PROSYS

Prosys Sentrol 4 Tutorial

Hello world!

This Tutorial shows you the very basics of Prosys Sentrol and also guides you to your first Sentrol applications.

Help Files & PDF

You might consider looking at additional documentation in the **Sentrol Help**[1] while you read these lessons. Start by looking at the 'Sentrol Framework' section, which talks about the main ideas and concepts behind the design and usage of the component set. Also note that this file is available as PDF from http://www.prosys.fi/downloads.html

About the Samples

The sample projects created in this tutorial are installed along with the product. You will find them under `<installation directory>\Tutorials`. Although this document only contains listings in Delphi language, you will also find the respective C++ examples from the installed projects (except for a couple of units).

Note that there a couple of additional samples also in `<installation directory>\Samples`.

Compiler versions

There are specific projects for different compilers in the Tutorial project directories. Look for the following project files, depending on your compiler:

Compiler	Project File
Delphi 5	Project1D50.dpr or Project1.dpr
C++Builder 5	Project1C50.bpr
Delphi 6 & 7	Project1.dpr
C++Builder 6	Project1.bpr
Delphi 2006	Project1.bdsproj
C++Builder 2006	Project1C.bdsproj
Delphi 2007	Project1.dproj
C++Builder 2007	Project1C.cbproj

Vista Notes

Prosys Sentrol 4 should install fine in Windows Vista, if you just make sure that you get to install it in administrator mode.

The tutorial and sample projects are also installed under the Prosys Sentrol installation directory, which is located in "C:\Program Files\" (or similar). In Vista, normal users are not allowed to write in these directories. So before you try to compile your tutorial and sample applications, you must edit the security settings, so that you are enabled to write in these directories.

[1] You can find it from the Delphi Help (Contents) after installation.

Table of Contents

Lesson 1. Introduction to Basics 3
Stage 1.1 Variables 3
Stage 1.2 GUI Controls 4
Stage 1.3 Linking Sentrol components 5
Stage 1.4 Buttons 6
Stage 1.5 Extra: Composite Controls 7
Stage 1.6 OnChange and OnValueChange Events 8

Lesson 2. OPC Client connection 9
Stage 2.1 OPC Components 9
Stage 2.2 Connecting to the OPC server 9
Stage 2.3 Defining the OPC Connector 11
Stage 2.4 Linking Variables to OPC Items 12
Stage 2.5 CSV Files 13
Stage 2.6 Copy & Paste to Excel 13
Stage 2.7 Create New Variables for OPC Items 13
Stage 2.8 Dynamic configurations 14

Lesson 3. Trends 16
Stage 3.1 PsHistoryArray: Online Trend Buffer 16
Stage 3.2 PsChart: Multipurpose Variable Charting 17

Lesson 4. Persistent Storage 21
Stage 4.1 Define a Storage 21
Stage 4.2 Object Persistence in Storage Tables 21
Stage 4.3 Link Variables to Any Table 23
Stage 4.4 History Table 25
Stage 4.5 Sample 26

Lesson 5. Byte Arrays 28
Stage 5.1 Usage of Byte Array 28
Stage 5.2 Usage of ByteArrayConnector 29

Lesson 6. Functions 31
Stage 6.1 TPsFunction 31
Stage 6.2 TPsParserFunction 32
Stage 6.3 TPsSumFunction 32
Stage 6.4 The sample application 33

Lesson 7. Animators 34
Stage 7.1 Color Animation 35
Stage 7.2 Position & Size Animations 36
Stage 7.3 The animation sample in action 37

Lesson 8. OPC Server 38
Stage 8.1 Adding OPC Server to your application 38
Stage 8.2 Customization 39

Lesson 9. Creating components at run-time 43
Stage 9.1 Adding Links to Connectors 43
Stage 9.2 “Helper” objects 43
Stage 9.3 Sample project 43

Lesson 1. Introduction to Basics

The objective of this lesson is to create a simple application that presents the information flow between Sentrol components.

Stage 1.1 Variables

Begin a New Application from the File Menu. You will see a single form (Form1), which you can use to draw the visible and non-visible components from the Delphi palette. If you are not familiar with the Delphi environment, yet, you should take some time to learn the basics with it first.

The Sentrol components are located on several pages in the Delphi component palette:

Sentrol Vars	Variable and function components
Sentrol Controls	GUI Controls for visual display
Sentrol Anims	Simple animation effects
Sentrol Functions	On-line function components
Sentrol OPC	OPC Client connections
Sentrol Storage	Database connections

As you can see from the Sentrol Framework description in the Help file, all Sentrol activities are based on Variable data. You will define the process data using variable components. After that you can specify how that data is used: where it comes from (the OPC components), what is done with it (Vars, Functions) how it is displayed (the Controls, Anims) and how it is stored (the Storage).

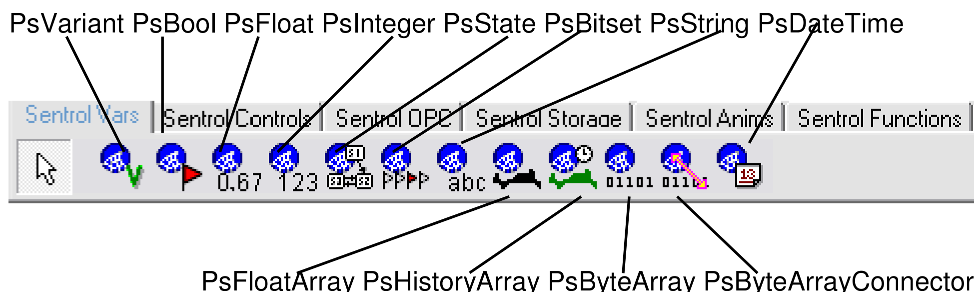


Figure 1. Sentrol components are on several palettes – the Variables are the main components.

Once you have located the 'Sentrol Vars', select a `PsFloat` from it and drop it on the form. `PsFloat` is the variable type that you will use most often. It is suited for standard analog signals and other floating point data. Check also the `PsBool`, which you can best use with binary signals. You may also take a look at the other variable types, but we will do with a single float for this first application.

Once you have the component on the form, you can set its *design time properties* with the Object Inspector.

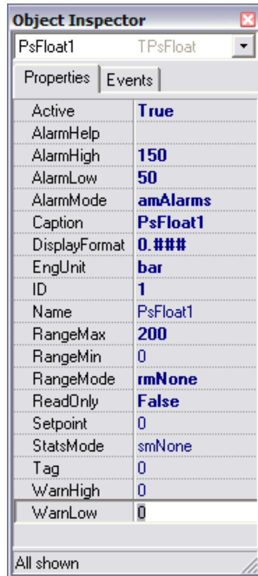


Figure 2. Properties of PsFloat1.

Set the value of `RangeMax` to `200`, `AlarmHigh` to `150` and `AlarmLow` to `50`. The alarm limits are used only in Lesson 3, but you should already set `AlarmMode` as well. Also you might like to limit the number of decimals shown on screen with `DisplayFormat`. See Figure 3, for an example and refer to Sentrol help for the format[2].

Remember that you can press *F1* anywhere in Delphi and it locates the Help description for the component that you have currently selected.

Stage 1.2 GUI Controls

After having modeled the measurement data in Stage 1.2, you can go and design the user interface!

Open Sentrol Controls from the Delphi component palette. Drop a `PsLabel`, `PsPanel` and `PsEdit` component on the Form as in Figure 4.

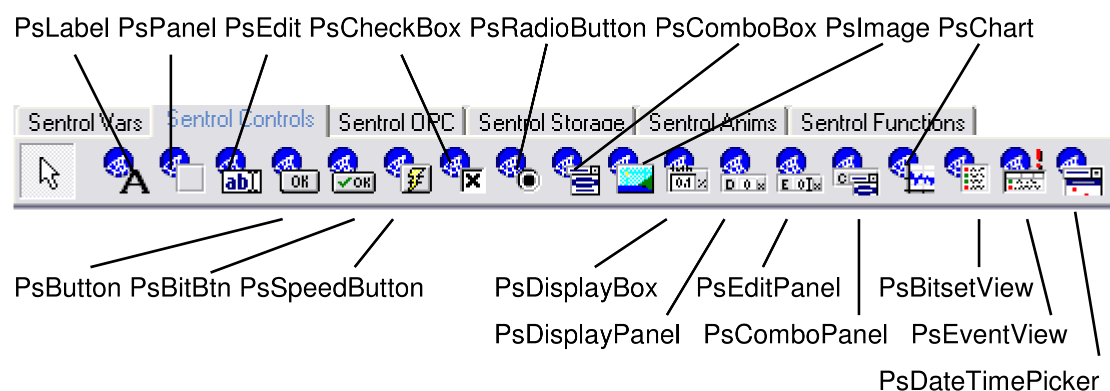


Figure 3. The Sentrol Controls are for displaying and modifying Variable data.

[2] The database fields use the same format. See TNumericField.DisplayFormat, for an example.

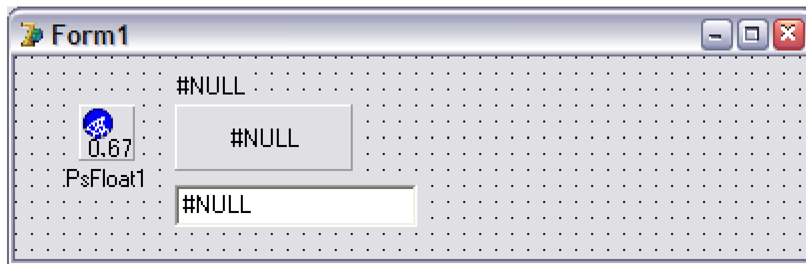


Figure 4. PsLabel, PsPanel and PsEdit placed on Form1 with PsFloat1.[3]

The GUI Controls have all the same properties that the standard Delphi VCL controls and in addition they know how to deal with Variable data.

Stage 1.3 Linking Sentrol components

All the GUI controls work the same way: they have a Variable and VarProp property, which you use to define the Variable data that they show or manipulate.

Select the `PsPanel1`, which you dropped on `Form1` and locate the property called `Variable` in the Object Inspector. Open the drop-down list and select `PsFloat1` from the list (you should not be able to miss it!). Repeat this with `PsEdit1` and `PsLabel1`.

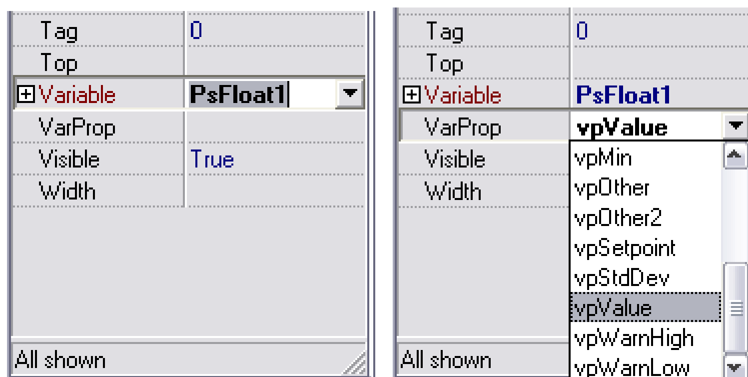


Figure 5. Setting the Variable and VarProp with the Object Inspector. All Variables in the active form are shown on the selection list. Also components in other forms and data modules will be shown, after the units are *used* in the form (from the menu, use `File->Use Unit`).

Now, select the `PsLabel1` and change its `VarProp` from `vpValue` to `vpCaption`. If you want, you can go and change the caption of `PsFloat1` to your choice.

You can also connect to other *run-time properties* of the Variable by changing the `VarProp` from the default of `vpValue`. This way you can define generic fields in the form, which can then be switched at run-time to display or modify any Variable – or any VarProp. Note however, that not all VarProps are useful with all variable types.

---

[3] '#NULL' means that the components are not connected to any Variable

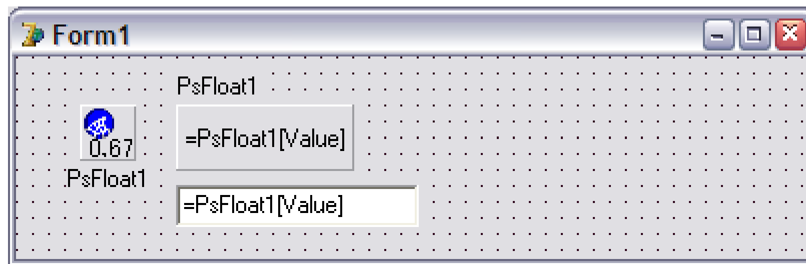


Figure 6. The controls show where they are connected to at design time.

Now you are ready to compile the program. Values fed in `PsEdit1` are seen in `PsPanel1` once you press <Enter>.

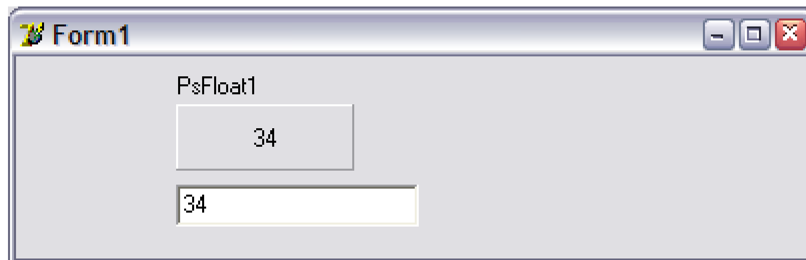


Figure 7. The first Sentrol application created in Lesson 1.

Stage 1.4 Buttons

To write values to the variables, you can use the edit box – or buttons. Buttons come in different flavors, corresponding to different Win Controls: `TPsButton`, `TPsBitBtn`, `TPsSpeedButton`. And there is also a `TPsRadioButton`, `TPsCheckBox` and `TPsComboBox`! In fact they all behave a bit differently, see the available properties and the help for details.

The idea in all the buttons is the same: You use a button to set a variable to a certain value. So, first connect the button to the variable you want to set, the same way that you did with the label, panel and edit box. And then, define the value to be set, in `ClickValue`.

