, which when pressed, executes the action.

5.1.3. Opening Diagram Without Use of Predefined Synoptic Window

It is also possible to open directly the synoptic diagram without prior definition of the window. This is achieved with the *OpenDiagram* action. As a result of its execution, temporary window displaying required diagram is internally created.

Example 1

OpenDiagram(machine,"mid=8",control_window,$Normal, &CursorLocation, &CursorLocation,"FixedSize", $None)

The action opens the *machine* diagram with the *mid parameter* equals to 8 in the automatically created window. The window size is consistent with the diagram size, and its location results from the cursor position. The window title, its appearance and behaviour are defined directly in the action contents. In this case the window title is the *control_window* text, the window is of fixed size and other appearance parameter are set to default. As with the *OpenWindow* action, there is a check performed if the diagram with the specified parameters is already open, before opening a new window.

The action functioning is similar to the *OpenWindow(window1,info,machine,"mid=8",true)* action and allows simultaneously opening of multiple stations.

The main difference between the *OpenWindow* and *OpenDiagram* actions depends on location where the appearance and set of available window system operations are defined. In the first case, it is specified in the parameters of the predefined window, in the second case, in the action contents. The definition in the action contents allows dynamically adapting the window appearance - in particular, the window title.

Example 2

OpenDiagram(machine,"varname="+Attribute(Name),"Variable "+Attribute(Name),$Normal, &CursorLocation, &CursorLocation,"FixedSize", $None)

In this case, the object main variable name (in the context of which the action is executed) is passed to the parameters of the diagram (*"varname="+Attribute(Name)* parameter). Simulatnously the variable name is added to the station title (*"Variable "+Attribute(Name)* parameter).

An additional advantage of the *OpenDiagram action is the ability to define the operating mode.* As shown previously in the *$Normal* mode, the window behaves like any other predefined application window. In the *$Dialog* mode, there is created a dialogue station - it is not possible to switch to the other window until the station is closed. In the *$Temp* mode, the window is automatically closed when selecting the other application window.

5.2. Controlling the Location and Size of Windows

The location of the new opened window is declared directly in the *OpenWindow* and *OpenDiagram* actions. The position can be declared in four ways:

a. The coordinates given directly

b. The coordinates resulting from the definition of window – *$Default* constant

c. The coordinates resulting from the current mouse position – *$CursorLocation* constant

d. The coordinates resulting from the location of window from which the opening action was executed – *$ActiveLocation* constant

In all variants of calling the opening action, if it comes to detection of situation that the appropriate window is already open, then the correction of the existing window position according to the specified parameters will follow.

Example 1

OpenDiagram(machine,"mid=8",control_window,$Normal, $ActiveLocation, $ActiveLocation,"FixedSize", $Closable)

The action opens the *machine* diagram in the location of the window from which the action was initiated. In addition the *$Closable* parameter will close the window. Such use of the action allows achieving the self-switching windows effect - even if the user has changed the position of the first window, the second window will be opened in the first window position. A similar effect can be achieved using the diagram switching in the panel of the permanently open window.

Both *$CursorLocation and $ActiveLocation* modes in a natural way open the window on the currently used screen. The *$Default* mode uses a predetermined monitor. The correct position can be dynamically calculated in the direct coordinates mode.

Example 2

OpenWindow(window1,info,machine,"mid=8",false,XOnScreen(0,0),YOnScreen(100,0),$None)

Using the *XOnScreen* and *YOnScreen* functions allows calculating the position according to selected monitor. The second parameter of both functions specifies the monitor number (0 indicates monitor indicated by the mouse cursor). The presented action opens the window on the monitor indicated by the mouse cursor at the left edge of the screen, from the line no. 100.

The size of the windows being opened follows from the size of the window predefined (*OpenWindow)* or size of the diagram (*OpenDiagram)*. However, the size can be changed using the *SetWindowSize* action after the window opening operation.

5.3. Closing Window and Diagram

The windows can be closed in several ways. These are:

a. Closing the window by standard system methods such as the closing button on the window frame.

b. The automatic closing as a result of diagram opening by the *OpenDiagram action* in the *$Temp mode.*

c. The use of a switching mode different than the *$None* mode in the *OpenDiagram* and *OpenWindow actions.*

d. Use of *CloseWindow action.*

Example 1

OpenWindow(window1,info,machine,"mid=8",false,$CursorLocation,$CursorLocation,$Closable)

The action opens the *window1* window and then close all windows without the closing lock, which are located on the screen, where the window has been opened (in this case the screen indicated by the mouse cursor). The *$ExceptCurrent* constant also can be used. Its functioning is similar, but the window from which the action was performed will not be closed.

The locked windows are those which have the *Closable* property set to *False*.

Example 2

CloseWindow(null,false)

The action closes the window in the context of which it was executed (the current window).

Example 3

CloseWindow("ST*",true,-1)

The action closes all windows, which names start with “ST”, on all monitors. The windows only on the selected monitor or the monitor indicated by the mouse cursor can also be closed (last parameter equals to 0).

5.4. Mutual Overlaying Control of Synoptic Windows

Another important element of window designing process is the control of their mutual overlaying. In a typical case, the active window is displayed in front of all other application windows. In some cases, it may be disadvantageous. For example, if there is a need to display a several windows of station type on a background of large start-up window, the standard operating rules are insufficient.

The mutual overlaying can be explicitly controlled using the *Background Window* and *Top Most* window properties. The *TopMost* attribute can also be used in the *OpenDiagram* action. The "on top" windows are always displayed in front of the normal windows, even if they are inactive. The "underneath" windows are always below all other windows.

The above remarks apply also to all other windows opened on a computer, which are not necessarily the Asix.Evo application windows. Therefore, great caution is advised when using the "on top" masks. They can overlay other important windows. Do not open large windows in this mode. The safer solution is to open the star-up windows of the application in the "underneath" mode rather than using the "on top" windows.

6 Organization of Control Operations

An important element of almost any application is to implement a mechanisms used to perform the operations that control the monitored object. Some objects have integrated some control mechanisms, for example *Text, Button, Bar*. The other objects can be extended with control function using the events handling and the *SetVariable* function. Examples of such parameterization are described in other sections of the documentation.

This section concerns the general organization rules for control operations of the application.

6.1. Immediate Control

The immediate control mode is the simplest mode of control operation execution. The appropriate value is sent to controlled variable immediately after selecting or entering a new setting, without any additional confirmation.

The parameterization of the immediate mode is restricted to determine the controlled variable name and selection of mode.

Basic Properties		
	Avtive	True
	Main Variable	Level
	Control Variable	LevelSet
	Immediate Control	True

The *Active property* set to *True* enables the interactive functions of the objects, especially the controlling ones. The *Immediate Control* property equals to *False* forces the operation in the immediate mode without confirmation.

In the presented example, the controlled variable name is given directly in the *Control Variable* property. This is correct parameterization but due to the general recommendation to control the object parameterization only via the main variable name and its parameters, the following structures are rather recommended.

Basic Properties		
	Active	True
	Main Variable	Level
	Control Variable	#Set
	Immediate Control	True

The controlled variable name is created as a result of combining the main variable name with the *Set* suffix. In particular, the use of only the # symbol would imply that the name of the controlled variable is the same as the name of the main variable.

Basic Properties		
	Active	True
	Main Variable	Level
	Control Variable	@ControlVariable
	Immediate Control	True

The controlled variable name is taken from the *ControlSetVariable* attribute of the variable database, from the row describing the definition of the *Level* variable. The *ControlSetVariable* attribute is not included in the standard attribute set - it must be added on the stage of database creation.

6.2. Delayed Control Operations (with Confirmation)

The delayed control mode is activated by setting the *Immediate Control* property to the *False* value. This mode is used in order to avoid coincidental initiation of operation or when the entire group of control operations is to be executed simultaneously.

The parameterization of the delayed control operations consists of two elements: indication of control operation execution pending and initiation of data transfer operation.

The indication of the pending state is based on the *HasWaitingControl* function. The object which waits for a send signal should modify its properties determining its appearance.

State Properties, primary state		
	Text	Unrecognised state
	Color	=HasWaitingControl()?Red:Black

In this example, the *Text* object change a colour in the waiting mode. Here, the conditional expression was used. This is the simplest form, but it can be only used if the object has no other states changing the colour property. It is also inconvenient if there is a need to change more than one property.

