PVIServices
TM712

Perfection in Automation
www.br-automation.com

Requirements

Training modules:	TM211 – Automation Studio Online Communication TM710 – PVI Communication
Software:	Visual Studio.NET 2003 / 2005 Windows 2000 / XP
Hardware:	Any SG3 or SG4 controller

Table of contents

1. INTRODUCTION

This training module describes PVI access in the Visual Studio .NET programming environment. All communication and diagnostics services, from the PVICOM interface to the B&R controller can be used.

The PVI functions are grouped together for the user in easy-to-use classes in the PVIServices Namespace.

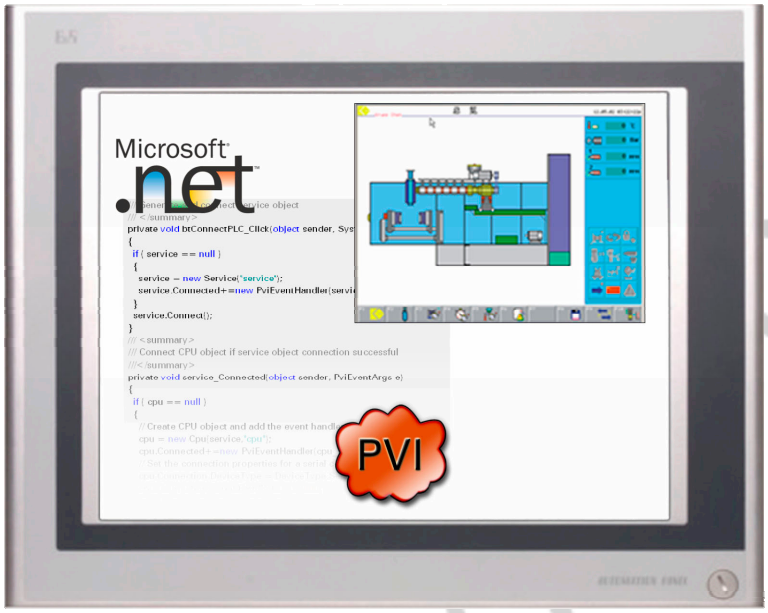


Fig. 1: PVIServices

1.1 Objective

Participants will be able to create their own PVI Client application in Visual Studio .NET using the practice examples.

The PVIServices User Documentation can be referred to for additional information regarding application possibilities in the existing PVIServices classes for the PVI Client application.

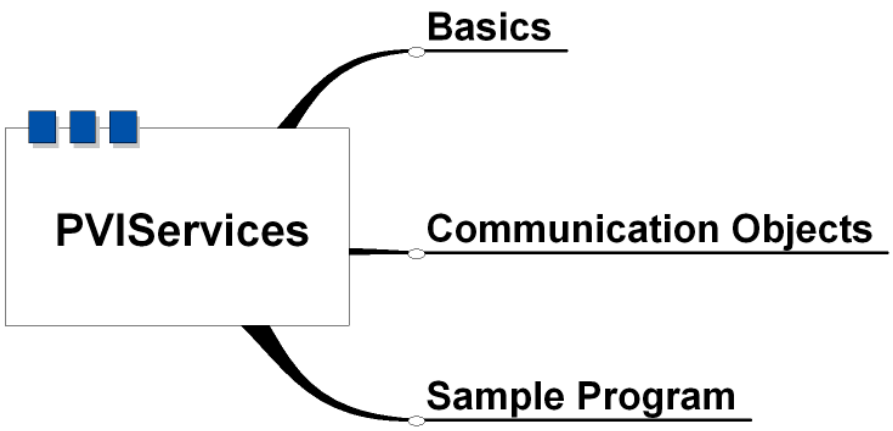


Fig. 2: Overview

2. PVISERVICES BASICS

PVIServices are used by all Visual Studio.NET based Windows applications for communication and diagnostics services on B&R controllers.

They are based on the PVICOM interface and are represented in the programming environment by an object-oriented structure.

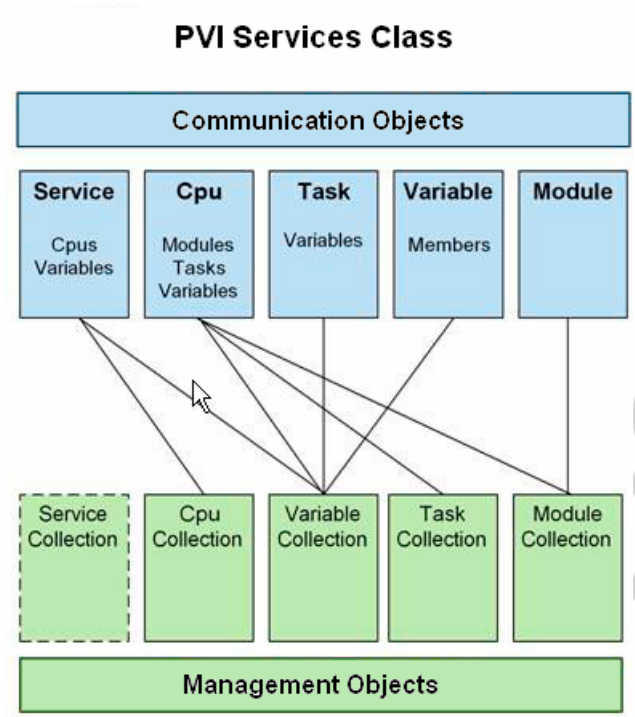


Fig. 3: PVIServices classes

Note:

The PviServices components can be used in Windows XP/2000 as well as Windows CE applications.

2.1 PVIServices installation files

The PVIServices DLLs are installed in the following directory after installing the PVI Server&Runtime / Development package:

PVI Server&Runtime:

BrAutomation/Pvi/PviServices/Win32 *for Windows XP/2000*
BrAutomation/Pvi/PviServices/WinCE *for Windows CE*

PVI Development (starting with PVI 2.5.2.3060):

Program Files\BrAutomation\PVI\%Version%\PVI\PviServices\Win32
Program Files\BrAutomation\PVI\%Version%\PVI\PviServices\WinCE

The component "**BR.AN.PVIServices.dll**" is added to the Visual Studio.NET project as a reference.

Only one PVI Runtime installation is necessary for the project's runtime because the PVIServices component is provided with the installation of the application.

2.1.1 Requirements for PVIServices programming

In this training module, the AR000 will be used for communicating with a controller. The variables used to do this and the corresponding data type are documented in this training module.

However, any existing Automation Studio project can be used – only the variables names have to be accordingly replaced.

A task "**pvitest**" with the variables "**Lifesign**" and "**PV1**" is required.

