Description of Function

DoF CB Open Interface Helper

SUMMARY

This document describes the object model “CB Open Interface Helper”.

The object model is a COM-DLL and the purpose of the object model is to make it easy to write client applications for the “Control Builder Open Interface”.

The document contains a lot of examples and is suitable as user documentation.

Type des.			Part no.			
Prep.	XAACS / Anders Crilfe	2012-02-17	Doc. kind	Function Description		No. of p.
Appr.	/ Gerding Christer	2012-02-17	Title	DoF CB Open Interface Helper		90
Resp. dept	XAACS	Approved				
ABB AB			Doc. no.	3BSE033316	Lang. en / Rev. ind. E	Page 1

FILE: 3BSE033316_E_en_DoF_CB_Open_Interface_Helper.doc; SAVEDATE: 2012-02-17 15:22; TEMPLATE: Techn_Doc_Stand_P.dot C; SKELETON: 3BSE040221 en B

CONTENTS

1 INTRODUCTION

The object model is a COM-DLL. The purpose of the object model is to make it easy to write client applications for the “CB Open Interface”. The benefits of the model are:

1. The object model takes care of all XML details so that clients (other applications EXE’s) don’t have to have any knowledge about XML. The object model is much easier to use compared with an XML DOM tree.
2. The object model provides users with a powerful “easy to use” object model, where objects reflect the logical structure of the control builder.

Client applications written in almost any language on the Microsoft platform (C++, VB 6.0, C# and VB.NET, …) can use the object model.

The object model is only a complement (a helper) to the “CB Open Interface”. You don’t have to use the object model. The alternative is to use a standard XML parser.

A reader of this document is assumed to be familiar with the “CB Open Interface” specification, according to ref [2]. The reader is also assumed to have some knowledge about COM, object models, and to have some experience of programming languages such as C#, Visual Basic, C++ or Java.

2 REFERENCES

2.1 Related documents

Number	Document Identity	Document Title
[2]	3BSE033313	DoF CB Open Interface
[3]	3BSE030902	DoF Common elements and POU
[4]	ISBN 1-861004-99-0	Professional C#
[5]	ISBN 0-672-32170-X	.NET and COM
[6]	3BSE040467	DoF Hardware Libraries

2.2 Input documents

Ref	Document Identity	Document Title
[In1]	3BSE023764 Rev C	ATLAS 044 PRS
[In2]	3BSE027868 Rev I	Product Requirement Specification Control IT
[In3]	3BSE063596 Rev -	Product Requirement Specification Control IT 5.1 FPHI
[In4]	3BSE062617 Rev -	800xA System Software Architecture - Control

3 TERMINOLOGY

Terms as POU, Function block, Module, Program etc are described in reference [3].

4 GENERAL DESCRIPTION

4.1 Conformity with requirement specifications

Req. Spec. No.	Req. ID	Description
3BSE023764	ATL-300j	**Object model for convenient use**

Req Spec ID	Req Item ID	Headline and Definition	Specific Func ID
3BSE063596	PA-FCT-	DIAGRAM - OPEN INTERFACE	C5.3.43

	749.09	SUPPORT	C5.3.44 C5.3.45 C5.3.46 C5.3.50 C5.3.51 C5.3.52 C5.3.53
3BSE027868	PA-FCT-140.01.11	CONFIGURATION - DEFINITION OF COMMUNICATION VARIABLES	C5.3.47

4.2 Product perspective

The “CB Open Interface helper” is a COM-DLL. The COM-DLL, and an installation program, will be included on the “Control Builder” CD.

4.3 Safety perspective

Not relevant, non-SIL.

4.4 Short description of the object model “CB Open Interface Helper”

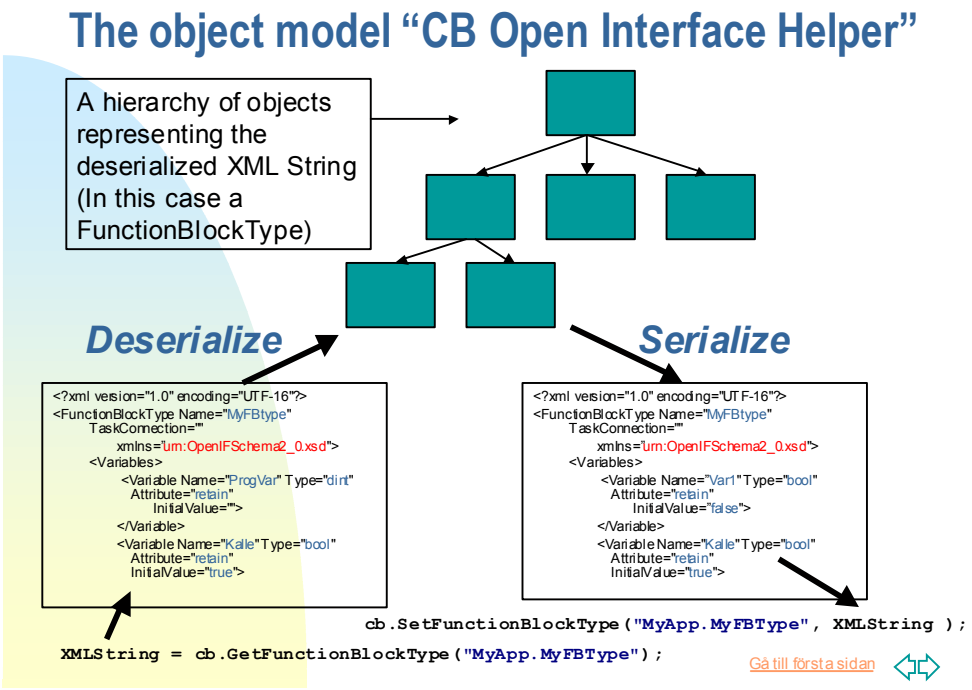


Figure1.

Figure 1 describes one typical usage of the object model and how it works together with the “CB Open Interface”

1. The “CB Open Interface” function “GetFunctionBlockType” is called. This function returns an “XMLString” describing the content of the type.
2. The object model is able to ***deserialize*** the XML String ***to a hierarchy of objects***, see figure 1. The objects are created in the client’s memory and have well known names such as “FunctionBlockType”, “Variable”, “Parameter” and so on.
3. The client application code can then use the objects i.e. append new objects, delete objects or modify the content of objects. See example 1.
4. The object model is able to ***serialize*** the hierarchy of objects to an “XMLString”.
5. The “CB Open Interface” function “SetFunctionBlockType” is called with the serialized “XMLString” as a parameter. The Control Builder professional EXE will now be updated according to the “XMLString”.

Example 1 shows a simple C# client application using both the “CB Open Interface” and the object model.

Example1.

```
private CONTROLBUILDERLib.CBOpenIF cb = null;
private CBOpenIFHelper.ObjectFactory ObjectFactory = null;

private void MyClient_Load(object sender, System.EventArgs e)
{
  try
  {
    // Create an object of the "CB Open Interface" class
    cb = new CONTROLBUILDERLib.CBOpenIF();
    // Create an ObjectFactory object.
    ObjectFactory = new CBOpenIFHelper.ObjectFactory();

    // Get an XML description of an existing function block type
    // from the Control Builder
    string XMLStr = cb.GetFunctionBlockType("MyLib.MyFBType");
    // Deserialize the XMLSTring into Objects
    FunctionBlockType fbType =
                        ObjectFactory.DeserializeFunctionBlockType(ref XMLStr);

    // Search for the variable named X and change the Variable's InitialValue
    Variable var = fbType.Variables.Find("X");
    if (var != null)
    {
      var.InitialValue = "10";
    }
    int nr = fbType.Variables.FindNr("Str");
    if (nr>0)
    {
      // Remove the Variable
      fbType.Variables.Remove(nr);
    }
    // Add a Parameter
    fbType.Parameters.Add2("MyParam", "Dint", "retain", DirectionValue.cbIn,
                           "7", "","", "Description of MyParam");
    // Add a FunctionBlock
    fbType.FunctionBlocks.Add1("RTC1","RTC");

    // Serialize the objects into an XMLString and update the
    // FunctionBlockType in the Control Builder
    string bucket = cb.SetFunctionBlockType("MyLib.MyFBType", fbType.Serialize());
  }
```

```
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

4.5 User characteristics

4.6 General constraints

There isn't any true object model for the function diagram code block language. Instead there is a possibility to describe the code block part as an XML String. For more information see chapter 5.3.44 How to create or modify a Function Diagram code block

4.7 Assumptions and dependencies

5 SPECIFIC FUNCTIONS

5.1 How do I learn about the object model?

The object model is huge and contains more than 100 classes. This specification only gives you an introduction into the subject. There are three great ways to learn more about the object model:

1. Use a graphical tool displaying the XML Schema and recall the fact that the object model is almost identical to the schema.
2. Use the "Object browser" of C# or VB 6.0. The browser shows all classes, all methods, and all parameters of the methods, all "enums" and so on.
3. Study the examples in this specification.

5.1.1 A Graphical tool presenting the object model

The classes and objects of the object model follow the XML Schema very closely. Fortunately, there exists graphical tools presenting schemas and a user-friendly HTML project named "CBOpenIFSchema3_0.html" is included on the Control Builder CD.

The following figure shows a tool displaying the schema for a "FunctionBlockType".

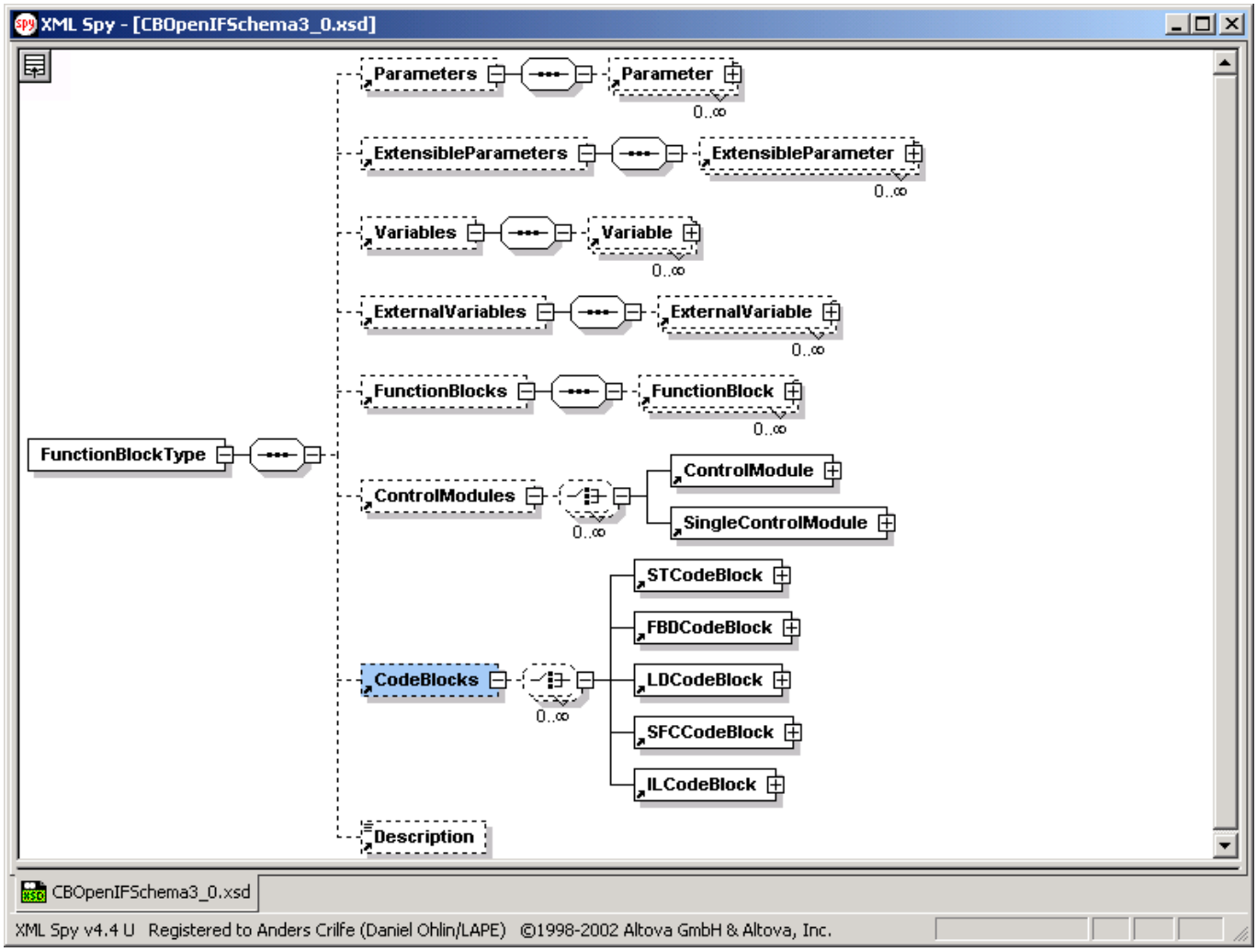


Figure 2.

The figure above shows the XML Schema for a “FunctionBlockType”. A graphical view of the object model class named “FunctionBlockType” would look the same. The “FunctionBlockType” class contains a “Parameters” object, a “Variables” object, a “CodeBlocks” object and so on. The class “Parameters “ is a collection of “zero to many” parameter object. A parameter object have the properties: “Name”, “TypeName”, “Direction”, “InitialValue” and so on according to figure 3.

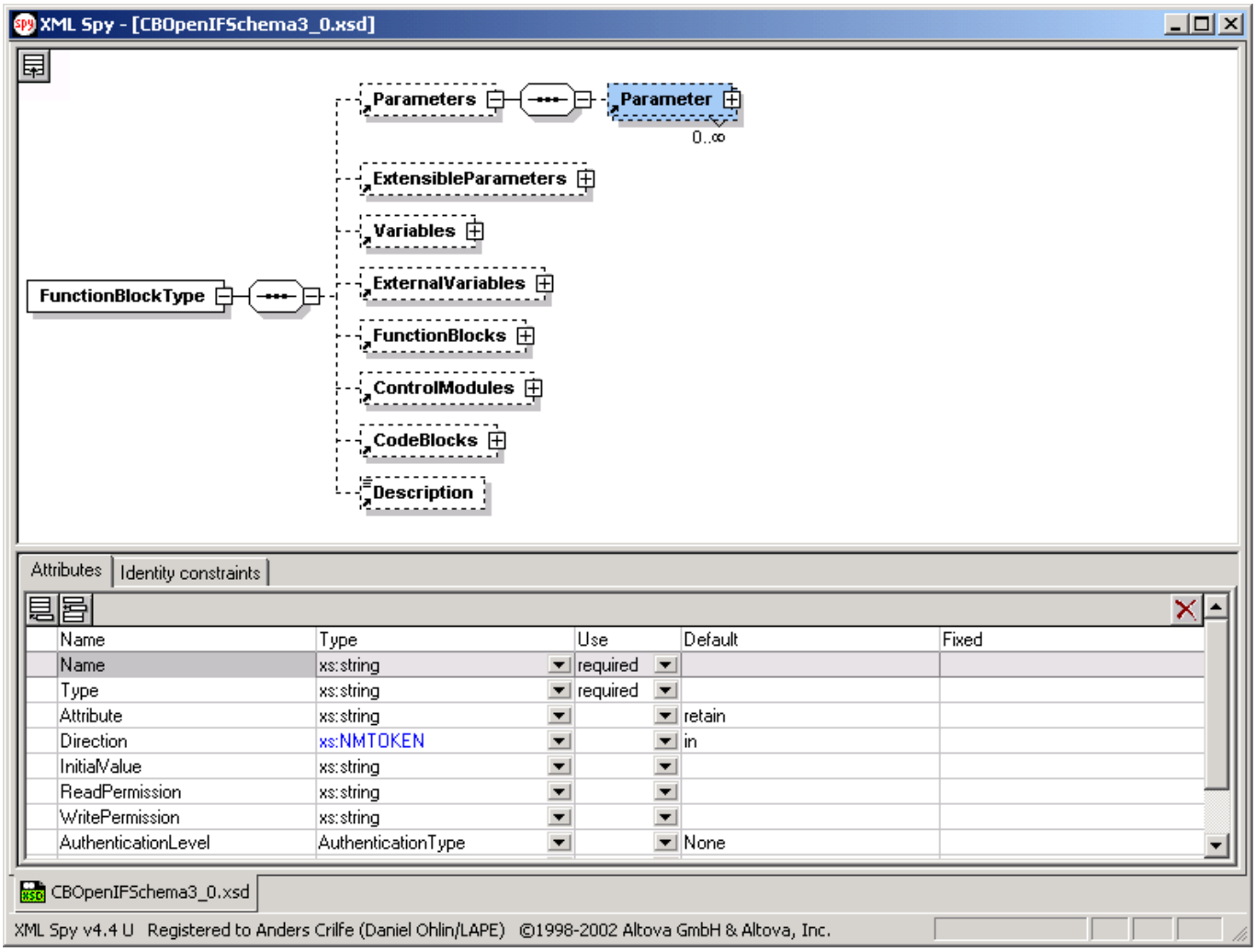


Figure 3.

Note! Although the classes and objects of the object model follow the XML Schema very closely some dissimilarities exists. One example is the “ProjectConstants” and “ProjectConstant” classes. An another example are the “SingleControlModuleType” and “SingleControlModuleInst” classes.

5.1.2 The “Object browser”

The “Object Browser” tool in Visual Studio .NET, and Visual Basic 6.0 IDE, shows all classes, all methods, all parameters of the methods, all “enums” and so on.

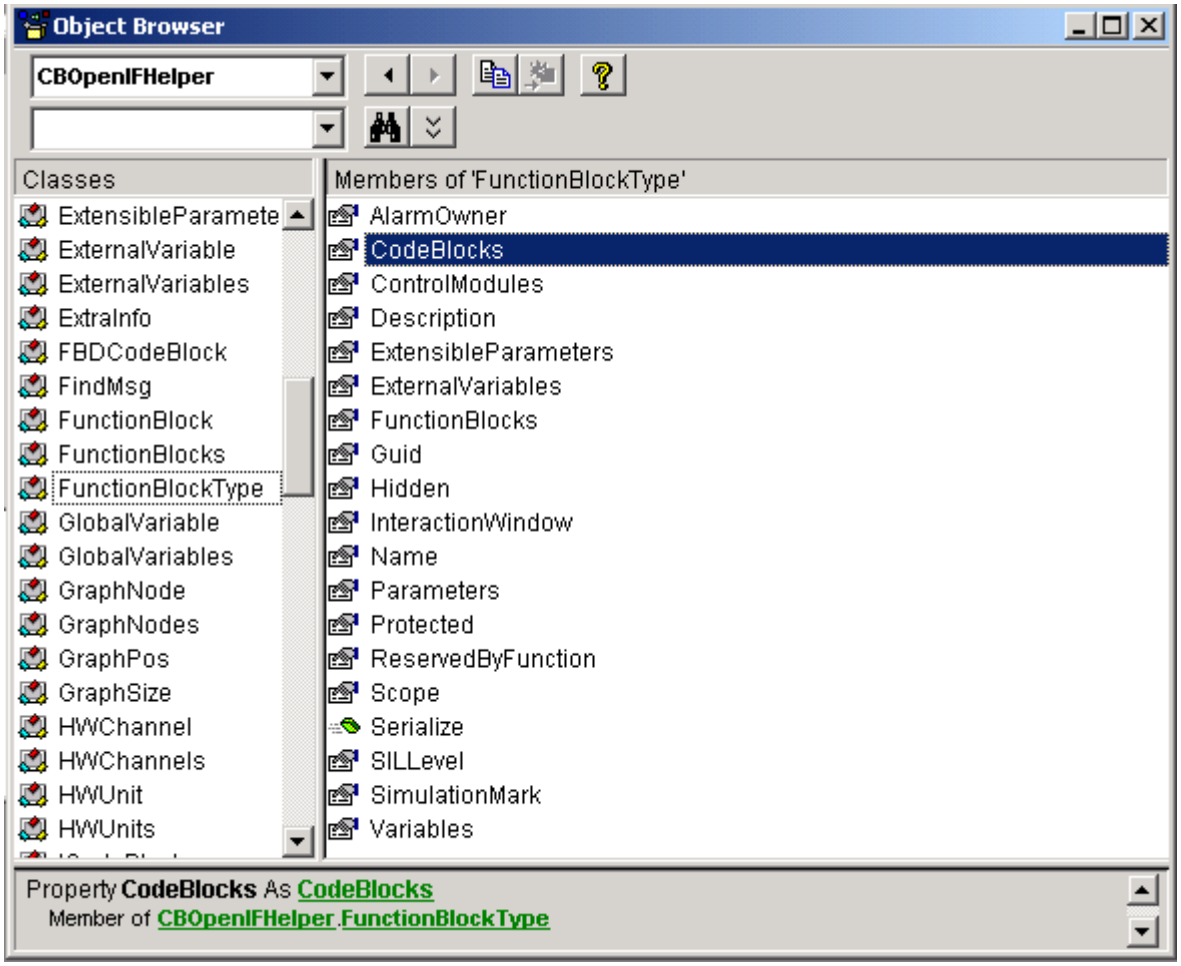


Figure 4. Shows the object browser tool in Visual Basic 6.0.

Figure 4 shows some of the classes of the object model. For instance, the class “FunctionBlockType” implements the “Serialize” function and the properties: “AlarmOwner”, “CodeBlocks”, …. , “Variables”.

5.2 General description of classes and interfaces

5.2.1 Class names

The classes and objects have well known names such as FunctionBlockType, Variable, Variables, Parameter, Parameters, CodeBlock, CodeBlocks and so on.

5.2.2 The “ObjectFactory” class

COM-classes don’t support parameterized constructors. The object model has one help class, called “ObjectFactory”, responsible for creating and initializing other objects. Example:

```
// First create an ObjectFactory object.
CBOpenIFHelper.ObjectFactory ObjectFactory = new CBOpenIFHelper.ObjectFactory();

// Create an object of the type "FunctionBlockType"
FunctionBlockType fbType = ObjectFactory.NewFunctionBlockType("MyFBType",
                                                    "Description of MyFBType");
```

The “ObjectFactory” often have several methods for creating objects of a certain class. For instance, objects of the “FunctionBlockType” class can be created by the **NewFunctionBlockType** method or by the **NewFunctionBlockType1** method. You can think of these methods as overloaded constructors. The difference is that the latter method has more parameters and thus gives you the opportunity to initialize the object more in detail. Please recall that COM doesn’t support overloading. For that reason the methods have to have different names. Example:

```
FunctionBlockType fbType1 = ObjectFactory.NewFunctionBlockType1("MyFBType",
                                        "Description of MyFBType", false, false,
                                         ScopeValue.cbPublic, "", true, "");
```

The “ObjectFactory” also contains ***deserialize*** methods. Such methods are able to ***deserialize*** an “XMLString” ***to a hierarchy of objects.*** Example:

```
 // Get an XML description of the HWUnit from the Control Builder
 string XMLString = cb.GetHardwareUnit("MyController.0.11.1", false);
 // Deserialize the XMLString to objects
HWUnit hw = ObjectFactory.DeserializeHWUnit(ref XMLString);
```

In the example above the “CB Open Interface” method **GetHardwareUnit** is called. This function returns the current content of the hardware unit **"MyController.0.11.1",** from the Control Builder EXE, and the result is stored in an “XMLString”. The **ObjectFactory.DeserializeHWUnit** is then called. This function builds a hierarchy of objects corresponding to the “XMLString” and returns a reference to the top-level object. The client application code can now use the objects. Example:

```
HWChannel ch = hw.HWChannels.Find("IW0.11.1.1");
// Change the connection to "Application1.Program1.Crilfe"
```

```
ch.ConVariable = "Application1.Program1.Crilfe";
ch.IODescription = "Description1";
```

5.2.3 Collection classes

The object model contains a lot of type safe collection classes. Some examples are: “Variables”, “Parameters”, “CodeBlocks”, “FunctionBlocks”, “ControlModules”, “HWUnits” and “HWChannels”.

Almost all collection classes have the “Add”, “Add1”, “Add2”, “AddBefore”, “Find”, “FindNr”, “Remove” functions and the “Count” property in common.

There are several “Add” functions named “Add”, “Add1”, Add2” and so on. The functions have different names because COM doesn’t support overloading. The difference is that some functions have more parameters and thus gives you the opportunity to initialize the object more in detail. Example:

```
FunctionBlockType fbType = ObjectFactory.NewFunctionBlockType("MyFBType",
                                                    "Description of MyFBType");

// Add some Varible objects
fbType.Variables.Add1("MyVariable", "bool");
fbType.Variables.Add2("X", "dint", "retain", "8", "", "", "Desc of X");
Variable var = fbType.Variables.Add1("Str", "string[32]");
var.Description = "Description of the variable";
var.ReadPermission = "My Read Permission";
```

The “Add” functions, in the example above, accomplish the following tasks:

1. Creates a “Variable” object.
2. Initializes the object according to the actual parameters.
3. Inserts the created “Variable” object into the collection.
4. Returns a reference to the created “Variable” object’s default interface.

Thus, the code above will create three “Variable” objects. The first variable object is given the name “MyVariable” and the type “bool”. The second variable object is given the name “X”, the type “dint”, the attribute “retain”, the initialvalue “8”, the ReadPermission “”, the WritePermission “”, and the description “Desc of X”.

The “Find” function returns a reference to an object in the collection (if found). Example:

```
Variable var = fbType.Variables.Find("MyVariable");
var.Description = "New Description of the variable";
```

The following example demonstrate the “FindNr” and “Remove” functions:

```
// Search for the Variable named "MyVariable" in the Variables collection
int Nr = fbType.Variables.FindNr("MyVariable");
// Remove this Variable from the collection
fbType.Variables.Remove(Nr);
```

All collections can be indexed, using [], just as an array. The index is 1-based i.e. the lowest index is 1. Example:

```
// All collections can be indexed, using [], just as an array.
for (int i=1; i<=fbType.Variables.Count; i++)
{
   // Print out the data, for one Variable, into the TextBox
   richTextBox1.Text += "Name: " + fbType.Variables[i].Name +
          " TypeName: " + fbType.Variables[i].TypeName + "\n" +
          "Attribute: " + fbType.Variables[i].CBAttribute +
          " InitialValue:" + fbType.Variables[i].InitialValue + "\n" +
          "Description: " + fbType.Variables[i].Description +
          " ReadPermission: " + fbType.Variables[i].ReadPermission +
          " WritePermission: " + fbType.Variables[i].WritePermission +"\n\n";
}
```

The collection classes implements the “**IEnumerable**” interface. The benefit is that the “foreach” statement of C# (and VB 6.0 and VB.NET) can be used in order to loop through all objects in the collection. Example:

```
//Use the foreach statement in order to loop trough all objects in
//the collection
foreach (Variable var1 in fbType.Variables)
{
   // Print out the data, for one Variable, into the TextBox
   richTextBox1.Text += "Name: " + var1.Name +
                    " TypeName: " + var1.TypeName + "\n" +
                    "Attribute: " + var1.CBAttribute +
                    " InitialValue:" + var1.InitialValue + "\n" +
                    "Description: " + var1.Description +
                    " ReadPermission: " + var1.ReadPermission +
                    " WritePermission: " + var1.WritePermission +"\n\n";

}
```

5.2.4 Polymorphism

Some of the collection classes contain objects of different types. One example is the “CodeBlocks” collection. This collection is able to hold objects of the following types: “STCodeBlock”, “FBDCodeBlock”, “LDCodeBlock”, “SFCCodeBlock” , “ILCodeBlock” and “FDCodeBlock”. See Figure 5.

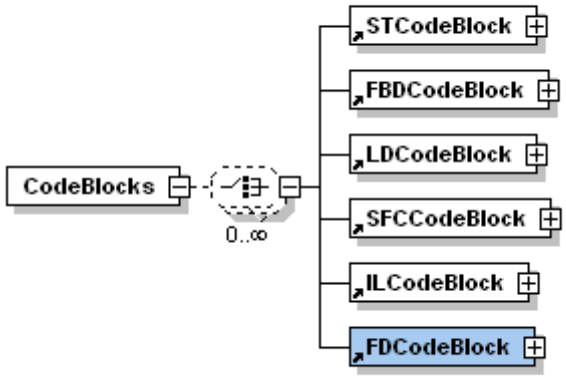


Figure 5. Note the “choice” symbol in the figure.

The important thing is to understand that the “CodeBlocks” collection only can hold objects of the types listed above. This behavior is achieved through ***polymorphism***. The classes: “STCodeBlock”, “FBDCodeBlock”, “LDCodeBlock”, “SFCCodeBlock”, “ILCodeBlock” and “FDCodeBlock” ***all implements an extra interface*** called **“ICodeBlock”** in addition to the default interfaces. The “CodeBlocks” collection class is still strong typed – it can only hold objects implementing the **“ICodeBlock”** interface.

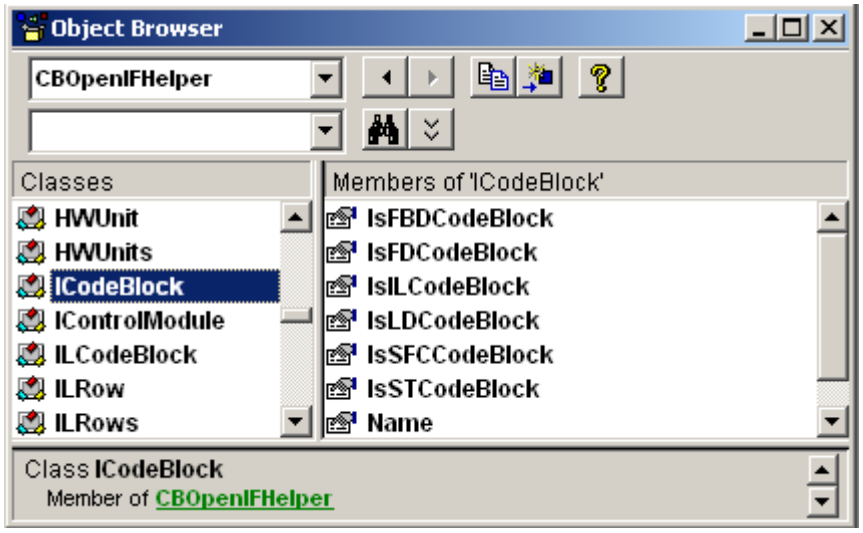


Figure 6.

The “SFCCodeBlock” class implements two interfaces: the “ICodeBlock” interface and the default “SFCCodeBlock” interface.

Figure 6 shows the members of the “ICodeBlock” interface i.e. the boolean properties: “IsFBDCodeBlock”, “IsILCodeBlock”, “IsLDCodeBlock”, “IsSFCCodeBlock”, “IsSTCodeBlock”, “IsFDCodeBlock” and the string property “Name”.

Please note that various properties and function, of the “CodeBlocks” collection, returns “ICodeBlock” references. You have to use this interface in order to investigate the objects type. When you know the type you can get a reference to the type’s default interface by means of a cast.

```
// Get an XML description of the program "MyProgram" from the
// Control Builder
string XMLStr = cb.GetProgram("MyApp.MyProgram");
// Deserialize the XMLSTring into Objects
Program prog = ObjectFactory.DeserializeProgram(ref XMLStr);

// Use the foreach statement in order to loop trough all objects in
// the CodeBlocks collection
foreach (ICodeBlock icb in prog.CodeBlocks)
{
   if (icb.IsSTCodeBlock)
   {
      //cast to the interface STCodeBlock
      STCodeBlock stcode = (STCodeBlock) icb;
      richTextBox1.Text += stcode.STcode;
   }
   else if (icb.IsSFCCodeBlock)
   {
      //cast to the interface SFCCodeBlock
      SFCCodeBlock sfccode = (SFCCodeBlock) icb;
      // work with the SFCCodeBlock's default interface
   }
   // else if ….. omitted in order to simplify the example
}
```

The code above loops through all objects in the collection. For each object a reference “icb” to the “ICodeBlock” is available. This interface is used in order to retrieve the type of the object. When the type is known the code query for the specific default interface of the object by means of casting the reference to the default interface.

The “CodeBlocks” collection is only one example of polymorphism. Another example is the “ControlModules” collection. The later collection class can hold objects of the types: “ControlModule” and “SingleControlModuleInst”. Both “ControlModule” and “SingleControlModuleInst” implements the “IControlModule” interface.

The table below shows all collection classes that can contain objects of different types.

Collection name	**Holds object implementing the interface**
CodeBlocks	ICodeBlock
ControlModules	IControlModule
MessageBucket	IMsg
SFCElements	ISFCElement
VAProtocols	IVAProtocol

5.3 Some C# examples

5.3.1 Getting started

You have to add a reference to the COM DLL “CB Open Interface Helper” before you can make use of the classes in the object model. Use the menu Project->Reference and a dialog will be shown.

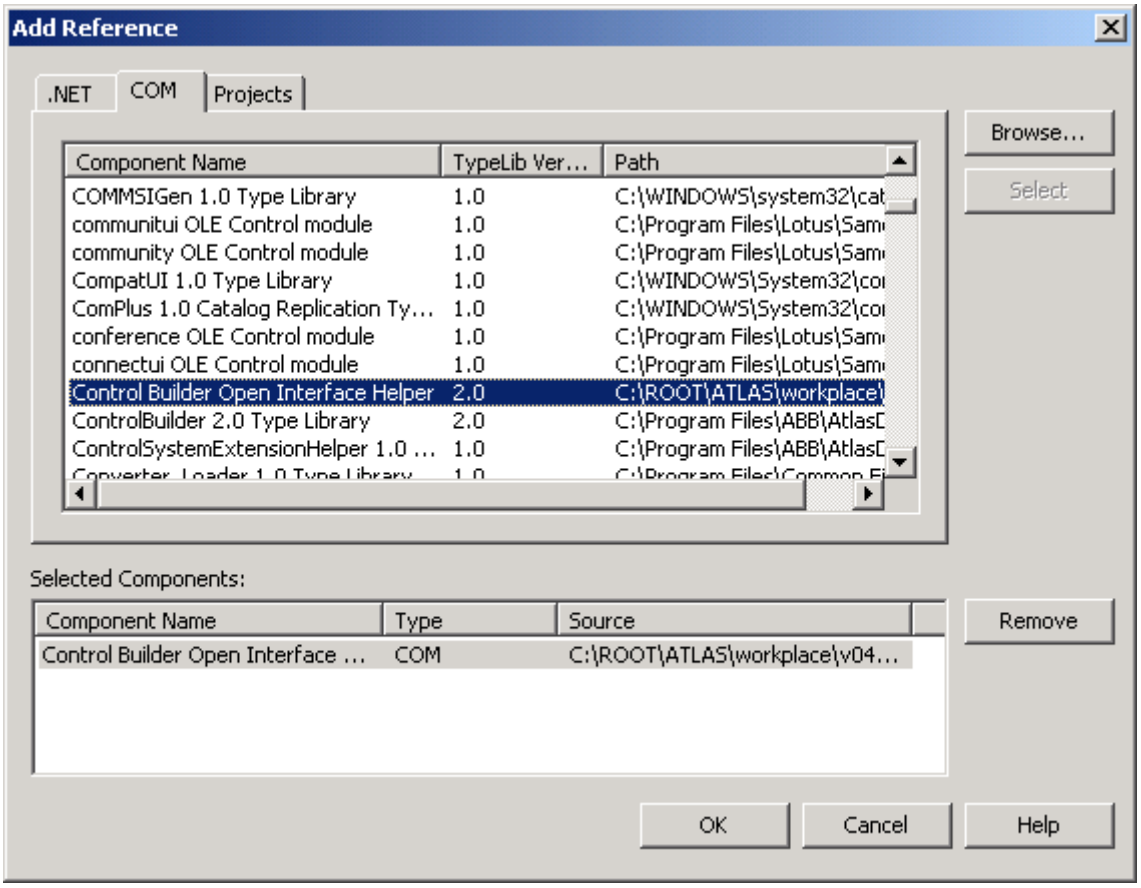


Figure 7.

Select the component (under the COM tab) and finish the dialog.

You also have to add a reference to the “Control Builder Professional” EXE in order to be able to use the “CB Open Interface” methods. Use the menu Project->Reference and the dialog will be shown again. Select the COM component “Control Builder 2.0 Type Library” in the dialog and press OK. Accept the offer to build a runtime callable wrapper for this classic COM component.

The Solution Explorer will now look like figure 8 and you are ready to use both the object model and the “CB Open Interface”. Try the “Object Browser” and you will be able to inspect all classes, all interfaces, all methods and so on.

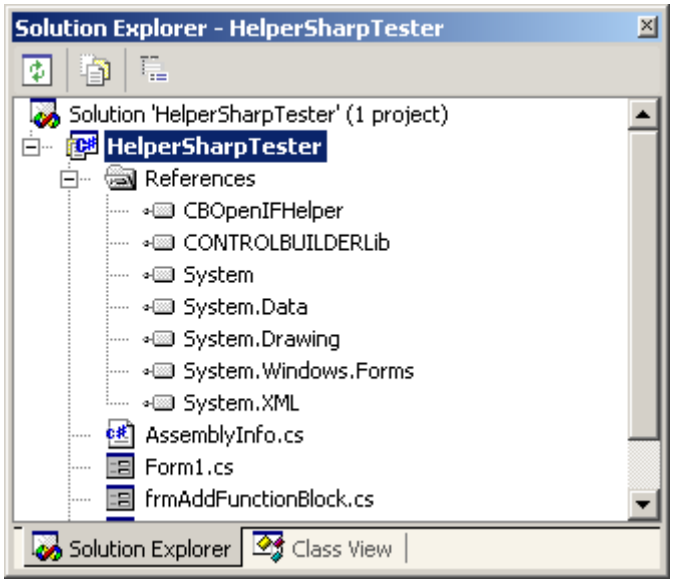


Figure 8.

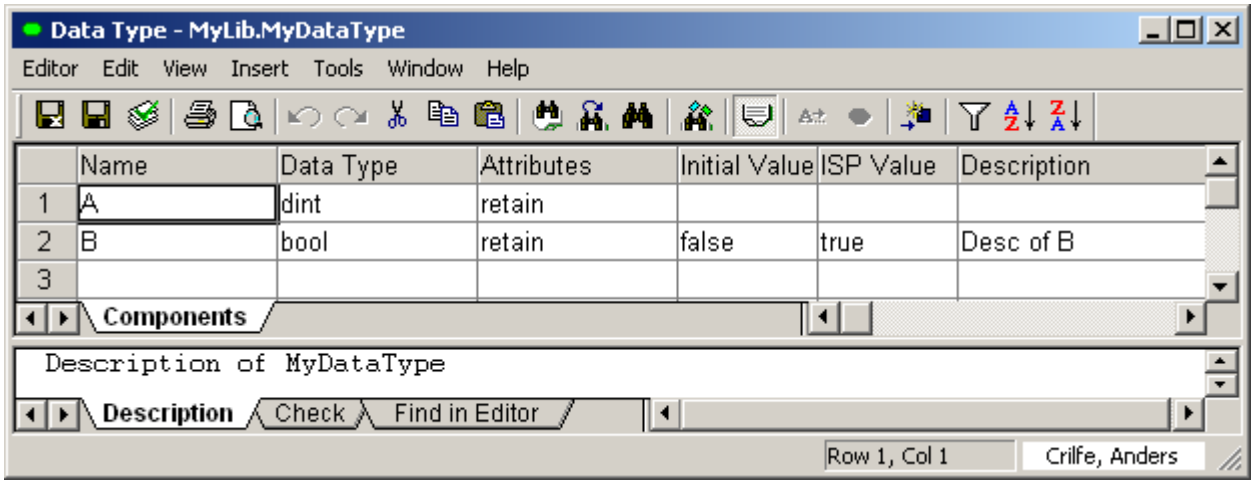
All examples below presume the presence of two reference variables.

```
private CONTROLBUILDERLib.CBOpenIF cb = null;
private CBOpenIFHelper.ObjectFactory ObjectFactory = null;
```

The first one is called “cb” (short for Control Builder) and is a reference to an object of the “CBOpenIF” type. The second one is called “ObjectFactory” and is a reference to an object of the “ObjectFactory” type. It is also assumed that the corresponding objects are created in a suitable function. The “Load” event function is a suitable place to create these objects for a project of the “WindowsApplication” type. See the code below.

```
using CBOpenIFHelper;
using CONTROLBUILDERLib;

namespace HelperSharpTester
{
  public class MyClient : System.Windows.Forms.Form
  {
    private CONTROLBUILDERLib.CBOpenIF cb = null;
    private CBOpenIFHelper.ObjectFactory ObjectFactory = null;

    private void MyClient_Load(object sender, System.EventArgs e)
    {
      ObjectFactory = new CBOpenIFHelper.ObjectFactory();
      cb = new CONTROLBUILDERLib.CBOpenIF();
    }

  }

}
```

5.3.2 Create a new DataType



Assume the task is to create a new data type in the library “MyLib” according to the following specification:

Figure 9.

```
bool Protected = false;
bool Hidden = false;
DataType dt = ObjectFactory.NewDataType1("MyDataType", "Description of MyDataType",
                                          Protected, Hidden, ScopeValue.cbPublic);
dt.Components.Add1("A","dint");
CBOpenIFHelper.Component comp1 = dt.Components.Add2("B", "bool", "retain", "false",
                                                    "Desc of B");
comp1.ISPValue = "true";
// Serialize the objects into an XMLString and create the DataType in
// the Control Builder EXE
string XMLBucket = cb.NewDataType(dt.Name, "MyLib", dt.Serialize());
```

5.3.3 Modify the content of an existing DataType

The task is now to modify the data type created in the previous example. The “A” component should be removed and two new components, “NewComp” and “C” should be added. The new content of the data type would be according to figure 10.

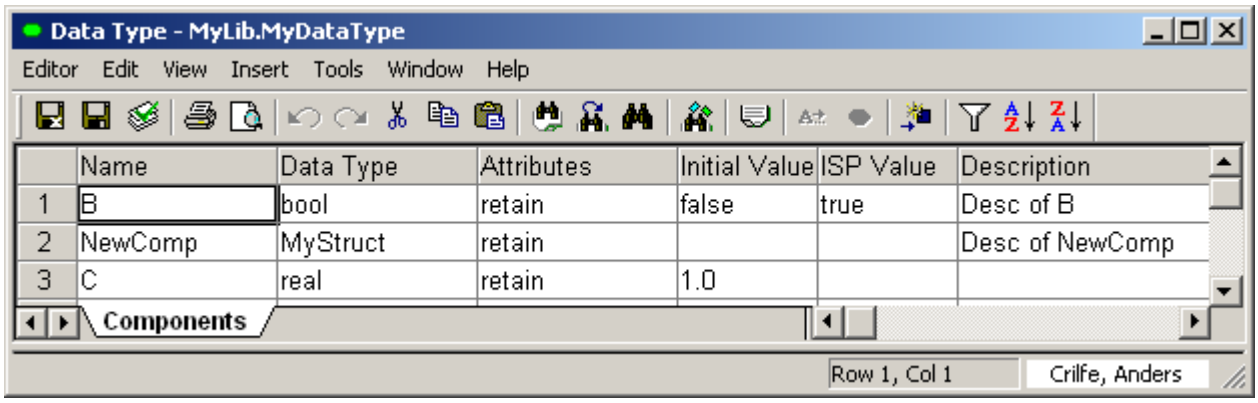


Figure 10.

```
// Get an XML description of the data type from the Control Builder
string XMLStr = cb.GetDataType("MyLib.MyDataType");
// Deserialize the XMLString into Objects
DataType dt = ObjectFactory.DeserializeDataType(ref XMLStr);
// Search for a Component named "A"
int nr = dt.Components.FindNr("A");
if (nr>0)
{
  // Remove the found Component
  dt.Components.Remove(nr);
}
// Add new components
dt.Components.Add2("NewComp", "MyStruct", "retain", "", "Desc of NewComp");
dt.Components.Add2("C", "real", "retain", "1.0", "");
// Serialize the objects into an XMLString and update the DataType in
// the Control Builder
string bucket = cb.SetDataType("MyLib.MyDataType", dt.Serialize());
```

5.3.4 List the components of an existing DataType

The task is now to display the components, of the data type created in the previous examples, in an edit box. The result would look like figure 11.

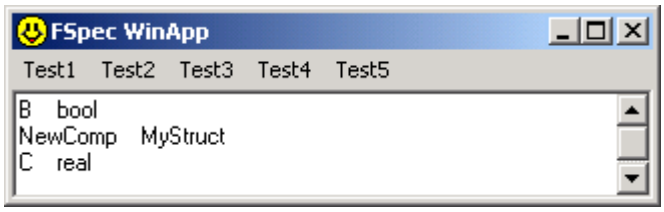


Figure 11.

```
// Get an XML description of the data type from the Control Builder
string XMLStr = cb.GetDataType("MyLib.MyDataType");
// Deserialize the XMLString into Objects
DataType dt = ObjectFactory.DeserializeDataType(ref XMLStr);
foreach (CBOpenIFHelper.Component cp in dt.Components)
{
   richTextBox1.Text += cp.Name + "    " + cp.TypeName + "\n";
}
```

Note! We have to use the namespace name “CBOpenIFHelper” in order to qualify the “Component” type due to name conflicts.

The code above made use of the “foreach” statement in order to loop through all objects in the “Components” collection. An alternative is to use the index operator. Example:

```
for (int i=1; i<= dt.Components.Count; i++)
{
   richTextBox1.Text += dt.Components[i].Name + "    " +
                        dt.Components[i].TypeName + "\n";
}
```

5.3.5 Create a new FunctionBlockType

Assume the task is to create a new function block type in the library “MyLib” according to figure 12.

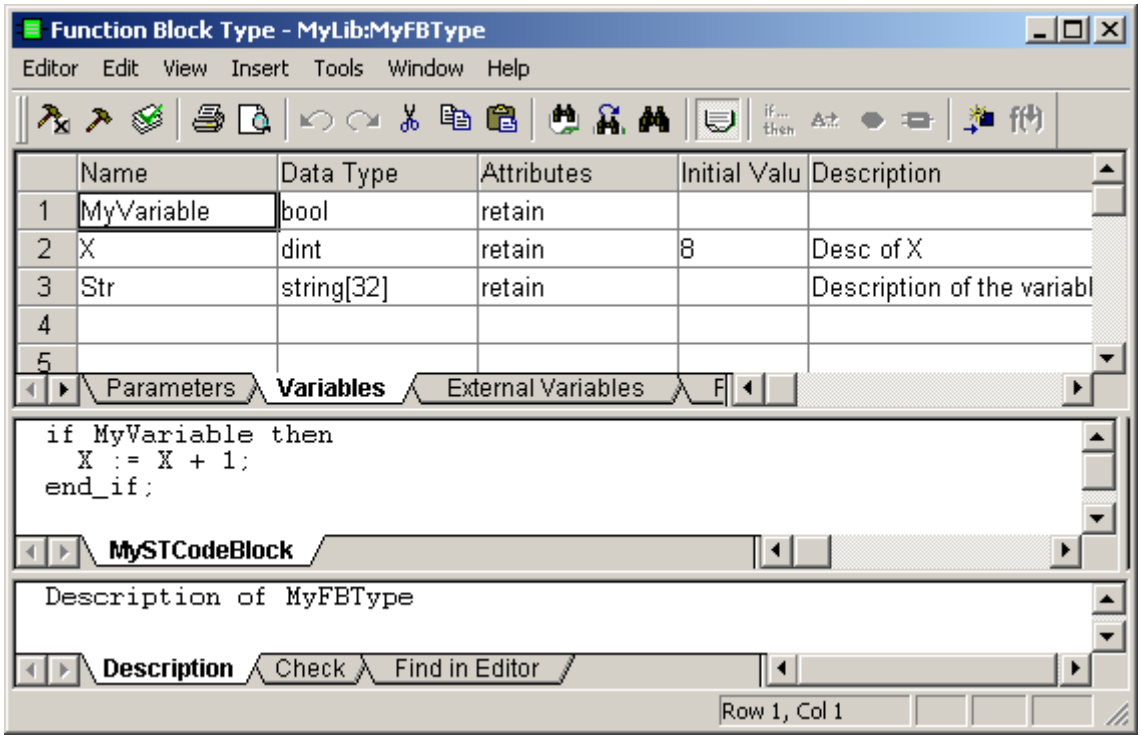


Figure 12.

```
// Create an object of the type "FunctionBlockType" in the client's
// process memory
FunctionBlockType fbType = ObjectFactory.NewFunctionBlockType("MyFBType",
                                                    "Description of MyFBType");

// Add some Variable objects
fbType.Variables.Add1("MyVariable", "bool");
fbType.Variables.Add2("X", "dint", "retain", "8", "", "", "Desc of X");
Variable var = fbType.Variables.Add1("Str", "string[32]");
var.Description = "Description of the variable";


// Add a ST CodeBlock
string stCode = "if MyVariable then\n" +
                "  X := X + 1;\n" +
                "end_if;";
fbType.CodeBlocks.AddSTCodeBlock2("MySTCodeBlock", ref stCode);

// Finally, serialize the object model into an XML String and
// call the OpenIF method "NewFunctionBlockType" in order to create the type
// in the Control Builder EXE.
string bucket = cb.NewFunctionBlockType(fbType.Name,"MyLib", fbType.Serialize());
```

5.3.6 Modify the content of an existing FunctionBlockType

The task is now to modify the function block type created in the previous example.

First the initial value of “X” should be changed to 10 and the variable “Str” should be removed. The variable list should look like the following figure.

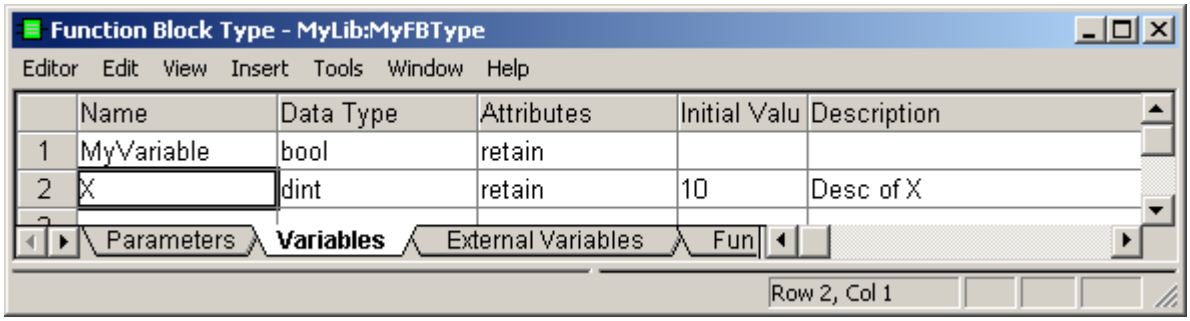


Figure 13

Second a new parameter, according to the figure below, should be added.

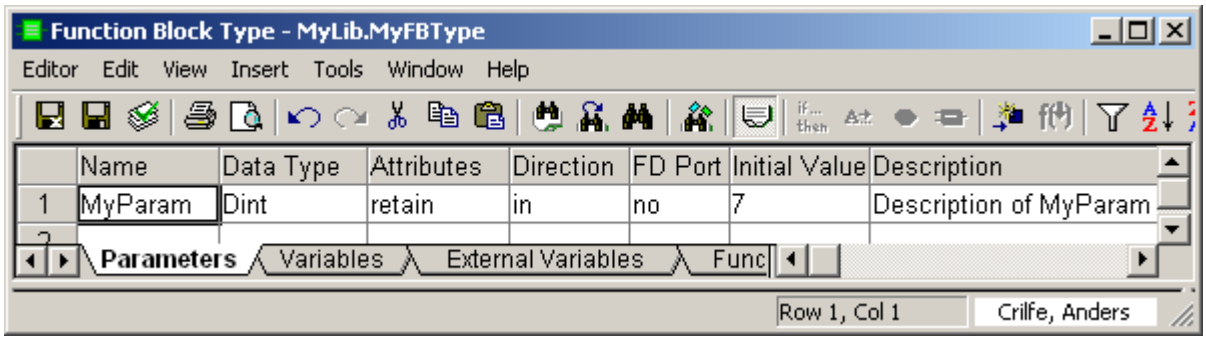


Figure 14

And a new function block, according to the figure below, should be added.

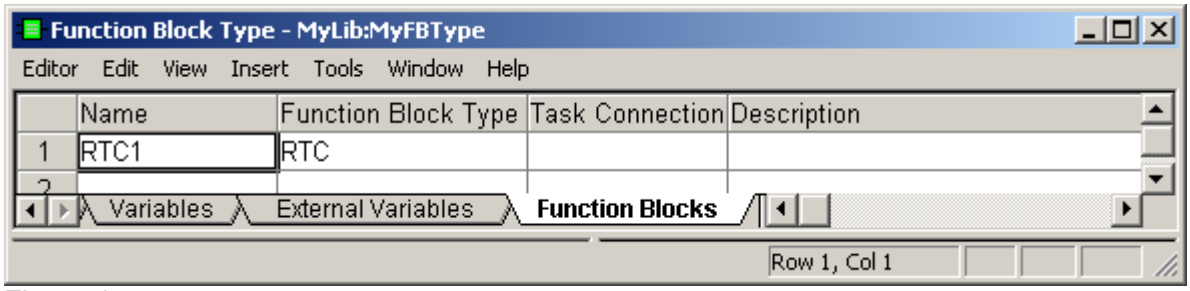


Figure 15