State Properties, state x		
	State Condition	=HasWaitingControl()
	Color	Cyan

It is easier to use the additional state with the *HasWaitngControl* condition. This state must be set as the last state of the object (or at least after all the states modifying the same properties).

Executing pending control operations is initiated with the *SendControls* action. Depending on the action parameter used, the button (or other object) executing the action may cause controls sending from all diagrams (*$All*), from current window (*$Window*), from current diagram (*$Diagram*) or from the objects of specified name.

Basic Properties		
	Active	True
	Cursor	Hand
	Button Kind	Rectangular 3d Butoon
	Switch Mode	False
State Properties, primary state		
	On	False
	Off Color	=HasWaitingControl(”*”)?LightCoral:LightGray
	Off Text	Send
Events		
	Button Off	^SendControls($Diagram)

The presented fragment of the *Button* object parameterization initiates the sending of controls from the diagram on which it was located. Additionally, the state indicating on pending controls is signalled. The call of the *HasWaitingControl* function with the parameter of the object name template allows checking if there are objects of compatible name pending for execution of control operation. The * parameter identifies all objects regardless of their name. This mechanism applies only to the objects located on the same diagram. It can not be used in case of the global initiation of the controls from the other diagrams.

The cancellation mechanism for pending controls is also available. The *CancelControls* action is used for this purpose, applied in a similar way to the *SendControls* action.

6.3. Control by Using *SetVariable* Operator Action

If the method of sending the settings not based on the built-in object mechanisms is applied, then it is necessary to use the *SetVariable* operator action. This function is available in two variants.

SetVariable(Alfa,1)

This variant sets the *Alfa* variable directly to the value of 1.

SetVariable(Alfa,1,0xf)

This variant sets/executes the bit modification of the *Alfa* variable. The four least significant bits are set to the value of 0001. Please note that the bit controls are highly inefficient in terms of execution time and bandwidth usage.

Although the direct entering of the variable name is correct, the context convention should be used whenever it is possible.

SetVariable(””,1)

SetVariable(”#”,1)

SetVariable(”#Set”,1)

The following examples use: name of object controlled variable, name of object main variable, name consisting of name of main variable and the *Set* suffix. It is also worth to define the main variable for the objects which are used only for control and use it in the *SetVariable* action.

6.4. Control Validity Check

In case of controls with delay, it is important to define how long the entered setting is valid. The button executing the *CancelControls* action may be added to the application. This will allow the operator to withdraw the control operation without closing the diagram. There is also an automatic method. The validity time for the controls can be specified in the *Control Timeout* field, in the *Stations Settings* panel, in the *Settings* tab. If it is set, the setting values will be invalid when the time specified expires.

There also could be a problem when the setting selected on the diagram of the station type remains valid but the station itself was overlaid by another window (i.e. the user cannot see the control signalling). The solution is to use the *SendControls* action with the *$AllVisible* parameter. This causes that the send signal applies only to objects on the windows that are not overlaid in any way by other window.

6.5. Controlling Permissions

The authority control for currently logged in user is very important in many applications when it goes to execute the control operations. The security system features the *Right to send control commands* privilege. The user must perform a role for which that privilege is active. Otherwise, every control operation executed by the user will fail.

From the perspective of the security it is enough to adequately parameterize the security system. However, it is better to take focus on the authority control before it comes to attempt to execute the control operations. The objects feature two properties that can be used for this purpose: *Active* and *Visible*. Both properties are of boolean type. The built-in object control functions as well as responses to the mouse click are blocked by setting the *Active* property to *false* (*NO*). The unwanted object may be completely hidden with the *Visible* property.

The proper procedure involves defining the role in which the *Control send right* permission is active and then adjoining to it all users who should have the ability to execute the controls. Then, that will be enough to use the expression using the *HasRole* function in the *Active* or *Visible* properties of the objects associated with the control operation.

Basic Properties		
	Active	=HasRole(SuperOperator)
	Main Variable	AlarmLimit
	Control Variable	#
State Properties, primary state		
	Text	#

The example above shows a part of the *Text* object parameterization. If the logged user does not perform the role of the *SuperOperator*, the object will be displayed the setting value but will not allow changing it.

The same mechanism may be used as well in a slightly different scheme of operation. Let's assume, that the standard operator has the control permissions. However, we want to make some control operations available only for some operators. To do this, the additional role is to be created (it is not required to feature any active privileges) and assigned to the appropriate users, then, using the *HasRole* function the critical objects should be blocked.

6.5.1. Double Confirmation of Control Operations

Occasionally, a different operating mode in which the operator controls the process but the execution of some operations requires additional confirmation of another person, is needed. However, we don't want to change the logged in user. The *ConfirmRole* operator action will be used for this purpose.

Basic Properties		
	Active	True
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	False
State Properties, primary state		
	On	False
	Off Text	Send
Events		
	Button Off	^Actions(ConfirmRole(Manager,$Always), SendControls($All))

The example above is the modification of the button which sends the signal of the delayed controls. In this case, the executed action is a combination of the *ConfirmRole* and *SendControls* actions. The use of the *ConfirmRole* action results in opening the login window. The check, whether the user performs the role specified in the call, will follow after the authorisation. If so, the further combination actions are executed in the context of the new user. After completion of all actions the user context will be restored to the state it had before the operation. Invalid authorisation or incorrect role causes an interruption of performing the combination actions. The mechanism of the action sets initiated by the *ActionSet* call may also be used instead of the *Actions* action.

Such method of use of the *SendControls* action allows the objects to be protected by built-in control mechanisms, for example *Text,* or *Bar*. The objects executing the controls (or other task) by using the immediate action execution are protected in a similar way.

Basic Properties		
	Active	True
	Switch Mode	False
State Properties, primary state		
	On	False
	Off Text	Enable
	On Text	
Events		
	Button Off	^Actions(ConfirmRole(Manager,$Always), SendControls($Diagram))

In this case the button object requests verification of the user role as the part of the *Button Off* event handling.

The *ConfirmRole* action may also be used in other ways. It can be used if the logged in operator is to be authorized again before executing the operation. This avoids the situation in which unauthorized person takes control over the computer when the operator has left position for a while.

7 Parameterization of Interactive Functions of Passive Objects

A large part of the available object classes do not feature any built-in mechanisms of interaction with the user. Their standard functionality is limited to displaying an information. However, it is possible to add the custom interactive functions to the objects of any type. The interactive functionality of the objects that are already provided with these kind of functions may also be extended (for example the *Buttons)*.

Adding interactive functions to an object is mainly based on handling events associated with the mouse operations (e.g. *right-click*) and pressing the keys. The supplementation is the set of functions which returns the state of the mouse and keyboard (for example *IsMouseOver* and *IsShiftPressed*), that when used in the definitions of the object properties allow an object appearance to be linked with the operator operations.

7.1. Navigation Through Text Links

The effect that will be achieved in this example is the use of the text links (similar to the links used in a browser) on diagrams. The object of the *Text* class displaying a static text will be used as a base object. Clicking on the text executes the action. At same time we want to visually present the link activity.

Basic Properties		
	Active	True
	Cursor	Hand
State Properties, primary state		
	Text	Fan scheme
State Properties, state number 1		
	State Condition	=IsMouseOver()
	Color	Blue
	Font Style	Underline
Events		
	Left Button Click	^OpenWindow(null,Info,Fan,null)

Setting the *Active* property to *True* enables full support of the interactive functions. Otherwise, only the mouse click event handling would be functional.

The *Cursor* property specifies the form of mouse cursor when it hovers over the area occupied by the object.

The state group controlled by the *=IsMouseOver()* conditional expression is activated when the cursor hovers over the object area. As a result, the text colour is changed into blue and the text is displayed as underlined.

The *OpenWindow* action in the handling of the *Left Button Click* event will open the *Fan* diagram in the *Info* panel of the current window.

7.2. Connecting Context Menu to Object

We want to obtain the commonly used mechanism of so-called context menu. Pressing the right mouse button when the cursor is in the area of the object should display the context menu.

The object of any class may be used as the base object (excluding the control objects, for example *Web Browser*, which take over the mouse handling process). We start with creating and defining the context menu. Using the menu in the object requires only using the *ShowMenu* action in the *Right Button Down* event handling.

Basic Properties		
	Main Variable	P1000
Events		
	Right Button Down	^ShowMenu(MenuCtx,"v="+Attribute(Name))

The *ShowMenu* action opens the menu of the *MenuCtx* name transferring into it the name of the main variable (as parameter) of the object in the context of which the menu has been opened.