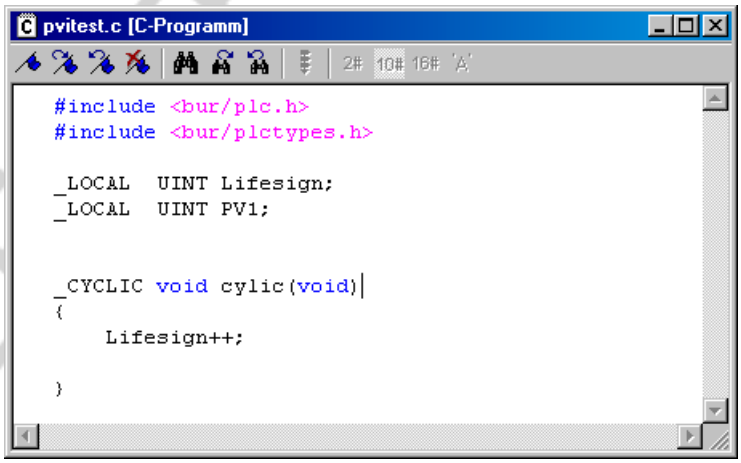


Fig. 4: Automation Studio test project

2.1.2 Visual Studio.NET programming environment

All of the functions described in this training module are explained using C# program code.

Any differences with the programming in VB.NET will be explicitly indicated and described.

Numerous PVIServices examples created in the programming language C# are also available for the user.

These examples can be found in the directory "..\Samples\PviServices" when installing the PVIServices.

2.1.3 Adding the "BR.AN.PviServices.dll"

The reference to the corresponding DLL must be added to access the PVIServices.

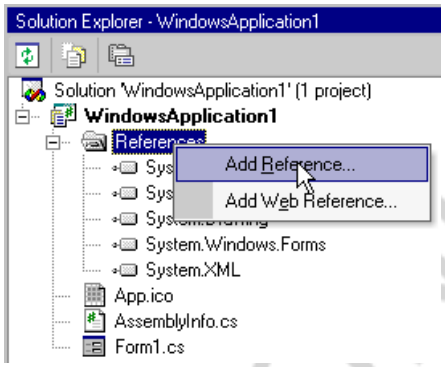


Fig. 5: Adding the BR.AN.PviServices.dll reference

Pressing the <**Browse**> button in the "**Add Reference**" dialog box allows you to select the reference from the "**BrAutomation\Pvi\PviServices\Win32**" directory (PVI Server&Runtime Installation) or "**Program Files\BrAutomation\PVI\%*Version*%\PVI\PviServices\Win32**" (PVI Development).

Caution:

If the component is displayed in the component list, make sure that the PviControl.NET also uses this DLL. There is no guarantee that the versions are identical.
This is why the directory described above should always be used.

3. PVISERVICES COMMUNICATION OBJECTS

A communication object represents an object located on the controller such as a task object or variable object.

PVIServices recognizes the following communication objects or classes:

- Service class
- CPU class
- Task class
- Variable class
- Module class

3.1 Properties, events and methods

Each of these communication objects is made up of the following:

- Properties
- Methods
- Events

This enables consistent and complete application of the objects.

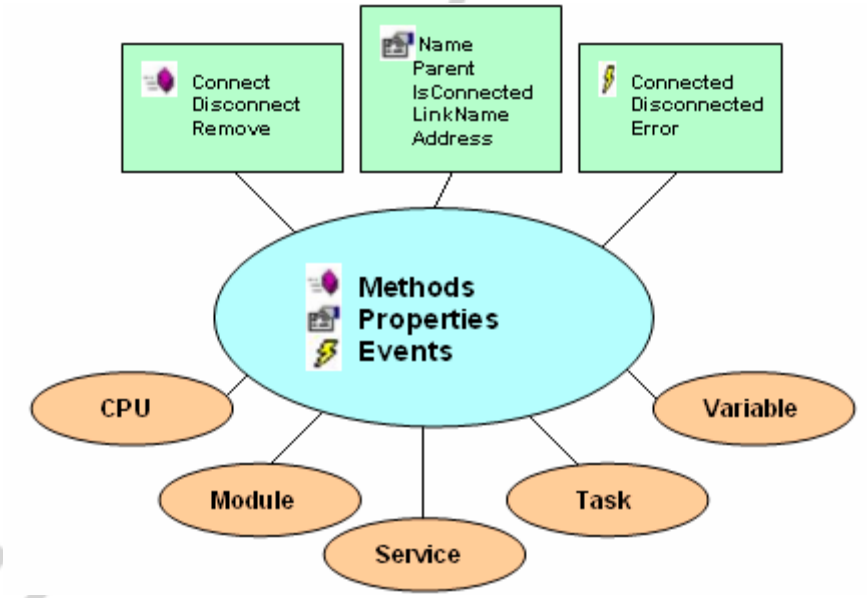


Fig. 6: PVIServices communication objects

Note:

All tasks are processed **asynchronously**. Responses from a task are made via events. For example, the **"Read" method** from a variable object sends response data in the **"ValueRead" event**.

Methods, properties and events from all communication classes are documented in the PVIServices help files.

3.1.1 Properties:

Each object is written with its specific properties.

```
private void myService_Connected(object sender, PviEventArgs e)
{
    myCpu = new Cpu(myService,"cpu");
    myCpu.Connection.DeviceType = DeviceType.TcpIp;
    myCpu.Connection.TcpIp.DestinationStation = 2;
    myCpu.Connection.TcpIp.DestinationPort = 11160;
    myCpu.Connection.TcpIp.DestinationIpAddress = "127.0.0.1";

}
```

After creating a new object (in this example, a "Cpu" type) the properties are defined according to the requirements.

```
myCpu = new Cpu(myService,"cpu");
myCpu.Connection.DeviceType = DeviceType.
```

3.1.2 Methods

The desired method or function of an object must be used to execute an action on a specific object.

```
private void myService_Connected(object sender, PviEventArgs e)
{
    myCpu = new Cpu(myService,"cpu");
    myCpu.Connection.DeviceType = DeviceType.TcpIp;
    myCpu.Connection.TcpIp.DestinationStation = 2;
    myCpu.Connection.TcpIp.DestinationPort = 11160;
    myCpu.Connection.TcpIp.DestinationIpAddress = "127.0.0.1";

    myCpu.Connect();
}
```

This is how a connection to the controller is established on the CPU object using the "**Connect**" method.

The "**Disconnect**" method is used to terminate the connection to the controller.

3.1.3 Events

The application is informed of the object's state by setting up events (**EventHandlers**).

There are a few differences between Visual Basic.NET and C# in how this is done.