```
// Get an XML description of the type from the Control Builder
string XMLStr = cb.GetFunctionBlockType("MyLib.MyFBType");
// Deserialize the XMLSTring into Objects
FunctionBlockType fbType = ObjectFactory.DeserializeFunctionBlockType(ref XMLStr);

Variable var = fbType.Variables.Find("X");
if (var != null)
```

```
{
   var.InitialValue = "10";
}
int nr = fbType.Variables.FindNr("Str");
if (nr>0)
{
   // Remove the Variable
   fbType.Variables.Remove(nr);
}
// Add a Parameter
Parameter par = fbType.Parameters.Add2("MyParam", "Dint", "retain",
                        DirectionValue.cbIn, "7", "","", "Description of MyParam");
par.FDPort = "no";
// Add a FunctionBlock
fbType.FunctionBlocks.Add1("RTC1","RTC");

// Serialize the objects into an XMLString and update the
// FunctionBlockType in the Control Builder
string bucket = cb.SetFunctionBlockType("MyLib.MyFBType", fbType.Serialize());
```

5.3.7 Add a new Code Block to an existing type

The task is to add a new SFC Code Block to the “FunctionBlockType” created in the previous example. The SFC code should be implemented according to figure 16.

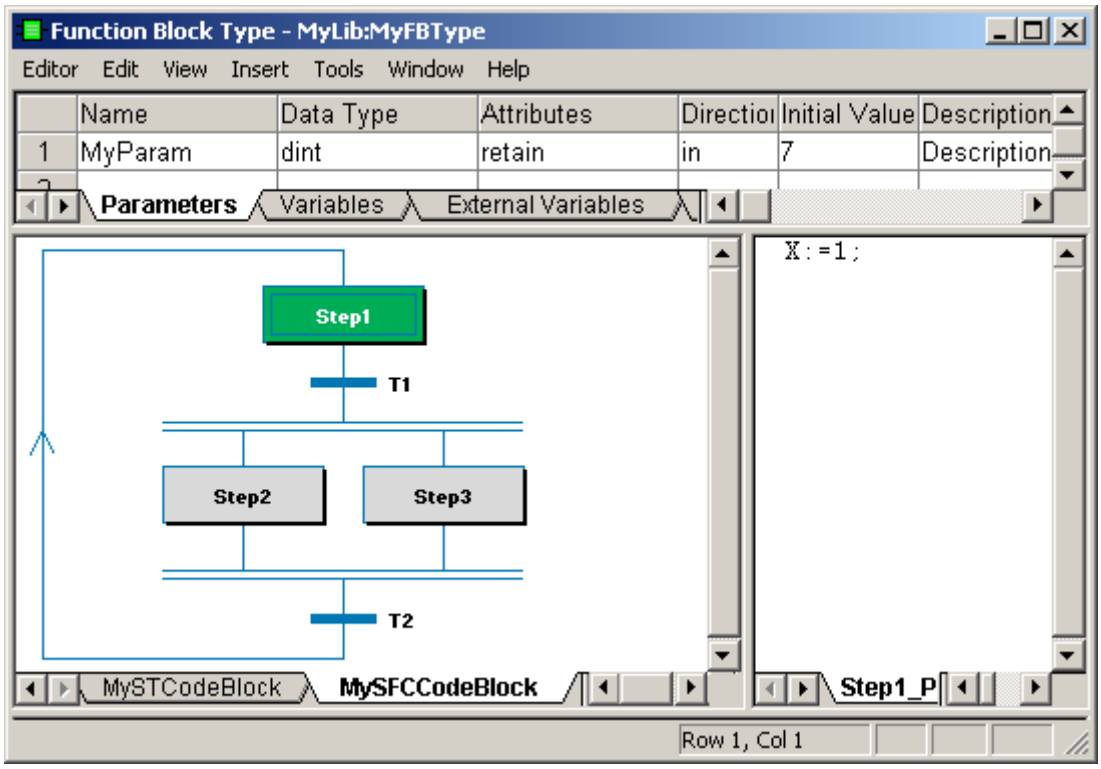


Figure 16.

We can solve this problem in two different ways. Both alternatives are presented below.

Alternative 1.

```
// Get an XML description of the type from the Control Builder
string XMLStr = cb.GetFunctionBlockType("MyLib.MyFBType");
// Deserialize the XMLString into Objects
FunctionBlockType fbType = ObjectFactory.DeserializeFunctionBlockType(ref XMLStr);
// Add a SFC CodeBlock
SFCCodeBlock sfccodebl= fbType.CodeBlocks.AddSFCCodeBlock1("MySFCCodeBlock");
sfccodebl.SFCElements.AddStep2("Step1", true, "X:=1;", "X:=X+1;", "");
sfccodebl.SFCElements.AddTransition2("T1", "X>100", "");
SFCSimultaneous sim = sfccodebl.SFCElements.AddSimultaneous1(2); // 2 branches

sim.SFCBranches[1].SFCElements.AddStep2("Step2",false,"","X:=X-1;","");
sim.SFCBranches[2].SFCElements.AddStep2("Step3",false,"","X:=X-1;","");
sfccodebl.SFCElements.AddTransition2("T2", "X<50", "");

// Serialize the objects into an XMLString and update the
// FunctionBlockType in the Control Builder
string bucket = cb.SetFunctionBlockType("MyLib.MyFBType", fbType.Serialize());
```

Alternative 2.

```
SFCCodeBlock sfccodebl = ObjectFactory.NewSFCCodeBlock("MySFCCodeBlock");
sfccodebl.SFCElements.AddStep2("Step1", true, "X:=1;", "X:=X+1;", "");
sfccodebl.SFCElements.AddTransition2("T1", "X>100", "");
SFCSimultaneous sim = sfccodebl.SFCElements.AddSimultaneous1(2);
// 2 branches
sim.SFCBranches[1].SFCElements.AddStep2("Step2",false,"","X:=X-1;","");
sim.SFCBranches[2].SFCElements.AddStep2("Step3",false,"","X:=X-1;","");
sfccodebl.SFCElements.AddTransition2("T2", "X<50", "");

// Serialize the objects into an XMLString and create the
// Codeblock in the Control Builder
string bucket = cb.NewCodeBlock(CBOpenIFCodeBlockType.OI_SFC, sfccodebl.Name,
                                "MyLib.MyFBType", sfccodebl.Serialize());
```

The second alternative is the most efficient solution in this example. However, if you would like to create several code blocks etc at a time, the first alternative is the most efficient.

5.3.8 Modify an existing CodeBlock

The task is to modify the SFC code block in the previous example. Assume we would like to add a step “S4” and a transition ”T3” at the end of the sequence. We also would like to change the condition of the “T2” transition to “X<45”. The SFC code should now look like figure 17.

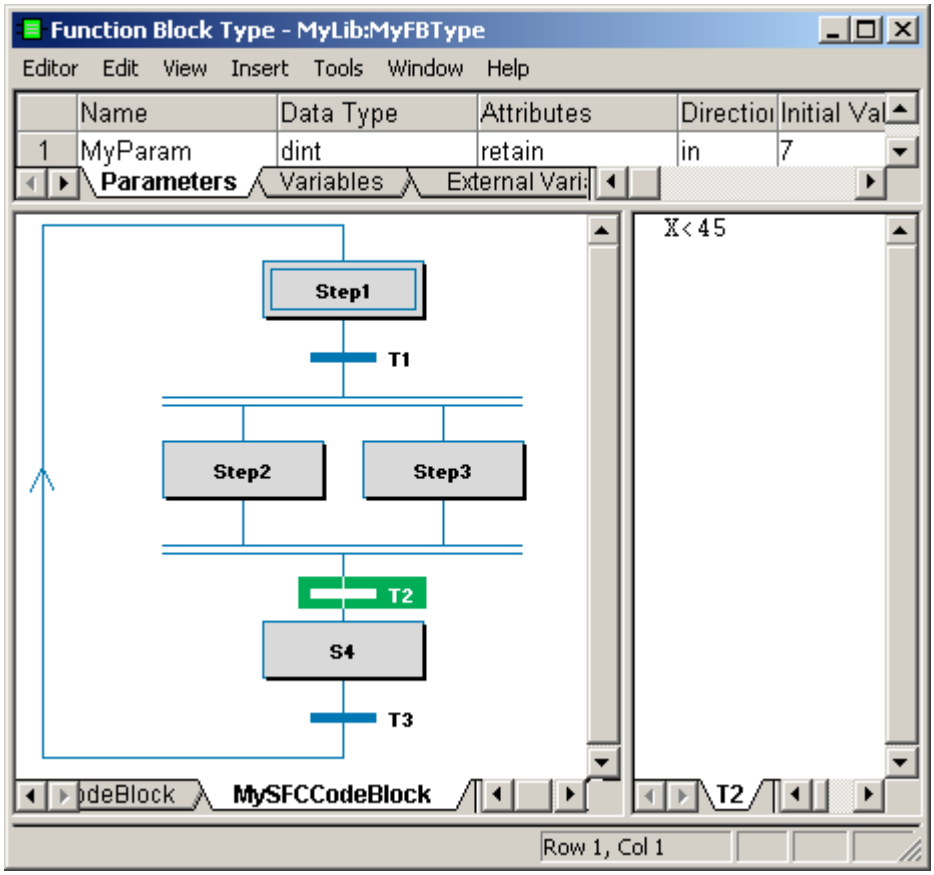


Figure 17

We can solve this problem in several different ways. One alternative is presented below.

```
// Get an XML description of the CodeBlock from the Control Builder
string XMLStr = cb.GetCodeBlock("MyLib.MyFBType.MySFCCodeBlock");

// Deserialize the XMLString into Objects
ICodeBlock codeBlock = (ICodeBlock) ObjectFactory.DeserializeCodeBlock(ref XMLStr);
if (codeBlock.IsSFCCodeBlock)
{
   // It is a SFCCodeBlock just as we expected
   SFCCodeBlock sfccodebl = (SFCCodeBlock) codeBlock;
   // Add a step "S4" at the end
   sfccodebl.SFCElements.AddStep2("S4", false, "", "X:=X+1;", "");
   // and add a transition "T3"
   sfccodebl.SFCElements.AddTransition2("T3", "X>75", "");

   // Now, change the "T2" transition code to "X<45"
   // The "SFCElements" collection has no Find method because all SFC elements
   // don't have names. We have to implement the search ourselves
   // Find the "T2" transition
   foreach (ISFCElement el in  sfccodebl.SFCElements)
```

```
    {
      if (el.IsSFCTransition)
      {
         SFCTransition tr = (SFCTransition) el;
         if (tr.Name == "T2")
         {
           tr.Transition_STCode = "X<45";
         }
      }
    }
    // Serialize the objects into an XMLString and update the
    // CodeBlock in the Control Builder
    string bucket = cb.SetCodeBlock("MyLib.MyFBType.MySFCCodeBlock",
                                    sfccodebl.Serialize());
}
```

5.3.9 Delete an existing CodeBlock

Assume the task is to delete an existing code block. The path to the code block is **"MyLib.MyFBType.MySFCCodeBlock".**

We can solve this problem in several different ways. One alternative is presented below.

```
// Call the "CB Open Interface" method DeleteCodeBlock
cb.DeleteCodeBlock("MyLib.MyFBType.MySFCCodeBlock");
```

5.3.10 Add a new Variable to an existing type

Assume the task is to add a new variable, according to the following specification, to an existing type. The path to the type is “MyLib.MyFBtype”.

Name	Type	Attribute	Initial Value	Description
ANewVariable	bool	retain	false	Description of my new Variable

Figure 18.

We can solve this problem in two different ways. Both alternatives are presented below.

Alternative 1.

```
// Create and initialize a Variable object
Variable var = ObjectFactory.NewVariable1("ANewVariable","bool", "retain",
                              "false","","","Description of my new Variable");
// Serialize the Variable object to an XMLSTring and call the
// "CB Open Interface" method NewVariable.
cb.NewVariable(CBOpenIFVariableType.OI_VARIABLE, var.Name, var.TypeName,
               "MyLib.MyFBType", var.Serialize());
```

Alternative 2.

```
// Get an XML description of the type from the Control Builder
string XMLStr = cb.GetFunctionBlockType("MyLib.MyFBType");
// Deserialize the XMLSTring into Objects
FunctionBlockType fbType = ObjectFactory.DeserializeFunctionBlockType(ref XMLStr);
// Add a Variable
fbType.Variables.Add2("ANewVariable","bool", "retain", "false","","",
                      "Description of my new Variable");
// Serialize the objects into an XMLString and update the
// FunctionBlockType in the Control Builder
string bucket = cb.SetFunctionBlockType("MyLib.MyFBType", fbType.Serialize());
```

The first alternative is the most efficient solution in this example. However, if you would like to create several variables etc at a time, the second alternative is the most efficient.

5.3.11 Modify an existing Variable

Assume the task is to modify an existing variable, according to the following specification. The path to the variable is `"MyLib.MyFBType.ANewVariable"`.

Name	Type	Attribute	Initial Value	Description
ANewVariable	dint	coldretain	7	The Variable is now an int

Figure 19.

We can solve this problem in several different ways. One alternative is presented below.

```
// Create and initialize a Variable object
Variable var = ObjectFactory.NewVariable1("ANewVariable","dint",
                  "coldretain", "7","","","The Variable is now an int");
// Serialize the Variable object to an XMLSTring and call the
// "CB Open Interface" method SetVariable.
string bucket = cb.SetVariable(CBOpenIFVariableType.OI_VARIABLE,
                               "MyLib.MyFBType.ANewVariable",
                               var.Serialize());
```

5.3.12 Delete an existing Variable

Assume the task is to delete an existing variable. The path to the variable is `"MyLib.MyFBType.ANewVariable"`.

We can solve this problem in several different ways. One alternative is presented below.

```
// Call the "CB Open Interface" method DeleteVariable
cb.DeleteVariable(CBOpenIFVariableType.OI_VARIABLE,
                  "MyLib.MyFBType.ANewVariable");
```

5.3.13 Add a new FunctionBlock to an existing type

Assume the task is to add a new function block, according to the following specification, to an existing type. The path to the type is `"MyLib.MyFBType"`.

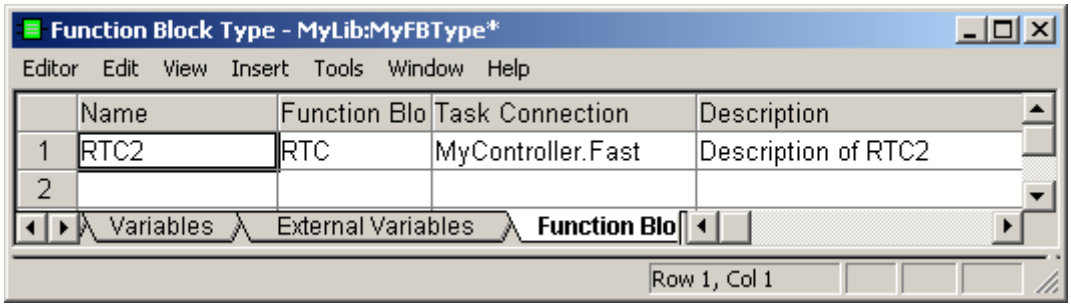


Figure 20.

We can solve this problem in several different ways. One alternative is presented below.

```
FunctionBlock fb = ObjectFactory.NewFunctionBlock1("RTC2", "RTC",
                                   "MyController.Fast", "", "Description of RTC2");
// Serialize the object into an XMLString and post it to the Control Builder
string bucket = cb.NewFunctionBlock(fb.Name, fb.TypeName,
                                       "MyLib.MyFBType", fb.Serialize());
```

5.3.14 Modify the content of an existing FunctionBlock

Assume the task is to modify the function block in the previous example. The “Task Connection” should be changed to “MyController.Normal” and the description to “New Description of RTC2”.

We can solve this problem in several different ways. Only one alternative is presented below.

```
// Get an XML description of the function block from the Control Builder
string XMLStr = cb.GetFunctionBlock("MyLib.MyFBType.RTC2");

// Deserialize the XMLString into Objects
FunctionBlock fb = ObjectFactory.DeserializeFunctionBlock(ref XMLStr);
fb.TaskConnection = "MyController.Normal";
fb.Description = "New Description of RTC2";

// Serialize the function block object to an XMLString and call the
// "CB Open Interface" method SetFunctionBlock.
string bucket = cb.SetFunctionBlock("MyLib.MyFBType.RTC2", fb.Serialize());
```

5.3.15 Delete an existing FunctionBlock

Assume the task is to delete an existing function block. The path to the function block is **"MyLib.MyFBType.RTC2".**

We can solve this problem in several different ways. One alternative is presented below.

```
// Call the "CB Open Interface" method DeleteFunctionBlock
cb.DeleteFunctionBlock("MyLib.MyFBType.RTC2");
```

5.3.16 List the variables of an existing type

The task is now to display the variables, of the function block type created in the previous examples, in an edit box. The result would look like figure 21.

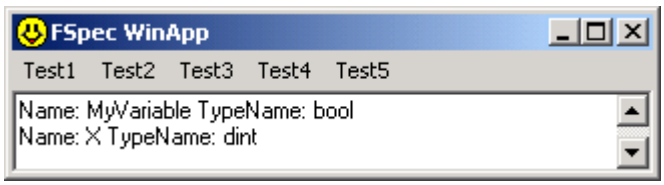


Figure 22.

```
// Get an XML description of the type from the Control Builder
string XMLStr = cb.GetFunctionBlockType("MyLib.MyFBType");
// Deserialize the XMLSTring into Objects
FunctionBlockType fbType = ObjectFactory.DeserializeFunctionBlockType(ref XMLStr);

//Use the foreach statement in order to loop trough all objects in the collection
foreach (Variable var in fbType.Variables)
{
   // Print out the data, for one Variable, into the TextBox
   richTextBox1.Text += "Name: " + var.Name +
                        " TypeName: " + var.TypeName + "\n";
}
```

The code above made use of the “foreach” statement in order to loop through all objects in the “Variables” collection. An alternative is to use the index operator. Example:

```
// Get an XML description of the type from the Control Builder
string XMLStr = cb.GetFunctionBlockType("MyLib.MyFBType");
// Deserialize the XMLSTring into Objects
FunctionBlockType fbType = ObjectFactory.DeserializeFunctionBlockType(ref XMLStr);

// Use the for statement in order to loop trough all objects in the collection
for (int i=1; i<=fbType.Variables.Count; i++ )
{
   // Print out the data, for one Variable, into the TextBox
   richTextBox1.Text += "Name: " + fbType.Variables[i].Name +
                        " TypeName: " + fbType.Variables[i].TypeName + "\n";
}
```

5.3.17 Create a new Program

Assume the task is to create a new program in the application “MyApp” according to figures below.

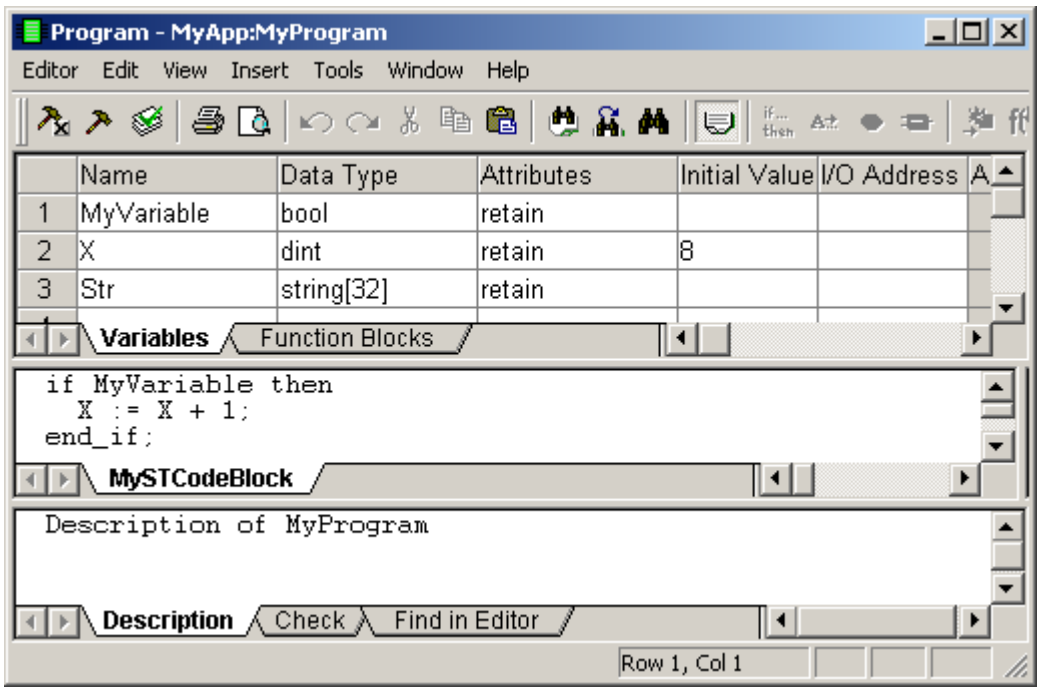


Figure 23.

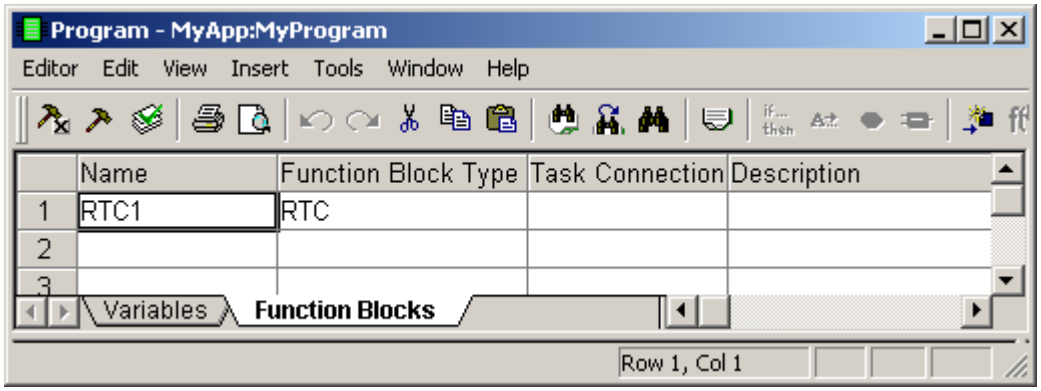


Figure 24.

```
// Create an object of the type "Program" in the client's process memory
Program prog = ObjectFactory.NewProgram("MyProgram", "Description of MyProgram");

// Add some Varible objects
prog.Variables.Add1("MyVariable", "bool");
prog.Variables.Add2("X", "dint", "retain", "8", "", "", "Desc of X");
Variable var = prog.Variables.Add1("Str", "string[32]");
var.Description = "Description of the variable";

// Add a FunctionBlock object
prog.FunctionBlocks.Add1("RTC1","RTC");


 // Add a ST CodeBlock
 string stCode = "if MyVariable then\n" +
```

```
                "  X := X + 1;\n" +
                "end_if;";
prog.CodeBlocks.AddSTCodeBlock2("MySTCodeBlock", ref stCode);

// Finally, serialize the object model into an XML String and
// call the OpenIF method "NewProgram" in order to create the type
// in the Control Builder EXE.
string bucket = cb.NewProgram(prog.Name,"MyApp", prog.Serialize());
```

5.3.18 Create a new ControlModuleType

Assume the task is to create two new control module types, in the library “MyLib”, according to figures below.

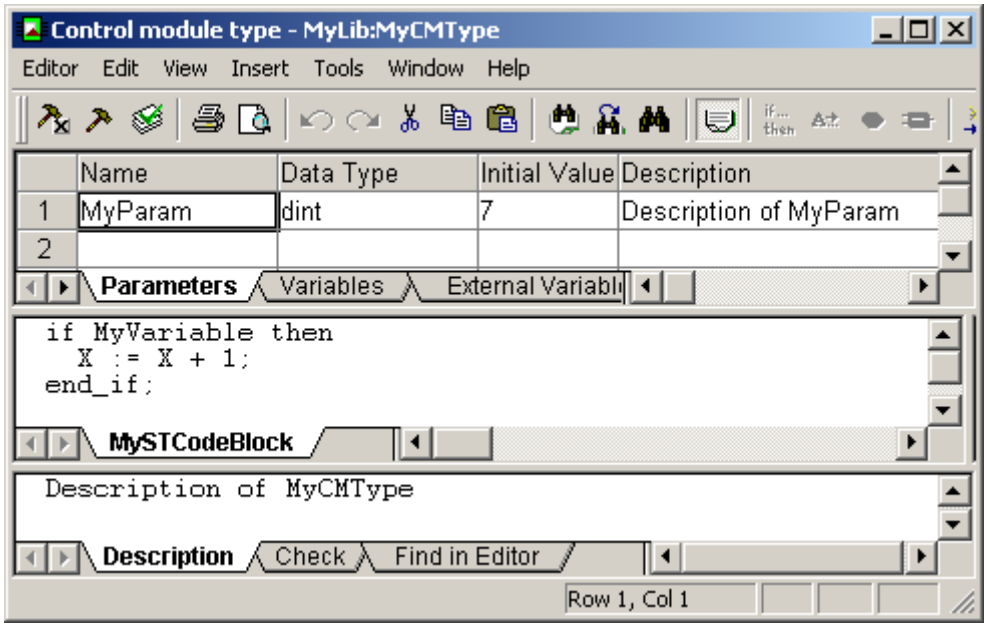


Figure 25

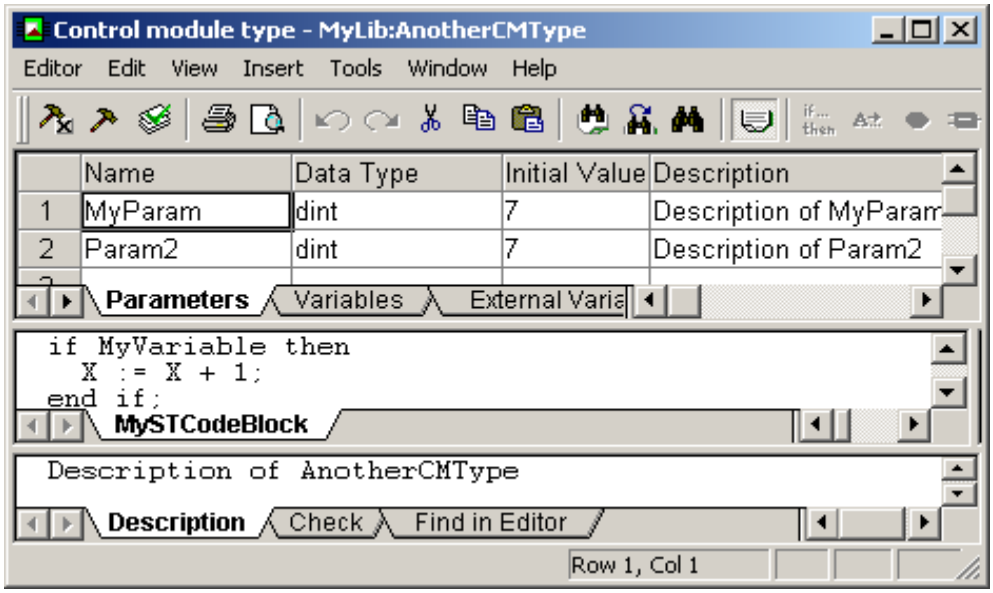


Figure 26

You should know that the default coordinate system for a Control Module Type ranges from -1,-1 to 1,1 corresponding to the lower left corner and the right upper corner respectively. Origo, in the middle, will then have the coordinates 0,0.