7.3. Diagram Activity Zones

In another example we want to create a mechanism based on the fact that clicking on a selected area of the diagram will execute the appropriate operator action. Any set of objects may be located in the diagram active area. The active area is to be visible only when the mouse cursor hovers over this area. The following illustration shows the activity area over the large picture.

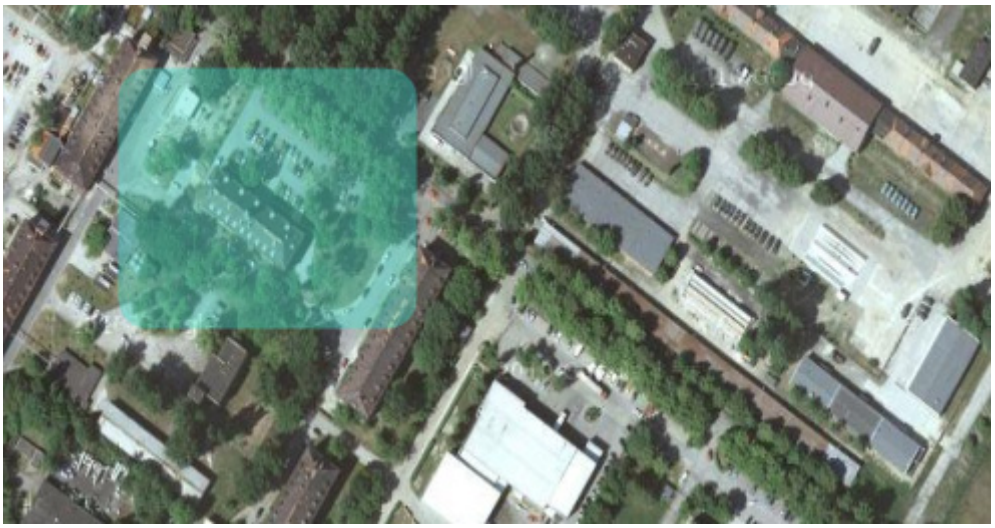

Fig. The activity area over the large picture.

The object of the *Shape* class will be used as the object on which the activity zone will be created.

Basic Properties		
	Active	True
	Cursor	Hand
	Shape Kind	Rectangle
	Rounding Radius	15
State Properties, primary state		
	Color	Transparent
State Properties, state number 1		
	State Condition	=IsMouseOver()
	Color	Turquoise
	Opacity	0.4
Events		
	Left button Click	^OpenWindow(null,Info,Askom,null)

Setting the *Active* property to *True* enables full support of the interactive functions. Otherwise, only the mouse click event handling would be functional.

The *Cursor* property specifies the form of mouse cursor when it hovers over the area occupied by the object.

In the basic group, the *Color* property is set to *Transparent*. This results in that the object in normal state remains invisible. The state group controlled by the *=IsMouseOver()* conditional expression is activated when the cursor hovers over the object area. As a result, the colour of the object changes and the shape becomes visible. At the same time the transparency level setting makes visible the objects located under the shape.

The *OpenWindow* action in the handling of the *Left Button Click* event will open the *Askom* diagram in the *Info* panel of the current window when the mouse button is clicked within the object area.

7.4. Self-Repetitive Operations

Most of the events associated with mouse handling is of one-time type. However, the events set includes one pair of the *Left Button Hold* and *Right Button Hold* events which are called repeatedly for the time when the appropriate button is pressed and the mouse cursor is in the object area.

The *Left Button Hold* event will be used to increment the variable value executed when the button is pressed. The object of the *Picture* class will be used.

Basic Properties		
	Active	True
	Curosr	UpArrow
	Main Variable	P1000
State Properties, primary state		
	Picture	UpHand
Events		
	Left Button Hold	^SetVariable("#",Variable()+1)

Setting the *Active* property to *True* enables full support of the interactive functions. The *Cursor* property specifies the form of mouse cursor when it hovers over the area occupied by the object.

The action being the part of the event handling changes a main variable value (abbreviated name "*#*" was used), by increasing its previous value, read out with the *Variable()* function, by one.

The period of the holding event is declared in a workstation settings.

An alternative to the method of the action iteration described herein is to use the object of the *Button* class which has the iteration mechanism built-in directly in its operation logic.

7.5. Keyboard Support

In addition to mouse related events handling, the definition of responses to the keyboard key presses is also possible. This example shows the object of the *Picture* class which increments the variable value every time the 'q' key is pressed, and decrements it every time the 'w' key is pressed.

Basic Properties		
	Active	True
	Main Variable	P1000
State Properties, primary state		
	Picture	Arrows
State Properties, state number 1		
	State Condition	=IsSelected()
	Opacity	0.5
Events		
	Key Press	^Actions(Perform(LastKeyPressed()==Q,SetVariable("#",Variable()+1),Nothing()), Perform(LastKeyPressed()==W,SetVariable("#",Variable()-1),Nothing()))

Setting the *Active* property to *True* enables full support of the interactive functions. It allows the user to select the object.

The state group controlled by the *=IsSelected()* conditional expression is activated when the cursor hovers over the object area. As result of this, the appearance of the picture is changed via modification of the *Colour* property.

The *Actions* action being the part of the event handling is a combination of the two *Perform* conditional actions. On the other hand, each of the *Perform* actions checks whether the correct key has been pressed and, provided that the check is successful, increments or decrements the main variable value.

An alternative methods of the keyboard handling are so-called global keys declared for the entire application and the keyboard shortcuts of the *Button* class objects. The main difference in the operation of the *Button* object shortcuts is that these shortcuts function even if the object is not selected.

8 Use of Transparency Effect

One of the mechanisms that can be used while developing the application is the transparency of objects and windows. It allows to achieve an interesting graphic effects and to efficiently use the screen surface by displaying different information, one on top of the other.

8.1. Window Transparency

The window transparency level is declared in the *Opacity* property. It assumes the value from 0 to 1, where 0 means a completely transparent window and value of 1 means a window without the transparency effect. Setting the value lower than 1 makes that the entire window (frame and displayed diagram with objects on it) is transparent. The method of defining diagram objects is arbitrary.

8.2. Object Transparency

Each object may have the individually controlled transparency level. It is set in the *Opacity* property, included in the group of the state properties. It assumes the value from 0 to 1, where 0 means a completely transparent object and value of 1 means an object without the transparency effect. The transparency effect relates to all graphic elements of the object.

Basic Properties		
	Main Variable	TranspValue
State Properties, primary state		
	Opacity	#

The transparency level is controlled by the value of the *TranspValue* variable. When combined with the mechanism of variable setting (e.g. *Bar* object with a slider enabled) allows the user to control smoothly the object transparency level.

8.3. Hiding Objects

Sometimes it is necessary to completely hide an object. This can be done by two methods.

Basic Properties		
	Main Variable	P1000
State Properties, primary state		
	Visible	False
	Text	Measurement error
State Properties, state number 1		
	State Condition	=VarIsNotGood()
	Visible	True

In this example, a *Text* object usually remains invisible. Only when the controlled variable has an incorrect status, the value of the attribute *Visible* is changed and the object displays a warning message concerning reading the variable.

Basic Properties		
	Main Variable	P1000
State Properties, primary state		
	Text	Measurement error
	Opacity	0
State Properties, state number 1		
	State Condition	=VarIsNotGood()
	Opacity	1

The visual effect of this example is identical to the previous one. Transparency is used to conceal an object. **The difference is that a transparent object (even completely) remains on the diagram and can respond to right mouse button clicks. The object with the *Visible* property set does not have this possibility.**

8.4. Use of the *Transparent* Colour

Another element which influences the visibility of the object is the use of the *Transparent* colour. Choosing this colour in the property associated with a component of an object usually means that this element will not be displayed.

State Properties, primary state		
	Visible	True
	Pointer Color	Red
	Background Color	Transparent
	Outline Color	Transparent
	Calibration Color	Transparent
	Font Color	Transparent
	Opacity	1

The parameter values of the *Gauge* object, which is shown above, causes only the pointer of the meter to be displayed. Putting such an object over another, fully visible meter, enables the two pointer meter.

9 Controlling the Behaviour of Objects

This chapter describes the methods of changing the appearance or behaviour of objects in response to the interactive activities of the operator. The methods described here enable customizing the appearance of the diagram when it is displayed in the application execution mode.