In C#, the event handler is set up by specifying the desired event, writing += and pressing the <**TAB**> key two times.

```
myCpu.Connected+=
myCpu.Connect();  new PviEventHandler(myCpu_Connected); (Press TAB to insert)
```

The event handler is automatically set up and the corresponding program code is added.

```
        myCpu.Connected+=new PviEventHandler(myCpu_Connected);
        myCpu.Connect();

        }

    private void myCpu_Connected(object sender, PviEventArgs e)
    {

    }
```

Note:

The "**Connected**" event should be added for each object. The objects can only be further accessed once this has been done.

Caution:

An event handler should always be set up BEFORE calling the method. Otherwise, events might not be received when sent in the method call.

In Visual Basic.NET, the **event handler** is set up by specifying **WithEvents**.

```
Friend WithEvents myCpu As BR.AN.PviServices.Cpu
```

From this point, the **event** can now be selected from the specified class "**myCpu**" and the program code that is needed for the event can be entered.

3.2 PVIServices classes and the PVI object hierarchy

As described in the earlier training modules TM700 and TM710, all objects are mapped in a PVI application using the PVI object hierarchy.

In the PVIServices classes the line object, the device object, the station object and the CPU object are mapped in the CPU class.

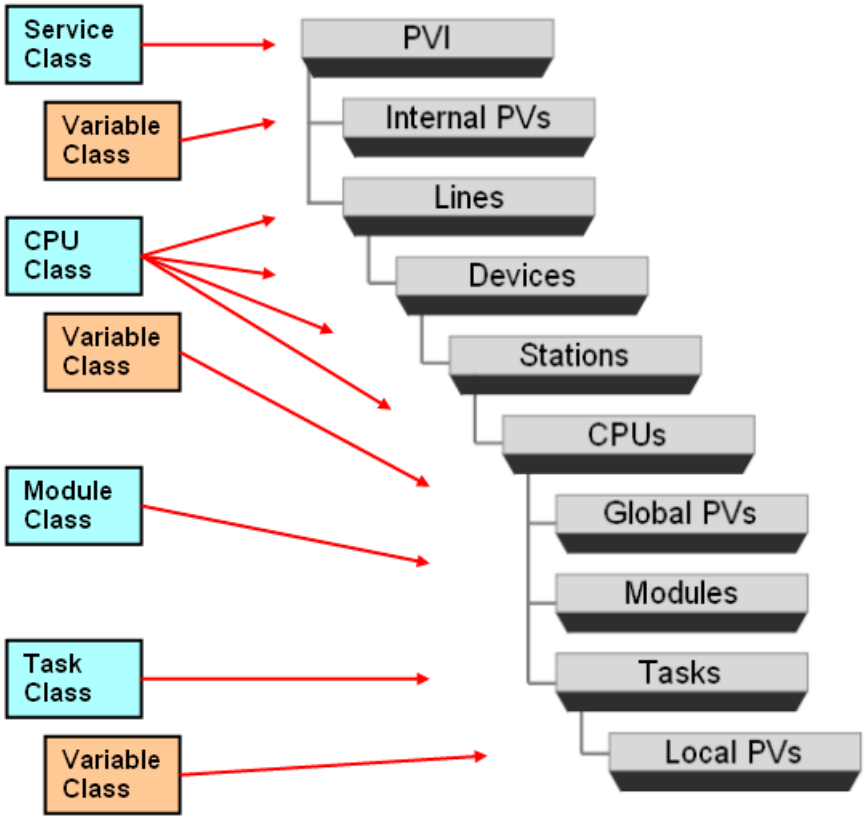


Fig. 7: PVIServices – PVI object hierarchy

Note:

PVIServices supports only the INA2000 line.

3.3 PVIServices – Variable declaration

Variables from the corresponding class must be created in order to create PVIService objects.

The classes and members can be accessed directly by declaring the Namespace "**BR.AN.PviServices**" in C#.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

using BR.AN.PviServices;
```

From this point on, PVIServices objects from the respective class can be set up.

```
public Service myService;
public Cpu myCpu;
public Task myTask;
public Variable myVariable;
```

The declaration looks like this in Visual Basic.NET:

```
Public myService As BR.AN.PviServices.Service
Public myTask As BR.AN.PviServices.Task

Friend WithEvents myCpu As BR.AN.PviServices.Cpu
Friend WithEvents myVariable As BR.AN.PviServices.Variable
```

When making the declaration, you must determine whether events in this object should be evaluated or not.

3.4 PviServices – Service class

A connection to the PVI Manager is established via the service class object. This object is the basis for all subsequent objects (CPU, tasks, variables) in a client application.

```
private void Form1_Load(object sender, System.EventArgs e)
{
    myService = new Service("service");
    myService.Connected+=new PviEventHandler(myService_Connected);
    myService.Connect();


}
```

A new service object "**myService**" is set up with the **logical name** "**service**".

A local connection to the PVI Manager is created if an argument is not transferred in the "**Connect**" method.

A remote connection can be established by specifying the **IP address** and **port number** of the remote PC.

```
myService.Connect("172.43.36.120",20000)
```

2 of 3 ▾ void Service.Connect (string server, **int port**)

3.5 PviServices – CPU class

The connection to the controller is established via the CPU class object. Furthermore, all global variables on the CPU object and all task objects are managed with the local or global variables.

Diagnostics services (memory, transferring modules, etc) are also managed by this class.

```
private void myService_Connected(object sender, PviEventArgs e)
{
    myCpu = new Cpu(myService,"cpu");
    myCpu.Connection.DeviceType = DeviceType.TcpIp;
    myCpu.Connection.TcpIp.DestinationStation = 2;
    myCpu.Connection.TcpIp.DestinationPort = 11160;
    myCpu.Connection.TcpIp.DestinationIpAddress = "127.0.0.1";
    myCpu.Connected+=new PviEventHandler(myCpu_Connected);
    myCpu.Connect();

    }
```

The reference to the higher-level service object is made by specifying the **parent name** ("**myService**" in this example).

The type of connection (i.e. the medium) is defined via the "**Connection.DeviceType**" property.

The properties should be defined in accordance to the selected "**DeviceType**".

3.6 PviServices – Task class

The PVIServices task class object represents a task on the controller. Global and local variables from the task are managed under this object.

```
private void myCpu_Connected(object sender, PviEventArgs e)
{
    myTask = new Task(myCpu,"pvitest");
    myTask.Connected+=new PviEventHandler(myTask_Connected);
    myTask.Connect();
}
```

The assignment to the respective **CPU object** is made by specifying the **parent name** of the higher-level object.

3.7 PviServices – Variable class

The PVIServices variable class object represents a variable on the controller.

The following variable types are available:

- Internal variables
- Global variables
- Local variables

A variable is assigned by specifying the **parent name** of the higher-level object.