```
// Create an object of the type "ControlModuleType"
ControlModuleType cmType = ObjectFactory.NewControlModuleType("MyCMType",
                                                    "Description of MyCMType");
// Accept the default value of the "GraphSize" i.e. ((-1,-1) , (1,1))
// Add some Variable objects
cmType.Variables.Add1("MyVariable", "bool");
```

```
cmType.Variables.Add2("X", "dint", "retain", "8", "", "", "Desc of X");
cmType.Variables.Add2("Y", "dint", "retain", "9", "", "", "Desc of Y");
cmType.Variables.Add2("Z", "dint", "retain", "0", "", "", "Desc of Z");


// Add some Parameter objects
cmType.CMParameters.Add2("MyParam", "Dint", "7",  "","", "Description of MyParam",
                         null);
// Add a FunctionBlock object
cmType.FunctionBlocks.Add1("RTC1","RTC");
// Add a ST CodeBlock
string stCode = "if MyVariable then\n" +
                "  X := X + 1;\n" +
                "end_if;";
cmType.CodeBlocks.AddSTCodeBlock2("MySTCodeBlock", ref stCode);
// Finally, serialize the object model into an XML String and
// call the OpenIF method "NewControlModuleType" in order to create the type
// in the Control Builder EXE.
string bucket = cb.NewControlModuleType(cmType.Name,"MyLib", cmType.Serialize());

// Create the other "ControlModuleType" named "AnotherCMType"
cmType.Name = "AnotherCMType";
cmType.Description = "Description of AnotherCMType";
cmType.CMParameters.Add2("Param2", "Dint", "7",  "","", "Description of Param2",
                         null);
bucket = cb.NewControlModuleType(cmType.Name,"MyLib", cmType.Serialize());
```

5.3.19 Add new ControlModules to an existing type

This example is a continuation of the previous example. Assume the task is to create two new control modules “CMInst1” and “CMInst2” as children of the father type “MyCMType”. Both “CMInst1” and “CMInst2” are objects of the type “AnotherCMType”.

The control modules should be placed in the father type’s coordinate system according to the following figure.

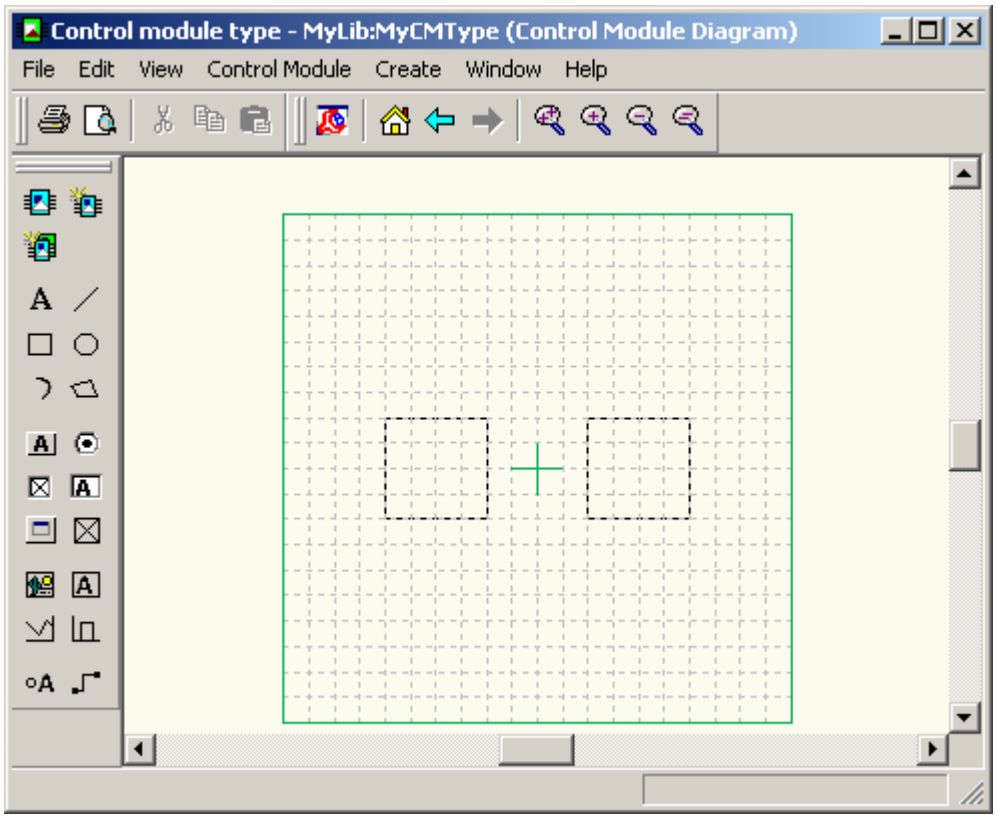


Figure 27

The figure shows the control modules position and sizes in the father’s coordinate system. The father’s coordinate system is “Lower left corner” **(X=-1, Y=-1)** and “Upper right corner” **(X=1, Y=1)**.Origo is **(X=0, Y=0).**

The “ModInst1” has a position **(XPos=-0.4, YPos=0.0)** in relation to the father type’s origo. The size is determined by the **(XScale=0.2, YScale=0.2).**

The “ModInst2” has a position **(XPos=0.4, YPos=0.0)** in relation to the father type’s origo. The size is determined by the **(XScale=0.2, YScale=0.2).**

Furthermore, the parameters of the control modules should be connected to variables and parameters according to the following figures.

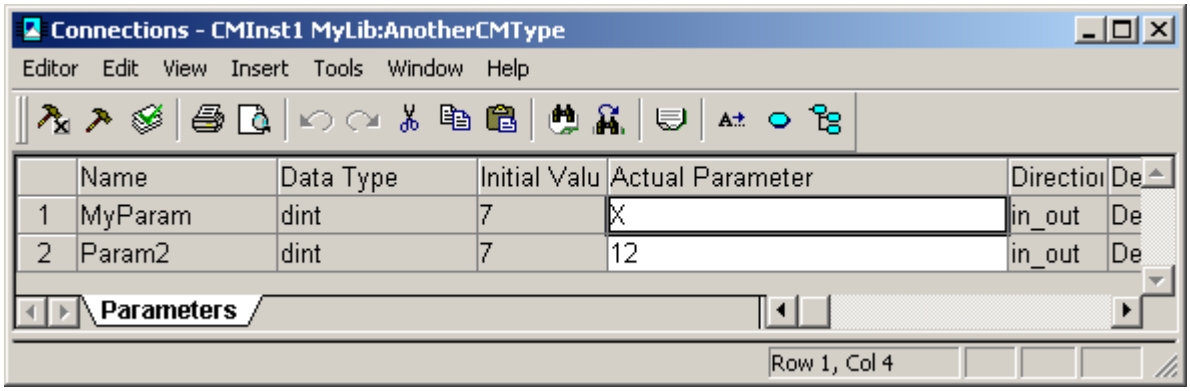


Figure 28

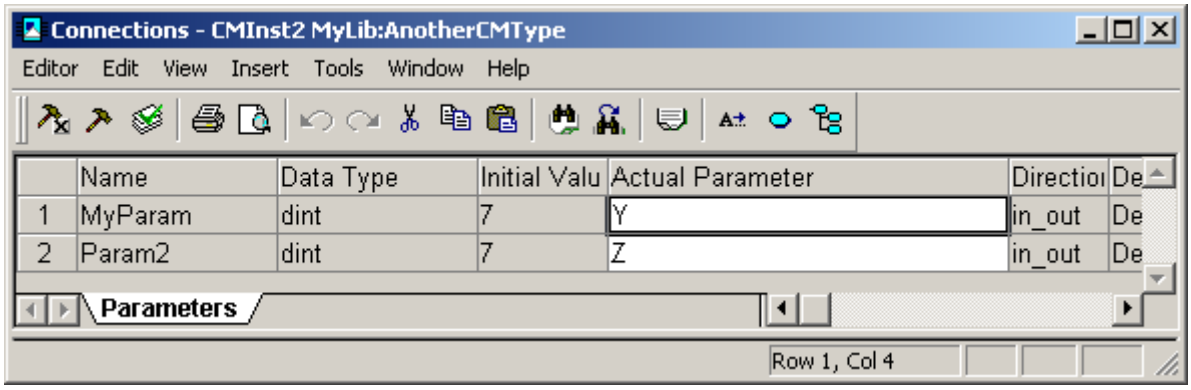


Figure 29

```
// Create an "ControlModule" object named "CMInst1"
ControlModule cm = ObjectFactory.NewControlModule("CMInst1", "AnotherCMType");
// Set the size and position to: XPos = -0.4, YPos = 0.0, XScale = 0.2, YScale = 0.2
cm.GraphPos = ObjectFactory.NewGraphPos(-0.4, 0.0, 0.0, 0.2, 0.2);
// Connect the parameters
cm.CMConnections.Add1("MyParam", "X");
cm.CMConnections.Add1("Param2", "12");
// Serialize to an XML String and update the Control Builder EXE.
string bucket = cb.NewControlModule(cm.Name,cm.TypeName,"MyLib.MyCMType",
                                    cm.Serialize());

// Create an "ControlModule" object named "CMInst2"
cm = ObjectFactory.NewControlModule("CMInst2", "AnotherCMType");
cm.GraphPos = ObjectFactory.NewGraphPos(0.4, 0.0, 0.0, 0.2, 0.2);
cm.CMConnections.Add1("MyParam", "Y");
cm.CMConnections.Add1("Param2", "Z");
// Finally, serialize to an XML String and update the Control Builder EXE.
bucket = cb.NewControlModule(cm.Name,cm.TypeName,"MyLib.MyCMType", cm.Serialize());
```

5.3.20 Modify the connections of an existing ControlModule

This example is a continuation of the previous examples. Assume the task is to change the connections of the control module “CMInst1” according to the following figure.

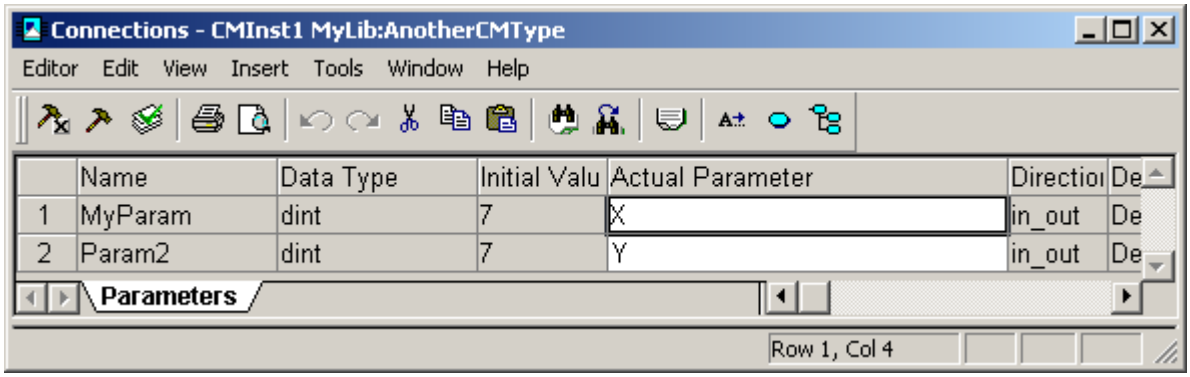


Figure 30

We can solve this problem in several different ways. Two alternatives are presented below.

Alternative 1.

```
// Create a "CMConnection" object
CMConnection cmConn = ObjectFactory.NewCMConnection("Param2", "Y");
// Finally, serialize to an XML String and update the Control Builder EXE.
string bucket = cb.SetCMConnection("MyLib.MyCMType.CMInst1.Param2",
                                   cmConn.Serialize());
```

Alternative 2.

```
// Get the content of the ControlModule "CMInst1" from the Control Builder EXE
string XMLStr = cb.GetControlModule("MyLib.MyCMType.CMInst1");
// Deserialize to objects
ControlModule cm = ObjectFactory.DeserializeControlModule(ref XMLStr);
// Find and change "Param2"
int Nr = cm.CMConnections.FindNr("Param2");
if (Nr > 0 )
{
  cm.CMConnections.Remove(Nr); // Remove the old connection
}
cm.CMConnections.Add1("Param2", "Y");
// Finally, serialize to an XML String and update the Control Builder EXE.
string bucket = cb.SetControlModule("MyLib.MyCMType.CMInst1", cm.Serialize());
```

5.3.21 Create a new ControlModuleType with graphical parameter nodes

The task is to create a “ControlModuleType”, with three graphical parameter nodes according to the figures below.

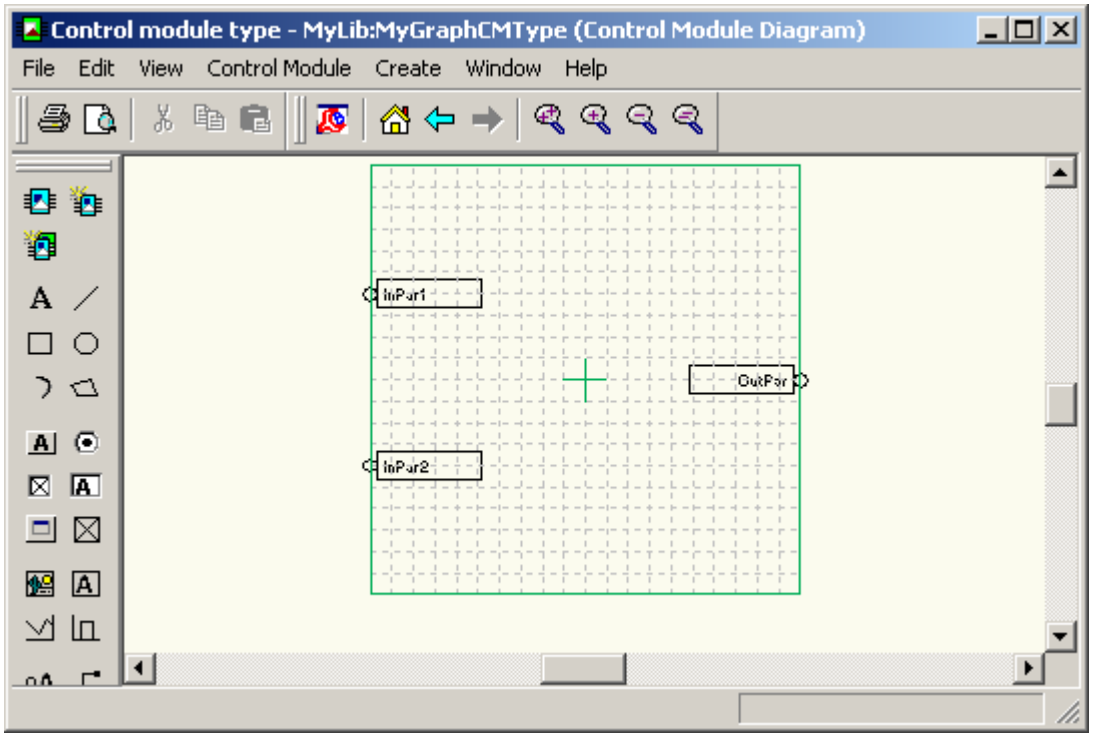


Figure 31

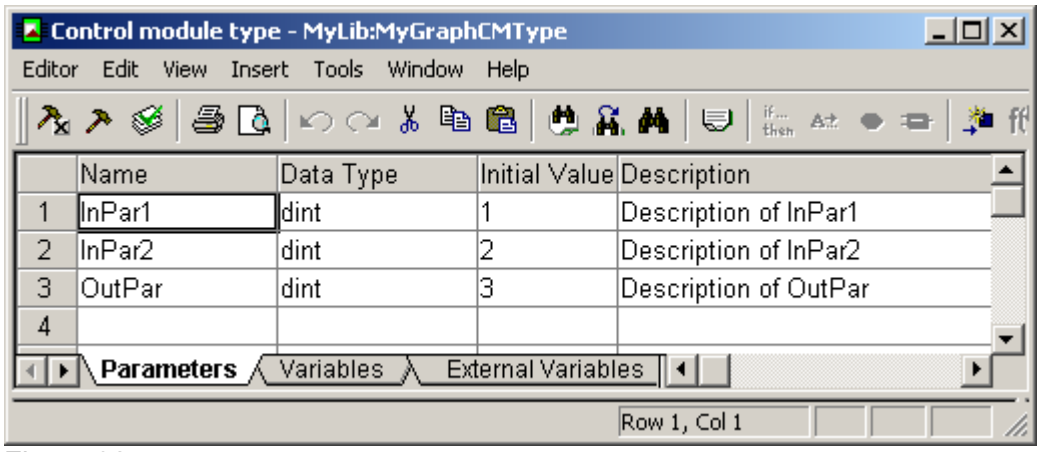


Figure 32

The example will focus on the graphical aspects i.e. the “ControlModuleType” will not contain code, variables etc. The reason is to keep the example small. Other examples in this document shows how to populate a type with variables, codes, function blocks etc.

```
// Create an object of the type "ControlModuleType"
ControlModuleType cmType = ObjectFactory.NewControlModuleType("MyGraphCMType",
"Description of MyGraphCMType");
// Accept the default value of the GraphSize i.e ( (-1,-1) , (1,1) )
```

```
// Add some Parameter objects
CMParameter par = cmType.CMParameters.Add2("InPar1", "Dint", "1",  "","",
                                           "Description of InPar1", null);
// Create a graphical node at (-1.0, 0.4)
par.GraphNodes.Add1("InPar1", -1.0, 0.4);

par = cmType.CMParameters.Add2("InPar2", "Dint", "2",  "","",
                               "Description of InPar2", null);
// Create a graphical node at (-1.0, -0.5)
par.GraphNodes.Add1("InPar2", -1.0, -0.5);

// Create a graphical node at the right edge
par = cmType.CMParameters.Add2("OutPar", "Dint", "3",  "","",
         "Description of OutPar", ObjectFactory.NewAutoPoint(AutoPosValue.cbRight));

// Finally, serialize to an XML String and update the Control Builder EXE.
string bucket = cb.NewControlModuleType(cmType.Name,"MyLib", cmType.Serialize());
```

The code above shows two different ways to assign a graphical node to a parameter. The first one is

```
// Create a graphical node at (-1.0, 0.5)
par.GraphNodes.Add1("InPar1", -1.0, 0.5);
```

In this case we place the graphical node at the coordinates `(X=-1.0, Y=0.5).` You might wonder why a parameter can have a collection of graphical nodes (GraphNodes). That's because a parameter can be a "struct" and several members of the "struct" can have different graphical nodes, with different names and coordinates.

The second alternative is to use the "AutoPoint" object as shown below.

```
// Create a graphical node at the right edge
par = cmType.CMParameters.Add2("OutPar", "Dint", "3",  "","",
         "Description of OutPar", ObjectFactory.NewAutoPoint(AutoPosValue.cbRight));
```

In this case we only specify that the node should be placed at the right edge of the type.

5.3.22 Graphical connections of ControlModules

This example is a continuation of the previous examples. Assume the task is to create a “Control Module Diagram” according to the figure below.

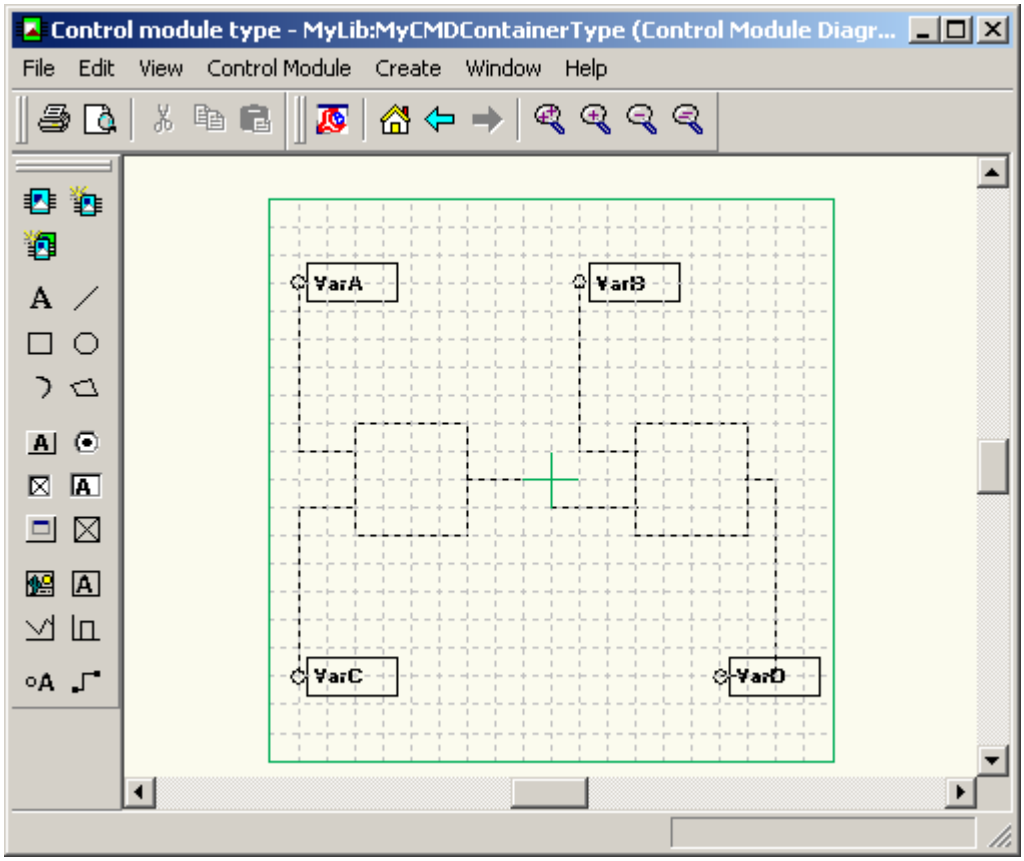


Figure 33

We solve this problem in the following steps:

1. The father type, that is the container, is created.
2. Four variables are created in the father type. Graphical nodes are assigned to the variables. Thus, the variables become visible in the CMD diagram.
3. Two control modules, named “CM1” and “CM2”, are created. Both are of the “MyGraphCMType” created in the previous example.
4. The graphical connections between the variables nodes and the parameters are defined. In some cases we give the system a clue how to draw the connection lines by means of defining some points. We don’t have to define points for the polygon but the diagram would look nicer if we do so.