9.1. Using Virtual Variables

A common mechanism that may be used is to control the appearance or behaviour of objects by additional variables defined in the virtual channel (type *None*). The condition of controlled objects depends on such variables and the values of the variables are controlled through other objects. One of the features of this method is that the settings selected by the user are permanent - they remain unchanged after closing and returning to the diagram.

An important element of the virtual variables method is to correctly set the initial values. You can use the action *SetVariable* performed under the task scheduler which is executed when starting the application or when handling the event *Diagram Opened*.

Basic Properties		
	Main Variable	A1000
State Properties, primary state		
	Visible	=Variable(ShowNames)
	Text	@Name

The visibility state of the fragment of a text object shown above depends on the value of the virtual variable *ShowNames*. If the value is other than zero, the name of the variable is displayed on the diagram.

Basic Properties		
	Active	True
	Main Variable	ShowNames
	Control Variable	#
	Cursor	Hand
	Button Kind	Standard Windows Button
	Switch Mode	False
	Immediate Control	True
State Properties, primary state		
	On	False
	Off Value	1
	Off Text	Show names
State Properties, state 1		
	State Condition	=Variable()==0
	Off Value	0
	Off Text	Hide names

The above *Button* class object is used to switch the values of the *ShowNames* variable from 0 to 1 or vice versa, which, as a result, allows or blocks the function of showing objects which display variable names on the diagram.

9.2. Modifying Object Properties

Another method of controlling objects in operation is using the operator action *SetProperty*. This action allows changing the value of any property in any of the named objects. The change only applies to property values. After closing and opening the diagram, the property values will be re-calculated on the basis of their definitions stored in the diagram file.

Basic Properties		
	Element Name	Chart
	Show legend	False

The above fragment of the definition of the *Chart* class object shows the *Chart* object, which in the normal state does not display the legend.

Basic Properties		
	Active	True
	Cursor	Hand
	Button Kind	Standard Windows Button
	Switch Mode	False
	Immediate Control	True
State Properties, primary state		
	On	False
	Off Text	Show Legend
Events		
	Button Off	^SetProperty(Chart,ShowLegend,true)

Basic Properties		
	Active	True
	Cursor	Hand
	Buuton Kind	Standard Windows Button
	Switch mode	False
	Immediate Control	True
State Properties, primary state		
	On	False
	Off Text	Hide the legend
Events		
	Button Off	^SetProperty(Chart,ShowLegend,false)

The above pair of *Button* objects changes the value of the property *ShowLegend* for all objects with *Chart* names. As a result the buttons control the display of chart legend by the chart objects placed on the diagram.

10 Application of Button Class Objects

One of the most frequently used objects in the application is an object of the *Button* class. The scope of its application is very wide, from organizing the application window switching schema up to a variety of process variable controls.

10.1. Single-Position Button

The simplest and probably the most common case is to use the single-position mode. It then works in a manner similar to standard Windows buttons. The button stays pressed. When the mouse button is pressed it changes its state to pressed and the moment the mouse button is released it automatically returns to the depressed state. The action executing the function of the button is executed during this process.

This mode is used for simple one-off operations such as switching windows or control operations without additional conditions.

10.1.1. Single-Position Button Executing Operator Actions

Basic Properties		
	Active	True
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Flat	=!IsMouseOver()
	Switch Mode	False
	Shortcut	Alt+Q
State Properties, primary state		
	On	False
	Off Color	LightGreen
	Off Text	Scheme
	On Text	
Events		
	Button Off	^OpenWindow(null,Info,Schemat,null)

The *Cursor* property causes the mouse cursor to change when hovering over the button area to indicate the readiness to perform the operation. The *Button Kind* property determines the shape of a button. At the same time the expression used in the *Flat* property causes the button to be displayed in flat form until the mouse hovers over an object. Only after the hovering over an object is it displayed in full three-dimensional form. This effect of course does not have to be used and one may set the value of the property *False* to *Flat*.

Setting the *Switch Mode* to *False* causes the button to work in single-position mode. Indicating the keyboard short cut causes that apart from using the mouse the user can execute the operation with the keyboard.

The *On* property determines how to display the button. In the case of the single-position mode it should be set to *False*.

Most of the state properties determine the attributes of button display. These usually occur in pairs, one for pressed and one for depressed buttons. The provided example the *Off Text* specifies the description of the button. No such text for the *On Text* property causes that regardless of the state of the button the *Off Text* will always be displayed.

Using the *Off Color* property causes that when a button is momentarily pressed, its background colour changes to further indicate the execution of the operation. Setting the colour and any other attributes for the inclusion is of course not necessary.

The operator action specified in the *Off Button* event handling replaces the diagram in the *Info* panel of the current window. In the case of *Button* objects it is necessary to use the events dedicated for this class of objects, i.e. *Button On* and *Button-Off.* Although in some cases similar results can be achieved by using universal mouse events (*Left Button Click*, etc.), it is better to use dedicated events, which account for the state of a button.

10.1.2. Single-Position Button Executing Control Actions

If it is necessary to perform an operation associated with sending control values to process variables, parameter values based on the execution of the *SetVariable* operator action can be used. Generally it is however easier to use the functionality built into the button object, which is used to perform direct controls.

Basic Properties		
	Active	True
	Control Variable	Command
	Cursor	Hand
	Buton Kind	Rectangular 3d Button
	Switch Mode	False
State Properties, primary state		
	On	False
	Off Value	0x8
	Off Text	Enable
	On Text	

Defining the parameter *Off Value* means that during the button on/off cycle, a hexadecimal value of 0x8 (16) is sent to the controlled variable *Command*.

Basic Properties		
	Active	True
	Main Variable	Level
	Control Variable	#
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	False
State Properties, primary state		
	On	False
	On Value	=Variable()+1
	Off Text	Increase
	On Text	

In another example the *Off Value* is created dynamically. It is equal to the current value of the main variable plus 1. Pressing and releasing the button will increase the value of the main variable by 1. The name of the controlled variable is defined by # - this means using the name of the main variable.

10.2. Single-Position Button with Repeat Function

The *Button* object also enables working in the auto operation repeat mode or operator action repeat mode.

Basic Properties		
	Active	True
	Main Variable	Level
	Control Variable	#
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	False
	Repeatable	True
State Properties, primary state		
	On	False
	On Value	=Variable()+1
	End repeating Value	0
	Off Text	Increase
	On Text	

The repeat mode is enabled by setting the *Repeatable* property. While holding down the button the value defined in *On Value* will be repeatedly sent to the controlled variable (which is identical with the main variable). As a result the value of the variable is incremented continuously. Upon releasing the button the value defined in the property *End Repeating Value* will be sent. A similar pattern applies to actions based on event handling. The *Button On* event is executed cyclically and the sequence is finished with the *End repeating* event.

The time of repetition of the control or action is defined in the positions parameters panel in the in the *Settings* tab.

10.3. Single-Position Button with Hold

In the case of important controls it may be important to prevent the button from being pressed accidentally. If the mechanism of delayed controls described further should not be used, pressing the button may be combined with the requirement of pressing a key on the keyboard simultaneously.

Basic Properties		
	Active	=IsAltPressed()
	Main Variable	Level
	Control Variable	#
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	False
State Properties, primary state		
	On	False
	Off Value	=Variable()+1
	Off Color	=IsAltPressed()?LightGreen:LightGray
	Off Text	Increase
	On Text	

This is a modification of the earlier example of a single-position simple button which increments the variable value. The *Active* property is set to *true* (YES) only if the user presses the *Alt* key. Only then the object enables pressing the button. Additionally *Off Color* is also modified according to the state of the *Alt* key in order to signal an activity of the button.

10.4. Two-Position Button

In the next example we shall build a two-position in which the pressed condition will be determined by the value of the process variable. When clicked the button changes to the opposite state, while performing the appropriate control operation.

Basic Properties		
	Active	True
	Main Variable	Flag1
	Control Variable	#
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	True
	Immediate Control	True
State Properties, primary state		
	On	=Variable()
	Off Value	0
	On Value	1
	Off Text	Off
	On Text	On

All functionality is defined in a single state. Setting the property *Switch Mode* to *True* causes the button to move in one direction and perform a single control.