```
private void myService_Connected(object sender, PviEventArgs e)
{
    myVariable = new Variable(myService,"IntVar1");
```

An internal variable is created if the service object is entered as parent name.

```
private void myCpu_Connected(object sender, PviEventArgs e)
{
    myTask = new Task(myCpu,"pvitest");
    myTask.Connected+=new PviEventHandler(myTask_Connected);
    myTask.Connect();
    myVariable = new Variable(myCpu,"Lifesign");
}
```

A global variable is created in this example because the Cpu object was specified as parent name.

3.7.1 Active / passive switching of variable objects

A variable object is switched to active /passive via the "**Active**" property.

```
private void myTask_Connected(object sender, PviEventArgs e)
{
    myVariable = new Variable(myTask,"Lifesign");
    myVariable.Address = "Lifesign";
    myVariable.Active = true;
    myVariable.RefreshTime = 200;
    myVariable.ValueChanged+=new VariableEventHandler(myVariable_ValueChanged);
    myVariable.Error+=new PviEventHandler(myVariable_Error);
    myVariable.Connect();
}
```

3.7.2 Evaluating data changes

The application is informed of data changes by setting up the "**ValueChanged**" event handler.

```
private void myVariable_ValueChanged(object sender, VariableEventArgs e)
{
    this.lblValLifesign.Text  = ((Variable)sender).Value.ToString();
}
```

The "**ValueChanged**" event is called until either a "**Disconnect**" takes place on the variable object or the "**Active**" property is set to **=false**.

3.7.3 Read and write access

Targeted, asynchronous access to a variable object is achieved using the "**ReadValue**" method.

```
private void cmdReadVar_Click(object sender, System.EventArgs e)
{
    myVariable.ReadValue();
    myVariable.ValueRead+=new PviEventHandler(myVariable_ValueRead);
}

private void myVariable_ValueRead(object sender, PviEventArgs e)
{
    this.lblReadVar.Text = ((Variable)sender).Value.ToString();
}
```

Another possibility is (with active variables) to directly access the "**Value**" property.

```
this.lblReadVar.Text = myVariable.Value.ToString();
```

Caution:

Make sure that the last value received for the PVIServices variable object is read during read access to the value property of the variable object – regardless whether the object is active or not (active property).

The assigned value is written to the variable by **defining** the "**Value**" property.

```
private void cmdWriteVal_Click(object sender, System.EventArgs e)
{
    myVariable.Value = 123;
}
```

A response is also possible if needed because write access is asynchronous. This is done by setting up the event handler for the "**ValueWritten**" event.

Note:

Operators cannot be assigned in Visual Basic.NET (7.0). However, there are two alternatives:

```
// Write value in VB.NET #1
myVal = new Value(123);
myVariable.Value = myVal;

// Write value in VB.NET #2
myVariable.Value.Assign(123);
myVariable.WriteValue();
```

3.8 PviServices – Module class

The PVIServices module class object defines a BR module located on the controller with its properties.

The following actions are possible on the module object:

- Module upload
- Module download
- Delete module

```
private void cmdDeleteModule_Click(object sender, System.EventArgs e)
{
    myModule = new Module(myCpu,"pvitest");
    myModule.Connected+=new PviEventHandler(myModule_Connected);
    myModule.Connect();
}

private void myModule_Connected(object sender, PviEventArgs e)
{
    myModule.Deleted+=new PviEventHandler(myModule_Deleted);
    myModule.Delete();
}

private void myModule_Deleted(object sender, PviEventArgs e)
{
    myModule.Deleted-=new PviEventHandler(myModule_Deleted);
    this.lblStatus.Text = myModule.Name + " deleted";
}
```

As seen in this example, an object is created from the PVIServices module class. The CPU object is used as parent name.

The module object is set up using the "**Connect**" method. The application is informed of successful setup in the "**Connected**" event.

The module on the controller is deleted in this event using the "**Delete**" method and the application is notified of a successful delete procedure in the "**Deleted**" event.

Note:

It is also necessary to evaluate the "**Error**" event because errors can occur when setting up (e.g. due to an incorrect module name). As a result, the "**Connected**" event is not called.

3.9 Collections

Collections allow the user to manage multiple objects with the same type together.

For example, if variables are grouped into a collection, then this collection has similar properties, methods and events as the base variable class.

This means that each variable does not have to be managed individually. Instead all of the variables in the collection are managed automatically when calling a method (e.g. the "**Connect**" method) for the collection.

Management of variables for multiple screens can be seen as an example. The variables for one screen can be managed in a collection in order to switch these variables to active / passive at the same time.

The directory "**BrAutomation\Samples\PviServices**" contains a few examples of using collections.

More detailed information can also be found in the PVIServices User Documentation.

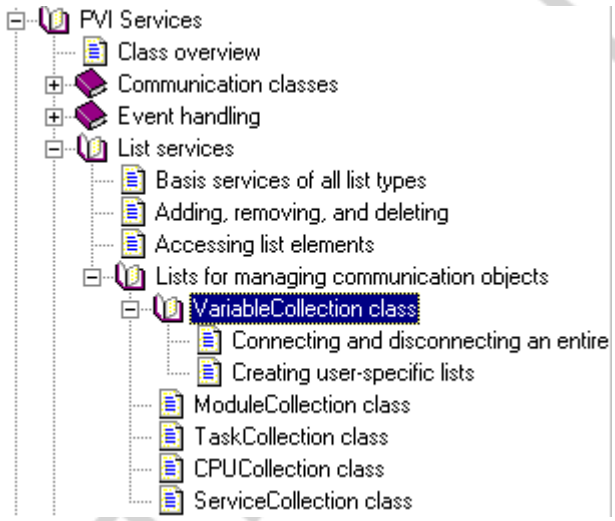


Fig. 8: PVIServices - Collections

3.10 Working with structures

If a variable object with the type "Structure" has been created, then the structure member can be accessed after receiving the "**Connected**" event.

At "**Connect**", PVIServices reads all of the information from the structure and manages it in the **hash table**.

This makes it possible to address a structure element directly using the element name or the index of the array. An example with structures is explained in the PVIServices sample program (item 4).

Detailed information is available in the PVIServices User Documentation.

A few examples of structures are provided in the directory "**BrAutomation\Samples\PviServices**". A control program that contains a structure variable is required.

4. PVISERVICES SAMPLE PROGRAM

In this section a PVIServices application will be created instead of performing exercises and tasks.

What will be accomplished in this example:

- Creating a Visual Studio.NET application
- Creating the PVIServices objects
- Error evaluation
- Evaluating data changes
- Read and write access
- Using a structure

4.1 Creating a new Visual Studio.NET application

We will now create a new Visual Studio.NET application. This example will be explained using a C# application. However, it should also be possible for the user to create the application using Visual Basic.NET with the information from the previous section.

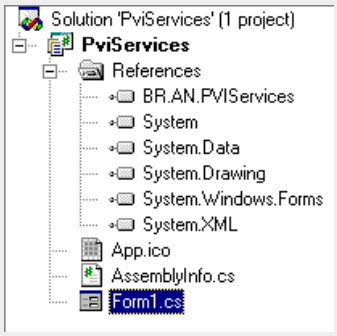
Task: Creating the Visual Studio.NET application and inserting the PVIServices reference.

The steps for this task should be followed with the help of the instructions in chapter 2.

The reference to the PVIServices should be defined in the namespace.

Result:

From this point on, the classes and members of PVIServices can be accessed.



4.2 Creating the PVIServices communication objects

In the next step, the PVIServices variables required for the program example must be declared in the form class.