```
// Create an object of the type "ControlModuleType". This type is a container for
// the control modules
ControlModuleType cmType = ObjectFactory.NewControlModuleType("MyCMDContainerType",
                                                    "Description MyCMDContainerType");
// Accept the default value of the GraphSize i.e ( (-1,-1) , (1,1) )
// Add some Variable objects
Variable var = null;

var = cmType.Variables.Add1("VarA", "dint");
var.GraphNodes.Add1("VarA", -0.9, 0.7);

var = cmType.Variables.Add2("VarB", "dint", "retain", "8", "", "", "Desc of B");
var.GraphNodes.Add1("VarB", 0.1, 0.7);

var = cmType.Variables.Add2("VarC", "dint", "retain", "9", "", "", "Desc of C");
var.GraphNodes.Add1("VarC", -0.9, -0.7);

var = cmType.Variables.Add2("VarD", "dint", "retain", "0", "", "", "Desc of D");
var.GraphNodes.Add1("VarD", 0.6, -0.7);

// Add two ControlModules of the type MyGraphCMType
ControlModule cm1 = cmType.ControlModules.AddControlModule("CM1", "MyGraphCMType");
// Set the size and position to: XPos = -0.5, YPos = 0.0, XScale = 0.2, YScale = 0.2
cm1.GraphPos = ObjectFactory.NewGraphPos(-0.5, 0.0, 0.0, 0.2, 0.2);

ControlModule cm2 = cmType.ControlModules.AddControlModule("CM2", "MyGraphCMType");
// Set the size and position to: XPos = 0.5, YPos = 0.0, XScale = 0.2, YScale = 0.2
cm2.GraphPos = ObjectFactory.NewGraphPos(0.5, 0.0, 0.0, 0.2, 0.2);

// Now connect the modules graphically (set the property GraphicalConnection=true)
CMConnection c = cm1.CMConnections.Add2("InPar1", "VarA", true);
c.Points.Add1(-0.9, 0.1);  // Add one polygon point

c = cm1.CMConnections.Add2("InPar2", "VarC", true);
c.Points.Add1(-0.9, -0.1); // Add one polygon point

c = cm1.CMConnections.Add2("OutPar", "CM2.InPar2", true);
c.Points.Add1(0.0, -0.1);  // Add one polygon point

c = cm2.CMConnections.Add2("InPar1", "VarB", true);
c.Points.Add1(0.1, 0.1); // Add one polygon point

c = cm2.CMConnections.Add2("InPar2", "CM1.OutPar", true);
c.Points.Add1(0.0, -0.1); // Add two polygon points
c.Points.Add1(0.0, 0.0);

c = cm2.CMConnections.Add2("OutPar", "VarD", true);
c.Points.Add1(0.8, 0.0); // Add one polygon point

// Finally, serialize to an XML String and update the Control Builder EXE.
string bucket = cb.NewControlModuleType(cmType.Name,"MyLib", cmType.Serialize());
```

5.3.23 SingleControlModules

There are two different COM Classes for SingleControlModules:

- SingleControlModuleInst. This class describes the ***instance*** part of a SingleControlModule
- SingleControlModuleType. This class describes the ***type*** part of a SingleControlModule

The following figure describes the properties and the methods of the SingleControlModuleInst class.

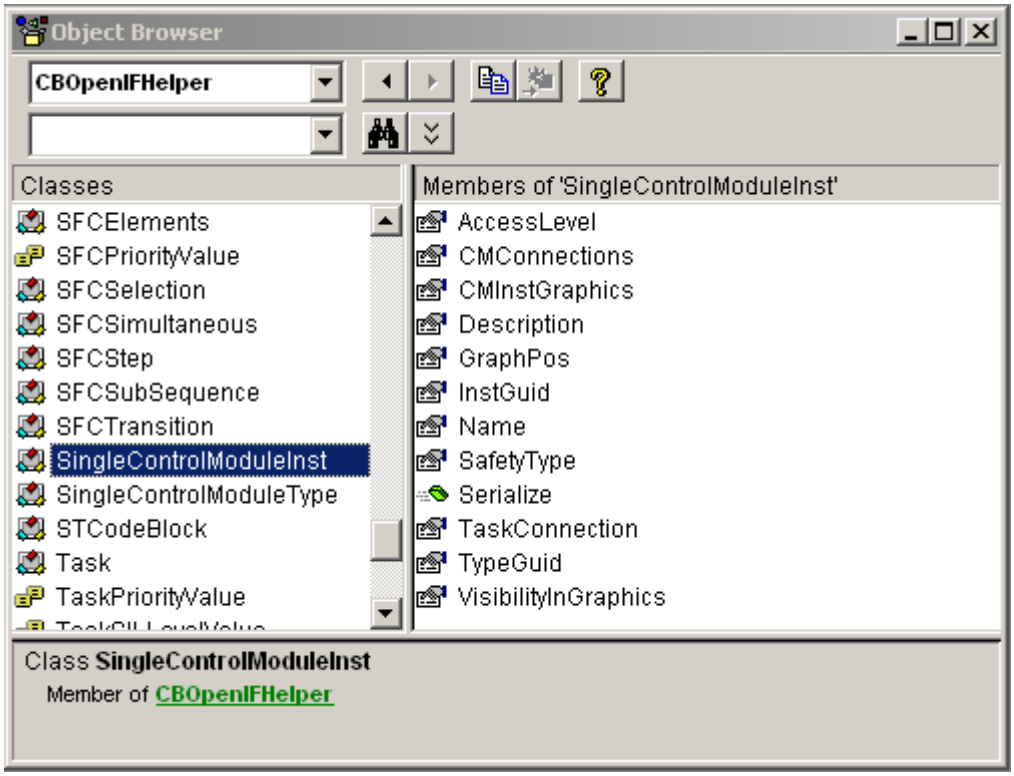


Figure 34

And the following figure describes the properties and the methods of the SingleControlModuleType class.

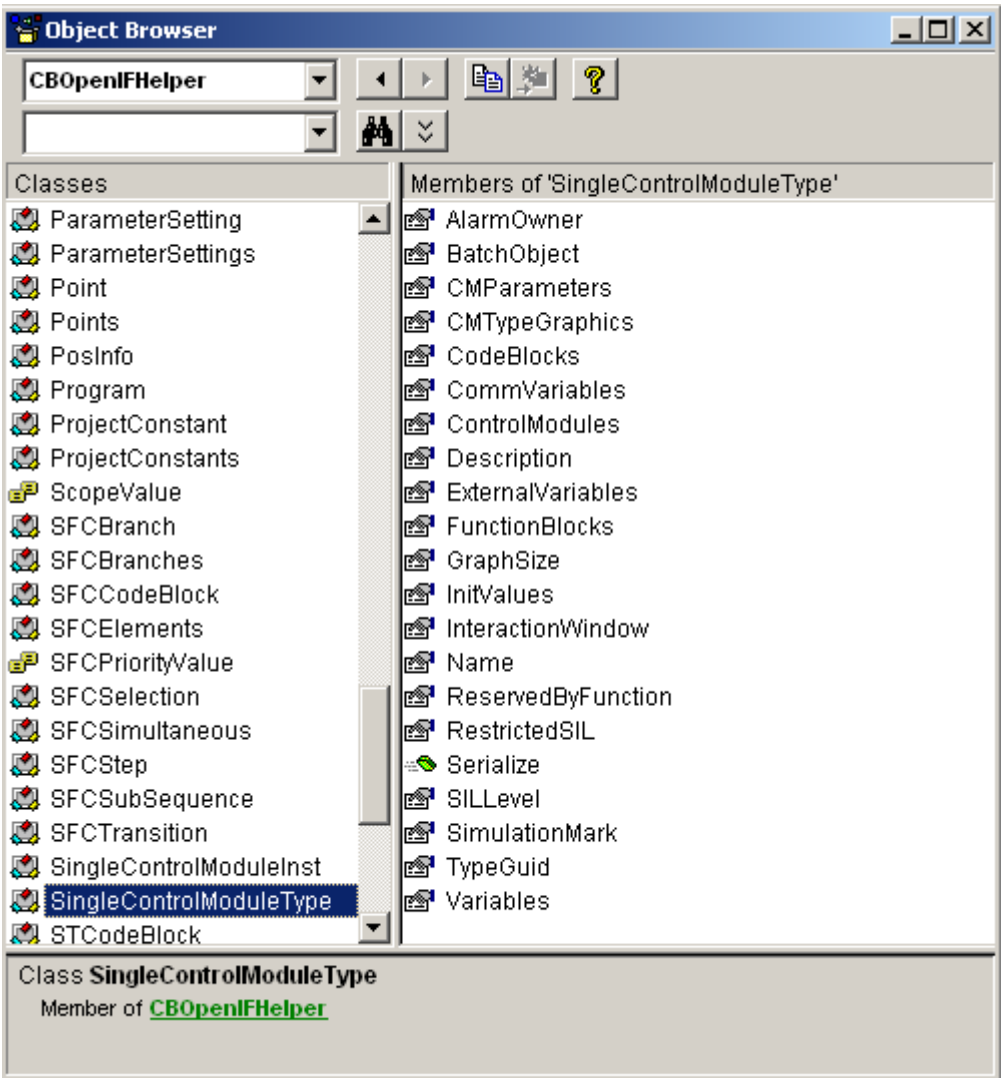


Figure 35

A SingleControlModule is a ***Singleton*** i.e. there is always exactly one instance of the type. The XML Schema describes both the type part and the instance part by only one XML Element, namely the “SingleControlModule” element.

If a SingleControlModule is created then both the type part and the instance part are created simultaneously, and if a SingleControlModule is deleted the both the type part and the instance part are deleted.

The following “CB OpenInterface” methods operates on **SingleControlModules**:

1. **NewSingleControlModule**. Operates on both the instance part and the type part.
2. **DeleteSingleControlModule**. Operates on both the instance part and the type part.
3. **DeleteControlModule**. Operates on both the instance part and the type part.
4. **GetControlModule**. Operates on the ***instance*** part only.
5. **SetControlModule**. Operates on the ***instance*** part only.
6. **GetSingleControlModule**. Operates on the ***type*** part only.
7. **SetSingleControlModule**. Operates on the ***type*** part only.

5.3.23.1 Create SingleControlModules in an existing type

Assume that a ControlModuleType named “CMType2” already exists in the library “MyLib”. “CMType2” has the following variables:

	Name	Data Type	Attributes	Initial Valu	Description
1	MyVariable	bool	retain		
2	X	dint	retain	8	Desc of X
3	Y	dint	retain	9	Desc of Y
4	Z	dint	retain	0	Desc of Z
5					

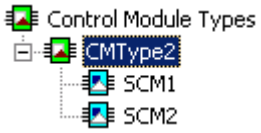
The task is now to make the following modifications of the ControlModuleType:

1. Add a SingleControlModule named “SCM1”. The “SCM1” has one parameter named “P1”. “P1” should be connected to the variable named “Z”.
2. Add a SingleControlModule named “SCM2”. The “SCM2” has two parameter named “P1”and “P2”. “P1” should be connected to the variable named “Y” and “P2” to the variable named “X”.
3. Add a new code block named “MySTCodeBlock” to the type

```
// Get an XML description of the type from the Control Builder
string XmlStr = cb.GetControlModuleType("MyLib.CMType2");
ControlModuleType cmType = ObjectFactory.DeserializeControlModuleType(ref XmlStr);

// Add a SingleControlModule named "SCM1", with one parameter "P1", to the type.
// Connect the parameter to the variable "Z"
SingleControlModuleInst scmInst1 = null;
scmInst1 = cmType.ControlModules.AddSingleControlModuleInst("SCM1");
scmInst1.CMConnections.Add1("P1","Z");

// Add a SingleControlModule named "SCM1", with two parameters "P1" and "P2",
// to the type. Connect the parameters to the variables "Y" and "X"
SingleControlModuleInst scmInst2 = null;
scmInst2 = cmType.ControlModules.AddSingleControlModuleInst("SCM2");
scmInst2.CMConnections.Add1("P1","Y");
scmInst2.CMConnections.Add1("P2","X");

// Add a new CodeBlock
string stCode = "if MyVariable then\n" +
          "  X := X + 1;\n" +
          "end_if;";
cmType.CodeBlocks.AddSTCodeBlock2("MySTCodeBlock", ref stCode);

// Finally, serialize the object model into an XML String and
// update the ControlBuilder
string bucket = cb.SetControlModuleType("MyLib.CMType2", cmType.Serialize());
```

5.3.23.2 Define the type parts if the SingleControlModules

This example is a continuation of the previous example. Two SingleControlModules were created in the previous example and the project explorer of the Control Builder looks like:



However, the ***Type*** parts of the “SCM1” and “SCM2” are empty.

The task is now to define the Type parts (Variables, CodeBlocks, FunctionBlocks and so on) of the SingleControlModules according to the following code:

```
// Get the type part of the SingleControlModule from the Control Builder
string xmlstr = cb.GetSingleControlModule("MyLib.CMType2.SCM1");
SingleControlModuleType scmType =
          ObjectFactory.DeserializeSingleControlModuleType(ref xmlstr);
scmType.Variables.Add1("Var1","dint");
scmType.Variables.Add1("Var2","dint");
scmType.CMParameters.Add1("P1","dint");
string STCodeStr = "P1:=P1+Var1+1;";
scmType.CodeBlocks.AddSTCodeBlock2("TheCodeBlock", ref STCodeStr);
// Update the Control Builder
cb.SetSingleControlModule("MyLib.CMType2.SCM1", scmType.Serialize());

// Get the type part of the SingleControlModule from the Control Builder
xmlstr = cb.GetSingleControlModule("MyLib.CMType2.SCM2");
scmType = ObjectFactory.DeserializeSingleControlModuleType(ref xmlstr);
scmType.Variables.Add1("Var1","dint");
scmType.CMParameters.Add1("P1","dint");
scmType.CMParameters.Add1("P2","dint");
STCodeStr = "P1:=P1+P2+Var1+7;";
scmType.CodeBlocks.AddSTCodeBlock2("TheCodeBlock", ref STCodeStr);
// Update the Control Builder
cb.SetSingleControlModule("MyLib.CMType2.SCM2", scmType.Serialize());
```

5.3.23.3 Create a new SingleControlModule into another SingleControlModule

This example is a continuation of the previous example. We will now create a new SingleControlModule named “SCM12” into the SingleControlModule named “SCM1” according to the figure below:

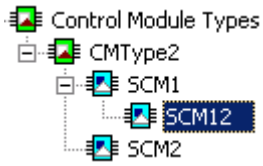


Figure 36

The task is to create the SingleControlModule and to define the type part and the instance part of the module according to the code below:

```
SingleControlModuleType scmType =
          ObjectFactory.NewSingleControlModuleType("SCM12", "Description of");

// Define the type part of the SCM. In this case a parameter called "P1" and a
// ST Code Block.
scmType.CMParameters.Add1("P1","dint");
string STCodeStr = "P1:=P1+1;";
scmType.CodeBlocks.AddSTCodeBlock2("TheCodeBlock", ref STCodeStr);

string s = scmType.Serialize();
// Create the SCM in the control builder. The type part of the SCM will be created
// according to the serialized XML
cb.NewSingleControlModule(scmType.Name,"MyLib.CMType2.SCM1", scmType.Serialize());

// Get the instance part and connect the parameter to
// a variable in the father single control module
string xmlstr = cb.GetControlModule("MyLib.CMType2.SCM1.SCM12");
SingleControlModuleInst scmInst =
       ObjectFactory.DeserializeSingleControlModuleInst(ref xmlstr);
scmInst.CMConnections.Add1("P1","Var2");
// Update the Control Builder
cb.SetControlModule("MyLib.CMType2.SCM1.SCM12", scmInst.Serialize());
```

5.3.24 Create new Access Variables

Assume the task is to create access variables, in the controller “MyController” according to the figures below.

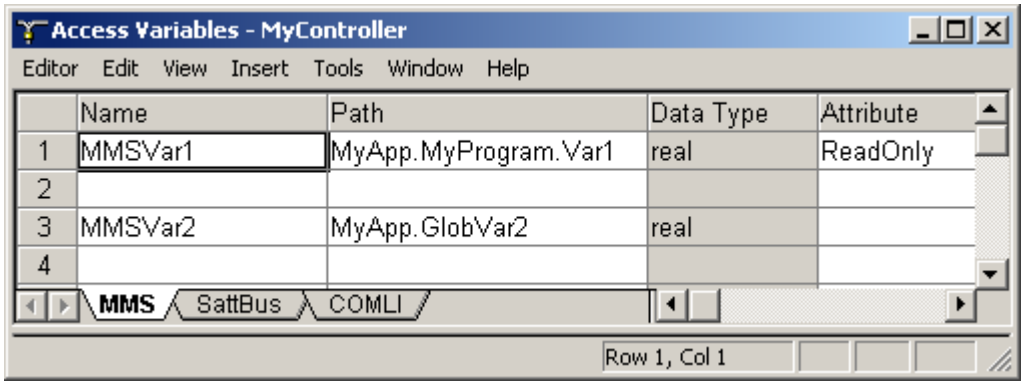


Figure 37

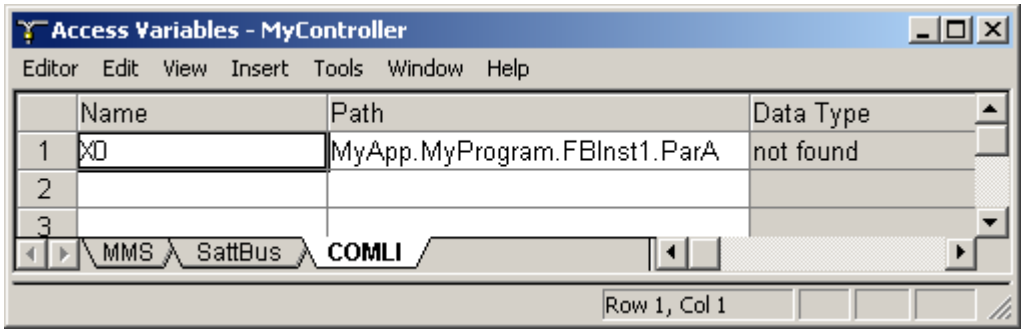


Figure 38

```
AccessVariables avars = ObjectFactory.NewAccessVariables();
// Add a "MMS" protocol" to the collection
VANamedProtocol mms = avars.VAProtocols.AddVANamedProtocol1("MMS");
// Define two MMS access variables named "MMSVar1" and "MMSVar2"
mms.Add2("MMSVar1", "MyApp.MyProgram.Var1", "readonly", 1);
mms.Add2("MMSVar2", "MyApp.GlobVar2", "", 3);

// Add a "COMLI" protocol" to the collection
VAAddressedProtocol comli = avars.VAProtocols.AddVAAddressedProtocol1("Comli");
// Define one COMLI access variable named "X0"
comli.Add2("X0", "MyApp.MyProgram.FBInst1.ParA", 1);

// Serialize the objects to an XMLString and update the Control Builder.
string bucket = cb.SetAccessVariables("MyController", avars.Serialize());
```

5.3.25 Modify existing Access Variables

Assume the task is to modify the access variables created in the previous example. The “Path” of the variables should now have values according to the figure below. In addition, the “MMSVar2” should be located on row 2.

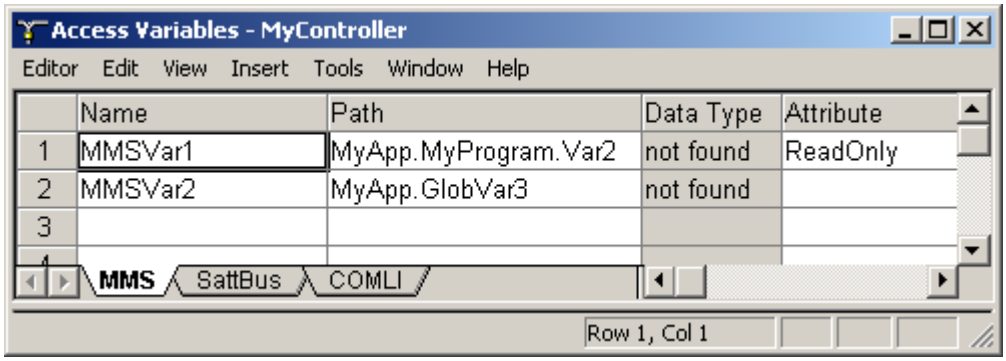


Figure 39

```
// Get an XML description of the access variables from the Control Builder
string XMLStr = cb.GetAccessVariables("MyController");
// Deserialize to objects
AccessVariables avars = ObjectFactory.DeserializeAccessVariables(ref XMLStr);
// Find the "MMS" Protocol
VANamedProtocol mms = (VANamedProtocol) avars.VAProtocols.Find("MMS");
VANamedVariable var = mms.Find("MMSVar1");
if (var != null)
{
  var.Path = "MyApp.MyProgram.Var2";
}
var = mms.Find("MMSVar2");
if (var != null)
{
  var.Path = "MyApp.GlobVar3";
  var.Row = 2;
}
// Serialize the objects to an XMLString and update the Control Builder.
string bucket = cb.SetAccessVariables("MyController", avars.Serialize());
```

5.3.26 Delete Access Variables

Assume the task is to delete one of the access variables created in the previous examples. The variable “MMSVar1” should be removed.

```
// Get an XML description of the access variables from the Control Builder
string XMLStr = cb.GetAccessVariables("MyController");
// Deserialize to objects
AccessVariables avars = ObjectFactory.DeserializeAccessVariables(ref XMLStr);
// Find the "MMS" Protocol
VANamedProtocol mms = (VANamedProtocol) avars.VAProtocols.Find("MMS");
int Nr = mms.FindNr("MMSVar1");
if (Nr > 0)
{
  mms.Remove(Nr);
}
// Serialize the objects to an XMLString and update the Control Builder.
string bucket = cb.SetAccessVariables("MyController", avars.Serialize());
```

5.3.27 Add a new Hardware unit

Assume the task is to create hardware units, in the controller “MyController”, according to the figures below.

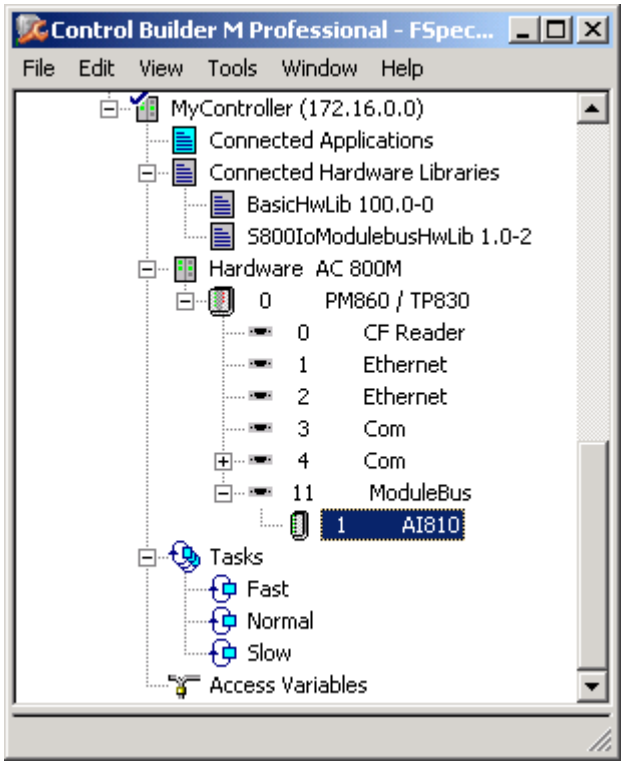


Figure 40

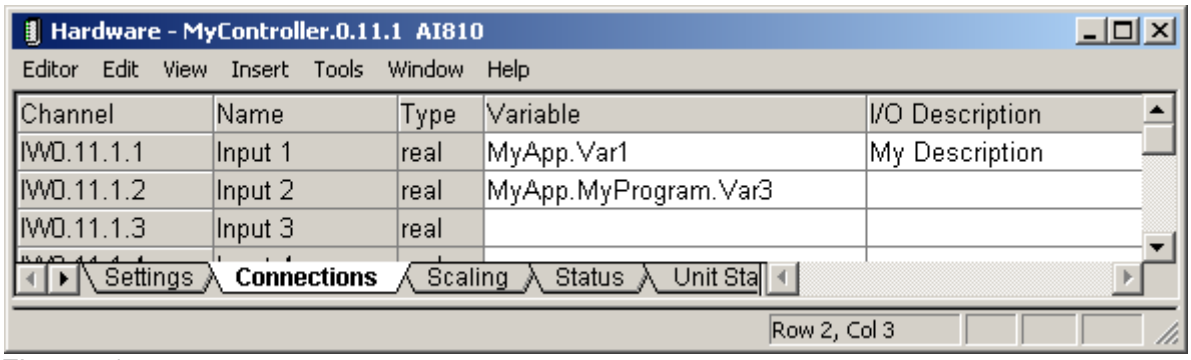


Figure 41

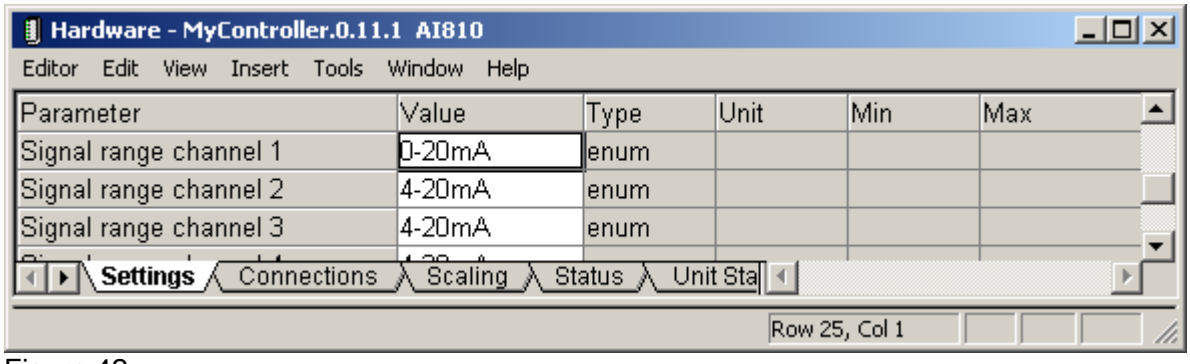


Figure 42

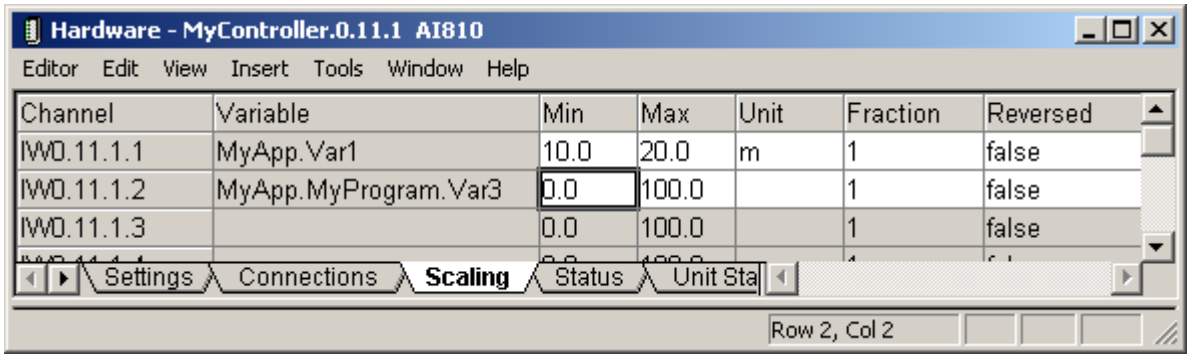


Figure 43

```
string bucket;
// Create a "PM860 / TP830" unit
bucket = cb.NewHardwareUnit("MyController.0", "PM860 / TP830", "","","");