If the variable *Flag1* is equal to 0, the button is depressed and the description *Off* is displayed. After pressing the mouse button, the button remains pressed and the description is changed to *On*. The moment the mouse button is released an *On Value* is sent to the *Flag 1* variable. As a result the button remains pressed (turned on) because the value of the main variable Flag1 changes to a non-zero value. After the operation is executed the button may temporarily return temporarily to its original state *Off* until the new value of the main variable will be read back.

10.5. Two-Position Button with Delayed Control

A common case is that a two-position button has to be used, where the control operation is not performed immediately but requires a separate confirmation. The functional diagram is very similar to that of an ordinary two-position button.

Basic Properties		
	Active	True
	Main Variable	Flag1
	Control Variable	#Control
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	True
	Immediate Control	False
State Properties, primary state		
	On	=Variable()
	Off Value	1
	On Value	2
	On Color	=HasWaitingControl()?LightGreen:LightGray
	Off Color	=HasWaitingControl()?LightCoral:LightGray
	Off Text	Off
	On Text	On

Immediate Control is set to *False*, which causes that clicking on the button only changes its state to the opposite. The Button is in this state until the control of the *SendControls* action is approved or cancelled by the *CancelControls* action. It is important to visually show that the object awaits confirmation of the operation. In the example the button changes colour. The conditional expression uses the function *HasWatingControl*, which takes the value *true* if the object is in waiting.

The example shows also that the control (command) is sent to a different variable than the underlying main variable which determines the state of a button. The name of the controlled variable is created by suffixation and in this example it is *Flag1Control*.

As in the previous example, after the operation is executed it may cause the button to momentarily return to the initial state for the time needed to execute the control, execute the command by the driver and read back the main variable.

10.6. Switch

This example will show the configuration of the *Button* object in a manner in which its function is in accordance with the function of the object *Switch* in the classic version of the Axis system. The button remains depressed and its description defines the operational current mode (stopped/started). After completing the control the button remains pressed until the operation is approved. At the same time the description of the button changes and shows which operation will be executed (start/stop).

Basic Properties		
	Active	True
	Main Variable	MStatus
	Control Variable	#Control
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	False
	Immediate Control	False
State Properties, primary state		
	On	=HasWaitingControl()
	Off Value	1
	On Color	LightGreen
	Off Text	Started
	On Text	Stop
State Properties, state 1		
	State Condition	=Variable()&1
	Off Value	2
	Off Text	Stopped
	On Text	Start

The object uses two variables. The main variable is used to determine the status of the device. Depending on its value the descriptions of the button and the values of the control command change. The object state was used. The control variable with the name composed of the combined main variable name and the *Control* suffix is used to send control commands (2 is the start command and 1 is the stop command for a device).

In the action diagram above it is important to use a *HasWaitingControl* expression in the *On* property. When the button is pressed an object enters a state where it has a valid control value. It remain in this state until the execution of the control (with the *SendControls* action) or its cancellation. During this time the value of the expression is *true* and will keep the button pressed. The condition of an important value is additionally indicated by the colour specified in the *On Color* property.

10.7. Bitwise Control

Sometimes the controls exercised may only change single bits of the controlled variable. In such cases it is necessary to use button objects in the bit control mode. The following example demonstrates a modified two-position button case.

Basic Properties		
	Active	True
	Main Variable	Status
	Bitwise control	True
	Control Variable	#
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	True
	Immediate Control	True
State Properties, primary state		
	On	=Variable()&1
	Off Value	1
	On Value	1
	Off Text	Off
	On Text	On

Setting the *Bitwise Control* property to *True* causes a change in the way of an executing the operation of sending a value to a controlled variable. The operation is performed in two steps. First the current value of the controlled variable is actively read, then it is calculated and a new variable value is sent. Keep in mind that this way of working is much less effective (in terms of communication with the driver) than regular controls.

The button state is controlled by the value of the first bit of the main variable. The expression *Variable ()&1* delivers a logical product of the variable and the number 1. As a result the button is pressed when the first bit is set. In the transition from the off (depressed) condition to the on condition these bits, the value of which in the *On Value* property is equal to 1 (in this case, only the least significant bit), are set depending on the variable. In the transition from the on (pressed) condition to the off condition these bits in the controlled variable are set at 0, the value of which in the *Off Value* property is equal to 1 (in this case the least significant bit of the variable will be set to 0).

In more complicated cases of bit controls use the operator action *SetVariable* to handle *Button On* and *Button Off* events. For example the action *SetVariable ("", 3, 0xf)* will change the four least significant bits of the controlled variable. The two least significant bits are set to 1 and the remaining are set to 0.

10.8. Grouping Buttons

Sometimes it is necessary to use a group of buttons to perform contradictory operations. The following illustration shows the buttons that switch the device to move forward or backward.

These buttons should work after a confirmation. As a result the user can enter the two buttons into a state of an important control value. Performing the *SendControls* action would cause the execution of two control operations - the final result would depend on the order of defining objects.

However it is also possible to enable pushing only one button at a time. Groups of controls are used for this purpose.

Button 1

Basic Properties		
	Active	True
	Main Variable	Move
	Control Variable	#
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	False
	Immediate Control	False
	Control Group	G1
State Properties, primary state		
	On	=HasWaitingControl()
	On Value	1
	On Color	LightGreen
	Off Text	<<<

Button 2

Basic Properties		
	Active	True
	Main Variable	Move
	Control Variable	#
	Cursor	Hand
	Button Kind	Rectangular 3d Button
	Switch Mode	False
	Immediate Control	False

	Control Group	G1
State Properties, primary state		
	On	=HasWaitingControl()
	On Value	2
	On Color	LightGreen
	Off Text	=”>>>”

In the presented set of parameter values the definition of the property *Control Group* is important. Property values are not important, it is only important for these to be identical for both buttons. At a given moment only one of the two buttons will remain pressed (that is the important control value checked by the *HasWaitingControl* function).

11 Control Operations in Text Class Objects

Text class objects offer two methods to perform the settings. In the first mode, which is used primarily in objects that show the numerical values of variables, the user enters the new variable value directly. In the second mode, which is used in objects providing text descriptions of bit states, the control consists in selecting a new setting from the list of proposed values.

11.1. Controlling the Numerical Value Entered by the User

In this mode it is important to clearly distinguish the different states of the object: the state showing the current value, the state of entering a new value and the state of awaiting the execution of a control.

Basic Properties		
	Active	True
	Main Variable	Level
	Control Variable	#
	Minimum Control Value	@SteeringRangeFrom
	Maximum Control Value	@SteeringRangeTo
	State Selection Mode	False
	Edit On Selection	False
	Initial Edit Value	Empty
	Immediate Control	False
State Properties, primary state		
	Text	#
	Color	Black
	Edit Color	Blue
State Properties, state 1		
	State Condition	=VarIsNotGood()
	Text	=Variable() + ”?”
	Color	Crimson
State Properties, state 2		
	State Condition	=HasWaitingControl()
	Color	Cyan
	Font Style	Italic

The *Active* property is set to *True*, which enables the control functions of the object. The name of the controlled variable and the main variable are identical. The limits for the control variable are also set - the limits are taken directly from the corresponding attributes of the main variable of the object. The property *State Selection Mode* equal to *False* means that the new value will be entered directly by the User. The *Edit On Selection* property defines how to start editing the set point. If *False*, if an object is selected, the *Enter* key needs to be pressed or a double-click of the mouse is necessary. For the *True* value editing will begin immediately after selecting an object. The property *Initial Edit Value* specifies the text displayed in the object after the start of editing. This may be an empty text, the text last entered or the current text to be displayed.

The property *Immediate Control* set to *False* means that after editing (with the *Enter* key) the new value of the set point will only be sent upon execution of the control action *SendControls*. The value *True* would mean sending immediately after editing is complete.

The basic condition properties determine the colour used to display variable values in the normal condition and the colour used when the user edits a new set point.

The property group number 1 describes the text display method when the status of the main variable is incorrect. The colour of the text is changed and the character '?' is added after the value of the variable. This condition is not necessary here, it only serves to illustrate the parameter values of the control in the multi-condition object.

Condition number 2 specifies how to display the new set point after the user finished editing and the operation execution order (action *SendControls*) have not yet been completed. The colour and font style are changed. The state controlling the waiting period for sending the value is usually placed at the end of the list of states, this way the settings take precedence. Reversing the order in our example would result in a situation, where, despite waiting for the control the text colour defined in the group controlling the variable status could be used.