```
public class Form1 : System.Windows.Forms.Form
{
    /// <summary>
    /// Required designer variable.
    ///

    public Service myService;
    public Cpu myCpu;
    public Task myTask;
    public Variable myVariable;
    public Module myModule;
```

Example: Creating the service object in the form load event.

The "**Connected**" event is set up so that the CPU object can be set up after "**Connect**" has been successfully executed.

We will be using static objects in this example.

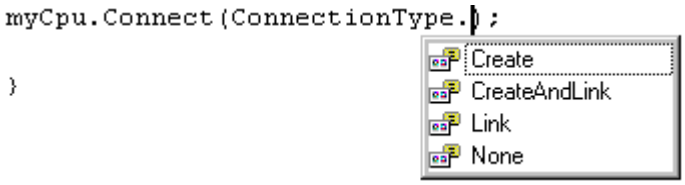
```
private void Form1_Load(object sender, System.EventArgs e)
{
    myService = new Service("service");
    myService.Connected+=new PviEventHandler(myService_Connected);
    myService.IsStatic = true;
    myService.Connect();


}
```

The "**IsStatic**" property is used to determine whether the subsequent objects were set up **static =true** or **temporary =false** (default).

"**Create**" and "**Link**" are automatically performed when **connecting** a new PVIServices object **without arguments** (i.e. the process object is created and a link object is placed at the same time).

The following definitions are possible for static process objects with the "**ConnectionType**" argument:



ConnectionType	Description
Create	Sets up the process object.
CreateAndLink	Sets up the process object and the link object.
Link	Sets up a link object on an existing process object.

An example of the separate "**Create**" and "**Link**" is described when setting up the variable objects. All other objects are set up without arguments.

Example: Creating the CPU object and the task object

A TCPIP connection is made to the AR000 for the CPU object.

The name of the user task "**pvitest**" is used as task name.

Each subsequent object is created in the "**Connected**" event of the preceding object. This ensures that the object has been successfully created.

The "**Error**" event is additionally evaluated for CPU and task objects in order to evaluate communication or configuration errors.

The error messages and "**Connected**" events are output to a MultiLine text box control with the name "**txtStatus**".

```
private void myService_Connected(object sender, PviEventArgs e)
{
    this.txtStatus.Text+= "Service Connected\r\n";
    myCpu = new Cpu(myService,"cpu");
    myCpu.Connection.DeviceType = DeviceType.TcpIp;
    myCpu.Connection.TcpIp.DestinationStation = 2;
    myCpu.Connection.TcpIp.DestinationPort = 11160;
    myCpu.Connection.TcpIp.DestinationIpAddress = "127.0.0.1";
    myCpu.Connected+=new PviEventHandler(myCpu_Connected);
    myCpu.Error+=new PviEventHandler(myCpu_Error);
    myCpu.Connect();
}

private void myCpu_Connected(object sender, PviEventArgs e)
{
    this.txtStatus.Text+= "CPU Connected\r\n";
    myTask = new Task(myCpu,"pvitest");
    myTask.Connected+=new PviEventHandler(myTask_Connected);
    myTask.Error+=new PviEventHandler(myTask_Error);
    myTask.Connect();

}

private void myTask_Connected(object sender, PviEventArgs e)
{
    this.txtStatus.Text+= "Task " + e.Name + " Connected\r\n";
```

The connected events for the respective objects are shown in the textbox after starting the application.

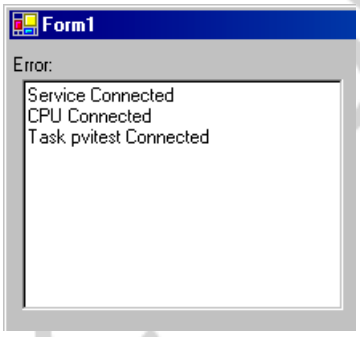


Fig. 9: Set up objects

4.3 Error evaluation

If an error occurs executing the program, the corresponding error code and the object are shown in the "**txtStatus**" textbox.

```
private void myCpu_Error(object sender, PviEventArgs e)
{
    this.txtStatus.Text+= e.Name + " Error:" + e.ErrorCode + "\r\n";
}

private void myTask_Error(object sender, PviEventArgs e)
{
    this.txtStatus.Text+= e.Name + " Error:" + e.ErrorCode + "\r\n";
}
```

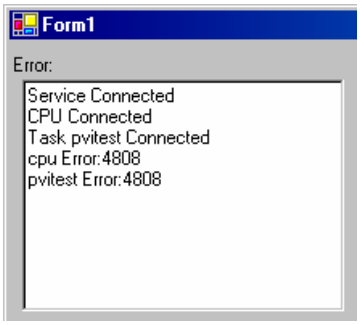


Fig. 10: Error event – Connection lost

The "**Connected**" event is called again once the connection has been reestablished.

However, this would also mean that the objects are newly set up again because they are created in the "**Connected**" event of the preceding object.

To prevent this from happening, a query must occur when creating the objects to determine whether or not the object already exists.´

```
private void myService_Connected(object sender, PviEventArgs e)
{
    this.txtStatus.Text+= "Service Connected\r\n";

    if (myCpu == null)
    {
        myCpu = new Cpu(myService,"cpu");
        myCpu.Connection.DeviceType = DeviceType.TcpIp;
        myCpu.Connection.TcpIp.DestinationStation = 2;
        myCpu.Connection.TcpIp.DestinationPort = 11160;
        myCpu.Connection.TcpIp.DestinationIpAddress = "127.0.0.1";
        myCpu.Connected+=new PviEventHandler(myCpu_Connected);
        myCpu.Error+=new PviEventHandler(myCpu_Error);
        myCpu.Connect();
    }
}
```

The "**Error**" event is called if the connection is lost while communicating with the controller (E=**4808**).

This event is created for all **active** objects. That means for this variable and for all higher-level objects (task object and CPU object) when a variable object's "**Active**" property is set to "**true**".

The "**Connected**" event for these objects is called automatically once the connection has been reestablished. At that point, the respective objects can be accessed again.

4.4 Evaluating data changes

The two variables "**Lifesign**" and "**PV1**" are registered in the next step.

Example: Setting up the variables "LifeSign" and "PV1"

The variable "**PV1**" is only set up with the "**ConnectionType.Create**" for the time being.

The "**ValueChanged**" and "**Error**" event should be used.

A new variable from the PVIService class "**Variable**" is required for the "**PV1**" variable.