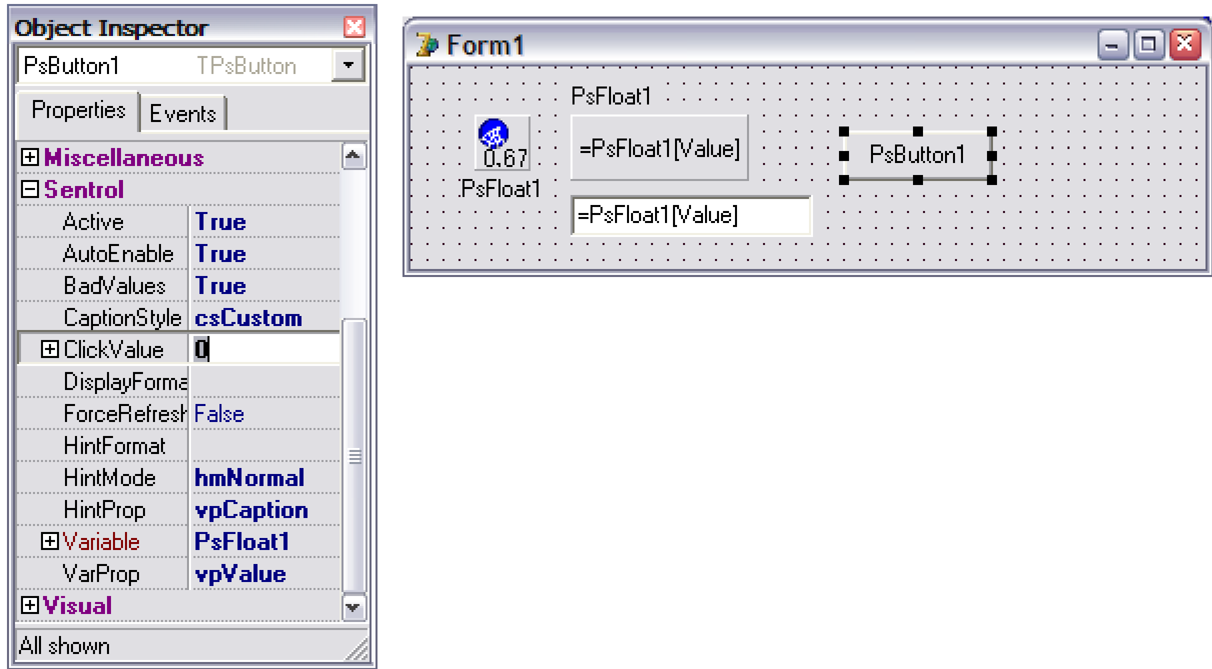


Figure 8. Defining ClickValue for a button. Note that if you arrange the Object Inspector by Category, you can see the Sentrol properties grouped together!

Run the application – and use the button to reset the variable to 0!

Stage 1.5 Extra: Composite Controls

Drop a `PsDisplayBox, PsDisplayPanel, PsEditPanel` – and possibly other controls on the form, select a Variable and VarProp for each and see how they behave!

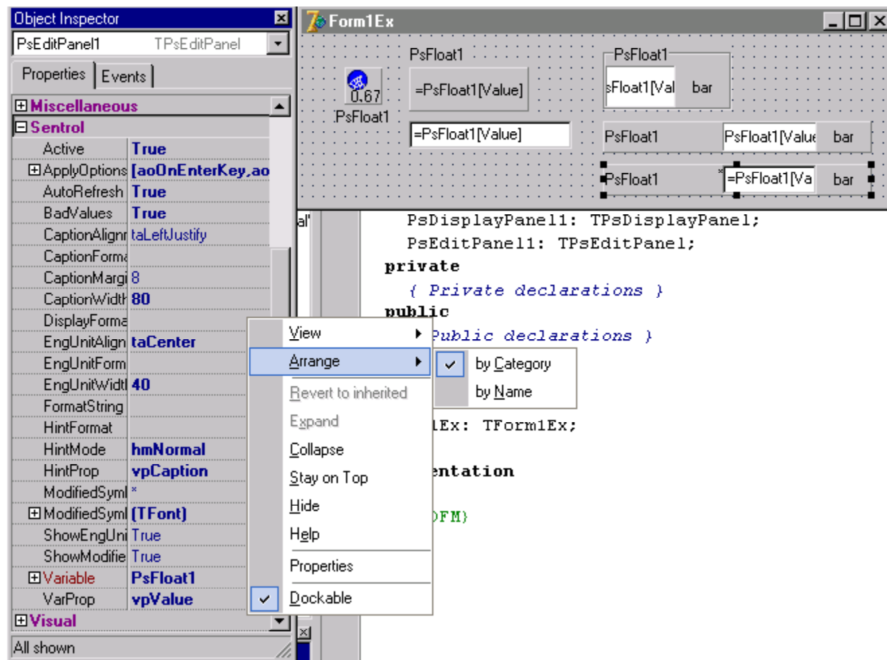


Figure 9. Object Inspector arranged by category helps to locate the special properties of the Sentrol components. Since Delphi 2005, the properties are arranged by category already by default.

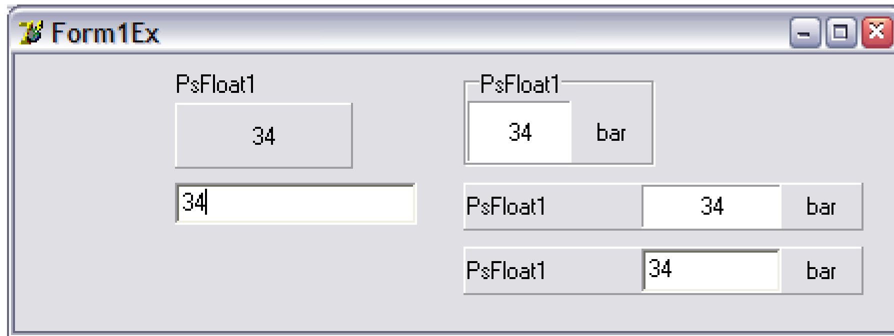


Figure 10. The extended application with some composite controls on it!

Stage 1.6 OnChange and OnValueChange Events

The variables also have event handlers that you can use to catch changes when you need to trigger actions in your own code. You can use `TPsVar.OnChange` to react to changes in any VarProp, but if you are only interested in the Value changes, then you will do better with `TPsVar.OnValueChange`.

For example:

```
procedure TForm1Ex.PsFloat1Change(Sender: TPsVar; props:
      TPsVarPropTypes);
begin
  // whenever the Caption or Value changes, update the Form1.Caption
  if (vpValue in props) or (vpCaption in props) then
    Caption := Format('Form1Ex - %s=%s',
      [Sender.Caption, Sender.ValueAsString]);
end;
```

So, in `OnChange` handler you should filter the interesting events yourself, as it will be called whenever any `VarProp` changes – and the `Value`, for example, may still be unset.

And to catch changes that are about to occur, you can also use `OnChanging`!

Lesson 2. OPC Client connection

The objective of this lesson is to read data from an OPC-server using Sentrol OPC components and the application created in Lesson 1.

Stage 2.1 OPC Components

Continue from Lesson 1. Locate a `PsOPCServer` and a `PsOPCConnector` from the Sentrol OPC palette and drop them on the Form.

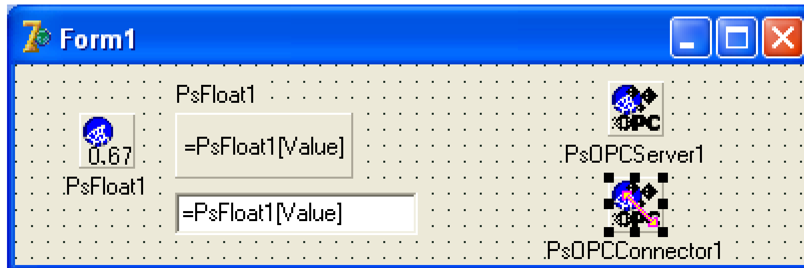


Figure 11. OPC components in Form1.

The `PsOPCServer` is used to define a connection to any OPC Data Access server and the `PsOPCConnector` is used to define how the OPC Items in the Server map to the variables in the application.

Stage 2.2 Connecting to the OPC server

Select the `PsOPCServer1` component on the form. Select the `ServerName` property in the Object Inspector. If there are OPC Servers installed in your PC[4], they will be seen in the drop-down list (see Figure 12). Pick one. You can connect to the server already at design time, by setting the `Connected` property to `True`.

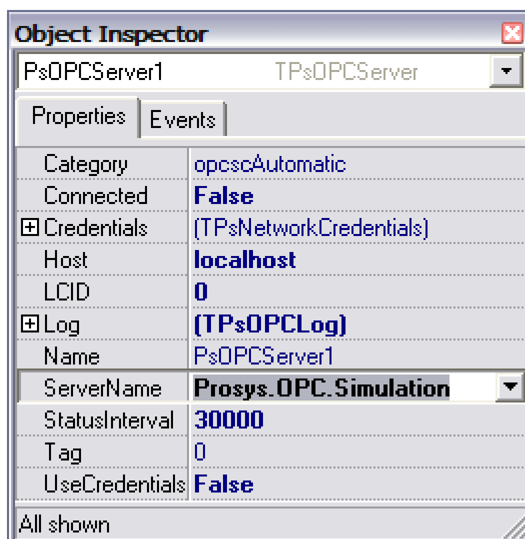


Figure 12. Select the OPC Server to connect to from the combo box – or just enter the ProgID (or CLSID) for the server.

---

[4] If you don't have any OPC Servers available, go and get a free simulation server from Prosys <http://www.prosys.fi/downloads.html>

You can also connect to remote OPC servers – just set the `Host` property before setting the `ServerName`. You must, however, be allowed to connect to the other host by DCOM and you will also need to allow data transfer back from that host to your local host. Check the OPC Foundation White Papers at http://www.opcfoundation.org/Downloads.aspx?CM=1&CN=KEY&CI=282 for how to do this, if you are not familiar with it.

Since Sentrol 4, you can also define additional user credentials for remote access. Set `UseCredentials=True` and define the username and password to `Credentials`.

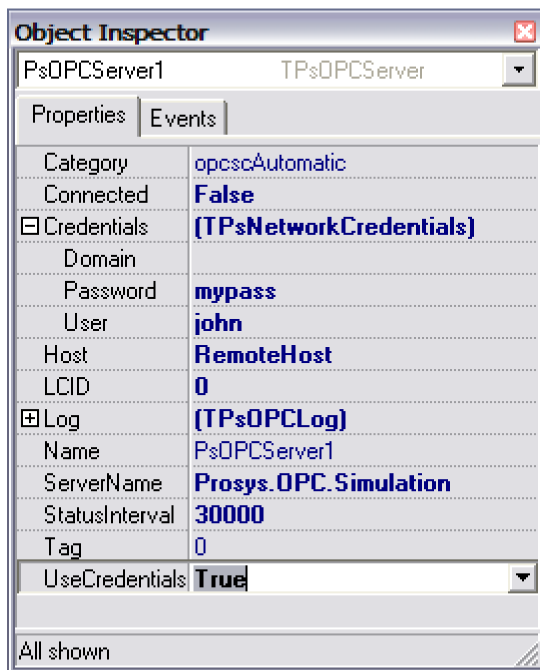


Figure 13. Defining alternate user credentials for remote access.

Stage 2.3 Defining the OPC Connector

Next, we will define the mapping between the OPC items in the server and the variables in our application.

Select the `PsOPCConnector1` component that you have on the form.

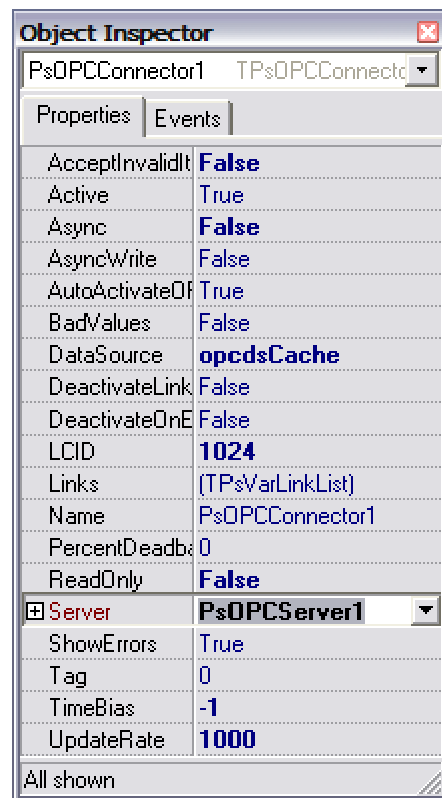


Figure 14. Properties of PsOPCConnector.

Use the Object Inspector to set `Active`, `Async` and `AsyncWrite` to `True` (asynchronous connections are preferred to synchronous, for efficiency, but you may also choose to use synchronous data transfer). Finally, set `Server` to `PsOPCServer1`.

Stage 2.4 Linking Variables to OPC Items

Next, you can define Links. Click the small button to the right of the property value box (the one with three dots in it) and the OPC Links Editor opens (Figure 15).

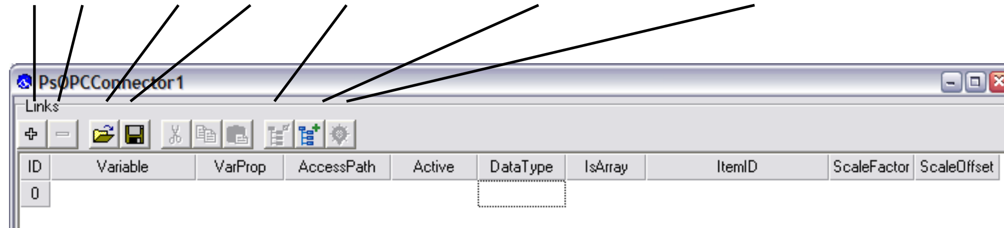
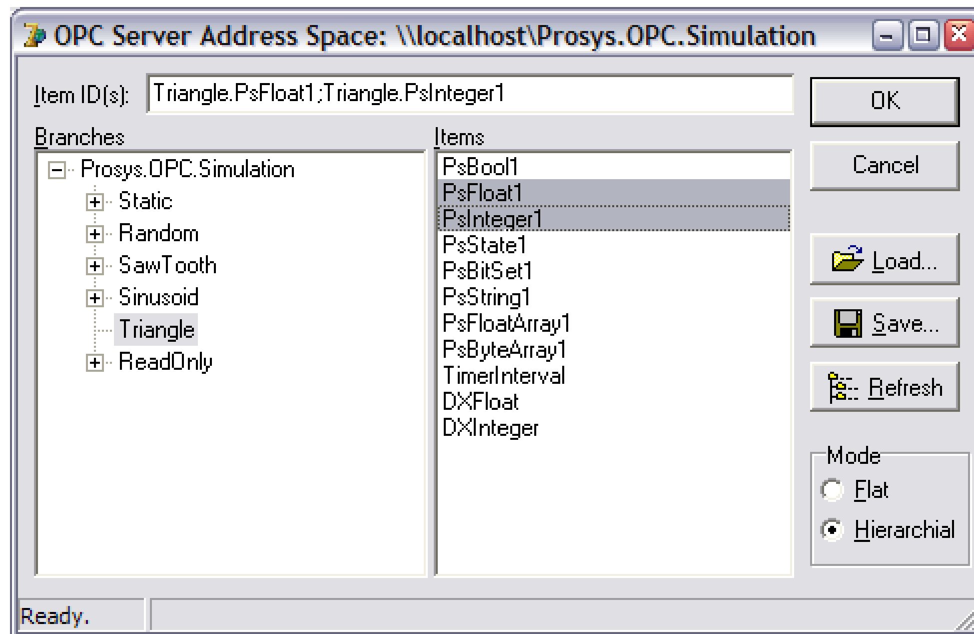


Figure 15. Links Editor for PsOPCConnector1. The editor is used to define how OPC Items are linked to the Sentrol Variables.