11.2. Controls From The Selection List

An alternative method of performing controls through the *Text* object is the use of state selection mode. This mode is usually used in objects that display textual descriptions of the states. Control involves selecting the correct item from a list of possible operations.

Fig. Text Object - The Selection List.

The contents of the list of controls result from the definition of the object states.

Basic Properties		
	Active	True
	Main Variable	LineState
	Control Variable	#Control
	Cursor	Hand
	State Selection Mode	True
	Immediate Control	True
State Properties, primary state		
	Text	Unrecognised state
	Color	Black
	State Selectable	False
State Properties, state 1		
	State Condition	=Variable()&1
	Text	Manual mode
	State Value	8
	State BitMask	8
	State Selectable	=!(Variable()&1)
State Properties, state 2		
	State Condition	=Variable()&2
	Text	Emergency mode
	State Value	4
	State BitMask	
	State Selectable	=!(Variable()&2)
State Properties, state 3		
	State Condition	=Variable()&4
	Text	Automatic mode
	State Value	0
	State BitMask	8
	State Selectable	=!(Variable()&4)

The *Active* property is set to *True*, which enables the control functions of the object. Controlled variable name is created based on the main variable name by adding the *Control* suffix. The property *State Selection Mode* chooses operation in selection mode and *Immediate Control* set to *True* causes the execution of the control to be effected immediately after selecting the state. Delayed control is of course possible. In this case it would be necessary to add waiting state signalling, e.g. by using the expression *HasWaitingControl()?Red: Black* in the definition of the property *Colour* or adding another state.

The basic state of an object defines the appearance of the object in case of an abnormal valueof the monitored variable. The next three states are used to decode the variable value to a text description, the condition of the state is a test of the settings for the selected bit. The property *Selectable State* determines whether the description of the state is to appear in the control operation selection list. The terms used cause the description to appear in the list if it is not the present state (the expressions are negations of the conditions of the state).

It is also possible to define states of the objects used only for control. To this end the *State Condition* should be set to *False* and the *State Selectable* should be set to *True*.

The properties *State Value* and *State BitMask* determine how are the controls executed. An undefined mask means the direct control of the state value. A defined mask means bit controls. In the example shown, the state no. 1 is the change of the fourth bit of the controlled variable to 1, the state no. 2 means sending the value of 4, and the state no. 3 is a fourth bit change to the value of 0.

An appearance of the state list (colours and font) is parametrised in the *Stations Settings* panel in the *Settings* tab of the *States Menu Settings* frame.

A similar functionality may be achieved by connecting a context menu to an object.

See also: *7.2. Connecting Context Menu to Object*

12 Using Sliders in Bar Class Objects

The main task of the *Bar* class objects is displaying the process variables values in the form of bars (horizontal or vertical). They may also be used as static objects for displaying a numerical scale on a diagram.

This section deals with the rules of using sliders built in the *Bar* class object.

12.1. Using Slider to Control Set Point Values

In this application, the bar is used to show the current setting value, and simultaneously, it allows changing the value using a slider.

Basic Properties		
	Active	True
	Main Variable	SliderSet
	Control Variable	#
	Minimum Control Value	@SteeringRangeFrom
	Maximum Control Value	@SteeringRangeTo
	Pointer Style	=IsMouseOver()?Holder:None
	Use Limits	False
	Bar Value	#
	Pointer Value	#
	Minimum Value	@SteeringRangeFrom
	Maximum Value	@SteeringRangeTo
	Immediate Control	True
State Properties, primary state		
	Pointer Color	Transparent

Since the bar size and slider position (pointer) depend on the same variable, the *Control Variable* is specified with the # short notation. The range of the slider variation depends on the range specified in the pair of properties *Minimum Control Value* and *Maximum Control Value*. The bar range depends on the pair of properties *Minimum Value* and *Maximum Value*. Since in our application the slider and bar show an identical value, both the ranges were identically set and are based on the settings of the *SliderSet* variable control range specified in the variable definition database. The current bar value is specified by the *Bar Value* property - using the # notation means that the main variable value was loaded. The slider position (pointer) in the *Pointer Value* property is specified in a similar way.

Using the *IsMouseOver()?Holder:None* conditional expression in the *Pointer Style* property causes that the slider in the normal state is not displayed, whereas when the cursor hovers over the object, the slider is displayed and allows the set point to be changed.

Using the *Transparent* colour in the *Pointer Color* property does not hide the slider, but only displays it in the semi-transparent form.

12.2. Using the Slider to Control Set Value With a New Set Point Preview

The following example shows how to connect on the single *Bar* class object the presentation of a measurement current value and measurement set value with a setting change possibility. Simultaneously, while changing the set value (slider dragging), its value in the form of text will be shown on the separate *Text* class object.

Basic Properties		
	Active	True
	Main Variable	Level
	Control Variable	#Set
	Preview Variable	#View
	Minimum Control Value	=Attribute("#Set",SteeringRangeFrom)
	Maximum Control Value	=Attribute("#Set",SteeringRangeTo)
	Pointer Style	=IsMouseOver()?Holder:Triangles
	Use Limits	True
	Bar Value	#
	Pointer Value	#Set
	Minimum Value	@DisplayRangeFrom
	Maximum Value	@DisplayRangeTo
	Value LL	@LimitLoLo
	Value L	@LimitLo
	Value H	@LimitHi
	Value HH	@LimitHiHi
	Immediate Control	True

The bar height will depend on the *Level* variable value, and the slider (pointer) position will depend on the *Level Set* variable specified in the suffix notation. In the *Minimum Control Value* and *Maximum Control Value* properties, the slider value variation range needs to be specified in order to determine the slider position. The used expressions refer to the *LevelSet* variable attributes stored in the variable definition database. The bar height depends on the range specified in the *Minimum Value* and *Maximum Value* properties. In our example, this range is defined by the main variable displaying range attributes. Both the bar and slider variation ranges must be consistent, e.g. the bar size may depend on an absolute value and the set value (slider position) may be specified as percentage value.

Settings of the *Use Limits*, *Value LL, Value L, Value H* and *Value HH* properties change the bar colour when the individual warning limits are exceeded. The limit values are loaded from the *Level* main variable attributes.

Using the *IsMouseOver()?Holder:Triangles* conditional expression in the *Pointer Style* property causes that in a normal state the slider position is displayed in the form of two triangles on the background of the bar strip but when the cursor hovers over the object, the slider changes the shape indicating readiness to perform a new set point.

The *Preview Variable # View* property definition causes that when dragging the slider, the setting value corresponding to its temporary position will be set in the *LevelView* variable. Such a variable should be defined in a virtual channel, and its value may be displayed, e.g. by the *Text* class object.

Basic Properties		
	Main Variable	SliderView
State Properties, primary state		
	Visible	=Variable()>-1e200
	Text	#

If the slider is not being dragged, the slider preview variable is set to the Double type minimum value. It is used in the *Visible* property definition of the *Text* object. The object is visible on a diagram, only when the slider new position is being selected.

13 Motion Animation and Object Resizing

The section presents the dynamic methods of changing an object position and size on a diagram. The specificity of this problem lies in the fact, that it is impossible to directly change the object properties responsible for its location and size. The *X, Y, Width, Height* properties are always entered directly.

To change the object position or size in the application run mode, the *Animation* event and the *SetPosition*, *SetSize* and *SetBounds* operator actions may be used. In the *Animation* event handling, one of the actions with appropriately calculated parameters of the object location and/or size should be used, as required.

13.1. Changing Position

In the following example, the position of the *Picture* type object will be changed. The position on the X-axis will be controlled by the *CarPosX* process variable of value ranging from 0 to 100 specifies the image shift in pixels measured from the start position.

Basic Properties		
	Main Variable	CarPosX
State Properties, primary state		
	Picture Name	Car
Events		
	Animation	^SetPosition(RelToAbsX(100+Variable(),null)

The start position is set to 100 The *CarPosX* variable value is added to the start position value. The position calculated in pixels in this method, must be then converted into an absolute value using the *RelToAbsX* function. All the function parameters of the position and size change are transmitted in the absolute values independent of the diagram size. The absolute value of 1 000 000 always corresponds to the right edge of diagram. The *null* value in the second parameter of the *SetPosition* action, indicates that the Y coordinate of an object is not a subject of change.

The object size change can be changed in a similar way, but in this case, the *SetSize* or *SetBounds* action should be used.