```
public Variable myVariablePV1;


private void myTask_Connected(object sender, PviEventArgs e)
{
    this.txtStatus.Text+= "Task " + e.Name + " Connected\r\n";

    if (myVariable == null)
    {
        myVariable = new Variable(myTask,"Lifesign");
        myVariable.Active = true;
        myVariable.RefreshTime = 200;
        myVariable.ValueChanged+=new VariableEventHandler
            (myVariable_ValueChanged);
        myVariable.Error+=new PviEventHandler(myVariable_Error);
        myVariable.Connect();
    }
    if (myVariablePV1 == null)
    {
        myVariablePV1 = new Variable (myTask,"VarCreateOnly");
        myVariablePV1.Address = "PV1";
        myVariablePV1.Connect(ConnectionType.Create);
    }
}
```

The **name of the variable with error** is returned in the PviEvent argument "**e.Name**". The **error number** is returned in the PviEvent "**e.ErrorCode**".

```
private void myVariable_Error(object sender, PviEventArgs e)
{
    txtStatus.Text += e.Name + " E#" + e.ErrorCode.ToString();
}
```

In the "ValueChanged" event, the PviEvent argument "**e.Name**" also determines which variables have changed.

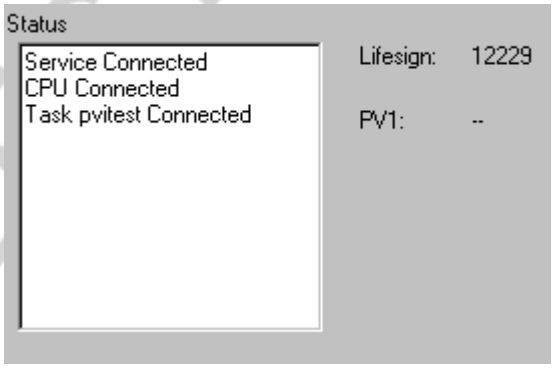
```
private void myVariable_ValueChanged(object sender, VariableEventArgs e)
{
    if (e.Name == "Lifesign")
    {
        lblValLifesign.Text  = ((Variable) sender).Value.ToString();
    }
    if (e.Name == "linkVarPV1")
    {
        lblPV1.Text = ((Variable) sender).Value.ToString();
    }
}
```

Note:

The variable "**linkVarPV1**" will be created in the next exercise when creating the link object.

The value of the "**Lifesign**" variable is displayed on the label control after starting the program.

There is still no value change for the variable "**PV1**" because it has only been created and not yet connected.



Example: Creating a connection object for the variable "PV1"

A link object is created using the button, "**cmdConnectPV1**". A new link object "**myLinkPV1**" is created with the connection type "**Link**".

```
private void cmdConnectPV1_Click(object sender, System.EventArgs e)
{
    Variable myLinkPV1 = new Variable(myVariable ,"linkVarPV1");
    myLinkPV1.LinkName = myVariablePV1.FullName;
    myLinkPV1.Active = true;
    myLinkPV1.ValueChanged+=new VariableEventHandler(myVariable_ValueChanged);
    myLinkPV1.Error+=new PviEventHandler(myVariable_Error);
    myLinkPV1.Connect(ConnectionType.Link);
}
```

In order to create a link to an already existing object, the **LinkName** property of the link variable object must be set to the variable object to be referenced (**FullName**) as a value of the hierarchical name.

The **Link** connection type must also be given when calling the Connect method so that the **LinkName** property is enabled.

The same event as for the "**Lifesign**" variable is applied to the "**ValueChanged**" and "**Error**" event handler.

Result:

A link object is created on the "**PV1**" link object by pressing the button "**cmdConnectPV1**". The "**Active**" property is used to determine that each value change in the "**ValueChanged**" event should be registered.

The "**Disconnect**" method must be called to delete the link object again.

4.5 Read and write access

Variable objects that are activated via the "**Active = true**" property (event mask EV=ed) are automatically monitored by the PVI Manager for data changes – regardless of whether these variables are polled by the PVI line (default) or monitored as event variable by the controller via the property "**Polling = false**" (attribute AT=re).

That means that these variables no longer have to be additionally read by the application. The application is automatically informed of any data changes in the "**ValueChanged**" event.

This training module will use different examples to cover three access methods for read and write access.

4.5.1 Reading variables

A variable object is read using the "**ReadValue**" method. The value is returned in the "**ValueRead**" event because PVIServices processes all tasks asynchronously.

Example: Reading the value of the "Lifesign" variable

The read process for the "**Lifesign**" variable is initiated by pressing the "**cmdReadVar**" button. The value is output to a label control "**lblReadVar**" by setting up the the event handler for the "**ValueRead**" event.

```
private void cmdReadVar_Click(object sender, System.EventArgs e)
{
    myVariable.ValueRead+=new PviEventHandler(myVariable_ValueRead);
    myVariable.ReadValue();
}

private void myVariable_ValueRead(object sender, PviEventArgs e)
{
    this.lblReadVar.Text = ((Variable)sender).Value.ToString();
}
```

Result:

Pressing the "**cmdReadVar**" button causes the current value of the "**Lifesign**" variable to be read and output to the label control.

4.5.2 Read the time from the controller

The date / time is read in the PVIServices CPU class.

The "**ReadDateTime**" method is called on the CPU object to read the time. The application is informed about the event by setting up an event handler "**DateTimeRead**".

Example: Reading the date and time on the controller

The read procedure for the time is initiated by pressing the new button "**cmdReadTime**". The read time is output to the form text property.

```
private void cmdReadTime_Click(object sender, System.EventArgs e)
{
    myCpu.DateTimeRead+=new CpuEventHandler(myCpu_DateTimeRead);
    myCpu.ReadDateTime();
}

private void myCpu_DateTimeRead(object sender, CpuEventArgs e)
{
    DateTime dt;
    dt = e.DateTime;
    this.Text = dt.ToString();
}
```

4.5.3 Writing a value

To define a variable, the "**Value**" property must be defined with the desired value.

Example: Resetting the "Lifesign" variable

The "**Lifesign**" variable is set with the value "**0**" by pressing the new "**cmdWriteVal**" button.

The write task is confirmed via the event "**ValueWritten**".