Add a new link with the Add OPC Items -button. The OPC Address Space Browser (see Figure 16) opens with a tree view that contains the address space of the connected OPC Server. Select a branch from the left and an item from the right (in this example we have used the Prosys Simulation OPC server). The OPC Item ID should appear in the edit box at the top of the dialogue. Press OK to accept the selection.



Note: You can select multiple items from the list and add them all at once!

Figure 16. Selecting the OPC Items from the OPC server address space.

Once you are back at the Links Editor, set `Variable` for the selected link to `PsFloat1`. You get a drop-down list when you click the `Variable` cell in the grid.

Note! Unless you define the connector to be `ReadOnly`, it will both read and write values to the OPC server automatically, whenever the other one changes (if the variable values in your application are changed, they will be written to the OPC server).

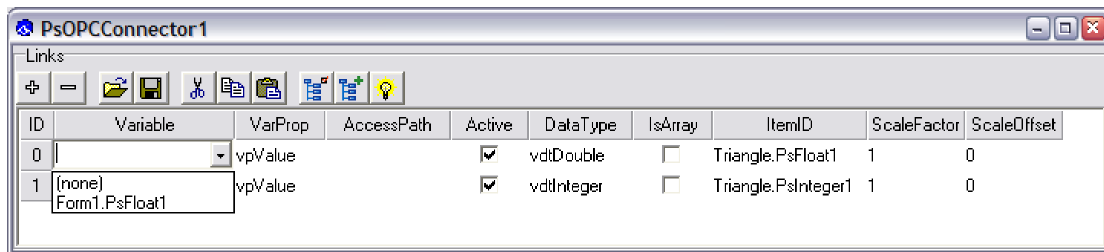


Figure 17. Defining the Variable for the link.

Close the Links Editor. Run the application. If you got everything right, you should see values updating on the panel as in Figure 18.

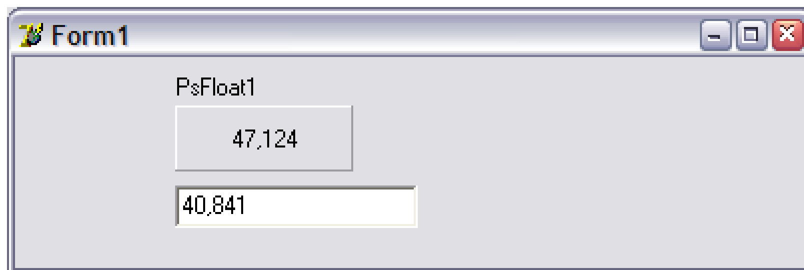


Figure 18. Application created in Lesson 2. You can change the value from the edit, to make the connector write the value to the OPC server. You will notice that the simulation in the server continues from the new value!

Stage 2.5 CSV Files

You can save and load the links from CSV files – and use other applications to modify the configuration. Use the Load and Save buttons in the Editor toolbar.

The CSV format of the used files is as follows[5]:

```
Variable;VarProp;AccessPath;Active;DataType;IsArray;ItemID;ScaleFactor;ScaleOffset
Form1.PsFloat1;vpValue;;True;vdtDouble;False;Triangle.PsFloat1;1;0
;vpValue;;True;vdtInteger;False;Triangle.PsInteger1;1;0
```

Note: When you load CSV files, make sure that the variables you refer to, already exist in the project!

Stage 2.6 Copy & Paste to Excel

The best way to handle the data, with for example, Excel can be with the CSV files, but you can also select separate lines to copy and paste between the editor and Excel.

Stage 2.7 Create New Variables for OPC Items

Did you notice the Create Variables button in the Editor Toolbar?!

It opens a dialog that lets you define which kind of variables you would like to create for the OPC Items you just brought from the address space. By default it will try to use the OPC Item DataTypes for selecting the appropriate variable type for each item. The Advanced Options define how the components will appear on the form or data module and how they are named.

[5] The CSV separator is defined in Windows Regional Settings; in this example it is ‘;’.

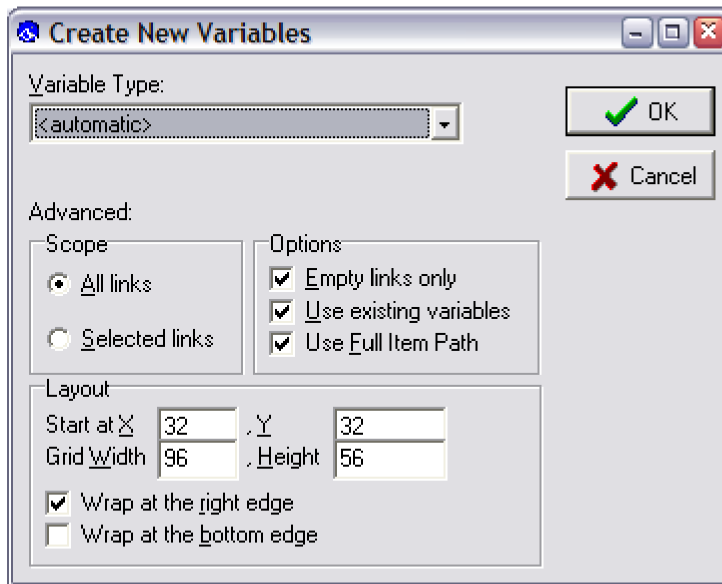
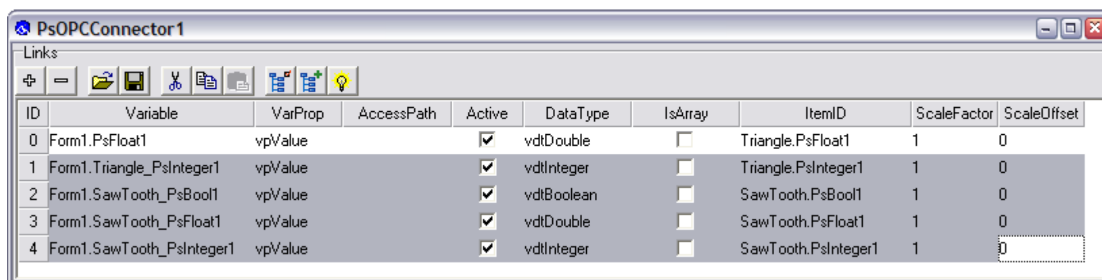


Figure 19. Create Variables Dialog.

The new variables appear on the form – and are linked to the OPC Items!



ID	Variable	VarProp	AccessPath	Active	DataType	IsArray	ItemID	ScaleFactor	ScaleOffset
0	Form1.PsFloat1	vpValue		☑	vdtDouble	☐	Triangle.PsFloat1	1	0
1	Form1.Triangle_PsInteger1	vpValue		☑	vdtInteger	☐	Triangle.PsInteger1	1	0
2	Form1.SawTooth_PsBool1	vpValue		☑	vdtBoolean	☐	SawTooth.PsBool1	1	0
3	Form1.SawTooth_PsFloat1	vpValue		☑	vdtDouble	☐	SawTooth.PsFloat1	1	0
4	Form1.SawTooth_PsInteger1	vpValue		☑	vdtInteger	☐	SawTooth.PsInteger1	1	0

Figure 20. The newly created variables are linked to the respective OPC Items.

Stage 2.8 Dynamic configurations

If you have noted that all the configurations are done at design-time, and wonder if the same could be done more dynamically at run-time only, please consult Lesson 9.

Stage 2.9 COM Threading Models

Because OPC is based on Microsoft COM technology, your application is actually a COM client application. You should therefore also be aware of the different threading models supported by COM.

The default threading model is the single threaded, but for OPC communication the recommended setting is the multi-threaded (also called "free threaded") model. It ensures that data callbacks from the OPC server get through to your application without delays in all situations.