// Create some objects
HWUnit hw = ObjectFactory.NewHWUnit("MyController.0.11.1");
hw.ParameterSettings.Add1("SignalRange_1", "0-20mA");
hw.HWChannels.Add2("IW0.11.1.1", "Output 1", "MyApp.Var1", "My Description",
                   "10.0", "20.0", "m", "1", false);
hw.HWChannels.Add1("IW0.11.1.2", "Output 2", "MyApp.MyProgram.Var3", "" );
// Serialize the objects to an XMLString and create an "AI810" unit,
// with content according to the XMLString, in the Control Builder.
bucket = cb.NewHardwareUnit(hw.Path, "AI810", "",
                            hw.Serialize(), "");
```

5.3.28 List the parameter setting names for a certain Hardware unit

If you read the previous example you might wonder “Which parameter setting names should I use?”

This example shows how to print out the parameter setting names for a certain hardware unit type. First create the hardware unit using the Control Builder. Then the code below produces the following print out.

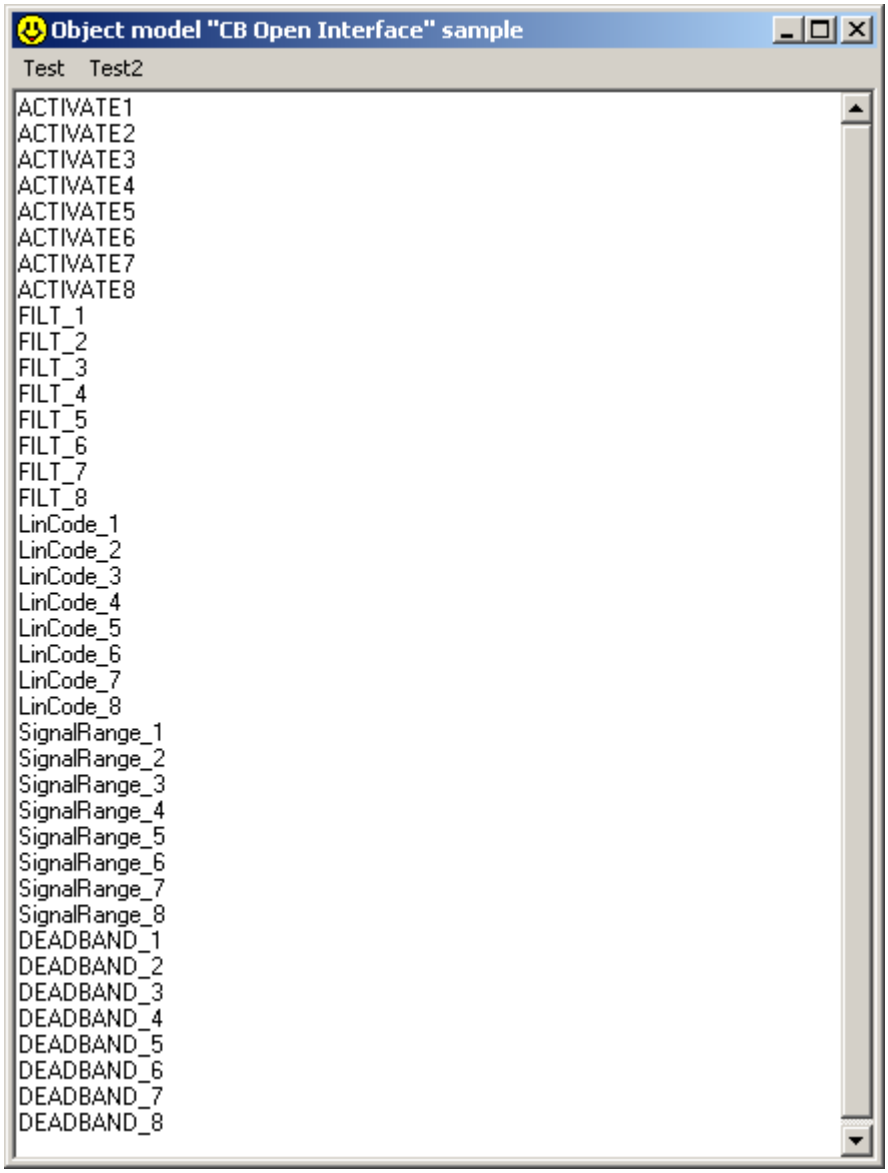


Figure 44.

```
// Get an XML description of the HWUnit from the Control Builder
string XMLString = cb.GetHardwareUnit("MyController.0.11.1", false);
// Deserialize the XMLString to objects
HWUnit hw = ObjectFactory.DeserializeHWUnit(ref XMLString);

foreach (ParameterSetting ps in hw.ParameterSettings)
{
  richTextBox1.Text += ps.Name + "\n";
}
```

5.3.29 Modify the settings and connections of an existing Hardware unit

Assume the task is to modify the hardware unit created in the previous example. The parameter setting “SignalRange_1” should now be “0-10V” and the variable connection of channel ”IW0.11.1.1” should be “MyApp.Prog2.FBInst3.Var4”.

```
// Get an XML description of the hardware unit "MyController.0.11.1"
// from the Control Builder
string XMLStr = cb.GetHardwareUnit("MyController.0.11.1", false);
// Deserialize to objects
HWUnit hw = ObjectFactory.DeserializeHWUnit(ref XMLStr);
ParameterSetting ps = hw.ParameterSettings.Find("SignalRange_1");
if (ps != null)
{
  ps.ParameterValue = "0-10V";
}
HWChannel ch = hw.HWChannels.Find("IW0.11.1.1");
if (ch != null)
{
  ch.ConVariable = "MyApp.Prog2.FBInst3.Var4";
}
// Serialize the objects to an XMLString and update the Control Builder.
string bucket = cb.SetHardwareUnit(hw.Path, hw.Serialize());
```

5.3.30 Delete a Hardware unit

Assume the task is to delete the hardware unit `"MyController.0.11.1"`.

```
cb.DeleteHardwareUnit("MyController.0.11.1", false);
```

5.3.31 Add a new Task

The task is to create a “Task” according to the following specification.

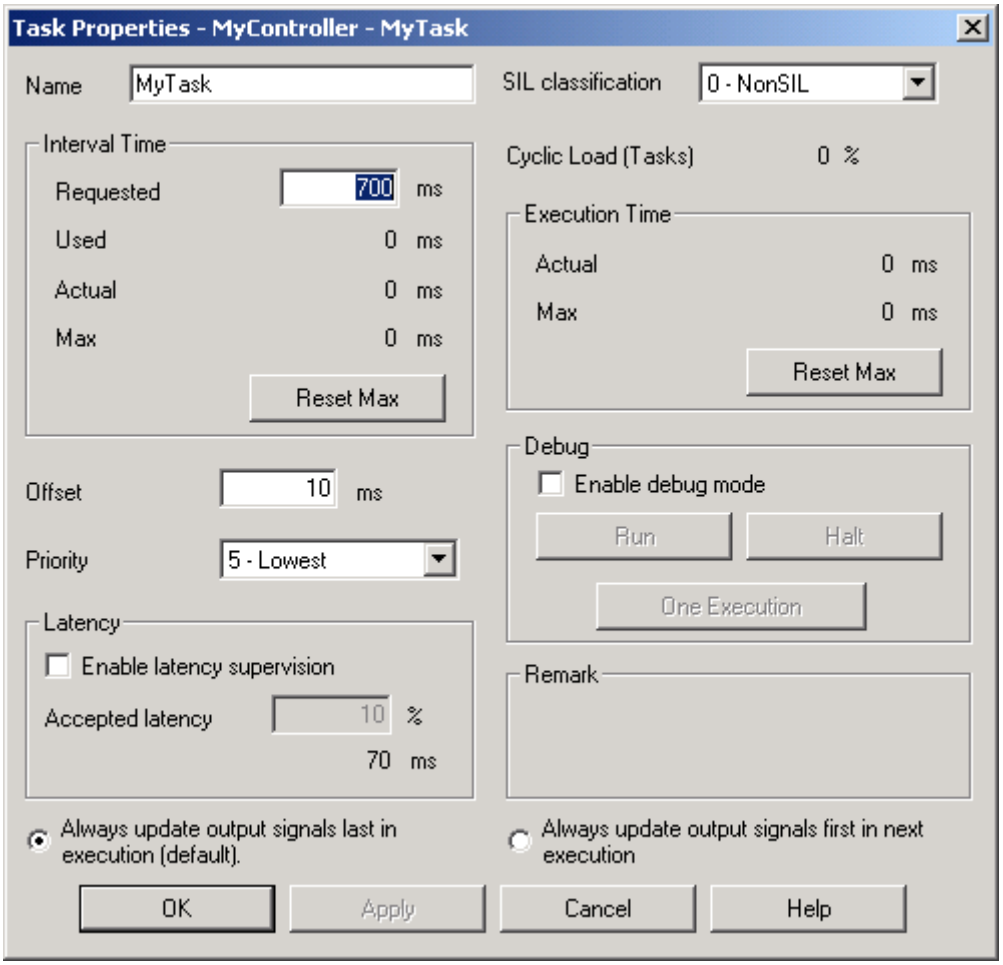


Figure 45

```
// Create, and initialize, a Task object.
Task t = ObjectFactory.NewTask1("MyTask", 700, TaskPriorityValue.cbTaskPriority5,
                                10, OutputUpdateValue.cbUpdateLast);
// Serialize the objects to an XMLString and create the task,
// with content according to the XMLString, in the Control Builder.
cb.NewTask(t.Name, "MyController", t.Serialize());
```

5.3.32 Modify an existing Task

Assume the task is to modify the “Task” we created in the previous example. The interval time should be 500 and the priority should be 3.

```
// Get an XML description of the task from the Control Builder
string XMLStr = cb.GetTask("MyController.MyTask");
// Deserialize to objects
Task t = ObjectFactory.DeserializeTask(ref XMLStr);
t.IntervalTime = 500;
t.Priority = TaskPriorityValue.cbTaskPriority3;
t.LatencySupervision = true;
t.LatencyPercentage = 44;
t.TaskSILLevel = TaskSILLevelValue.cbTaskSIL2;
// Serialize the objects to an XMLString and updater the task in the Control Builder.
cb.SetTask("MyController.MyTask", t.Serialize());
```

5.3.33 Delete a Task

Assume the task is to delete the Task **"MyController.MyTask"**.

```
cb.DeleteTask("MyController.MyTask");
```

5.3.34 Connect Applications to a Controller

Assume the task is to connect some applications, to the controller named “MyController”, according to the figure below.

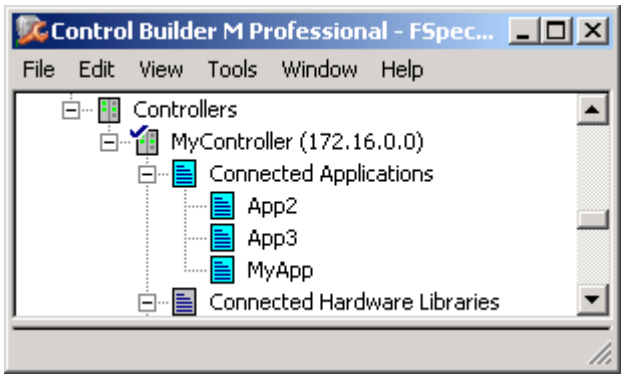


Figure 46

```
// Get an XML description of the Connected Applications from the Control Builder
string XMLStr = cb.GetConnectedApplications("MyController");
// Deserialize to objects
ConnectedApplications conApplics =
                        ObjectFactory.DeserializeConnectedApplications(ref XMLStr);
conApplics.Add1("App2");
conApplics.Add1("App3");
conApplics.Add1("MyApp");
// Serialize the objects to an XMLString and update the Control Builder.
cb.SetConnectedApplications("MyController", conApplics.Serialize());
```

5.3.35 Modify Connected Applications

Assume the task is to modify the “connected applications” in the previous example. The “App2” should be removed and the application “App4 2.1/0” should be added.

```
// Get an XML description of the Connected Applications from the Control Builder
string XMLStr = cb.GetConnectedApplications("MyController");
// Deserialize to objects
ConnectedApplications conApplics =
        ObjectFactory.DeserializeConnectedApplications(ref XMLStr);
int Nr = conApplics.FindNr("App2");
if (Nr > 0)
{
  conApplics.Remove(Nr);
}
conApplics.Add1("App4");
// Serialize the objects to an XMLString and update the Control Builder.
cb.SetConnectedApplications("MyController", conApplics.Serialize());
```

5.3.36 Connect Hardware Libraries to a Controller

Assume the task is to connect some Hardware Libraries, to the controller named “MyController”, according to the figure below.

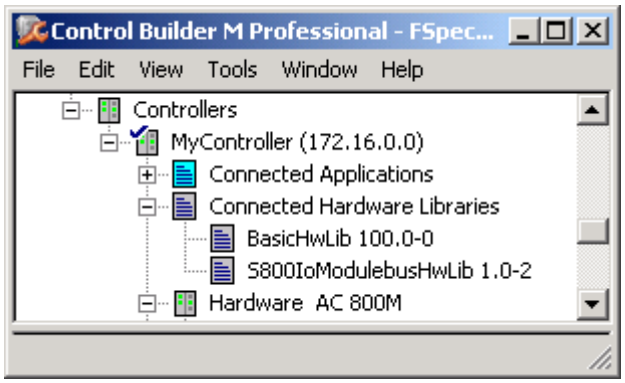


Figure 47

```
// Get an XML description of the Connected Hardware Libraries from the Control
// Builder
string XMLStr = cb.GetConnectedHardwareLibraries("MyController");
// Deserialize to objects
ConnectedHWLibraries conLibs =
          ObjectFactory.DeserializeConnectedHWLibraries(ref XMLStr);
conLibs.Add2("BasicHwLib",100,0,0);
conLibs.Add2("S800IoModulebusHwLib",1,0,2);
// Serialize the objects to an XMLString and update the Control Builder.
cb.SetConnectedHardwareLibraries("MyController", conLibs.Serialize());
```

5.3.37 Connect Libraries to an Application or to a Library

Assume the task is to connect some libraries, to the application named “MyApp”, according to the figure below.

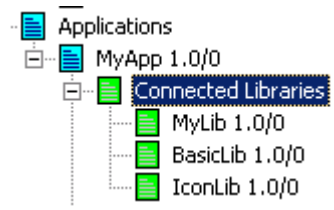


Figure 48

```
// Get an XML description of the Connected Libraries from the Control Builder
string XMLStr = cb.GetConnectedLibraries("MyApp");
// Deserialize to objects
ConnectedLibraries conLibs =
          ObjectFactory.DeserializeConnectedLibraries(ref XMLStr);
conLibs.Add2("MyLib",1,0,0);
conLibs.Add2("BasicLib",1,0,0);
conLibs.Add1("IconLib");
// Serialize the objects to an XMLString and update the Control Builder.
cb.SetConnectedLibraries("MyApp", conLibs.Serialize());
```

5.3.38 Modify Connected Libraries

Assume the task is to modify the “connected libraries” in the previous example. The “Mylib 2.0/3” should be connected instead of the “Mylib 1.0/0” library and “IconLib 1.0/0” should be removed. The result should be according to the figure below.

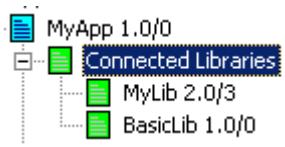


Figure 49

```
// Get an XML description of the Connected Libraries from the Control Builder
string XMLStr = cb.GetConnectedLibraries("MyApp");
// Deserialize to objects
ConnectedLibraries conLibs =
                   ObjectFactory.DeserializeConnectedLibraries(ref XMLStr);
ConnectedLibrary conLib = conLibs.Find("MyLib");
if (conLib != null)
{
   // The library was found. Change the Major, Minor and Revision numbers
   conLib.MajorVersion = 2;
   conLib.MinorVersion = 0;
   conLib.Revision = 3;
}
int Nr = conLibs.FindNr("IconLib");
if (Nr > 0)
{
   // The "IconLib" was found. Remove the library
   conLibs.Remove(Nr);
}

// Serialize the objects to an XMLString and update the Control Builder.
cb.SetConnectedLibraries("MyApp", conLibs.Serialize());
```

5.3.39 Create some new project constants

Assume the task is to create some new project constants according to the figure below. The code below will create all constants in the “MyConstantsGroup” and the “Pi” constant.

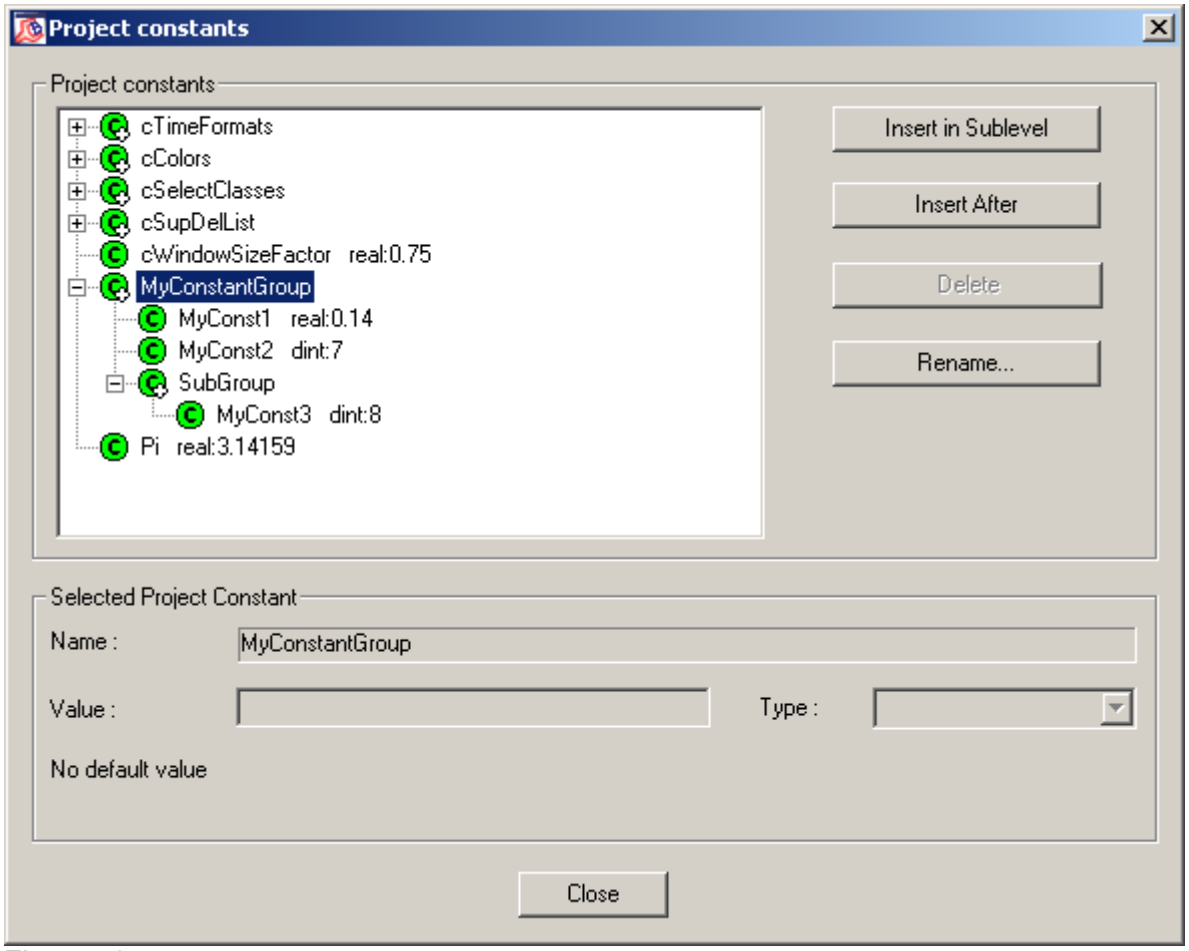


Figure 50

```
// Get an XML description of the Project Constants from the Control Builder
string XMLStr = cb.GetProjectConstants();
// Deserialize to objects
ProjectConstants projConsts = ObjectFactory.DeserializeProjectConstants(ref XMLStr);
projConsts.Add1("MyConstantGroup.MyConst1", "real", "0.14");
projConsts.Add1("MyConstantGroup.MyConst2", "dint", "7");
projConsts.Add1("MyConstantGroup.SubGroup.MyConst3", "dint", "8");
projConsts.Add1("Pi", "real", "3.14159");
// Serialize the objects to an XMLString and update the Control Builder.
string bucket = cb.SetProjectConstants(projConsts.Serialize());
```

5.3.40 Modify existing project constants

Assume the task is to modify some of the project constants in the previous example. The constant “MyConstantsGroup.MyConst2” should be a “real” and have the value “8.0”. The “Pi” constant should be removed.

```
// Get an XML description of the Project Constants from the Control Builder
string XMLStr = cb.GetProjectConstants();
// Deserialize to objects
ProjectConstants projConsts = ObjectFactory.DeserializeProjectConstants(ref XMLStr);
ProjectConstant pc = projConsts.Find("MyConstantGroup.MyConst2");
if (pc != null)
{
  pc.PCType = "real";
  pc.Value = "8.0";
}
int Nr = projConsts.FindNr("Pi");
if (Nr > 0)
{
  projConsts.Remove(Nr);
}
// Serialize the objects to an XMLString and update the Control Builder.
string bucket = cb.SetProjectConstants(projConsts.Serialize());
```

5.3.41 Message buckets

Almost all “New” and “Set” methods of the “CB Open Interface” returns a “Message bucket”. There are a lot of such examples in the previous chapters:

```
string bucket = cb.SetProjectConstants(projConsts.Serialize());
bucket = cb.SetDataType("MyLib.MyDataType", dt.Serialize());
bucket = cb.SetFunctionBlock("MyLib.MyFBType.RTC2", fb.Serialize());
bucket = cb.NewFunctionBlockType(fbType.Name,"MyLib", fbType.Serialize());
bucket = cb.NewHardwareUnit(hw.Path, "AI810", "", hw.Serialize(), "");
```

The Control Builder will perform an internal “compilation check” of the applied content in the examples above. The “check” result is returned as a “Message bucket” string.

A “Message bucket” is a container for information, warning and error texts. The code below shows how to display some of the content of a “Message bucket” in a textbox.

```
private void DisplayBucket_Click(string bucket)
{
  MessageBucket msgbucket =
                ObjectFactory.DeserializeMessageBucket(ref bucket);

  richTextBox1.Text = "NrOfErrors: " + msgbucket.NoOfErrors + " NrOfWarnings :"
                    + msgbucket.NoOfWarnings + "\n";
  foreach (IMsg m in msgbucket)
  {
    richTextBox1.Text += "Message: " + m.Message + " ";
    if (m.IsErrorMsg)
    {
       ErrorMsg emsg = (ErrorMsg) m;
       richTextBox1.Text += " ErrorNr : " + emsg.ErrorNo;
       DisplayPosInfo(emsg.PosInfo);
       DisplayExtraInfo(emsg.ExtraInfo);
    }
    else if (m.IsWarningMsg)
    {
       WarningMsg wmsg =(WarningMsg)m;
       richTextBox1.Text += " WarningNr : " + wmsg.WarningNo;
       DisplayPosInfo(wmsg.PosInfo);
       DisplayExtraInfo(wmsg.ExtraInfo);
    }
  }
}
```

```
private void DisplayPosInfo(PosInfo p)
{
  if (p != null)
  {
```

```
    richTextBox1.Text += " FOUName: " + p.FOUName;
    richTextBox1.Text += " POUName: " + p.POUName + "\n";
    richTextBox1.Text += " Row: " + p.Row;
    richTextBox1.Text += " Col: " + p.Col;
    richTextBox1.Text += " TabName: " + p.TabName + "\n";
    richTextBox1.Text += " StartPos: " + p.StartPos;
    richTextBox1.Text += " EndPos: " + p.EndPos;
    richTextBox1.Text += " ElementName: " + p.MessageType.ToString() + "\n\n";
  }
}

private void DisplayExtraInfo(ExtraInfo e)
{
  if (e != null)
  {
    richTextBox1.Text += " JumpDest: " + e.JumpDest;
    richTextBox1.Text += " VarName: " + e.VarName + "\n";
    richTextBox1.Text += " FunctionName: " + e.FunctionName;
    richTextBox1.Text += " ExpectedType: " + e.ExpectedType + "\n";
    richTextBox1.Text += " TraverseNo: " + e.TraverseNo + "\n\n";
  }
}
```

5.3.42 Example of using the ReservedByFunction property

If the ***ReservedByFunction*** property is set to an not empty string then the corresponding type (or singleton or HWUnit) becomes “Read-only”. That is, the type (or the singleton) can not be deleted, can not be renamed and can not be altered via the graphical user interface of the Control Builder.

The example below shows how to set this property for an existing function block type.