```
private void cmdWriteVal_Click(object sender, System.EventArgs e)
{
    myVariable.Value = 0;
    myVariable.ValueWritten+=new PviEventHandler(myVariable_ValueWritten);
}

private void myVariable_ValueWritten(object sender, PviEventArgs e)
{

}
```

A write task is performed immediately after defining this property.

For example, automatic definition can be suppressed by setting the property "**WriteValueAutomatic = false**" in order to write the member of an array or structure.

The array or structure is only written once when the "**WriteValue**" method is called.

4.6 Using a structure

The Automation Studio project can be expanded for the exercise examples with a structure. A structure called "**Pv_Struct**" is created with three members (elements).

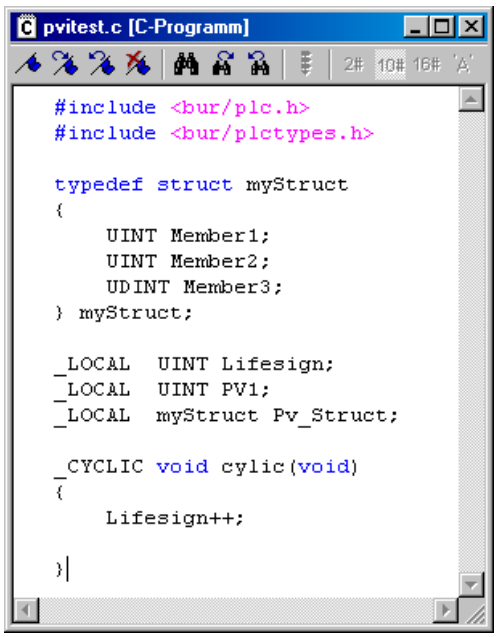


Fig. 11: AS test project with a structure

Example: Defining the variable and creating the structure variable

Create a structure variable for the PVIServices class variable. The structure variable "**myStructPV**" is created in the task's "**Connected**" event.

The new BR.AN.PviServices variable is defined in the form class.

```
public Service myService;
public Cpu myCpu;
public Task myTask;
public Variable myVariable;
public Variable myVariablePV1;
public Variable myStructPV;
```

The variable is then set up. The same event handlers are used for the "**ValueChanged**" and "**Error**" event as with other variables.

```
if (myStructPV == null)
{
    myStructPV = new Variable(myTask,"Pv_Struct");
    myStructPV.Active = true;
    myStructPV.RefreshTime = 1000;
    myStructPV.ValueChanged+=new VariableEventHandler
        (myVariable_ValueChanged);
    myStructPV.Error+=new PviEventHandler(myVariable_Error);
    myStructPV.Connect();
}
```

Example: Evaluating the value change in the structure

Every value change in the structure is registered in the "**ValueChanged**" event. The value of each member in the structure is output in the text box "**txtStatus**".

In the "**ValueChanged**" event, the variable is checked to see if it is a **"Structure" data type** and the value of each member is read.

```
private void myVariable_ValueChanged(object sender, VariableEventArgs e)
{
    Variable tmpVar = (Variable)sender;

    if (e.Name == "Pv_Struct")
    {
        if (tmpVar.Value.DataType == DataType.Structure)
        {
            foreach (Variable member in tmpVar.Members.Values)
            {
                txtStatus.Text += member.Value.ToString() + "\r\n";
            }
        }
    }
```

"**e.ChangedMember**" is used to notify the application of the member name on which a value change occurred if data changes only in the affected members should be output.

```
foreach (String  membername in e.ChangedMembers)
{
    if (membername != null)
    {
        txtStatus.Text += tmpVar.Value[membername].ToString() + "\r\n";
    }
}
```

Example: Defining a structure element

The structure variable "**Pv_Struct.Member3**" is defined with the value of the "**Lifesign**" variable using a new button, "**cmdSetStruct**".

```
private void cmdSetStruct_Click(object sender, System.EventArgs e)
{
    myStructPV.Value["Member3"] = myVariable.Value;
}
```

The entire structure is written by defining the "**Value**" property – not only the individual member as might be assumed here.

To write multiple members in the structure at the same time, set the "**WriteValueAutomatic**" property to the value "**false**" and after making the setting, all members are written with the "**WriteValue**" method.