Fortunately, defining the threading model is easy in Delphi; you can simply set the value of `CoInitFlags` at your project source, for example as follows:

```
program Project1;

uses
  ActiveX, // defines COINIT_* constants
  ComObj,  // defines the CoInitFlags variable
```

```
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  // Define the COM threading model
  CoInitFlags := COINIT_MULTITHREADED;
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

See the Help for more information on CoInitFlags.[6]

[6] In C++ Builder, Help is available under "multi-thread apartment threading model", and the necessary include files are ”objbase.h” and ”activex.hpp”.

Lesson 3. Trends

The objective of this lesson is to draw a trend of a variable. Again we will continue with the application created in Lessons 1 and 2.

The trends are handled in two ways: `PsHistoryArray` (Sentrol Vars) is used to keep a finite length of Variable history for display and analysis (e.g. sliding average). On the other hand, `PsHistorian` (Sentrol Storage) is used to define a persistent, long-term storage of variable values. A third component, the `PsChart` (Sentrol Controls), is used to display history data on screen[7].

Stage 3.1 PsHistoryArray: Online Trend Buffer

We will continue with the project from Lesson2. First, enlarge the form a little to make room for the chart. Then drop a `PsHistoryArray` (Sentrol Vars) and a `PsChart` (Sentrol Controls) on the form.

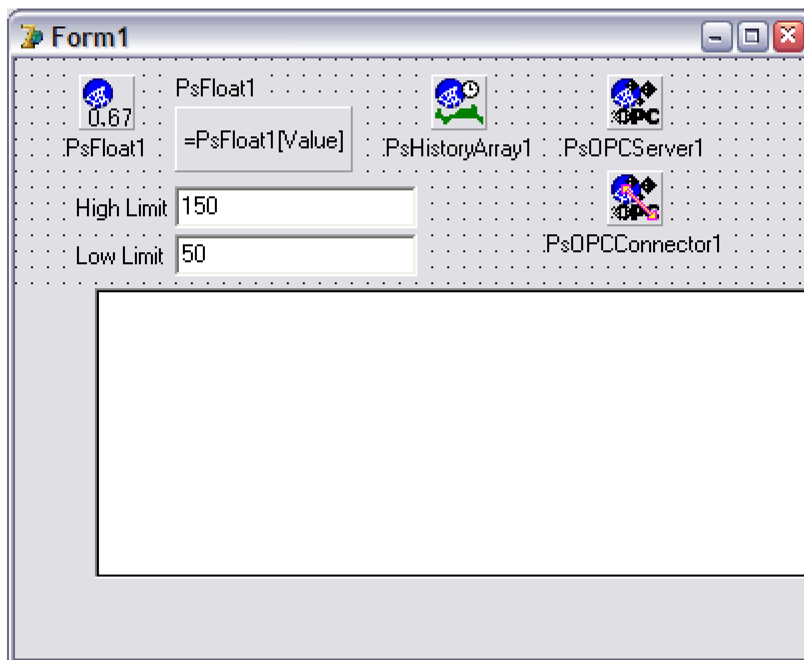


Figure 21. PsHistoryArray and PsChart added on form. The chart does not display anything until a variable is linked to it. The edit boxes for limits are used to modify the alarm limits at run-time.

Connect `PsFloat1` to `PsHistoryArray1` by setting the `Variable` property with the Object Inspector.

Also set `Circular` to `True` and `Capacity` to 60, which sets the array length to a minute (equals to `Capacity * SampleWidth` [in milliseconds]).

The `Range` and `Alarm` limits are copied from the variable, when `VarRange` and `VarSpecs` are set. You must, however, define `AlarmMode` by yourself.

---

[7] Actually, you can also use PsChart to display any variable values, although only PsFloat and PsFloatArray are currently meaningful in addition to PsHistoryArray.

Object Inspector

PsHistoryArray1 TPsHistoryArra

Properties | Events

Property	Value
Active	True
Aggregate	agRaw
AlarmHelp	
AlarmHigh	150
AlarmLow	50
AlarmMode	amAlarms
ArraySpecs	True
Capacity	60
Caption	PsFloat1
Circular	True
Deadband	0
DisplayFormat	
EngUnit	
FirstIndex	0
ID	6
Name	PsHistoryArray1
RangeMax	200
RangeMin	0
RangeMode	rmNone
ReadOnly	False
SamplingInterv	1000
SamplingMode	smChange
Setpoint	0
StatsMode	smNone
Tag	0
VarCaption	True
VarEngUnit	True
⊞Variable	PsFloat1
VarProp	vpValue
VarRange	True
VarSpecs	True
WarnHigh	0
WarnLow	0

All shown

Copied from Variable (if VarSpecs is True)

set these

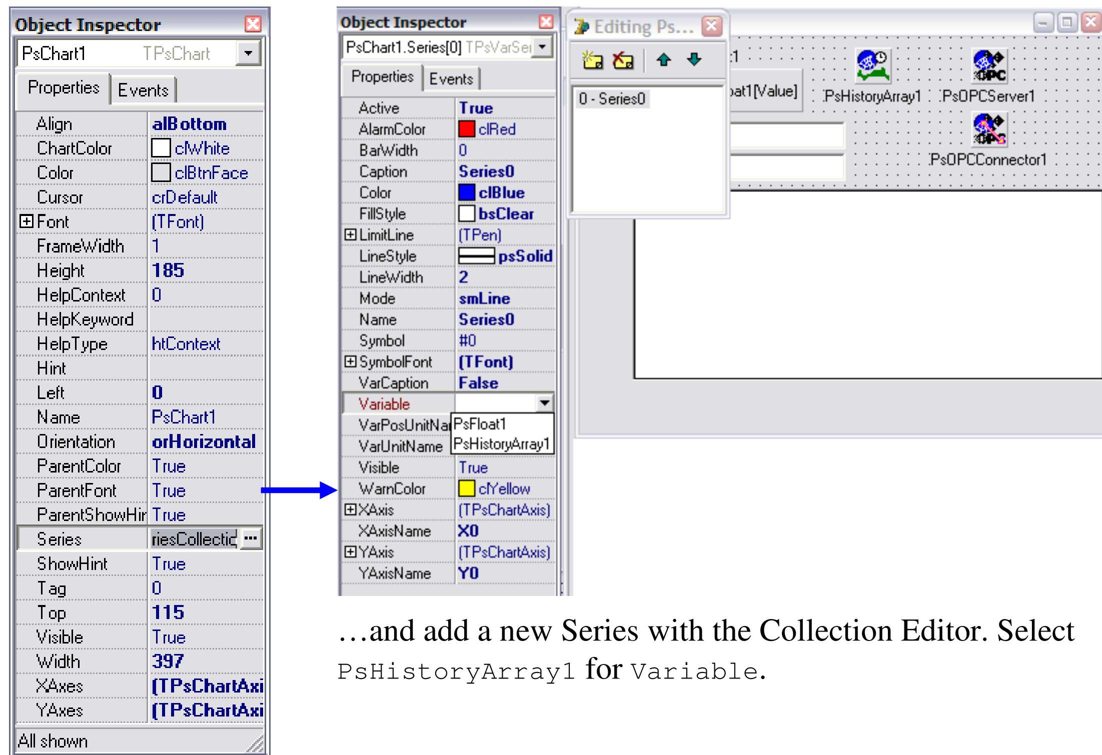
Copied from Variable (if VarRange is True)

set this

Figure 22. Setting PsHistoryArray to keep an online trend of PsFloat1 for the last minute.

Stage 3.2 PsChart: Multipurpose Variable Charting

Next select the chart, which is already on the form. First you must add one series on the chart:



…and add a new Series with the Collection Editor. Select `PsHistoryArray1` for `Variable`.

Double click Series…

Figure 23. Chart properties. Open the Series to get the Series Collection Editor. You can define any number of series to draw. Each can be connected to any variable.

Last, change the `PsEdits` to modify the `AlarmHigh` and `AlarmLow` of `PsFloat1`: you should be able to modify the limits at runtime!

Compile and run the application. You will see a variable trend updating in the chart.

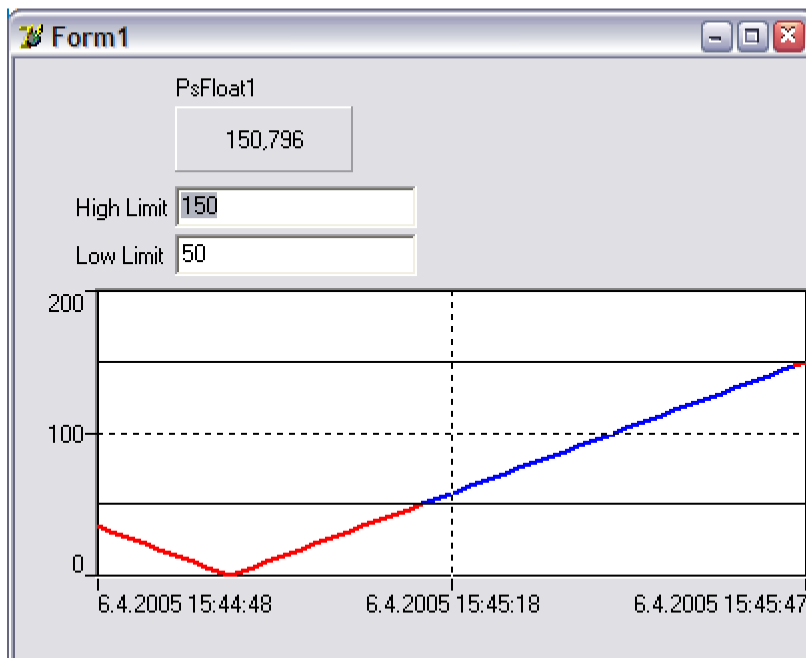


Figure 24. Trend curve in the application of Lesson 3.

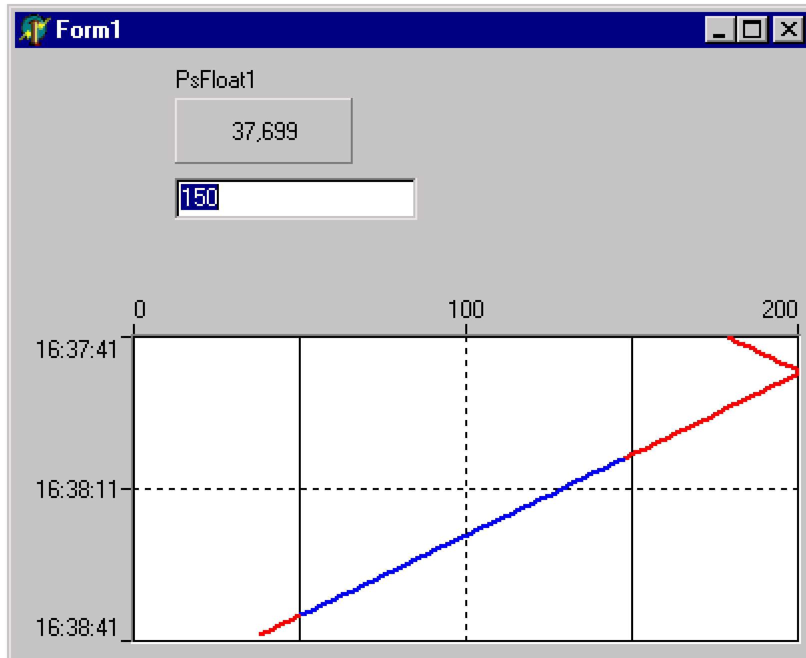


Figure 25. Another trend style. Here Orientation = orVertical, Series.Fill.Style = bsClear, Series.Line.Width = 2, XAxis.DateTimeFormat='hh:mm:ss', XAxis.Width = 50, XAxis.Mirror=True and YAxis.OtherSide=True. Go ahead and try!

3.2.1 Trend time scale

The axes are by default aligned to the range of the data, i.e. Y-axis runs from `Variable.RangeMin` to `RangeMax` and X-axis to the `Positions` of the variable – or time interval of a `HistoryArray`. But for trending purposes, it is best to specify `XAxis.AlignMin` = `aaInterval` – and use `Interval` or `IntervalMs` to define a fixed time interval for the axis. In this case, the length of the X-axis will be fixed, and it won't stretch from a narrow to full trend interval as new data is appended to the history array, as is otherwise the case.

In this context, it is also better to set the `TickMode` to `tmInterval` and define a good interval into `TickInterval` or `TickIntervalMs`. See Figure 26.

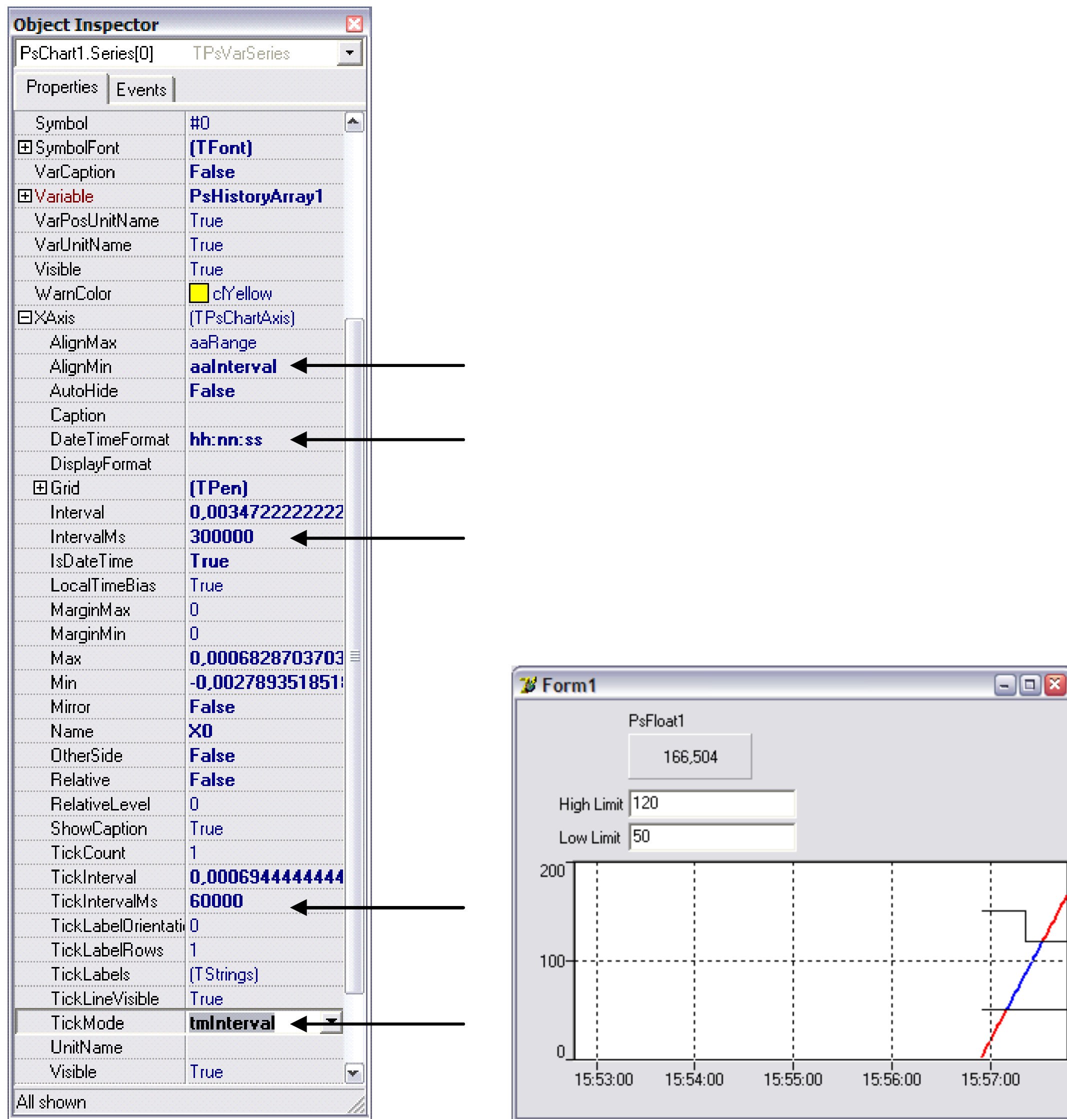


Figure 26. Check the marked properties (the arrows) to define a fixed Interval in the X-Axis. The sample application demonstrates this and also the trend of alarm limits!

Lesson 4. Persistent Storage

The function of the Sentrol Storage components is to enable connections between all database types and tables and Sentrol Variables.

The current set includes several storage variants, for specific database drivers, such as `PsBDEStorage` for BDE and `PsSQLStorage` or dbExpress[8]. The idea is that the other components in the Sentrol Storage palette will function with any Storage component – and you will be free to choose the database driver you like to use.

To link your variables into the database, you have two components: the generic `PsStorageConnector`, which defines correspondences between variable props and database fields; and `PsHistorian`, which is used for recording variable histories.

Stage 4.1 Define a Storage

Drop the `PsBDEStorage` or `PsSQLStorage`[9] (Sentrol Storage) on the form and rename it as Storage1. Also add a `TDatabase` or `TSQLConnection` component, respectively, to define the DB connection. In this example, we use the `IBLocal` database, which is included in Delphi installation.

Set `PsBDEStorage.DatabaseName` or `PsSQLStorage.SQLConnection` to point to the DB connection component from the drop-down list.

4.1.1 DatabaseProduct

In order to be able to create tables (see below) the DatabaseProduct that the storage component is connecting to, must be defined. This enables correct SQL syntax and data types to be used. Currently, Sentrol supports Interbase/Firebird (all versions), MS SQL Server (all versions) and MS Access (all versions).

Note: You can still use the storage components with any database that is supported with the mentioned database drivers. You just cannot use the automatic table creation feature for these databases.

Stage 4.2 Object Persistence in Storage Tables

The Storage components can also be used to manage database tables. This includes:

- Creating and dropping tables
- Adding and dropping columns
- Executing statements: insert, update, select, etc.
- Storing object data in tables
- Creating objects from the tables!

These operations are all independent of the storage variant you choose (and therefore also of the DB driver). You only need to define the `DatabaseProduct` for the Storage, so that it can use the correct SQL syntax for your database.

---

[8] ADO, IBExpress are also supported (and ZeosDBO, which requires that you install the ZeosDBO components first), if the respective drivers are available in your Delphi installation.

[9] SQLStorage uses dbExpress, which is not available in Delphi/C++Builder 5!

See the help for `TPsCustomStorage`, `TPsStorageRepository` as well as `TPsStorageTable`, `TPsObjectTable` and `TPsComponentTable` on what is available.

4.2.1 Variable tables

By default, the storage components define three tables for storing variables and the related information. The tables are defined in the following properties:

- `ModulesTable` for adding information of DataModules and Forms in the application
- `VariablesTable` for adding information of all variables in your application to be stored in the database
- `VarPropsTable` for just declaring the VarProp values and their meanings

This information is necessary, if you store, for example, history data in the database, since they use mostly IDs to refer to the components. Thus, these tables define which variable has which ID, so that the data can be used also external to your application, for example to make joined queries to the data tables.

4.2.2 Enabling tables

In order to take these metadata tables in use, you just need to enable them. You can also define the table structure in the field objects.

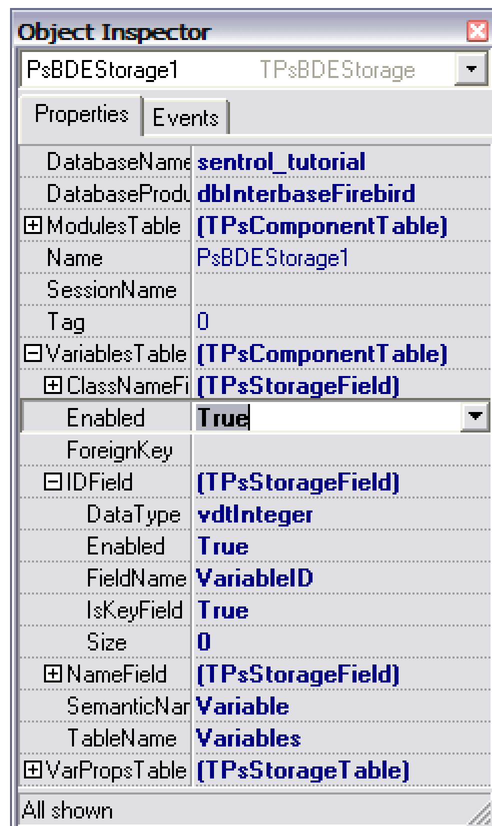


Figure 27. Enabling VariablesTable.

The tables have a special property, `SemanticName`, which defines the table semantics(!): the `TableName` and `FieldName` of `IDField` will be derived from that, unless you change them to something else.

4.2.3 Creating Tables

The storage will create all **enabled** tables automatically when it first connects to the database at runtime, unless the tables already exist. You can also call `CreateTable` in the storage or in any table object directly, to create the table. The table must have a valid field definition (including specific `DataType`s) in order to make table creation succeed.

The components in the following examples also define table properties. To create these tables, call:

```
PsHistorian1.HistoryTable.CreateTable;
PsStorageConnector1.Table.CreateTable;
```

The table will be created according to the definitions in the `Table.Fields`. Fields that have `IsKeyField` set, will also be included in the tables **primary key**.

If you wish to have more control over the table structure, it is better to use the database provider's tools for creating the tables for you. Then you just need to map the fields correctly to the storage component definitions using the `FieldName` properties.

4.2.4 Dropping Tables

To remove the tables from the database, you can use `DropTable`, which is also available in the storage and in the tables.

4.2.5 Adding data to the storage tables

You can add variables and data modules to the storage by calling manually `Storage.VariablesTable.SaveComponent` or `Storage.ModulesTable.SaveComponent` with your components. Or you can let the `PsHistorian` and `PsStorageConnector` components to do this automatically (which they will do, when you add variables to them).

Stage 4.3 Link Variables to Any Table

Drop a `PsStorageConnector` on the form. Set the Storage property to `Storage1`. Set `Table.TableName` to `TEST`. Open the Links Editor and define a link between `PsFloat1` and field `'F'` – as well as `PsBool1` and `'B'`.

Also define a couple of additional fields: `ID` and `ReportTime`. Just add them to `Links`, but without any variable connections. Set `IsKeyField` on for `ID`. See Figure 28.

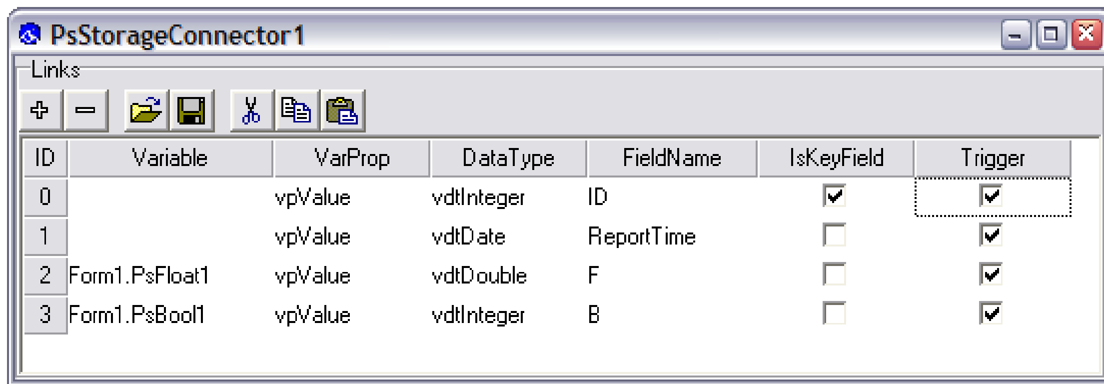


Figure 28. Defining links between variables and database fields. DataType must be defined if the variable datatype does not match with the field type (i.e. in Interbase we use Integer fields for Boolean data).

As discussed earlier, you can use the CreateTable method to create the table into the database. Or define it yourself: For example, create a table corresponding to the following SQL definition in the `IBLOCAL` database using Database Explorer:

```
CREATE TABLE TEST (
  ID INTEGER NOT NULL,
  REPORTTIME TIMESTAMP,
  F FLOAT,
  B INTEGER,
 PRIMARY KEY(ID)
)
```
[10]

The connector can be used for saving variable data into the table as well as loading data from the table to variables. It will not be very useful to use the same component for both directions, though. Normally, you will copy data manually, using Load or Save. If you wish to save data into the table automatically, define `TriggerMode` either to `tmDelayed` or `tmImmediate`. If you are not sure, use `tmDelayed`, which can wait for all changes occurring at the same time in all variables, before it will function. We will keep to the manual savings here – and leave the trigger to `tmManual`.

For saving, you have two alternatives: either to **insert new records** or **update existing ones**. Use `SaveType` property for this. Now, leave it as `stInsert`.

To trigger the saving, we use a Button: drop it on the form and define its `OnClick`-event as follows (here we have added a `TEdit` component (`IDText`) to supply the `ID` values) :

```
  PsStorageConnector1.SetSaveParamByName('ID',
StrToInt(IDEdit.Text));
  PsStorageConnector1.SetSaveParamByName('ReportTime', UTCNow);
  PsStorageConnector1.Save;
```

That will provide the values for the Links that are not connected to any variables (usually the key fields) – and saves the new record to the table.

---

[10] If you decide to use a different database than Interbase/Firebird, you will need to modify the table definition accordingly. Since Sentrol 3.2, you can also create the table from your application, see above, Stage 4.2.3.

4.3.1 Loading values back

To load values from the database back to the variables (or to set the limits and setpoint according to a recipe, for example) goes in a similar way using the Load method. You will have to define how the record is selected with the key field values, using SetLoadParamByName first, for example:

```
  PsStorageConnector1.SetLoadParamByName('ID',
StrToInt(IDEdit.Text));
  PsStorageConnector1.Load;
```

Stage 4.4 History Table

The Historian works as a "narrow" historian, where it records changes in variable properties that it is set to "watch".

Now you can drop a `PsHistorian` on the form. Set `Storage` again to `Storage1` and `TableName` to the table you just created. Finally, define `Links` for all the variable properties that you wish to record in the table. In this tutorial, you will just need to add two links: one for `PsBool1` and one for `PsFloat1`. See Figure 29.

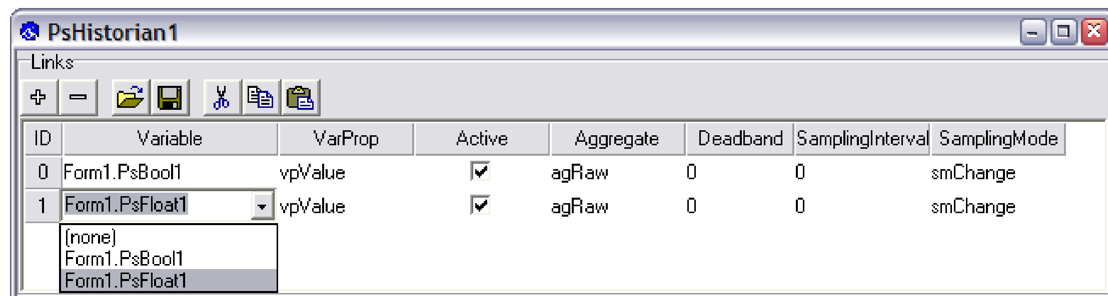


Figure 29. Defining history data to be recorded by the PsHistorian.

To create the table in the database, run Historian.HistoryTable.CreateTable in your application (set Storage.DatabaseProduct first) – or create the respective table using your database tools.

The default table is defined according to the following Interbase/Firebird definition:

```
CREATE TABLE HISTORY (
  VARIABLEID INTEGER NOT NULL,
  VARPROP INTEGER NOT NULL,
  CHANGETIME TIMESTAMP NOT NULL,
  NEWVALUE FLOAT,
 PRIMARY KEY(VARIABLEID, VARPROP, CHANGETIME)
)
```
[11]

You can customize this, using the `HistoryTable` property in the historian. For example, if you wish to also include a reference to the data modules (stored in `Storage.ModulesTable`), just enable `ModuleIDField`.

[11] Note: the order of the fields in the primary key has some effect on the order in which data is in the table. This order is fastest, if you will be fetching individual variable histories, but it will not show the changes in a time line, if you want to see all variables at the same time, as is the case with the sample application (see Figure 30). Unless by using SQL statements with "ORDER BY".

Compile and run the application and you should see new records appear in the `HISTORY` table as the `Variable` changes. Click the button and you should get a new record into the `TEST` table.

Stage 4.5 Sample

The sample application enables you to create the tables into '`IBLOCAL`' and experiment with the storage connector and historian. Clicking the Report button will add a new row into the 'TEST' report table – and refresh the datasets. Use the ID field to determine the ID of the report.

4.5.1 Use PsStorageDataset for data aware components

Note that the sample uses a generic `TPsStorageDataset`[12] component for displaying the table data in the data aware. PsStorageDataset fetches the data from the storage component, so if you switch to a different DB driver, you only need to change the Storage!

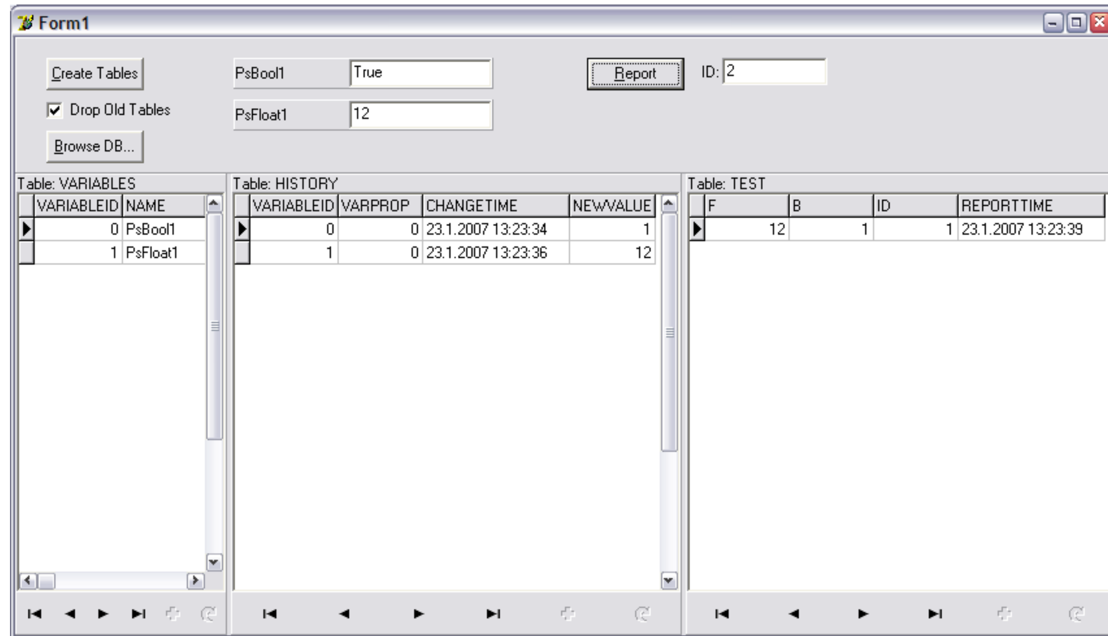


Figure 30. Storage sample.

Note that there is also a generic database browser available in the tutorial. The browser form is a standard part of Sentrol installation. Look for `TPsStorageBrowser` in `PsStorageBrowserForms.pas` – and add it to your application as a generic DB browser!

[12] The storage dataset and storage browser (which uses the dataset) are not available in Delphi/C++Builder 5. The dataset is based on TClientDataset which uses the MIDAS library. You must include unit 'midaslib' in the uses clause of your project source or distribute the midas.dll (included in Delphi installation) along with your application to be able to use it.

Lesson 5. Byte Arrays

Byte arrays are used to handle raw device (or PC) memory data.

Sentrol has a specific variable type for using byte arrays, `TPsByteArray`, and also a specific connector for mapping the byte arrays to other variable types, `TPsByteArrayConnector`.

Stage 5.1 Usage of Byte Array

You can use `TPsByteArray` to work with device data, for example, in connection with third party or your own device drivers – or when accessing raw data from OPC servers. It can handle byte and bit endianness and respectively provides methods to get and set data in the memory area to and from native data types.

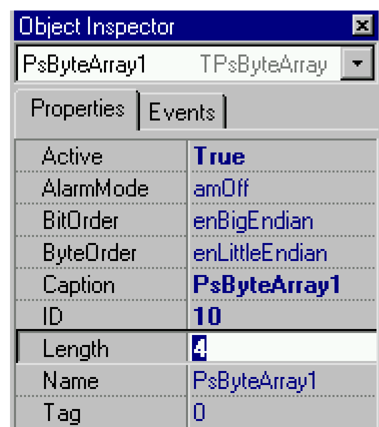


Figure 31. Properties of a Byte Array.

You just need to set the length of the byte array – and optionally change endianness from the Intel default values, if you are accessing data from a field device. Field devices typically use `BitOrder&ByteOrder=enBigEndian`, but you should refer to the documentation for the device to ensure that you have the correct settings (which you can, of course find out by just trying out). The `ByteOrder` will affect when mapping the data to native data types larger than one byte, e.g. Word, Integer, Float etc.

You can write data to the byte array simply using for example, `TPsByteArray.Bytes`:

```
procedure TForm1.SetRandomBytesBtnClick(Sender: TObject);
var
  I: Integer;
begin
  PsByteArray1.BeginUpdate;
  try
    for I := 0 to PsByteArray1.Length-1 do
      PsByteArray1.Bytes[I] := Random(256);
  finally
    PsByteArray1.EndUpdate;
  end;
end;
```

(Use `BeginUpdate`/`EndUpdate` to postpone the change notification to the end of all your changes)

Other applicable properties are `TPsByteArray.Values` (takes in a variant array) and `TPsByteArray.Lock`, which provides direct access to the internal memory using a `PsByteArray` pointer (just remember to Unlock it after you've done with it), e.g.

```
procedure TForm1.SetRandomLockBtnClick(Sender: TObject);
var
  I: Integer;
  P: PByteArray;
begin
  P := PsByteArray1.Lock;
  try
    for I := 0 to PsByteArray1.Length-1 do
      P[I] := Random(256);
  finally
    PsByteArray1.Unlock;
  end;
end;
```

Now, you can take data out of the ByteArray simply using GetAsDataType, e.g.