```
// Get an XML description of the type from the Control Builder
string XMLStr = cb.GetFunctionBlockType("MyLib.MyFBType");
// Deserialize the XMLString into Objects
FunctionBlockType fbType = ObjectFactory.DeserializeFunctionBlockType(ref XMLStr);
// If the ReservedByFunction property is set to an not empty string then the type
// becomes "Read-only" i.e. the editors and dialogs of the Control Builder
// will prohibit a user from altering the content of the type or from deleting
// or Renaming the type.
fbType.ReservedByFunction = "Functional Designer";
// Serialize the objects into an XMLString and update the
// FunctionBlockType in the Control Builder
string bucket = cb.SetFunctionBlockType("MyLib.MyFBType", fbType.Serialize());
```

If the ***ReservedByFunction*** property is set to **an empty** string then the corresponding type (or singleton or HWUnit) no longer becomes “Read-only”.

5.3.43 Add a new Diagram

Assume the task is to create a new Diagram with content according to the figures below

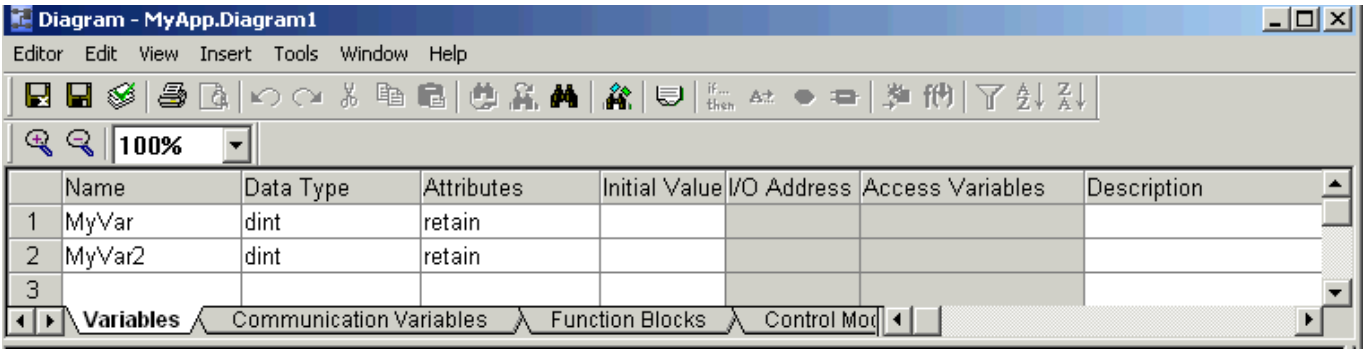


Figure 51 – shows the variables belonging to the diagram

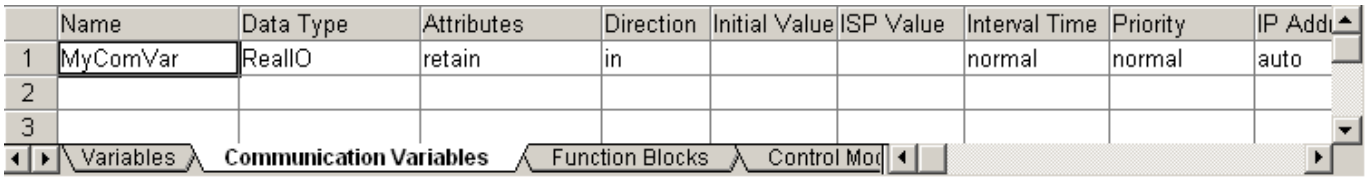


Figure 52 – shows the communication variable(s) belonging to the diagram

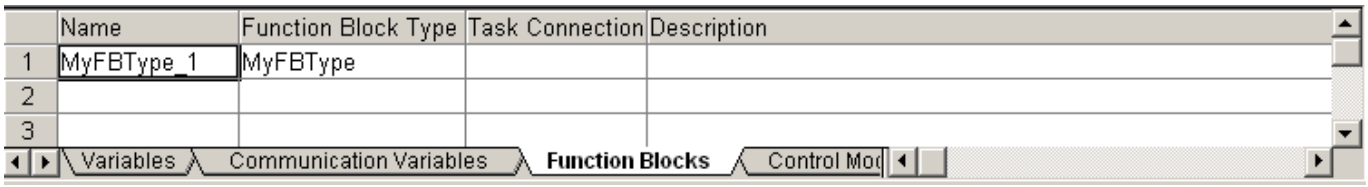


Figure 53 – shows the Function Block(s) belonging to the diagram

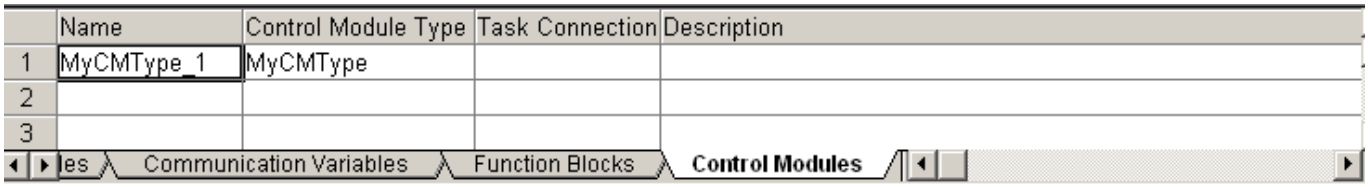


Figure 54 – shows the Control Module(s) belonging to the diagram

```
// Create a "Diagram" object in the client's process memory
Diagram diag = ObjectFactory.NewDiagram("Diagram1", "description of");
// Add variables, communication variables, function blocks and control modules
diag.Variables.Add1("MyVar", "dint");
diag.Variables.Add2("MyVar2", "dint","retain","","","","");
diag.CommVariables.Add1("MyComVar", "RealIO", "in");
diag.FunctionBlocks.Add1("MyFBType_1", "MyFBType");
diag.ControlModules.AddControlModule("MyCMType_1", "MyCMType");

// Finally, serialize the object model into an XML String and
// call the OpenIF method "NewDiagram" in order to create the diagram
// in the Control Builder EXE.
string bucket = cb.NewDiagram(diag.Name, "MyApp", diag.Serialize());
```

5.3.44 How to create or modify a Function Diagram code block

The previous example didn't create any code block. Unfortunately, due to development costs and resources available in SV5.1 we haven't yet developed any true object model for the function diagram code block language. Instead there is a possibility to describe the code block part as an XML String. The example below is a continuation of the previous example. Figure 55 shows the function diagram in an editor and figure 56 provides the XML necessary to define the code block shown in Figure 55.

```
// Create a "Diagram" object in the client's process memory
Diagram diag = ObjectFactory.NewDiagram("Diagram1", "description of");
// Add variables, communication variables, function blocks and control modules
diag.Variables.Add1("MyVar", "dint");
diag.Variables.Add2("MyVar2", "dint","retain","","","","");
diag.CommVariables.Add1("MyComVar", "RealIO", "in");
diag.FunctionBlocks.Add1("MyFBType_1", "MyFBType");
diag.ControlModules.AddControlModule("MyCMType_1", "MyCMType");

// This part defines the Function Diagram code block
FDCodeBlock fdCodeBlock = new FDCodeBlock();
fdCodeBlock.Name = "MyFDCodeBlock";
// Figure 55-56 i.e. the examples below contains a possible FDAsXMLStr
fdCodeBlock.FDAsXMLStr = TheContentOfFDXMLExampleAsAString;
diag.CodeBlocks.AddFDCodeBlock(fdCodeBlock);

// Finally, serialize the object model into an XML String and
// call the OpenIF method "NewDiagram" in order to create the diagram
// in the Control Builder EXE.
string bucket = cb.NewDiagram(diag.Name, "MyApp", diag.Serialize());
```

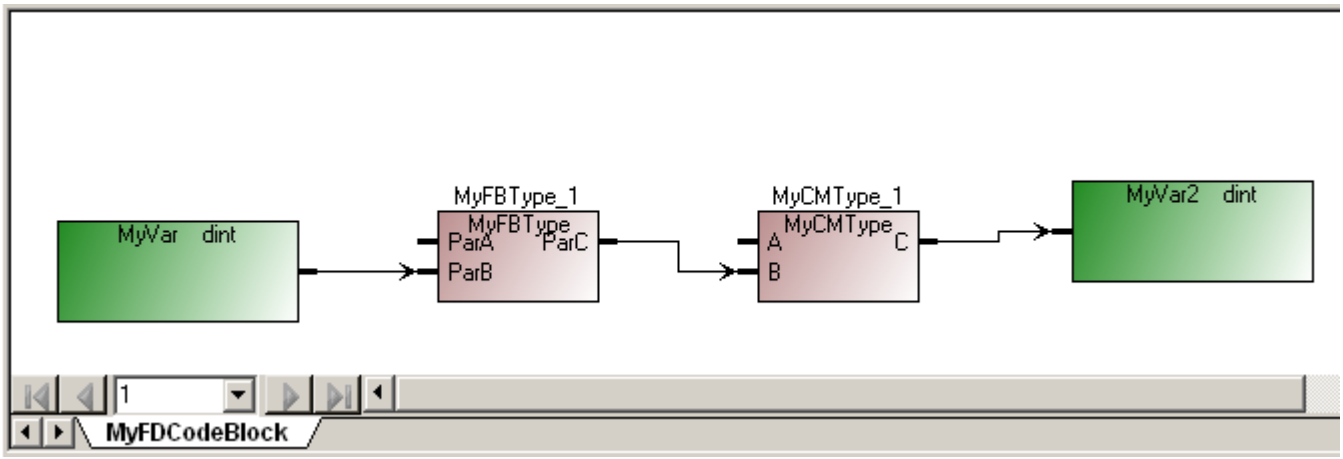


Figure 55 – shows the code block in an Editor

```
<FunctionDiagram Name="MyFDCodeBlock" >
    <Pages>
      <Layout FormatName="A4" Orientation="Portrait" Unit="0.01 mm" Width="21000"
              Height="29700" Top="100" Bottom="100" Left="100" Right="100"/>
      <Page PageNo="1">
        <Blocks>
          <Invocation Name="MyFBType_1" DataFlowOrder="1" Id="">
            <Layout X="5156" Y="2794" Width="100" Height="45">
              <Ports>
                <Port Name="ParA" Visible="true"/>
                <Port Name="ParB" Visible="true"/>
                <Port Name="ParC" Visible="true"/>
              </Ports>
```

```
            </Layout>
          </Invocation>
          <Invocation Name="MyCMType_1" DataFlowOrder="2" Id="">
            <Layout X="9220" Y="2794" Width="100" Height="45">
              <Ports>
                <Port Name="A" Visible="true"/>
                <Port Name="B" Visible="true"/>
                <Port Name="C" Visible="true"/>
              </Ports>
            </Layout>
          </Invocation>
          <DataRef Name="MyVar" Id="">
            <Layout X="330" Y="2921" Width="140" Height="50"></Layout>
          </DataRef>
          <DataRef Name="MyVar2" Id="">
            <Layout X="13208" Y="2413" Width="140" Height="50"></Layout>
          </DataRef>
        </Blocks>
        <DataConnections>
          <DataConnection Src="MyFBType_1.ParC" Dest="MyCMType_1.B"/>
          <DataConnection Src="MyVar" Dest="MyFBType_1.ParB"/>
          <DataConnection Src="MyCMType_1.C" Dest="MyVar2"/>
        </DataConnections>
      </Page>
    </Pages>
  </FunctionDiagram>
```

Figure 56 – shows the XML defining the code block

Note! Figure 56 is only an XML example valid at the time this document was written. The XML Schema might be modified in the future so please read the XML Schema for the version you use in order to get appropriate information.

5.3.45 Modify an existing Diagram

Assume the task is to modify the Diagram we have created in the previous two examples. The task is to add a variable named `MyVar3` and a communication variable named `MyComVar3` and a function block named `MyFBType_3`. Possibly changes of the FD Code block could be achieved as described in the previous example.

```
string XMLStr = cb.GetDiagram("MyApp.Diagram1");
Diagram diag = ObjectFactory.DeserializeDiagram(ref XMLStr);
// Add variable, communication variable and a function block
diag.Variables.Add2("MyVar3", "dint", "retain", "", "", "", "");
diag.CommVariables.Add1("MyComVar3", "RealIO", "in");
diag.FunctionBlocks.Add1("MyFBType_3", "MyFBType");
string bucket = cb.SetDiagram("MyApp.Diagram1", diag.Serialize());
```

5.3.46 Delete an existing Diagram

Assume the task is to delete the diagram **"MyApp.Diagram1"**.

```
cb.DeleteDiagram("MyApp.Diagram1");
```

5.3.47 Communication Variables

Communication Variables exists in Diagrams, Programs and top level SingleControlModules. Communication variables are handled in the same way as ordinary variables from an object model perspective i.e. the previous examples regarding variables reveals a pattern valid for communication variables as well.

This example shows how to add a Communication Variable, according to the following specification, to an existing Program. The path to the type is “MyApp.Program1”.

Name	Type	Attribute	Direction	IntervalTime	ISP Value
CommVarNew	dint	retain	in	normal	32

Figure 57.

We can solve this problem in two different ways. Both alternatives are presented below.

Alternative 1.

```
// Create and initialize a Communication Variable object
CommVariable cvar2 = ObjectFactory.NewCommVariable("CommVarNew", "dint", "in");
cvar2.IntervalTime = "normal";
cvar2.ISPValue = "32";
// Serialize the Variable object to an XMLString and call the
// "CB Open Interface" method NewVariable.
cb.NewVariable(CBOpenIFVariableType.OI_COMMUNICATIONVARIABLE, cvar2.Name,
               cvar2.TypeName, "MyApp.Program1", cvar2.Serialize());
```

Alternative 2.

```
// Get an XML description of the Program from the Control Builder
string XMLStr = cb.GetProgram("MyApp.Program1");
// Deserialize the XMLString into Objects
Program prog = ObjectFactory.DeserializeProgram(ref XMLStr);
// Add a Communication Variable
CommVariable cvar = prog.CommVariables.Add1("CommVarNew", "dint", "in");
cvar.IntervalTime = "normal";
cvar.ISPValue = "32";
// Serialize the objects into an XMLString and update the
// Program in the Control Builder
string bucket = cb.SetProgram("MyApp.Program1", prog.Serialize());
```

The first alternative is the most efficient solution in this example. However, if you would like to create several variables etc at a time, the second alternative is the most efficient.

5.3.47.1 Set or modify SIL properties for Communication Variables

Assume the task is to set the following properties of an existing Communication Variable:

Property Name	Value
ExpectedSIL	"SIL2"
AcknowledgeGroup	2
UniqueID	798
ISPValue	2.0

Figure 58

```
// Get an XML description of the Program from the Control Builder
string XMLStr = cb.GetProgram("MyApp.Program1");
// Deserialize the XMLString into Objects
Program prog = ObjectFactory.DeserializeProgram(ref XMLStr);
// Add a Communication Variable
CommVariable cvar = prog.CommVariables.Find("CV6");
if (cvar == null)
{ // We didn't find the variable. Create a new one instead!
  cvar = prog.CommVariables.Add1("CV6", "real", "in");
}
cvar.ExpectedSIL = "SIL2";
cvar.AcknowledgeGroup = "2";
cvar.UniqueID = 798;
cvar.ISPValue = "2.0";

// Serialize the objects into an XMLString and update the
// Program in the Control Builder
string bucket = cb.SetProgram("MyApp.Program1", prog.Serialize());
```

5.3.48 Init Values (Instance specific init values)

It is possible to define ***InitValues*** (also called instance specific initial values) for the following kind of POUs: Diagrams, Programs and SingleControlModules. In the case of SingleControlModules the ***InitValues*** collection is available in the type part i.e. is present in the SingleControlModuleType class and is only allowed for single control modules belonging to singleton types.

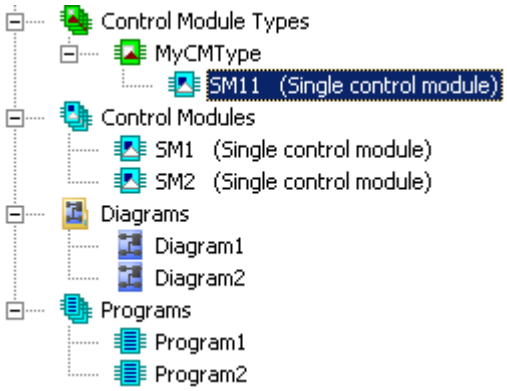


Figure 59

Consider figure 59 were it is possible to define ***InitValues*** for SM1, SM2, Diagram1, Diagram2, Program1 and Program2. But it isn't possible to define ***InitValues*** for SM11 belonging to MyCMType.

The following example shows how to set init values for a program with the following specification

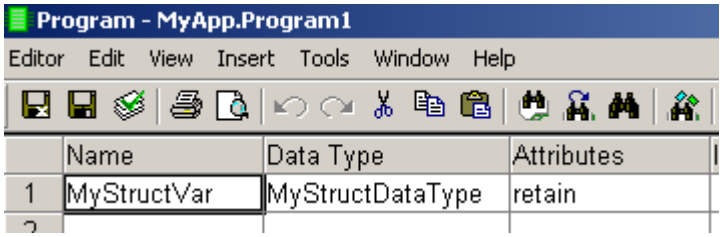


Figure 60

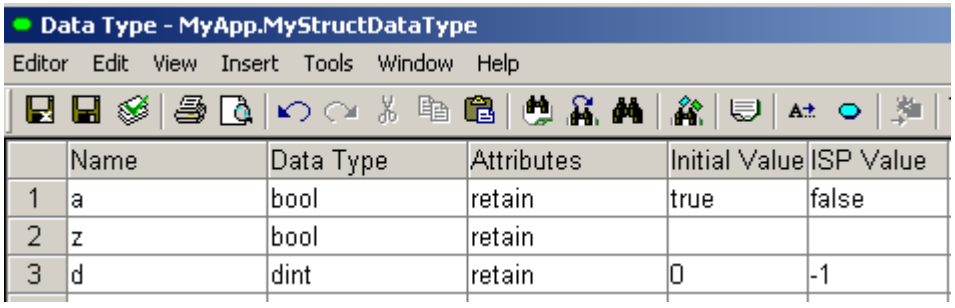


Figure 61

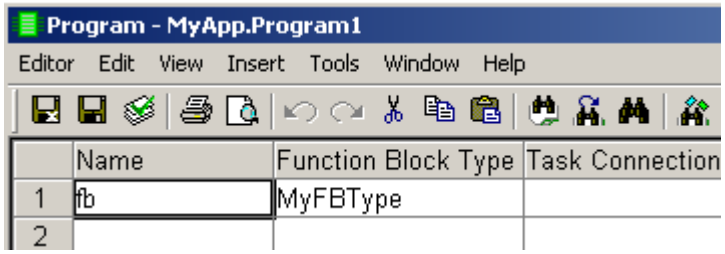
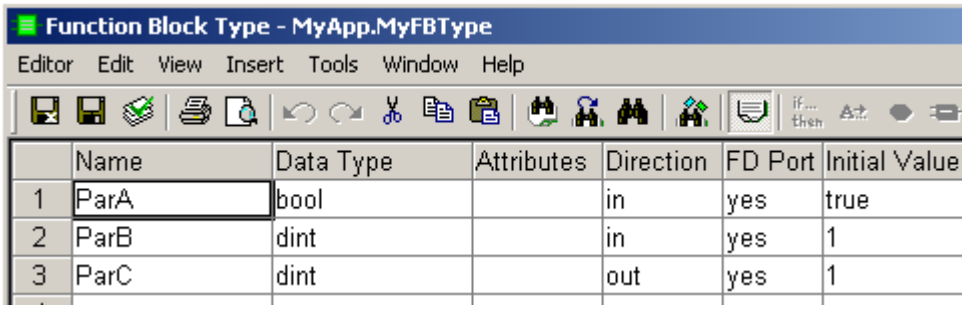


Figure 62



	Name	Data Type	Attributes	Direction	FD Port	Initial Value
1	ParA	bool		in	yes	true
2	ParB	dint		in	yes	1
3	ParC	dint		out	yes	1

Figure 63

Consider figure 59-62 above. The task in this example is to assign init values to the variable and function block parameters in the program according to the following table:

Name	**Value**
MyStructVar.a	false
MyStructVar.d	7
fb.ParA	false
fb.ParB	-1

Table 2

```
// Get an XML description of the Program from the Control Builder
string XMLStr = cb.GetProgram("MyApp.Program1");
// Deserialize the XMLString into Objects
Program prog = ObjectFactory.DeserializeProgram(ref XMLStr);
// Define init values according to table 2 above
InitValue ival = prog.InitValues.Find("", "MyStructVar.a");
if (ival != null)
{ // The Init value already exist but might have another value
  ival.Value = "false";
}
else
{ // Create a Init value
  prog.InitValues.Add1("", "MyStructVar.a", "false");
}

ival = prog.InitValues.Find("", "MyStructVar.d");
if (ival != null)
{ // The Init value already exist but might have another value
  ival.Value = "7";
}
else
{ // Create a Init value
  prog.InitValues.Add1("", "MyStructVar.d", "7");
}

ival = prog.InitValues.Find("fb", "ParA");
if (ival != null)
{ // The Init value already exist but might have another value
```

```
    ival.Value = "false";
  }
  else
  { // Create a Init value
    prog.InitValues.Add1("fb", "ParA", "false");
  }

  ival = prog.InitValues.Find("fb", "ParB");
  if (ival != null)
  { // The Init value already exist but might have another value
    ival.Value = "-1";
  }
  else
  { // Create a Init value
    prog.InitValues.Add1("fb", "ParB", "-1");
  }
  // Serialize the objects into an XMLString and update the
  // Program in the Control Builder
  string BucketXML = cb.SetProgram("MyApp.Program1", prog.Serialize());
```

5.3.49 Execution order

This chapter describes how to change the execution order between diagrams. Assume a an Application have the following diagrams connected to the following tasks

The diagrams executes in the following order within the task named “Normal”:

1. Diagram1
2. Diagram2

The diagrams executes in the following order within the task named “Fast”:

1. Diagram3
2. Diagram4
3. Diagram5
4. Diagram6

The task is to change the task connections and the execution order according to the table below. We assume all tasks already exist in the controller:

The diagrams shall execute in the following order within the task named “Normal:

1. Diagram2

The diagrams shall execute in the following order within the task named “FastRenamed”:

1. Diagram5
2. Diagram3
3. Diagram6
4. Diagram4

The diagrams shall execute in the following order within the task named “NewTask:

1. Diagram1

One solution is presented below.

```
// Get an XML description of the execution order from the Control Builder
string execOrderXML = cb.GetExecutionOrder(CBOpenIFExecutionInstanceType.OI_DIAGRAMS,
                                           "MyApp");
// Deserialize the XMLString into Objects
ExecutionOrder execOrder = ObjectFactory.DeserializeExecutionOrder(ref execOrderXML);
// Locate the execution group (task connection) named "MyController.Fast"
ExecutionGroup egFast = execOrder.Find("MyController.Fast");
// Move Diagram5 before Diagram3 i.e first in list
egFast.Remove(egFast.FindNr("Diagram5"));
int nr3 = egFast.FindNr("Diagram3");
egFast.AddBefore(ObjectFactory.NewExecutionInstance("Diagram5"), nr3);
// and move nr4 (Diagram4) to the end
ExecutionInstance ei4 = egFast.Find("Diagram4");
egFast.Remove(egFast.FindNr("Diagram4"));
egFast.Add(ei4);

// Rename the "fast" taskconnection name to "FastRenamed"
egFast.TaskName = "MyController.FastRenamed";

// Add a new execution group named "MyController.NewTask"
// to the execution order list
ExecutionGroup egNewGroup = execOrder.Add1("MyController.NewTask");

// Search for a diagram named "Diagram1". If found then remove it from the
// current execution group and move it at the end of the
// execution group named "MyController.NewTask"
bool found = false;
for (int k = 1; k <= execOrder.Count && !found; k++)
{ // Note! 1-based collection
  ExecutionGroup execGroup = execOrder[k];
  int nrDiagram1 = execGroup.FindNr("Diagram1");
  found = nrDiagram1 >= 1;
  if (found)
  {
    execGroup.Remove(nrDiagram1);
    egNewGroup.Add1("Diagram1");
  }
}
// Serialize the objects into an XMLString and update the
// execution order in the Control Builder
cb.SetExecutionOrder(CBOpenIFExecutionInstanceType.OI_DIAGRAMS,
                     "MyApp",
                     execOrder.Serialize());
```

5.3.50 Add a new Diagram Type

Assume the task is to create a new Diagram Type with content according to the figures below

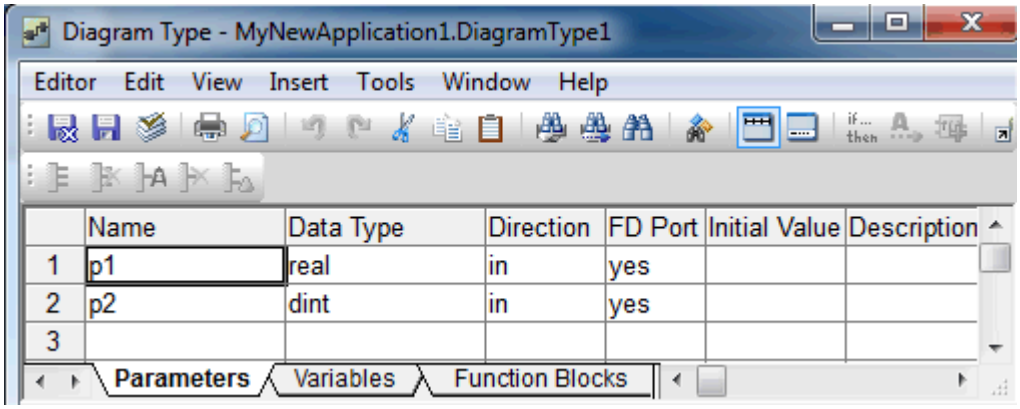


Figure 64 – shows the parameters belonging to the diagram type

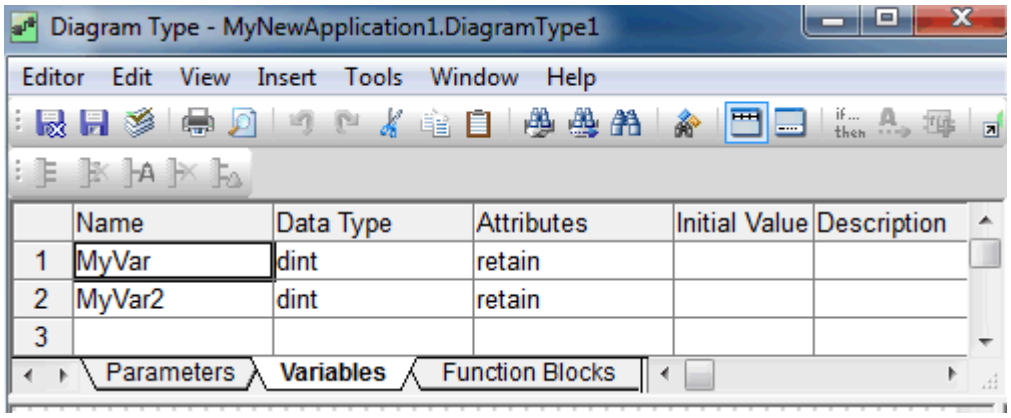


Figure 65 – shows the variables belonging to the diagram type

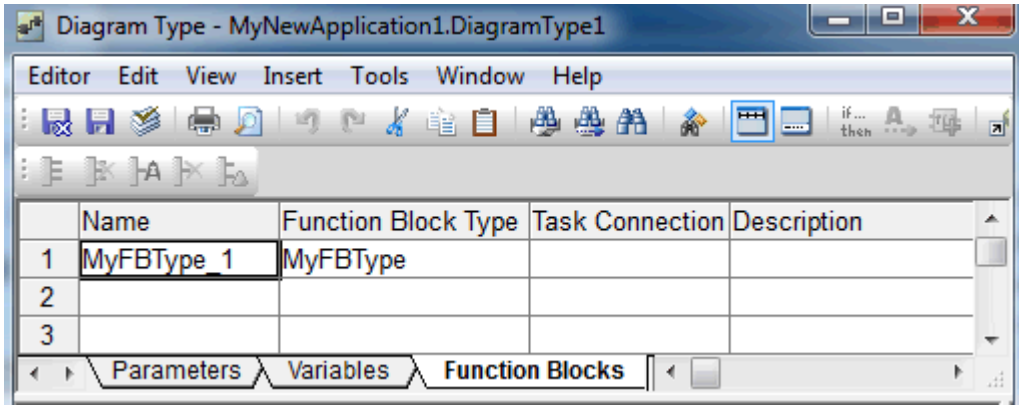


Figure 66 – shows the Function Block belonging to the diagram type

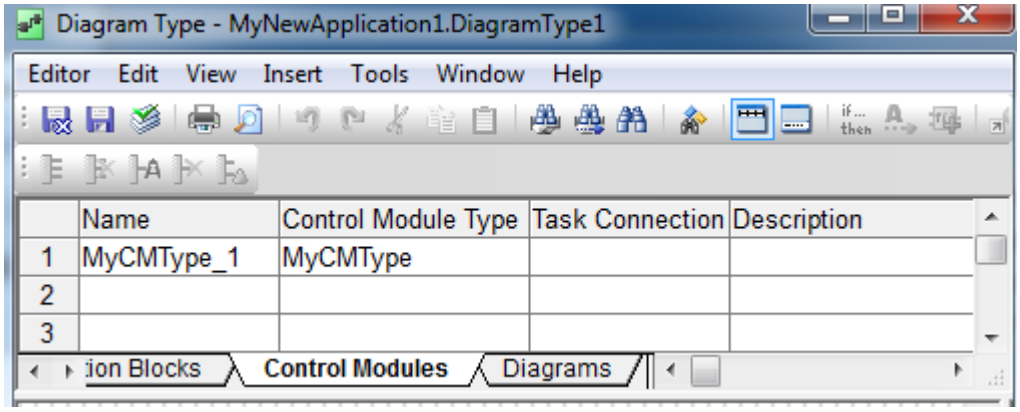


Figure 67 – shows the Control Module belonging to the diagram type

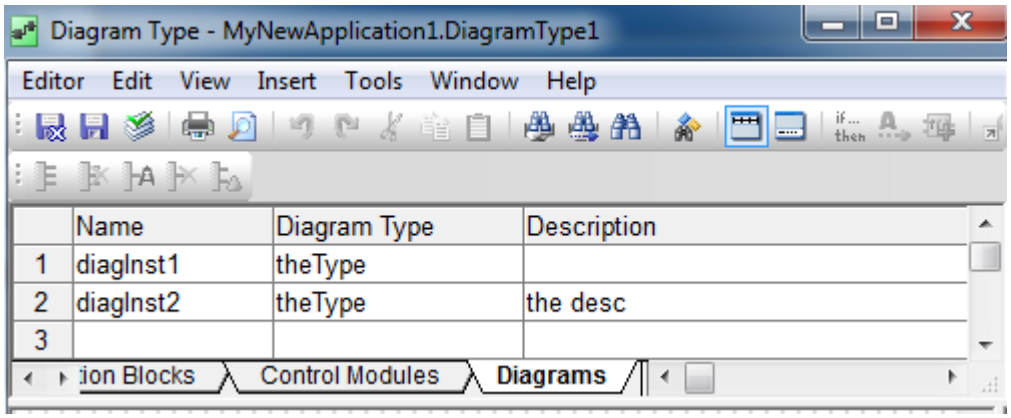


Figure 68 – shows the Diagram instances belonging to the diagram type