The action shown in the example above has one fundamental disadvantage - it only runs properly if the diagram has a fixed size. In the case of a scaled diagram, the position calculations must be based on the absolute coordinates.

Basic Properties		
	Main Variable	CarPosX
State Properties, primary state		
	Picture Name	Car
Events		
	Animation	^SetPosition(500000+100000*(Variable()/100.0), null)

In the example above, the image base position is the diagram centre (absolute value of 500 000). With the *CarPosX* variable value of 100, the extreme right position is located at the 60% of diagram width.

13.2. Changing Position Within Area Defined by Another Object

The above problem of the coordinates conversion can be avoided using the object relative positioning method. The motion range is determined by a special object (it may be invisible) within which, a specific object is positioned.

State Properties, primary state		
	Picture Name	Car
Events		
	Animation	^SetPosition(LocalProperty(area,X) + (LocalProperty(area,Width) * (Variable(CarPosX)/100.0)), LocalProperty(area,Y) + (LocalProperty(area,Height) * (Variable(CarPosY)/100.0)))

This example assumes that the object named *area* is located on the diagram. It can be e.g. the *Shape* class rectangular object. To convert the image position, the coordinates of the *area* object are used, read out with the *LocalProperty* function. The variables controlling the position on both *CarPosY* and *CarPosX* axes are within the range from 0 to 100.

The advantage of this solution is that the change of motion range, only requires the *area* object position change; changing the positioning expression in the object image is not required.

13.3. Positioning Groups and Templates

The methods of handling the object groups and embedded templates are identical as in the case of single objects. The *Animation* event for groups and templates allows controlling their position and size.

14 Alarm State Indication and Handling

The *Active Alarms Viewer* and *Historical Alarms Viewer* objects are the basic mechanisms for viewing and handling alarms. They show the state of alarms in a tabular form and are provided with a toolbar.

See also:

- *Asix.Evo_Getting_Started.PDF/CHM, "4. Application with Alarm Handling";*

- *Asix.Evo_Objects.PDf/CHM, " Active Alarms Viewer Object", " Historical Alarms Viewer Object";*

- *Asix.Evo_System_of_Alarms.PDF/CHM*.

This section, however, deals with the methods of handling and signalling alarms directly on the synoptic diagrams, using universal objects.

14.1. Single Alarm State Monitoring and Handling

In the following example, a typical mechanism for alarm signalisation using the *Picture* class object will be shown. The object state will be controlled using the *IsAlarm*, *IsAlarmUnaccepted* and *IsAlarmExcluded* functions. An additional object will be used to acknowledge an alarm with the *AcceptAlarm* operator action. The other operator actions related to the alarm handling include: *ExcludeAlarm* and *IncludeAlarm*.

State Properties, primary state		
	Visible	False
	Picture Name	
State Properties, state 1		
	State Condition	=IsAlarm(A1001)
	Visible	=!IsAlarmExcluded(A1001)
	Picture Name	AlrRed
State Properties, state 2		
	State Condition	=IsAlarmUnaccepted(A1001)
	Visible	=!IsAlarmExcluded(A1001)
	Picture Name	=IsBlinkOff()?AlrYellow:null
Events		
	Right Button Click	^AcceptAlarm(null,A1001)

In the basic state, a picture is not visible. The state no. 1 is activated when an alarm of the *A1001* ID changes its state to Active (initiated). The state no. 2 is activated when the alarm is not acknowledged. Because the state with a higher number takes precedence, in the period in which the alarm remains unacknowledged (terminated or not), the *AlrYellow* image is displayed. Additionally, the use of the conditional expression with the *IsBlinkoff* function causes the image blinking. If the alarm is Active but acknowledged, the *AlrRed* image is displayed. Please note that the terminated alarms state monitoring depends on the used active alarm log operation mode. If a terminated alarm is removed from the log (after the terminated alarm storage time elapses), the *IsAlarmUnaccepted* function will return the *false* value, even if the alarm is unacknowledged.

Since the *IsAlarm* and *IsAlarmUnaccepted* functions return an alarm actual state, regardless of whether it was excluded by the Operator from handling or not, it is necessary to use the *IsAlarmExcluded* function in the *Visible* property. If exclusions were not used in the application, using both the states of the *True* value in the *Visible* property would be sufficient.

The *AcceptAlarm* action used in the *Right Button Click* event handling, acknowledges the *A1001* alarm.

The example assumes that only one alarm domain is defined in the application. Otherwise, the *IsAlarm* function variants and similar, should be used along with an explicit specification of the domain name.

The above example, although correct, has one major disadvantage. An attempt to use an object to control another alarm, requires modification of five properties. The recommended procedure is to convert an object into a template with the parameter that specify the alarm ID and the *Parameter* function call wherever the ID was used. The example below may be also followed.

Basic Properties		
	Main Variable	A1001
State Properties, primary state		
	Visible	False
	Picture Name	
State Properties, state 1		
	State Condition	=IsAlarm(LocalProperty(MainVar))
	Visible	=!IsAlarmExcluded(LocalProperty(MainVar))
	Picture Name	AlrRed
State Properties, state 2		
	State Condition	=IsAlarmUnaccepted(LocalProperty(MainVar))
	Visible	=!IsAlarmExcluded(LocalProperty(MainVar))
	Picture Name	=IsBlinkOff()?AlrYellow:null
Events		
	Right Button Click	^AcceptAlarm(null, LocalProperty(MainVar))

This variant works exactly the same way as the previous one. The alarm ID, however, was only specified in the *Main Variable* property. The fact that such a variable does not exist is of no significance. The context of use of the property is important. In this case, the property value is retrieved using the *LocalPropert* function and it is used as an alarm ID.

14.2. Indicating States of Alarm Group

When using the *AlarmsGroupState* function, it is possible to control alarm group state in the selected group, or even in the entire domain. In this example, the semi-transparent *Shape* class object will be used, to indicate alarms in a section of the controlled system.

Basic Properties		
	Layer	10
State Properties, primary state		
	Visible	False
	Opacity	0.1
State Properties, state 1		
	State Condition	= AlarmsGroupState (Block1, "Zone = S1")&1
	Visible	True
	Color	Cyan
	Outline Color	Blue
	Opacity	0.5
State Properties, state 2		
	State Condition	= AlarmsGroupState (Block1, "Zone = S1")&4
	Visible	True
	Color	Coral
	Outline Color	Red
	Opacity	0.5

Create the *Shape* class object in such a location, so it overlays the relevant section of a diagram. For this purpose, changing the *Layer* property may be necessary. In the basic state, the object is to remain invisible. Because the *Visible* property value is ignored when editing the diagram, the *Opacity* property should be changed as well - it will facilitate the diagram editing, the objects underneath the *Shape* objects will be visible. Then, add the two states used for indicating the alarms. Each of them has the individually set colour attributes, the *Visible* property set to *True*, and the appropriate level of transparency set. In the state conditions, the *AlarmsGroupState* function was used. This function controls the alarm states of the *Block1* domain for which the *Zone* grouping attribute is equal to *S1*. The value returned via the function, can be interpreted in different ways. In our case, a bitwise AND operator with the value of 1 (the least significant bit test) checks whether a group contains an unacknowledged alarm. Test with the value of 4 (number 2 bit test) checks whether a group contains at least one unacknowledged Active alarm. Due to the state definitions sequence, if a group contains the unacknowledged Active alarm, a "glass" in the *Coral* colour will be displayed. If the group does not contain such an alarm, but contains at least one terminated and acknowledged alarm, the "glass" will be in the Cyan colour.

For the alarm group state indication, the *AlarmsCount* function which returns the number of alarms in the selected group including the activity and acknowledgement states can be used as well

15 Controlling Chart Class Objects

The *Chart* class objects have their own built-in interfaces for the interoperability with the user. The alternative is also the *Chart controller* object which allows controlling the chart functions. In some operation scenarios, however, adding custom controlling mechanisms may be necessary.

15.1. Controlling Time Range of Chart

In the standard application, the chart objects upon opening a diagram show a defined period of time in the object, calculated from the current moment. The user can further select any other time range, provided that the function was not locked. However, additional mechanisms for selecting the time range by the user may be added, alternatively, the time range may be forced based on the data retrieved from various sources.

15.1.1. Modifying Chart Object Properties

The time range shown on the object depends on the three properties. The time range length is specified in the *Time range [min]* property. The range end (the right section) is defined by the *Manual base time* property. However, it is only of significance, if the *Refresh mode* property is additionally set to *Manual*, otherwise the chart end point will be defined by the current moment.