```
private void cmdSetStruct_Click(object sender, System.EventArgs e)
{
    myStructPV.WriteValueAutomatic = false;
    myStructPV.Value["Member1"] = 10;
    myStructPV.Value["Member2"] = 20;
    myStructPV.Value["Member3"] = myVariable.Value;
    myStructPV.WriteValue();
}
```

A value can also be assigned before calling the method "**WriteValue**" using for example, "**myStructPV.Value["Member2"].Assign(20)**".

5. SUMMARY

PVIServices can be used to create a Windows Client application from the visualization up to the creation of the service tools within the programming environment VisualStudio.NET.

This training module covers a small scope of the PVIServices classes. Refer to the PVIServices User Documentation and the PVIServices samples included in the PVI Server&Runtime / Development installation to expand basic knowledge of this topic.

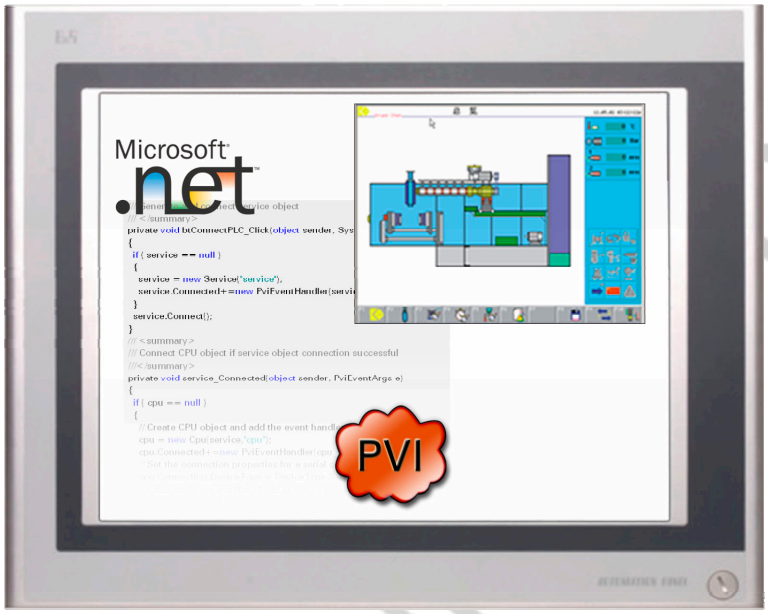


Fig. 12: PVIServices

Notes

Notes

Overview of training modules

TM200 – B&R Company Presentation **
TM201 – B&R Product Spectrum **
TM210 – The Basics of Automation Studio
TM211 – Automation Studio Online Communication
TM212 – Automation Target **
TM213 – Automation Runtime
TM220 – The Service Technician on the Job *
TM221 – Automation Components and Sources of Errors *
TM223 – Automation Studio Diagnostics
TM230 – Structured Software Generation
TM240 – Ladder Diagram (LAD)
TM243 – Sequential Function Chart (SFC) *
TM245 – Instruction List (IL) *
TM246 – Structured Text (ST)
TM247 – Automation Basic (AB) *
TM248 – ANSI C
TM250 – Memory Management and Data Storage
TM260 – Automation Studio Libraries I

TM400 – The Basics of Motion Control
TM402 – Dimensioning Motion Control Systems *
TM410 – The Basics of ASiM
TM440 – ASiM Basic Functions
TM441 – ASiM Multi-Axis Functions
TM445 – ACOPOS ACP10 Software
TM450 – ACOPOS Control Concept and Adjustment
TM460 – Starting up Motors *

TM600 – The Basics of Visualization
TM610 – The Basics of ASiV
TM630 – Visualization Programming Guide
TM640 – ASiV Alarm System
TM650 – ASiV Internationalization
TM660 – ASiV Remote
TM670 – ASiV Advanced

TM700 - Automation Net PVI
TM710 - PVI Communication
TM711 - PVI DLL Programming
TM712 - PVIServices
TM730 - PVI OPC

TM800 – APROL System Concept
TM801 – APROL Engineering Basics
TM810 – APROL Setup, Configuration and Recovery
TM811 – APROL Runtime System
TM812 – APROL Operator Management
TM813 – APROL XML Queries and Audit Trail
TM830 – APROL Project Engineering
TM840 – APROL Parameter Management and Recipes
TM850 – APROL Controller Configuration and INA
TM860 – APROL Library Engineering
TM865 – APROL Library Guide Book
TM870 – APROL Python Programming *
TM880 – APROL Report *

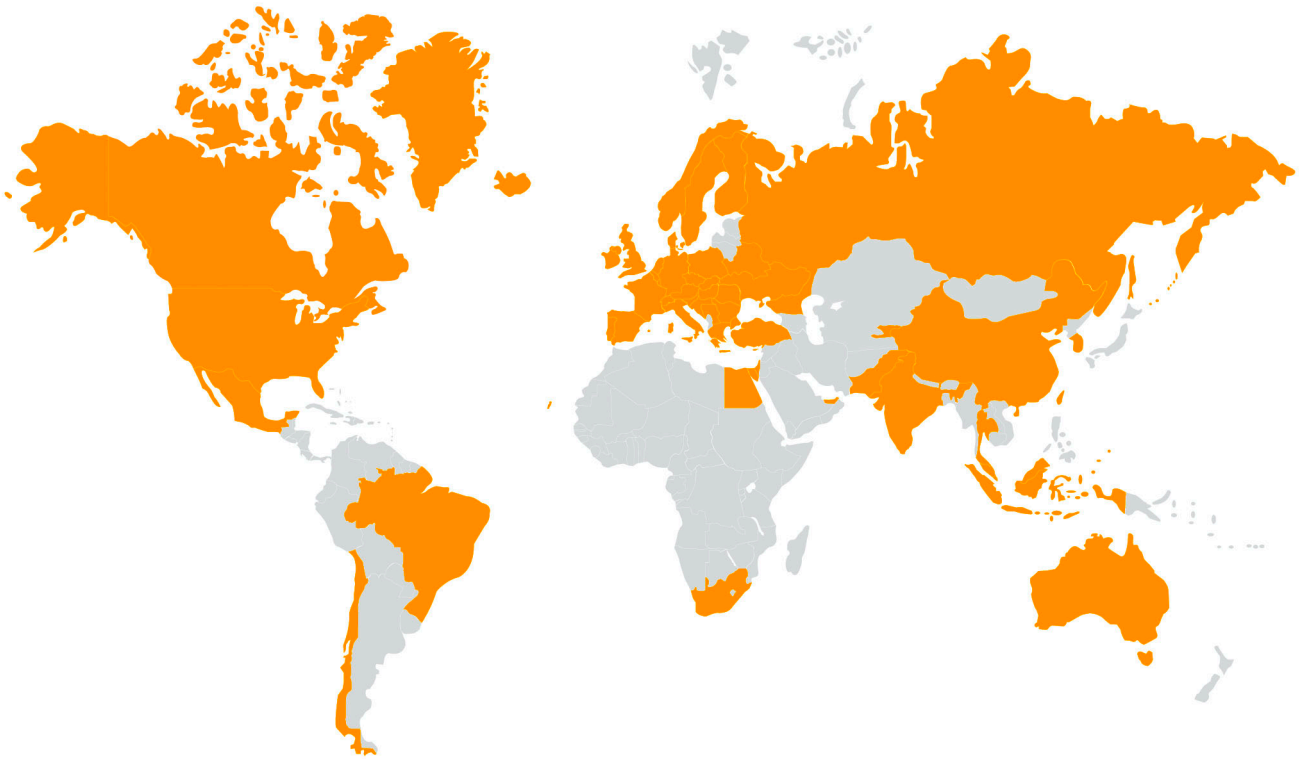
*) upon request
**) see Product Catalog

CORPORATE HEADQUARTERS

Bernecker + Rainer Industrie-Elektronik Ges.m.b.H.
B&R Straße 1
A-5142 Eggelsberg
Tel.: +43 (0) 77 48 / 65 86-0
Fax: +43 (0) 77 48 / 65 86-26
info@br-automation.com
www.br-automation.com

TM712TRE.00-ENG 0706

120 offices in more than 50 countries - **www.br-automation.com/contact**



Australia · Austria · Belarus · Belgium · Brazil · Bulgaria · Canada · Chile · China · Croatia · Cyprus · Czech Republic
Denmark · Egypt · Emirates · Finland · France · Germany · Greece · Hungary · India · Indonesia · Ireland · Israel · Italy · Korea
Kyrgyzstan · Malaysia · Mexico · The Netherlands · Norway · Pakistan · Poland · Portugal · Romania · Russia · Singapore
Slovakia · Slovenia · South Africa · Spain · Sweden · Switzerland · Taiwan · Thailand · Turkey · Ukraine · United Kingdom · USA