```
// Create a "Diagram Type" object in the client's process memory
DiagramType diag = ObjectFactory.NewDiagramType("DiagramType1", "desc");

// Add parameters, variables, function blocks and control modules and
// diagram instances
diag.Parameters.Add1("p1", "real");
diag.Parameters.Add1("p2", "dint");
diag.Variables.Add1("MyVar", "dint");
diag.Variables.Add2("MyVar2", "dint", "retain", "", "", "", "");
diag.FunctionBlocks.Add1("MyFBType_1", "MyFBType");
diag.ControlModules.AddControlModule("MyCMType_1", "MyCMType");
diag.DiagramInstances.Add1("diagInst1", "theType");
DiagramInstance diagInst = diag.DiagramInstances.Add2("diagInst2", "theType", "",
"the desc");
diagInst.AspectObject = false;
diagInst.ExposePropertiesInParent = true;

// Finally, serialize the object model into an XML String and
// call the OpenIF method "NewDiagramType" in order to create the diagram
// in the Control Builder EXE.
string bucket = cb.NewDiagramType(diag.Name, "MyNewApplication1", diag.Serialize());
```

5.3.51 How to create or modify a Function Diagram code block

The previous example didn't create any code block. Unfortunately, due to development costs and resources available in SV5.1 we haven't yet developed any true object model for the function diagram

code block language. Instead there is a possibility to describe the code block part as an XML String. This is described in chapter 5.3.44

5.3.52 Modify an existing Diagram Type

Assume the task is to modify the Diagram we have created in the previous two examples. The task is to add a variable named `MyVar3` and a diagram instance named **`DiagInst3`** and a function block named `MyFBType_3`. Possibly changes of the FD Code block could be achieved as described in chapter 5.3.44

```
string XMLStr = cb.GetDiagramType("MyNewApplication1.DiagramType1");
DiagramType diag = ObjectFactory.DeserializeDiagramType(ref XMLStr);
// Add a variable, a function block and a diagram instance
diag.Variables.Add2("MyVar3", "dint", "retain", "", "", "", "");
diag.FunctionBlocks.Add1("MyFBType_3", "MyFBType");
diag.DiagramInstances.Add1("DiagInst3", "Type2");
string bucket = cb.SetDiagramType("MyNewApplication1.DiagramType1",diag.Serialize());
```

5.3.53 Delete an existing Diagram Type

Assume the task is to delete the diagram **"MyNewApplication1.DiagramType1"**.

```
cb.DeleteDiagramType("MyNewApplication1.DiagramType1");
```

5.4 Some Visual Basic 6.0 examples

Only a few Visual Basic 6.0 examples are presented. I am convinced that you are able to translate the C# examples to Visual Basic 6.0 with help of the clues below.

5.4.1 Getting started

You have to add a reference to the COM DLL “Control Builder Open Interface Helper” before you can make use of the classes in the object model. Use the menu Project->Reference and a dialog will be shown.

You also have to add a reference to the “Control Builder Professional” EXE in order to be able to use the “CB Open Interface” methods. Use the menu Project->Reference and the dialog will be shown again. Select the COM component “Control Builder 2.0 Type Library” in the dialog and press OK.

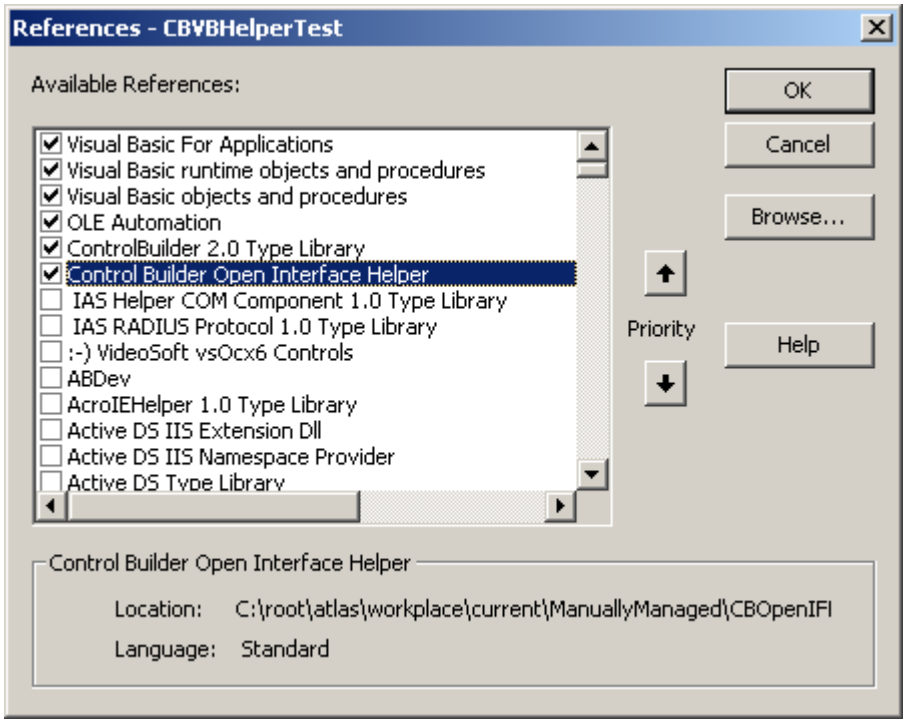


Figure 69

All examples below presume the presence of two reference variables.

```
Private cb As CONTROLBUILDERLib.CBOpenIF
Private ObjectFactory As CBOpenIFHelper.ObjectFactory
```

The first one is called “cb” (short for Control Builder) and is a reference to an object of the “CBOpenIF” COM Class. The second one is called “ObjectFactory” and is a reference to an object of the “ObjectFactory” COM Class. It is also assumed that the corresponding objects are created in a suitable function. The “Load” event function is a suitable place to create these objects for a project of the “Standard EXE” type. See the code below.

```
Option Explicit

Private cb As CONTROLBUILDERLib.CBOpenIF
Private ObjectFactory As CBOpenIFHelper.ObjectFactory

Private Sub Form_Load()
  Set cb = New CONTROLBUILDERLib.CBOpenIF
  Set ObjectFactory = New CBOpenIFHelper.ObjectFactory
End Sub
```

5.4.2 Create a new FunctionBlockType

The task is described in the C# chapter “Create a new FunctionBlockType”. The Visual Basic 6.0 solution is presented below.

```
  ' Create an object of the type "FunctionBlockType" in the client's
```

```
' process memory
Dim fbType As FunctionBlockType
Set fbType = ObjectFactory.NewFunctionBlockType("MyFBType", _
                                                "Description of MyFBType")
' Add some Variable objects
Call fbType.Variables.Add1("MyVariable", "bool")
Call fbType.Variables.Add2("X", "dint", "retain", "8", "", "", "Desc of X")
Dim var As Variable
Set var = fbType.Variables.Add1("Str", "string[32]")
var.Description = "Description of the variable"
' Add a ST CodeBlock
Dim stCode As String
stCode = "if MyVariable then" & vbCrLf & _
         "  X := X + 1;" & vbCrLf & _
         "end_if;"
Call fbType.CodeBlocks.AddSTCodeBlock2("MySTCodeBlock", stCode)
' Finally, serialize the object model into an XML String and
' call the OpenIF method "NewFunctionBlockType" in order to create the type
' in the Control Builder EXE.
Dim bucket As String
bucket = cb.NewFunctionBlockType(fbType.Name, "MyLib", fbType.Serialize())
```

5.4.3 Modify the content of an existing FunctionBlockType

The task is described in the C# chapter “Modify the content of an existing FunctionBlockType”. The Visual Basic 6.0 solution is presented below.

```
' Get an XML description of the type from the Control Builder
Dim XMLStr As String
XMLStr = cb.GetFunctionBlockType("MyLib.MyFBType")
' Deserialize the XMLString into Objects
Dim fbType As FunctionBlockType
Set fbType = ObjectFactory.DeserializeFunctionBlockType(XMLStr)

Dim var As Variable
Set var = fbType.Variables.Find("X")
If Not var Is Nothing Then ' (var != null)
   var.InitialValue = "10"
End If

Dim nr As Long
nr = fbType.Variables.FindNr("Str")
If nr > 0 Then
   ' Remove the Variable
   Call fbType.Variables.Remove(nr)
End If
' Add a Parameter
Call fbType.Parameters.Add2("MyParam", "Dint", "retain", DirectionValue.cbIn, _
                           "7", "", "", "Description of MyParam")
' Add a FunctionBlock
Call fbType.FunctionBlocks.Add1("RTC1", "RTC")

' Serialize the objects into an XMLString and update the
' FunctionBlockType in the Control Builder
Dim bucket As String
bucket = cb.SetFunctionBlockType("MyLib.MyFBType", fbType.Serialize())
```

5.4.4 Add a new Hardware unit

The task is described in the C# chapter “Add a new Hardware Unit”. The Visual Basic 6.0 solution is presented below.

```
Dim bucket As String
' Create a "PM860 / TP830" unit
bucket = cb.NewHardwareUnit("MyController.0", "PM860 / TP830", "", "", "")

' Create some objects
Dim hw As HWUnit
Set hw = ObjectFactory.NewHWUnit("MyController.0.11.1")
Call hw.ParameterSettings.Add1("SignalRange_1", "0-20mA")
Call hw.HWChannels.Add2("IW0.11.1.1", "Output 1", "MyApp.Var1", "My Description", _
                        "10.0", "20.0", "m", "1", False)
Call hw.HWChannels.Add1("IW0.11.1.2", "Output 2", "MyApp.MyProgram.Var3", "")
' Serialize the objects to an XMLString and create an "AI810" unit,
' with content according to the XMLString, in the Control Builder.
bucket = cb.NewHardwareUnit(hw.Path, "AI810", "", hw.Serialize(), "")
```

5.5 A C++ example

Only one C++ example is presented. I am convinced that you are able to translate the C# examples to Visual C++ 6.0 with help of the clues below.

5.5.1 Getting started

You have to import the ”Control Builder Open Interface Helper” DLL before you can make use of the classes in the object model. You also have to import the “Control Builder Professional” EXE in order to be able to use the “CB Open Interface” methods.

This is achieved by entering the following statements in an appropriate h-file:

```
#import "C:\Program Files\ABB Industrial IT\Engineer IT\Control Builder M
Professional 5.0\Bin\ControlBuilderPro.exe" rename_namespace("CBProf")
using namespace CBProf;

#import"C:\WINDOWS\system32\CBOpenIFHelper.dll" rename_namespace("CBProfHelper")
using namespace CBProfHelper;

#ifdef _DEBUG
#include "debug\controlbuilderpro.tlh"
#include "debug\CBOpenIFHelper.tlh"
#else
#include "release\controlbuilderpro.tlh"
#include "release\CBOpenIFHelper.tlh"
#endif
```

You might have to adjust the path to the files according to your specific installation of the Control Builder.

The example below assumes the presence of two smart pointers:

```
ICBOpenIFPtr cb;
_ObjectFactoryPtr ObjectFactory;
```

The first one is called “cb” (short for Control Builder) and is a reference to an object of the “CBOpenIF” COM Class. The second one is called “ObjectFactory” and is a reference to an object of the “ObjectFactory”COM Class. It is also assumed that the corresponding objects are created in a suitable function. “InitInstance” is one suitable place to create these objects for a standard MFC EXE application.

The code might be as follows:

```
// Make an instance of the CB Open Interface (co)class
hr = cb.CreateInstance(__uuidof(CBProf::CBOpenIF));
if (hr != S_OK)
{
      // Add appropriate error handling here
      AfxMessageBox("Unable to connect to CB Open Interface");
}

// Make an instance of the CB Open Interface Helper ObjectFactory (co)class
hr = ObjectFactory.CreateInstance(__uuidof(CBProfHelper::ObjectFactory));
if (hr != S_OK)
```

```
{
      // Add appropriate error handling here
      AfxMessageBox("Unable to connect to Object Factory");
}
```

It is strongly recommended to qualify the class ID with the name space as in **CBProfHelper::ObjectFactory**

5.5.2 Modify the content of an existing FunctionBlockType

The task is described in the C# chapter “Modify the content of an existing FunctionBlockType”. The Visual C++ 6.0 solution is presented below.

```
BSTR tmpStr;
_bstr_t XMLStr;
_bstr_t Bucket;
HRESULT hr;

_FunctionBlockTypePtr fbType;
_VariablePtr var;

try
{
  // Get an XML description of the type from the Control Builder
  XMLStr = cb->GetFunctionBlockType("MyLib.MyFBType");
  // Deserialize the XMLString into Objects
  tmpStr = XMLStr.GetBSTR();
  fbType = ObjectFactory->DeserializeFunctionBlockType(&tmpStr);

  var = fbType->Variables->Find("X");
  if (var != NULL)
  {
    var->InitialValue = "10";
  }
  long nr;
  nr = fbType->Variables->FindNr("Str");
  if (nr > 0)
  {
     // Remove the Variable
    fbType->Variables->Remove(nr);
  }
   // Add a Parameter
  fbType->Parameters->Add2("MyParam", "Dint", "retain", cbIn, "7",
                           "", "", "Description of MyParam");
  // Add a FunctionBlock
  fbType->FunctionBlocks->Add1("RTC1", "RTC");

  // Serialize the objects into an XMLString and update the
  // FunctionBlockType in the Control Builder
  Bucket = cb->SetFunctionBlockType("MyLib.MyFBType", fbType->Serialize());

}
catch (_com_error ce)
{
  _bstr_t ErrMsg("An Error Occured: ");
  ErrMsg += ce.Description();
  AfxMessageBox(ErrMsg);
}
```

5.6 Major changes between SB2 and SB3

This chapter describes the major changes between SB2 (System Baseline 2) and SB3.

SingleControlModules are described by two different COM Classes due to changes in the “CB Open Interface”. This is described in a new chapter.

Two new classes “**ConnectedLibrary**” and “**ConnectedLibraries**” are introduced. The possibility to connect libraries is described in a new chapter.

Instances (Variables, ControlModules etc) have got two new read only properties named “TypeGuid” and “**TypePath**”. The “TypePath” holds the full path to the type, for instance “System.dint”. Note! The “Type” property doesn’t hold the full path in SB3. The “TypeGuid” value is valid only if a special CB Open Interface setting is set.

The property “**AspectObject**” has been removed because all objects are aspect objects in SB3. A new property “**AlarmOwner**” has been introduced.

The “Component” class has three new properties: “**ReadPermission**”, “**WritePermission**” and “**AuthenticationLevel**”.

The Variable, ExternalVariable, Parameter etc, classes have one new property named “**AuthenticationLevel**”.

The “ConnectedApplication” class has three new properties: “**MajorVersion**”, “**MinorVersion**” and “**Revision**”.

All Type classes (DataType, ControlModuleType, Program, FunctionBlockType, SingleControlModuleType) have the following new properties: “**SILLevel**”, “**SimulationMark**” and “**ResevedByFunction**”. The “ReservedByFunction” property is described in a new chapter.

The HWUnit class also has the new “**ResevedByFunction**” property, described above.

The Task class have the following new properties: “**LatencySupervision**”, “**LatencyPercentage**” and “**TaskSILLevel**”.

The ControlModuleType class and the SingleControlModuleType class have a new property named “**CMTypeGraphics**”. This property holds the old CMD Graphics for the type. A user of these classes should not alter the content of this property.

The ControlModule class and the SingleControlModuleInst class have a new property named “**CMInstGraphics**”. This property holds the old CMD Graphics for the instance. A user of these classes should not alter the content of this property.

The HWUnit class have the following new properties: “**RedundantPos**”, “**HWSimulation**”, “**HWSimulationSupported**” and “**ReservedByFunction**”. The “ReservedByFunction” property is described in a new chapter.

The SFCCodeBlock class has the following new property: “**SFCViewerAspect**”.

The SFCBranch class has the following new property: “**SFCPriority”.**

5.7 Changes between SV4 and SV5

This chapter describes the changes between SV4 and SV5.

The concept of Hardware Libraries has been introduced according to [Ref 6] DoF Hardware Libraries. A number of new CB Open Interface methods have been introduced according to [Ref 2] DoF CB Open Interface.

Furthermore, a new parameter named "**hwQualifier**" has been added to the existing "NewHardwareUnit" method according to [Ref 2]. That is, the method can be called with a hardware library qualifier as in the following example:

```
bucket = cb.NewHardwareUnit(hw.Path, "AI810", "S800IoModulebusHwLib 1.0-2",
                            hw.Serialize(), "");
```

But the method can also be called without a hardware library qualifier as in the following example:

```
bucket = cb.NewHardwareUnit(hw.Path, "AI810", "",
                            hw.Serialize(), "");
```

A new chapter describing how to connect hardware libraries to a controller has been added to the specification.

5.8 Changes between SV5.0 and SV5.1

A new singleton POU named ***Diagram*** has been introduced. Diagrams are described in a separate chapter.

Communication Variables have been introduced for Diagrams, Programs and top level SingleControlModules. This is described in a separate chapter.

InitValues (also called instance specific initial values) have been introduced for Diagrams, Programs and SingleControlModules. This is described in a separate chapter.

A new property named ***ExposePropertiesInParent*** has been added to function blocks and control modules. This property makes sense for function blocks or control modules which not are ***Aspect Objects***. If ***ExposePropertiesInParent*** is true then the corresponding PPA properties are exposed in the father object instead.

The ***CMParameter*** class has been extended with a new ***Direction*** property. The property can have the values "In", "Out", "In_Out" or "Unspecified".

The ***CMParameter*** class and the ***Parameter*** class have been extended with a new ***FDPort*** property.

The ***FDPort*** property can have the values are "yes" or "no" and controls whether the parameter should be visible as a port or not.

A new ***ISPValue*** property has been added to the ***Component*** class i.e. it is possible to configure ISP (Input Set Predefined) Values for structured data type components.

The ***PosInfo*** class have got two new properties named ***PageNo*** and ***Id*** to be used by function diagrams.

6 Future development

7 How to Use

The whole specification describes how to use.

REVISION

Rev. ind.	Page (P) Chapt.(C)	Description	Date Dept./Init.
-d0	All	Initial version	2002-02-08,ACL Anders Crilfe and Mats Segerstein
-	All	Updated according to review record. Approved version	2002-02-20, ACL Anders Crilfe
Ad0	All	All examples updated according to the release version of .NET Framework 1.0	2002-02-22, ACL Anders Crilfe
A		Approved version.	2002-0-25, ACL Anders Crilfe
-d1	All	New document number and new file name for the PLUTO (ATLAS v0.45) version. The new document is based on 3BSE027673 version A. The major modifications are: • A new chapter named "Major changes between SB2 and SB3" is included. • A new chapter named "SingleControlModules" is included • New chapters about "Connect Libraries" are included. • The new "ReservedByFunction" property is described in a new chapter. • The task chapters have been modified. • A new chapter "Safety perspective" is included • The references chapter have been spitted up into input documents and related documents • Many figures have been updated.	2003-06-30, ACL Anders Crilfe
-d2	C 5.6, C5.3.41	New attributes for HWUnits are included. New attribute for SFCCodeBlock is included.	2003-07-18, ACL Anders Crilfe
-d3		Updated before review. Small changes, mostly the numbering of figures.	2003-08-26, ACL Anders Crilfe
-	All	Updated according to review record. Approved version	2003-09-01, ACL Anders Crilfe

Rev. ind.	Page (P) Chapt.(C)	Description	Date Dept./Init.
Ad1		CR #20744 OLU - Hardware Libraries. Updated for SV5. Converted to TTT	2005-11-24, ACL Anders Crilfe
	C2.1	DoF Hardware Libraries is added as related document	
	C5.3.27, C5.3.41, C5.4.4	Method NewHardwareUnit has a new “hwQualifier” parameter Method NewHardwareUnit has a new “hwQualifier” parameter	
	C5.3.34, C5.3.35	Method NewHardwareUnit has a new “hwQualifier” parameter	
	C5.3.36	Connect applications without version number (Use Add1) Connect applications without version number (Use Add1)	
	C5.7	A new chapter describing how to Connect Hardware Libraries to a Controller is included.	
	C7	A new chapter describing Changes between SV4 and SV5 is included How to use chapter is included	
Ad2	C 2.2, C 4.1	Corrections according to review remarks.	2005-12-07 XA/ACL Joakim Welin
A		Ready for approval	2005-12-07 ATPA/XA/ACP Anders Nessmar
Bd1	C5.5.1	CR #23392. Path to import directive and includes are changed Minor changes of “hwQualifier”	2006-03-18 ATPA/XA/ACL Anders Crilfe
B		Ready for approval	2006-03-20 PA/XA/ACP Anders Nessmar

Rev. ind.	Page (P) Chapt.(C)	Description	Date Dept./Init.
Cd1		Updated for SV5.1 according to CR: AC 800M #37868	2009-08-31
	C5.3.43 C5.3.44 C5.3.45 C5.3.46 C5.3.47 C5.3.48	New chapters describing Diagrams, Communication variables and Init Values.	XA/ACL Anders Crilfe
	C5.8	New chapter describing Changes between SV5.0 and SV5.1	
	C2.2 C4.6 C5.2.4 C5.3.2 C5.3.3 C5.3.6 C5.3.23	Minor changes in the following chapters: (C2.2) Input document updated (C4.6) General constraint updated (C5.2.4) Polymorphism (figure 5 and 6 updated) (C5.3.2) ISP Value for component in a DataType (C5.3.3) Figure 10 updated (C5.3.6) Extended with FDPort property (C5.3.23) Figure 35 updated	
C	No changes	The document is ready for review	2009-09-17 PA/XAACO Harriet Lindgren Larsson
Dd1	C5.3.49	Updated for SV5.1 according to CR: AC 800M #40458 New chapter describing support of Execution order	2010-02-11 XAACS Anders Crilfe
D	No changes	The document is ready for review	2010-02-16 PAOC/XAACO Harriet Lindgren Larsson
Ed1		Updated for SV5.1 FPHI according to CR: AC 800M #45536	2011-12-15
	C5.3.50 C5.3.51 C5.3.52 C5.3.53 C5.3.47.1	New chapters describing support for Diagram Types and Diagram Instances New HI Properties for Communication Variables	XAACS Anders Crilfe
Ed2	C5.3.47.1	Updated after formal review according to review record. Figure 58 updated	2012-02-17 XAACS Anders Crilfe
E	-	Ready for approval	2012-02-17 XAACS Johan Gren