The chart time ranges can be changed using the *SetProperty* operator action.

Basic Properties		
	Element Name	*Chart*
	Manual base time	
	Time range [min]	*10*
	Refresh mode	*Manual*

The above fragment of the *Chart* class object definition shows the object named *Chart* which upon a diagram opening displays a chart for the last 10 minutes. It is important to name the object and select the refresh manual mode.

The following *Button* class objects are used for switching the time range.

Basic Properties		
	Active	*True*
	Button Kind	*Standard Windows Button*
	Switch Mode	*False*
	Immediate Control	*True*
State Properties, primary state		
	On	*False*
	Off Text	*Last hour*
Events		
	Button Off	^Actions(SetProperty(Chart,ManualBaseTime,OPCTime("HOUR+1H")), SetProperty(Chart,TimeRange,60))

Basic Properties		
	Active	*True*
	Button Kind	*Standard Windows Button*
	Switch Mode	*False*
	Immediate Control	*True*
State Properties, primary state		

	On	*False*
	Off Text	*Last 15 minutes*
Events		
	Button Off	^Actions(SetProperty(Chart,ManualBaseTime,OPCTime("Minute15+15M")), SetProperty(Chart,TimeRange,15))

The both objects use the multiply *Actions* action with the *SetProperty* component actions to modify the *Manual base time* property value (internal name *ManualBaseTime*) and the *Time range [min]* (internal name *TimeRange*). The *Time range [min]* property change is obvious. For the *ManualBaseTime* property, a new value must be converted into the *DateTime* format. It can be specified in a text form, loaded from the *DateTime* type variable, calculated using the FromAsix6Date function. In the examples, the *OPCTime* function which in an easy way allows specifying the period of time calculated on the basis of the current moment is used. The *OPCTime("HOUR+1H")* expression calculates the end of current hour (e.g. for the 13:20 current moment, returns the 14:00 value), i.e. the chart will show the period from the beginning of the current hour to its end). The *OPCTime("Minute15+15 M")* expression which displays the current, full 15-minute period, function in a similar way.

15.1.2. Controlling Via Virtual Variable

It is also possible to control the time range via the *DateTime* type variables, as an alternative to setting the chart property values directly. This allows full operation automation.

Basic Properties		
	Manual base time	*=Variable(TimeMax)*
	Time range [min]	*10*
	Refresh mode	*Manual*

In the above case, the base time (the range end) results form the *TimeMax* variable value. The variable should be of the *DateTime* type value. Alternatively, a relevant conversion should be performed, e.g. using the *ToDateTime* function. Each change of the *TimeMax* variable value automatically alters the chart range accordingly.

The *TimeMax* variable value can be set using a script, loaded directly from a controller (with appropriate conversions) or entered using the *Text* object.

Basic Properties		
	Active	*True*
	Main Variable	*TimeMax*
	Control Variable	*#*
	Initial Edit Value	*Current*
	Immediate Control	*True*
State Properties, primary state		
	Text	*#*

If the *TimeMax* variable type was defined as *DateTime*, a new value should be specified in accordance with any system date and time text format. After verifying the correctness of the new value form, it will be converted into the *DateTime* type and stored in the *TimeMax* variable, thus affecting the time range displayed in the chart.

15.2. Controlling Trend Patterns Displaying

The *Chart* objects can also display the so-called trend patterns. Specifying the correct anchor point of the trend pattern is then critical.

Basic Properties		
	Customizable trend patterns	*False*
Trend patterns, series # 1		
	Pattern Name	*Trend1*
	Pattern Anchor	*&PatternTime*

The *Customizable trend patterns* property determines, whether the user can change a pattern curve and its anchor point (using the commands from the object context menu). If controlling the object operation with custom mechanisms is preferred, it is recommended to disable the User support.

For the pattern curve selection the *Pattern Name* property is responsible, and for its anchor point the *Anchor Pattern* property is responsible. In our example, the anchor point is set based on the *PatternTime* variable value which should be of *DateTime* type.

Basic Properties		
	Active	*True*
	Button Kind	*Standard Windows Button*
	Switch Mode	*False*
	Immediate Control	*True*
State Properties, primary state		
	On	*False*
	Off Text	*Anchor pattern*
Events		
	Button Off	^SetVariable(PatternTime,OPCTime("NOW-5M"))

Basic Properties		
	Active	*True*
	Button Kind	*Standard Windows Button*
	Switch Mode	*False*
	Immediate Control	*True*
State Properties, primary state		
	On	*False*
	Off Text	*Hide pattern*
Events		
	Button Off	^SetVariable(PatternTime,"2000-1-1")

The first of the *Button* objects sets the *PatternTime* to the value of 5 minutes before the current moment. This will display the pattern curve from this moment.

The second button hides the pattern trend by moving it outside the time range visible on the *Chart* object. An alternative method of removing pattern curve is to change the trend name to an empty name (the trend name can be also specified via a virtual variable).

16 Using Templates

The objects templates facilitate the application development. They allow creating "blocks" from which the application is built. A template can combine object groups, or in simpler cases, can contain a single object, but with the pre-set properties. Templates can have parameters which are used to specify the template functioning in its embedment place on a diagram.

In the example, the structure of a single object template based on the *Gauge* class object will be dealt with. The template will indicate when two alarm limits are exceeded, and the limit values and value ranges will be loaded from the variable definition database.

Table: Template parameters.

Template parameters		
EndAngle		
	Default value	405
StartAngle		
	Default value	135
Variable		
	Default value	

The template will have 3 parameters. *StartAngle* and *EndAngle* determine the extreme angles of the gauge scale. They have specific default values; in the template embedment place, the value of these parameters will need to be specified, provided that they are different from the default values. The *Variable* parameter is used to transmit the process variable name which value will be shown on the gauge.

When preparing (editing) the pattern, the parameters test values may be specified. This allows checking whether the template functions, as it was expected.

The partial parameterization of the *Gauge* object which constitutes the template contents is shown in the table below.

Table: Gauge object example parameters.

Basic Properties		
	Main Variable	%Variable
	Background Picture	!GaugeBkImage

	Show Limits	True
	Value	#
	Minimum Value	@DisplayRangeFrom
	Maximum Value	@DisplayRangeTo
	Value LL	@DisplayRangeFrom
	Value L	@LimitLo
	Value H	@LimitHi
	Value HH	@DisplayRangeTo
	From Angle	%StartAngle
	To Angle	%EndAngle
	Proper Value Color	!GaugeProperColor
	L Color	Yellow
	H Color	Yellow
State Properties, primary state		
	Pointer Color	*!GaugePointerColor*
	Background Color	*!GaugeBkColor*
	Outline Color	*=Property(GaugeBkImage) == "" ? Property(GaugeOutlineColor) : Transparent*
	Calibration Color	*!GaugeScaleColor*
	Font Color	*!GaugeFontColor*

The main variable of the object is defined as *%Variable*. This means a reference to the template parameter named *Variable*. The variable name will be forwarded to the template, in its embedment place on the diagram. The *From Angle* and *To Angle* properties are defined in a similar way.

Some of the properties, among others *Background Picture*, are defined with a reference to a global property. References of such type are defined by the ! prefix. As a result, the template appearance may be modified by changing the global property value. This method of parameterization is of special importance when the global properties are shared by many templates and objects - it allows changing the entire Application appearance.

The *=Property(GaugeBkImage) == "" ?* expression *Property(GaugeOutlineColor) :* The *Transparent* value used in the *Outline Color* property shows the conditional colour setting, depending on the global variable value (loaded by the *Property* function). If the *GaugeBkImage* global property is not defined (is blank), the outline colour determines the value of the *GaugeOutlineColor* global property. Otherwise, the outline is hidden by selecting the transparent colour.

The *Value* property is specified in a typical way, the # notation means the main variable value. Since a main variable is determined by a template parameter, the position of gauge pointer will result from a variable value of the name transmitted in via template parameter. The alarm range and limit properties definitions refer to the variable attributes (the @ prefix). Here, the indirect reference to a variable attribute specified by a template parameter, also occurs.

The *L Color* and *H Color* properties are defined directly. If it is necessary to use other colours, two strategies are available. An identical template, with different colours set for alarm trigger limits, can be created, or two additional colour parameters can be added and thus a single universal template created.