```
procedure TForm1.CopyToIntegerBtnClick(Sender: TObject);
var
  I: Integer;
  B: Boolean;
begin
  I := PsByteArray1.GetAsDataType(dtInt16, {Signed:}True, {Byte:}0);
  PsInteger1.Value := I;
  B := PsByteArray1.GetAsDataType(dtBit, True, 2, {Bit:}0);
  PsBool1.Value := B;
end;
```

Here, we have mapped the first two bytes of the data to `PsInteger1` and the first bit of the third byte (Byte number #2) to `PsBool1`.

`SetAsDataType`, obviously, can be used to set values to the ByteArray, respectively.

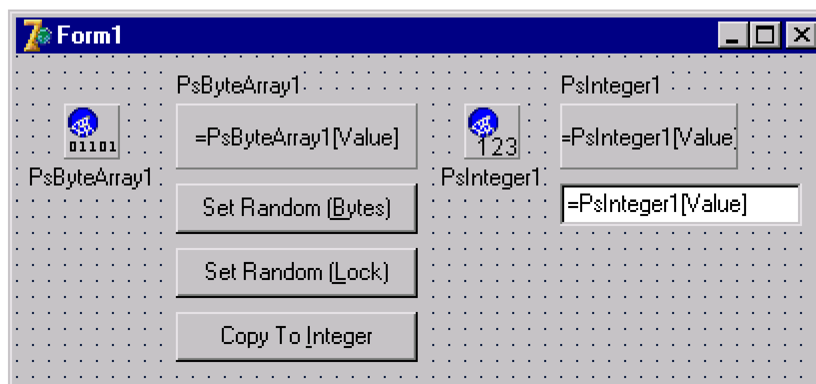


Figure 32. Sample application that uses a Byte Array variable.

Stage 5.2 Usage of ByteArrayConnector

Use `TPsByteArrayConnector` to build automatic parsing logic between native data type variables and byte array variables, instead of doing it all manually as in the previous sample.

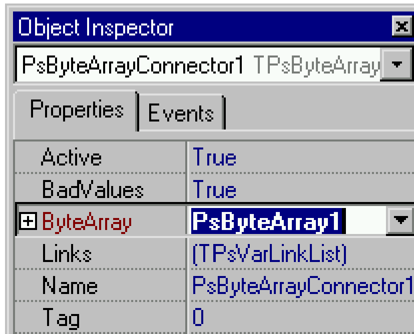


Figure 33. Properties of TPsByteArrayConnector.

You must first connect the component to a `TPsByteArray` as in Figure 33. Next, open the Links Editor to define the mapping between the byte array and the variables. To perform the same mapping as in the previous section, the links should look like in Figure 34.

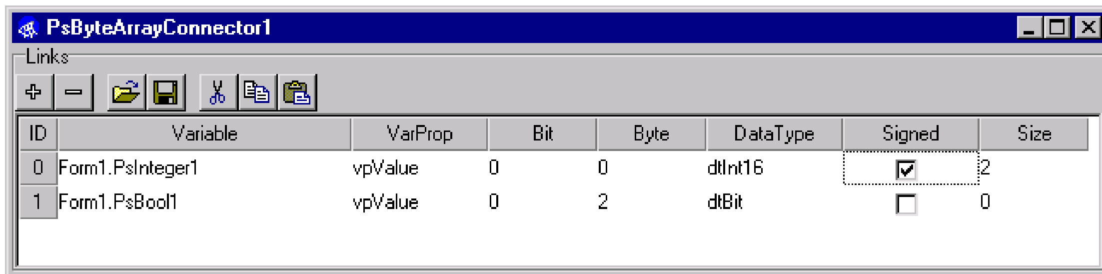


Figure 34. Linking ByteArray to other Sentrol variables.

Now, whenever the byte array data is changed, the change is reflected to the variables. The sample application uses `TPsByteArrayConnector.Active` to define when the connector is in use.

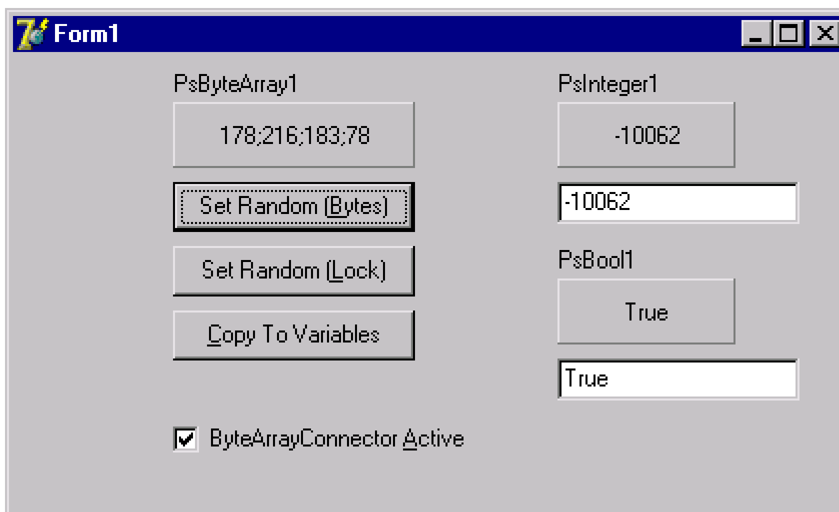


Figure 35. Sample application with ByteArrayConnector activated.

Lesson 6. Functions

The function components enable easy online analysis with Sentrol variables. You have a selection of predefined functions, and you have a couple of generic function components: TPsFunction and TPsParserFunction.

Stage 6.1 TPsFunction

TPsFunction is a generic component, which enables you to write your own function definition between any number of input variables and an output variable.

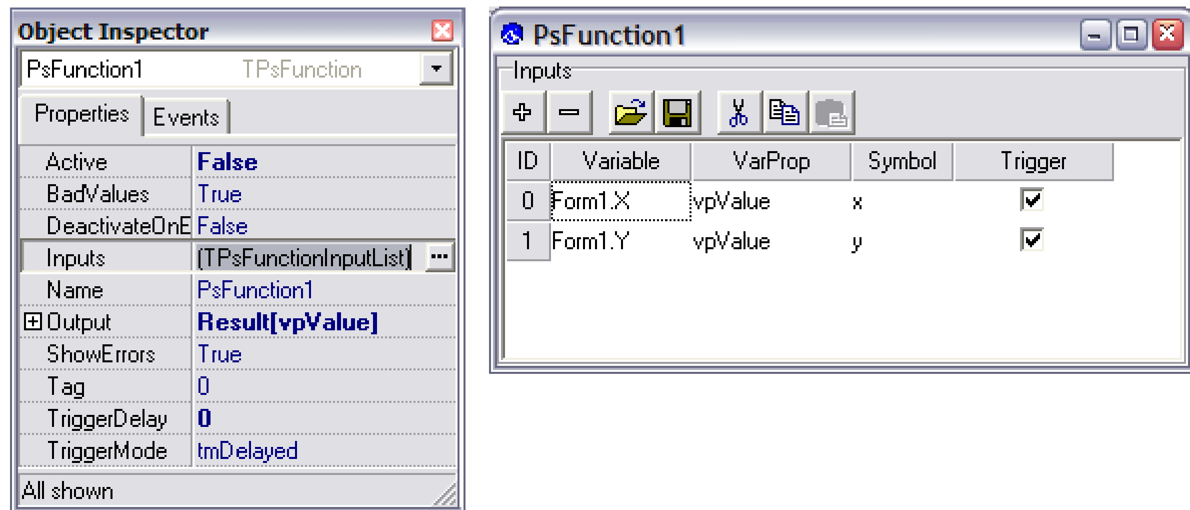


Figure 36. Properties of TPsFunction and Inputs.

Define the `Variable` and `VarProp` for `Output` and use the Inputs Editor to add the input links to the component. If you provide symbols for the links, you can refer to them using `TPsFunctionInputs.Values` (see the `OnCalc` example below). `TriggerMode` specifies when the function calculation is triggered: `tmDelayed` is suitable for most cases; it "waits" until all the inputs have changed[13] before the function output is recalculated. Using `tmManual` you specify that the output is calculated only when you say so, e.g.

```
procedure TForm1.CalcFunctionBtnClick(Sender: TObject);
begin
  PsFunction1.CopyToExternal(nil);
end;
```

TPsFunction requires that you define the algorithm in the `OnCalc` event handler, for example:

```
procedure TForm1.PsFunction1Calc(Sender: TPsCustomFunction;
  Inputs: TPsFunctionInputList; var OutputValue: Variant;
  var OutputQuality: TPsVarQuality);
begin
  OutputQuality := Inputs.WorstQuality;
```

[13] Actually, it dispatches a Windows message to itself, whenever an input changes. The calculation is triggered when the message is dispatched to the function. Effectively, it waits that all inputs that are going to change in reaction to a single operation, e.g. OPC Data Change, have changed before triggering the calculation.

```
  if OutputQuality <> vpqbad then
    OutputValue := Inputs.Values['x'] - Inputs.Values['y'];
end;
```

As you can see, it is important to check the quality of the inputs first and only calculate the result when appropriate!

Stage 6.2 TPsParserFunction

The Parser Function simplifies your job a little bit. It takes the free `TParser`[14] component in good use, and let's you define the `Expression` to calculate with the inputs already in the Object Inspector.

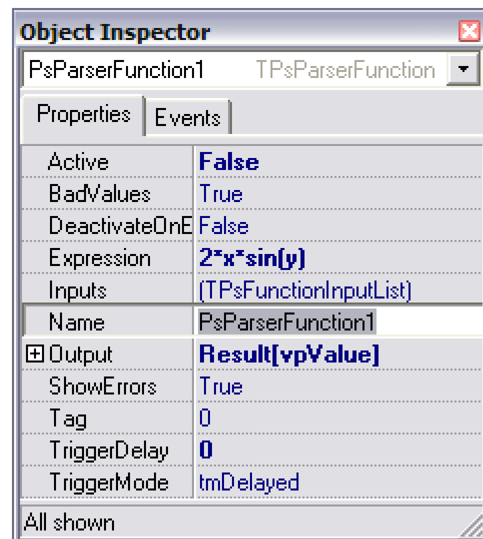


Figure 37. Parser Function let's you define the Expression in the Object Inspector.

Now, you don't need to use `OnCalc`, unless you need to further customize the result.

Stage 6.3 TPsSumFunction

Sum is one of the pre-built function components available in Sentrol. It simply overrides `TPsCustomFunction.CalcOutput` as follows:

```
procedure TPsSumFunction.CalcOutput(var Value: Variant;var Quality:
TPsVarQuality);
var
  sum: Double;
  i: Integer;
begin
  Quality := vpqBad;
  sum := -1.0;
  for i := 0 to Inputs.Count-1 do
    if Inputs[i].Quality <> vpqBad then
    begin
      if (Quality = vpqBad) then
      begin
```

[14] TParser is a free component, which is currently available from DATALOG's Delphi resources, http://www.datalog.ro/delphi/delphires.html, and included in Sentrol Tools directory.

```
        sum := Inputs[i].Value;
        Quality := Inputs[i].Quality;
      end
      else
      begin
        sum := sum + Inputs[i].Value;
        Quality := WorseQuality(Quality, Inputs[i].Quality);
      end;
    end;
  if Quality <> vpqBad then
    Value := sum;
end;
```

This way you can define your own function components, no matter how complicated they are…

Stage 6.4 The sample application

The sample application let's you select which f the functions is active – and try it out yourself.

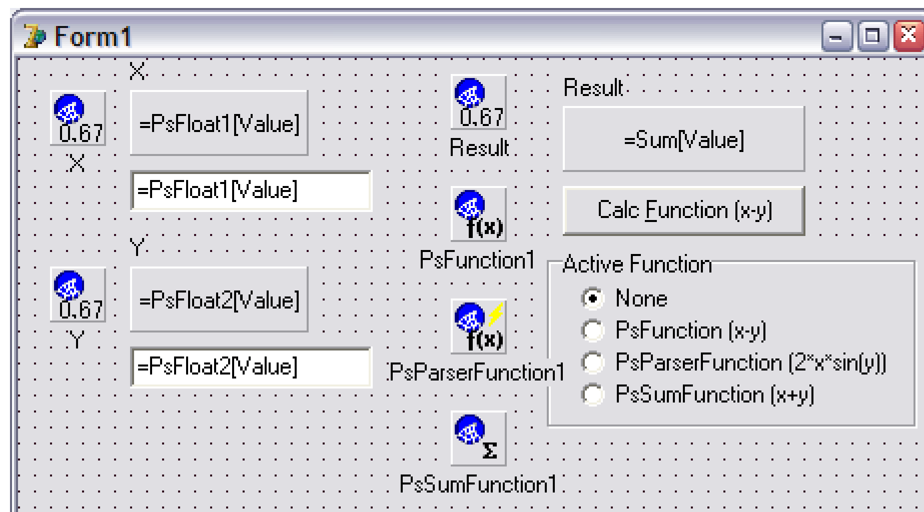


Figure 38. Function sample.

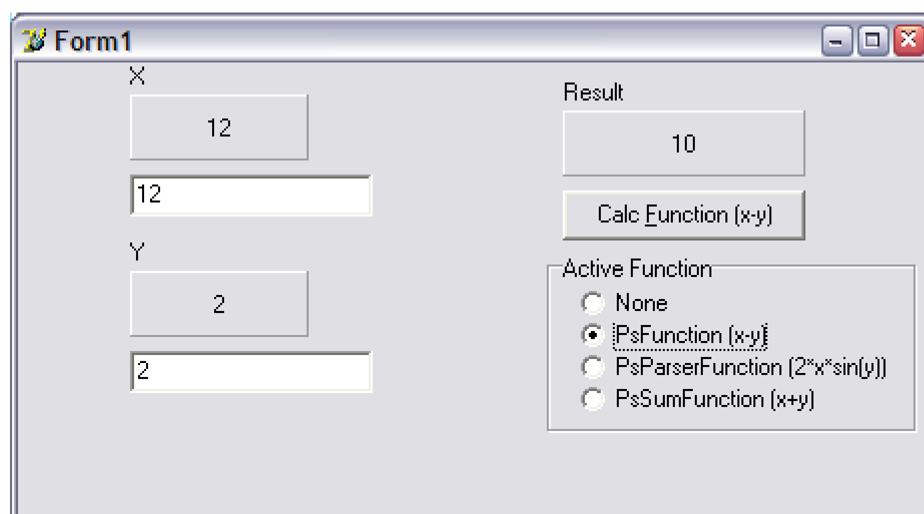


Figure 39. Sample in action.

Lesson 7. Animators

Sentrol Animators are used to create different animation effects according to variable values. There are different animators for different effects: position, size, color and visibility.

They all share the same principle: you connect the animator to a Sentrol Variable and to any GUI Control. The animator then modifies the properties of the Control according to the current value of the Variable as you have parameterized.

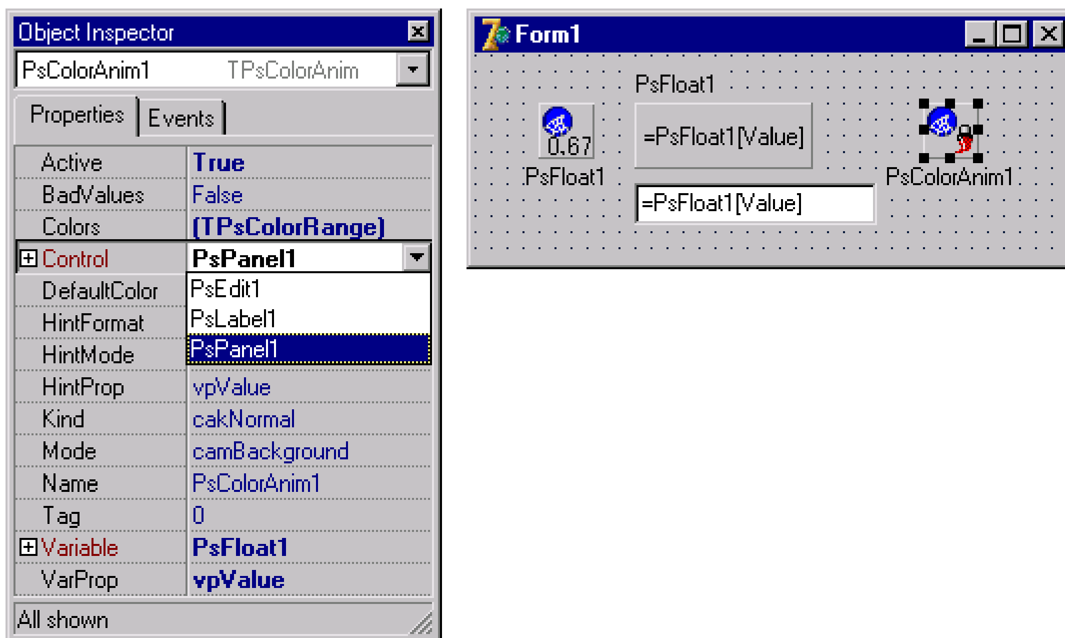


Figure 40. Connecting the animator to a GUI Control (PsPanel1). It is already connected to a Variable (PsFloat1). It will now modify the Control according to the current value of the Variable.

Stage 7.1 Color Animation

Coloring effects can be added easily with `TPsColorAnim`. You can define whether you want to affect the Background, Font or some other color property with `Mode`. If you select `camCustom`, you must define the coloring action in `OnSetColor`.

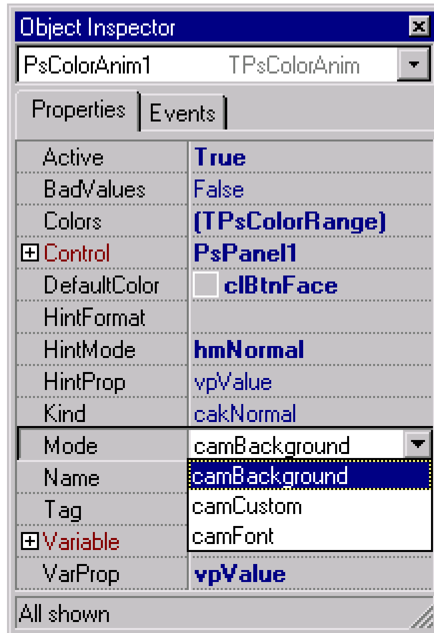


Figure 41. Selecting the Coloring Mode.

Use `Colors` to define the coloring range. Just add a number of entries to the collection and select the respective colors. Each color will be used as long as the variable value stays below the defined limit. In case the value is bad, the `DefaultColor` of the animator will be used. The last color in the range is applied to values above the limit as well.

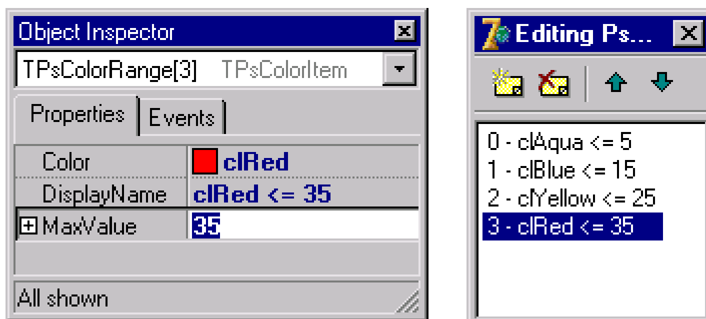


Figure 42. Defining the coloring range.

7.1.1 OnGetColor & Kind

In addition or instead of using the `Colors`, you can provide customized coloring rules with the `OnGetColor` event.

You can use `TPsColorAnim.Kind` to modify the effects a little bit: `cakAlarm` predefines the color range to suite for alarming purposes (and sets `VarProp` to `vpAlarmState`); `cakGradient` makes the coloring to find the color by interpolating

between the defined colors in the range, instead of changing in steps; `cakCustom` omits the `Colors` and only uses `OnGetColor`.

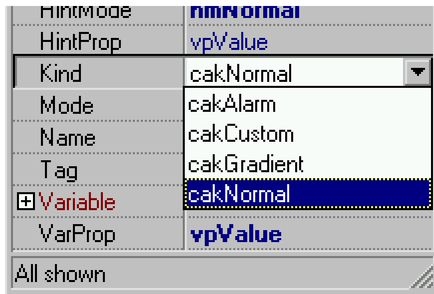


Figure 43. Kind specifies alternate behaviour.

Stage 7.2 Position & Size Animations

`TPsPosAnim` & `TPsSizeAnim` function quite similarly. They modify the position and size of the `Control` according to the rules. The default rule is a linear mapping between the variable range (`VarMax` & `VarMin`) and the position or size range of the `Control` (`PosMax`, `PosMin` / `SizeMax`, `SizeMin`).

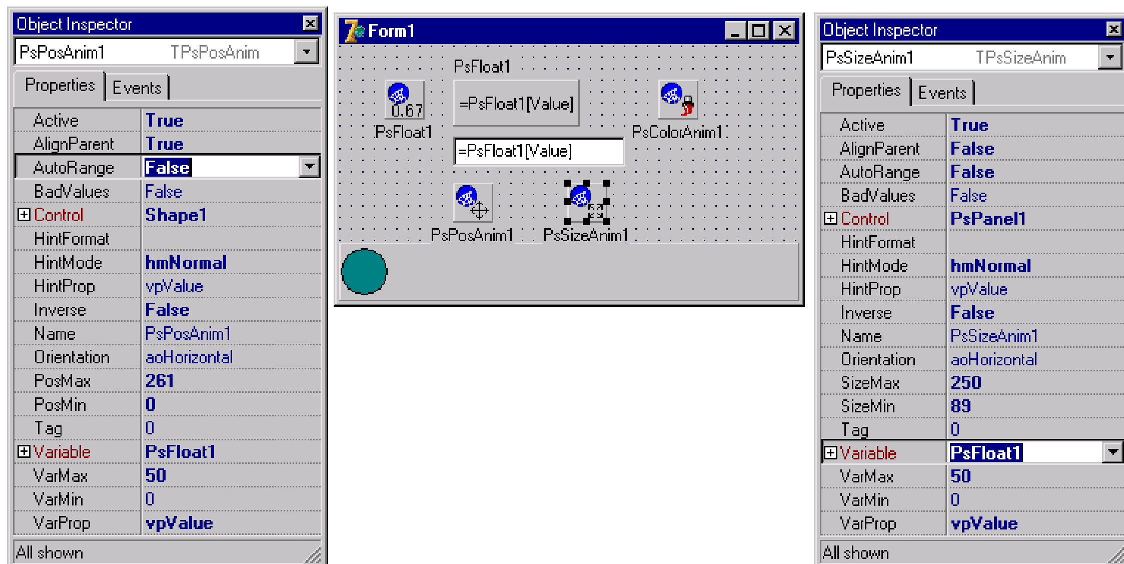
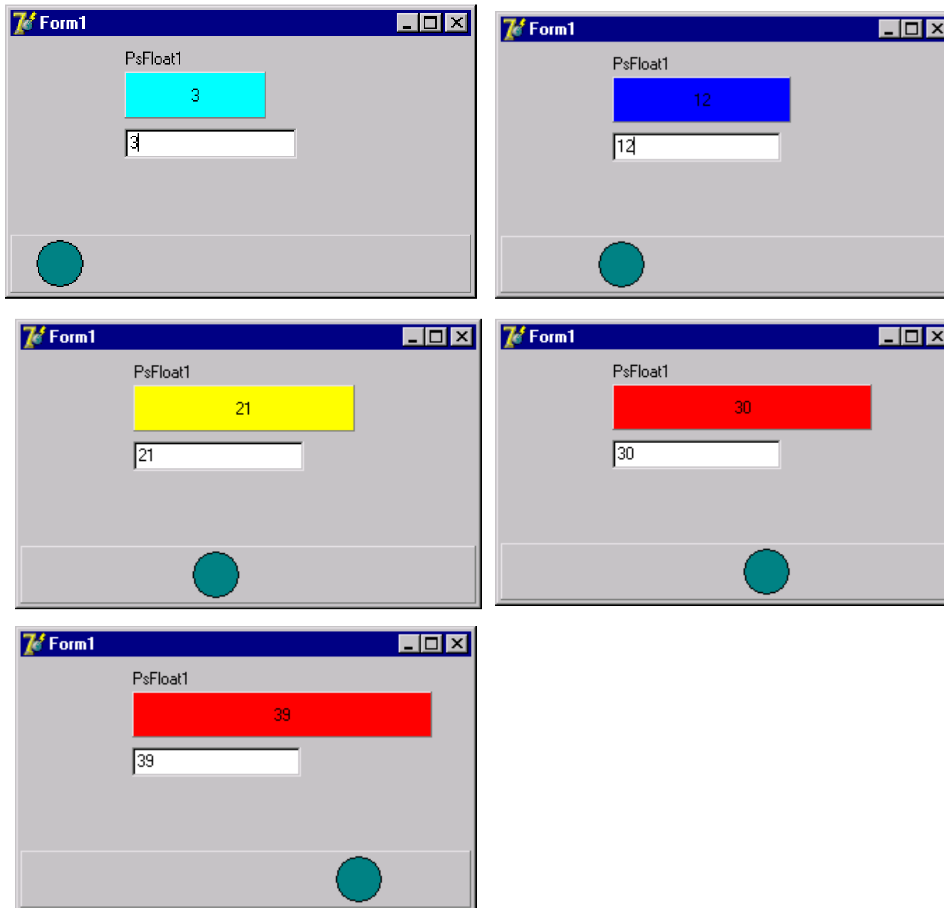


Figure 44. Position animator defined to move the circle Shape inside a panel – and size animator to resize the display panel.

`Orientation` and `Inverse` affect the direction of animation. `AlignParent` and `AutoRange` help to copy the ranges from the parent of the `Control` and the `Variable`, respectively.

Stage 7.3 The animation sample in action

The sample includes a color, position and size animator. You can see how they work in



Lesson 8. OPC Server

Sentrol includes OPC Data Access Server implementation in `TPsOPCProvider`. This is not a component, but an object that must be created in application initialization phase.

Stage 8.1 Adding OPC Server to your application

You will just need to create the OPC provider in your unit **`initialization`** as follows[15]:

```
uses PsOPCProvider;

…

var
  MyOPCProvider: TPsOPCProvider;

initialization
  MyOPCProvider := TPsOPCProvider.Create;
  with MyOPCProvider do
    begin
      AutoRun := True; // Default = True
      CLSID := '{CBC804CA-623B-4D2B-9E60-B54A55398EF5}';
      ServerName := 'My.OPC'; // ProgID
      Description := 'My OPC Server';
      Vendor := 'Me';
      VendorInfo := 'My Test Server';
      AddressSpace.IncludeProps := True; // Default=False
      Initialize;
    end;
finalization
  MyOPCProvider.Free; // Notifies Shutdown
```

You must provide the object the `CLSID` (use Shift-Ctrl-G to generate a new GUID in Delphi editor) and `ServerName` (the ProgID) – and call `Initialize`. That will register the component and make it available to COM Clients.

Also, remember to free the object at **`finalization`**: it will call the clients with `IOPCShutdown` callback.

You can now run the application and connect to it using an OPC Client.

Note: if you only wish to register the server and let it start automatically when the client connects to it, run it with the option '/regserver', e.g. 'Project1 /regserver'. To uninstall it, use '/unregserver'.[16]

---

[15] In C++, you don't have any initialization section, so you must just create the OPCProvider before the application is run. See the C++ version of the sample code included in the installation package.

[16] In Windows Vista, you must register the server with administrator rights. For example, run it once "as Administrator" from the Windows Explorer.

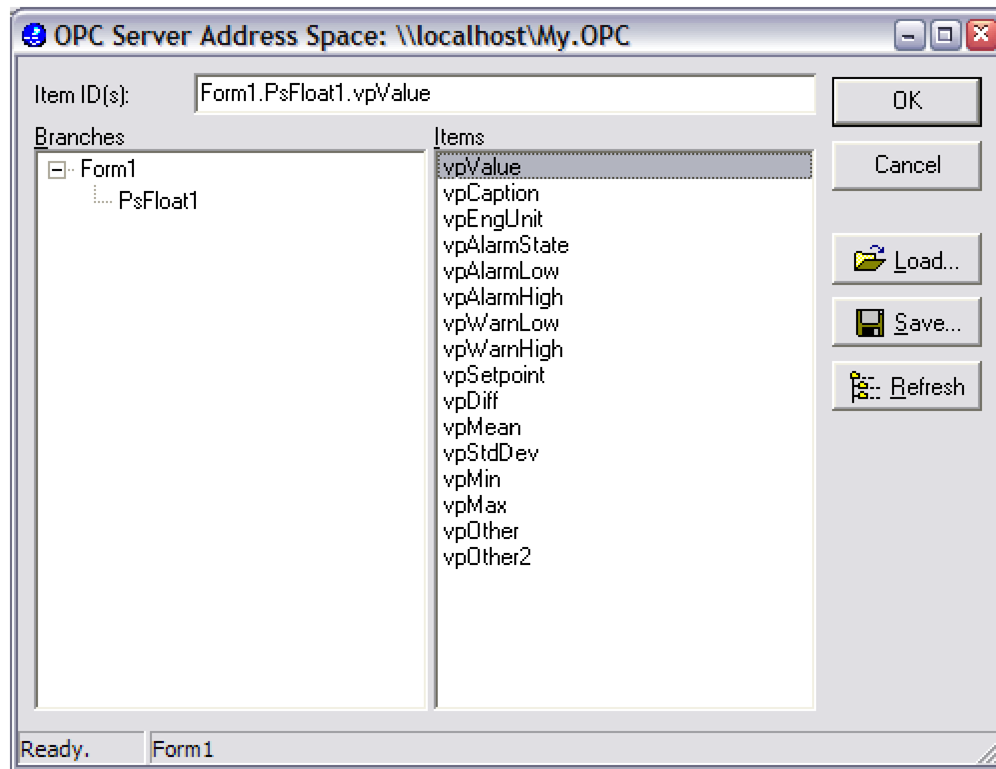


Figure 45. Address Space of the sample OPC Provider.

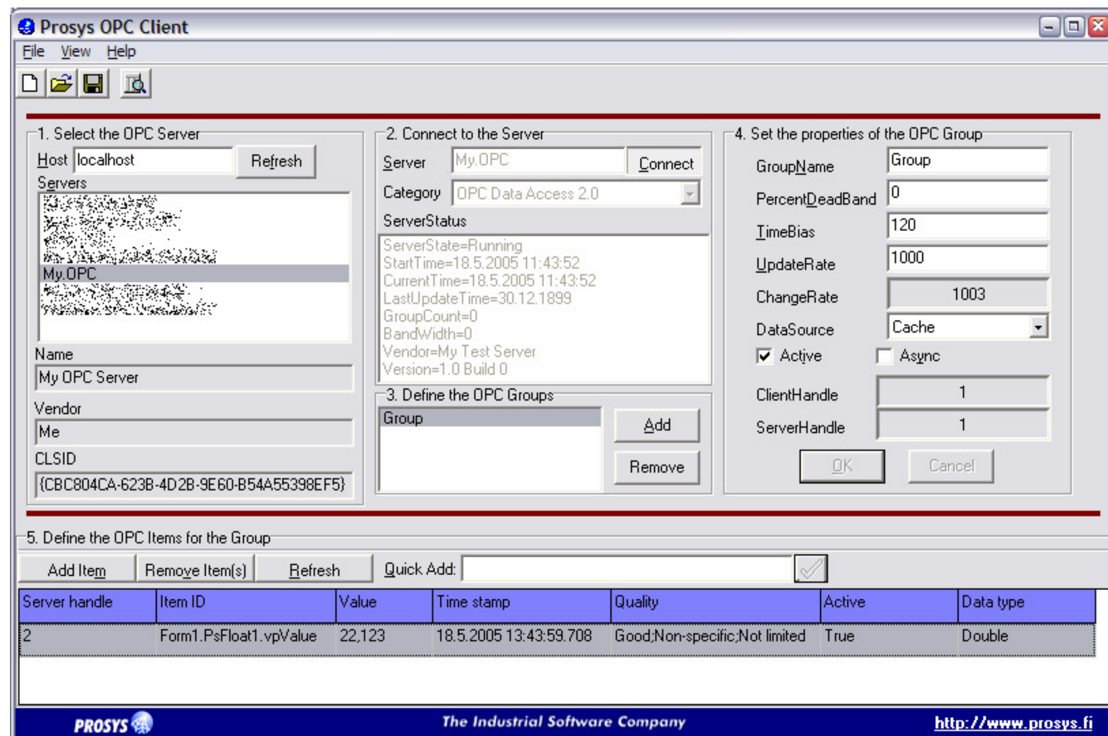


Figure 46. Connected to My OPC Server (with Prosys OPC Client).

Stage 8.2 Customization

In principle, the OPC provider will just publish all variables in the project as shown in Figure 45.

The Item Access Rights are set automatically depending on the selected VarProp, but they can also be overridden using `TPsOPCProvider.OnItemAccessRights`. You will need to create the event handler manually and connect it to `MyOPCProvider`, for example in `FormCreate`:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MyOPCProvider.OnItemAccessRights := MyOPCProviderItemAccessRights;
end;

procedure TForm1.MyOPCProviderItemAccessRights(Sender: TObject;
Variable:
    TPsVar; VarProp: TPsVarPropType; var AccessRights:
TPsOPCAccessRights);
begin
  if VarProp = vpValue then
    AccessRights := arReadOnly;
  // else use the default logic
end;
```

Alternatively, to guarantee that the clients cannot modify anything, you could just set:
`MyOPCProvider.ReadOnly := True;`

The server will be enabled immediately after startup for client connections. If you want to prepare your data first, you can set `AutoRun` to `False` and call `Run` yourself, when the time is appropriate. `AutoRun` tries to wait until all forms have been created, but it may not satisfy your needs. If the clients are allowed to connect to the server too early, the server address space may not contain everything, yet.

Note: After Sentrol 3.1, the address space should react dynamically to changes in the data modules. Even new data modules added to Application should be automatically added to the address space, if variable components are inserted in them. This functionality is controlled by the properties `AutoAddApplication` and `AutoUpdate` in the `AddressSpace`.

`AddressSpace.IncludeProps` is used to define whether the variable properties are shown in the address space. By default it is `False`, and the address space only includes the variables (no `vpValue`, etc. under that). The individual properties can always be accessed, though, as OPC item properties or just by adding the respective extensions to the variable IDs, e.g. `Form1.PsFloat1.`**`vpValue`**.

There are also a few other properties in `TPsOPCProvider` that affect how the server works. Check the respective documentation to learn about them.

8.2.1 Hierarchical address space

The OPCProvider will find all data modules and forms that are owned by Application – and create the address space according to that. But, it will also locate data modules and forms owned by those modules and forms also, so you can use the ownership hierarchy to initialize a deeper hierarchy, too. For example, if you initialize a SubModule under Form1 like this:

```
Application.CreateForm(TForm1, Form1);
SubModule := TSubModule.Create(Form1);
```

You will get an address space as in Figure 47.

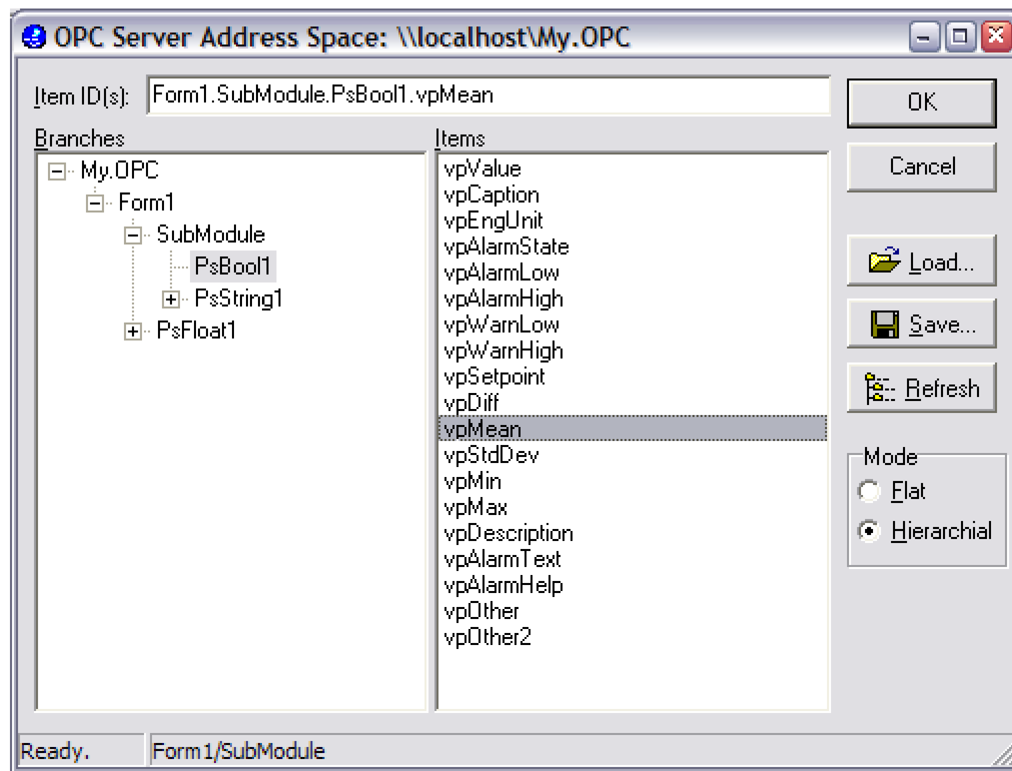


Figure 47. Hierarchical address space.

8.2.2 Fully customized address space

TPsOPCProvider.AddressSpace can be used to customize the address space contents even more. In order to create a customized hierarchy without the relation to Application object, you must first disable the default behavior:

```
  MyOPCProvider.AddressSpace.AutoAddApplication := False;
  MyOPCProvider.AddressSpace.AutoUpdate := False;
```

You can then add your modules and variables using the calls to

```
  RootIndex := MyOPCProvider.AddressSpace.AddModule(Self, -1,
'MyDevice');
  MyOPCProvider.AddressSpace.AddVariable(RootIndex, PsFloat1,
'Temperature');
```

You can also create new components or variables on the fly and add them to the address space:

```
  Pressure := TPsInteger.Create(nil);
  MyOPCProvider.AddressSpace.AddVariable(RootIndex, Pressure,
'Pressure');
```

You can also customize the item id separator:

```
  MyOPCProvider.AddressSpace.PathSeparator := '/';
```

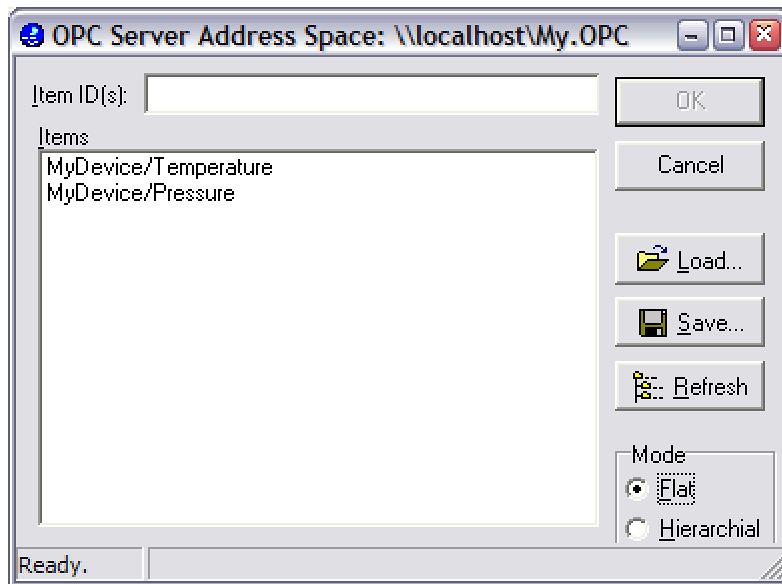


Figure 48. Fully customized address space in flat mode.

8.2.3 Simulation Server

You should also take a look at the Prosys OPC Simulation server, provided at http://www.prosys.fi/downloads.html. The full source code is included and all the material required for installing the servers. The simulation server demonstrates the use of `TPsOPCProviderForm`, which can be used to display client connections to your server.

Lesson 9. Creating components at run-time

Most of the lessons here have shown you how to create a project by defining the components at design time. Although this is very convenient, especially for small projects, it may become necessary quite soon to learn to configure your project dynamically at run-time.

Stage 9.1 Adding Links to Connectors

In principle, you can just create the same components and configure their properties the same way that you do at design time. One note is necessary, though: The `Links` property in the different connectors (as well as some other list type properties, such as `TPsState.States`) require that you cast the items to correct type. For example, if you add a new link to `TPsOPCConnector`, you need to cast the result to `TPsOPCVarLink`, in order to get access to all the properties

```
var
  Connector: TPsOPCConnector;
  Variable: TPsVar;
  Link: TPsOPCVarLink;
begin
  //…
  // TPsVarLinkList.Add is defined to return a generic TPsVarLink
  Link := Connector.Links.Add as TPsOPCVarLink;
  Link.ItemID := 'YourItemID';
  //…
end;
```

Stage 9.2 “Helper” objects

There are of course some objects that you can only access from the code at run-time. Take a look at the unit `PsClasses`, for example: you will find components such as `TPsStringList`, `TPsStringTable`, `TPsStringTree`, which are all used in the Sentrol internals, but you can also benefit from them yourself.

While working with Variables, you should consider using `TPsVarList` whenever you need to keep lists of your variables. If you `Attach` it to a Component (e.g. Data Module or Form), it will always automatically contain the variables owned by that component. Useful, isn’t it!

Please, consult the Sentrol Help for more about these objects.

Stage 9.3 Sample project

The sample project related to this lesson shows you how to create a number of OPC Connectors and Variables at runtime and how to link them. It also demonstrates the usage of standard string lists and `TPsVarList` to keep and show the components in GUI controls (using `Assign` methods).

Also note that the `Connector.Links` can be assigned to string lists and vice versa (if all the referred components just exist). And you can save and load them to files with `Links.SaveToFile` & `LoadFromFile` (or …`Stream`), respectively.

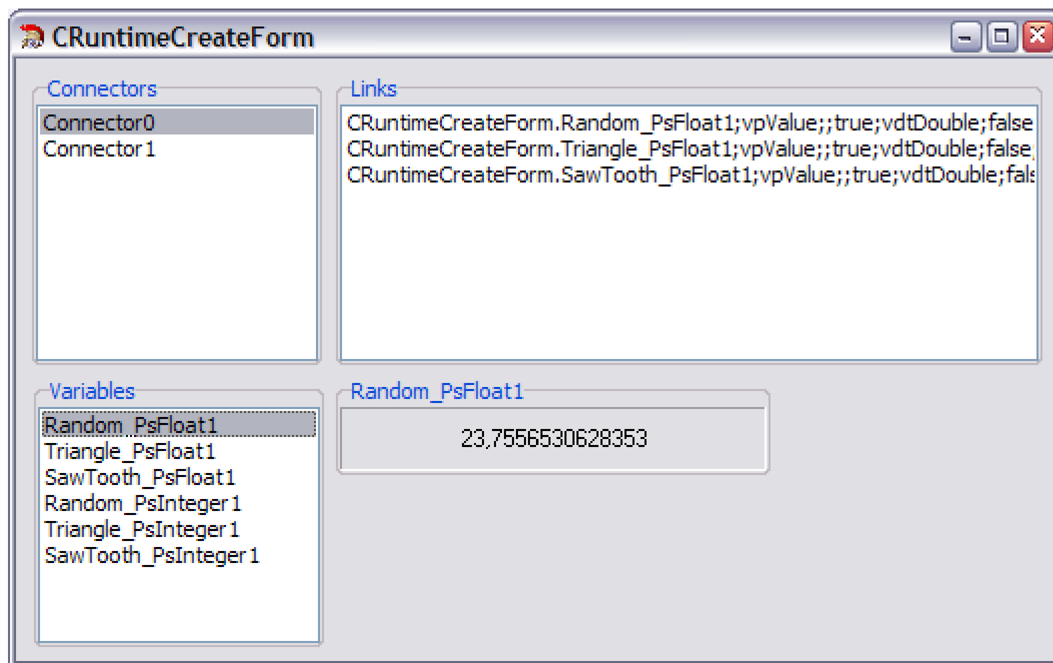


Figure 49. The run-time sample project (screen shot from the C++ version).