ibaPDA-Plugin

Custom Functions for ibaPDA

Manual

Issue 2.3

Measurement Systems for Industry and Energy

www.iba-ag.com

Manufacturer

iba AG

Koenigswarterstrasse 44

90762 Fuerth

Germany

Contacts

Main office	+49 911 97282-0
Fax	+49 911 97282-33
Support	+49 911 97282-14
Engineering	+49 911 97282-13
E-mail	iba@iba-ag.com
Web	www.iba-ag.com

Version	Date	Revision	Author	Version SW
2.3	08-2023	Plugin paths	IJ	8.4

Contents

1 About this documentation

This documentation describes the function and application of the software

ibaPDA-Plugin.

1.1 Target group and previous knowledge

This manual is aimed at qualified professionals who are familiar with handling electrical and electronic modules as well as communication and measurement technology. A person is regarded as professional if he/she is capable of assessing safety and recognizing possible consequences and risks on the basis of his/her specialist training, knowledge and experience and knowledge of the standard regulations.

1.2 Notations

In this manual, the following notations are used:

Action	Notation
Menu command	Menu *Logic diagram*
Calling the menu command	*Step 1 – Step 2 – Step 3 – Step x* Example: Select the menu *Logic diagram – Add – New function block*.
Keys	<Key name> Example: <Alt>; <F1>
Press the keys simultaneously	<Key name> + <Key name> Example: <Alt> + <Ctrl>
Buttons	<Key name> Example: <OK>; <Cancel>
Filenames, paths	`Filename`, `Path` Example: `Test.docx`

1.3 Used symbols

If safety instructions or other notes are used in this manual, they mean:

Danger!

The non-observance of this safety information may result in an imminent risk of death or severe injury:

- Observe the specified measures.

Warning!

The non-observance of this safety information may result in a potential risk of death or severe injury!

- Observe the specified measures.

Caution!

The non-observance of this safety information may result in a potential risk of injury or material damage!

- Observe the specified measures

Note

A note specifies special requirements or actions to be observed.

Tip

Tip or example as a helpful note or insider tip to make the work a little bit easier.

Other documentation

Reference to additional documentation or further reading.

2 System requirements

The following system requirements are necessary for the use of *ibaPDA-Plugin*:

- *ibaPDA* v8.4.0 or higher
- License for *ibaPDA-Plugin*

Licenses

Order no.	Product name	Description
30.681210	ibaPDA-Plugin	Plugin system to perform custom calculation on data measured by *ibaPDA*

3 Introduction

The *ibaPDA* plugin system has been designed to enable *ibaPDA* users to create their own functions that perform custom calculations on data measured by *ibaPDA*. These functions are executed by *ibaPDA* in real time. The functions can be used in expressions for virtual signals just as they were built-in functions. They also appear in the expression builder.

To create custom functions you have to create a .NET dll. You can write this dll in any .NET language (C#, C++ CLI, VB.NET …).

For information on implementing a plugin, chapter ➚ *How to implement a plugin*, page 8.

For information on the different ways you can debug your plugin, see ➚ *How to debug a plugin*, page 15.

There is a sample plugin provided with this package. You can find the sample solution in the sample directory of this package. It contains two projects.

- SamplePlugin is the plugin project that implements six functions: AddMultiple, Integrate, CountCharacter, GenerateText, LookupText and StoreValue.
- MainProgram is a windows forms application that hosts the plugin and that can be used to test the functions from the plugin.

4 How to implement a plugin

4.1 Create plugin project

The plugin consists of a .NET dll. You can create this dll with Visual Studio .NET 2017 or later.

Requirements for the dll plugin:

- The dll targets .NET framework 4.8.
- The dll is built for *Any CPU* because *ibaPDA* can run as a 32 bit or 64 bit application.

1. Create a new project for a class library in your preferred language.
2. Add a reference to `ibaPdaPluginInterface.dll`.
   You can find this dll in the server subdirectory of the *ibaPDA* installation directory (default: `C:\Program Files\iba\ibaPDA\Server`).

   This dll contains the definitions of the required interfaces and some required helper classes.
3. Optionally, you can find an xml documentation file `ibaPDAPluginInterface.xml` in the bin directory of this package. This file contains some extra information used by the intel-lisense of Visual Studio.

4.2 Implement IPlugin

Create a new class and derive it from IPlugin. This class is responsible for providing a list of all functions implemented in this dll and for creating function objects that can calculate the func-tions.

The interface definition looks like this in C# syntax:

```
public interface IPlugin
{
    string Name {get;}
    PluginFunctionInfo[] GetFunctions();
    IPluginFunction CreateFunctionObject(string functionName);
    void ClearAllFunctionObjects();
}
```

The interface definition looks like this in VB.NET syntax:

```
Public Interface IPlugin
   ReadOnly Property Name() As String
   Function GetFunctions() As PluginFunctionInfo()
   Function CreateFunctionObject(ByVal functionName As String) As
           IPluginFunction
   Sub ClearAllFunctionObjects()
End Interface
```

4.2.1 Property Name

This property must return the name of the plugin. This name is shown in the function tree of the expression builder dialog of *ibaPDA*.

4.2.2 Function GetFunctions()

This function must return a list of all the functions implemented by this plugin. The list is an array of PluginFunctionInfo objects. The PluginFunctionInfo class has following properties:

`string` Name: The name of the function. This name appears in the function tree.

`int` MinArguments: The minimum number of arguments for this function.

`int` MaxArguments: The maximum number of arguments for this function. This number can be different from MinArguments if the function supports default arguments.

`ResultTypeEnum` ResultType: The result type of this function. This can be:

- *Analog*: for analog results
- *Digital*: for digital results
- *Invariant*: if the result type is the same as the type of the first argument
- *Invariant2*: if the result type is the same as the type of the second argument
- *Invariant3*: if the result type is the same as the type of the third argument
- *Text*: for text results

`string` Prototype: The prototype of the function. This must be a string that looks like this: *Add('expr1', 'expr2', ‘expr3=0’)*. The prototype must start with the name of the function. The function arguments must be between single quotes (‘) and they must be separated by a comma (,). If there are optional function arguments, then they must come last and the default value should be mentioned as ‘arg=value’. Text arguments should be surrounded by double quotes (“) e.g. “’TextArg’”. Notice the single quotes inside of the double quotes.

`string` Description: The description of what the function does. This text is displayed in the expression builder dialog of *ibaPDA.*

`FunctionFlags` Flags: Some flags that determine how the function can be used.

- *None*: No special flags are needed
- *CanOptimize*: Set this if the result of the function is constant when the arguments are constants. The function is then evaluated only once at the start of the acquisition in *ibaPDA*.
- *Hidden*: Use this for obsolete functions that should not be shown in the expression builder but can still be used.
- *IsRetentive*: Set this for a function that can save the last state and reuse it when the acquisition is restarted. Either IRetentiveFunction or IRetentiveStateFunction must be implemented.
- *OnlyEvaluateWhenInputsChanged*: Set this if the Calculate function should only be called when the value of the arguments has changed. This flag is only relevant for functions that return text.

4.2.3 Function CreateFunctionObject()

This function must return an object that can calculate the requested function. The requested function is identified by its name. This function is called by *ibaPDA* for every expression the function is used in.

For information on the IPluginFunction interface, see ➚ *Implement IPluginFunction*, page 10.

4.2.4 Function ClearAllFunctionObjects()

This function is called when the acquisition is stopped in *ibaPDA*. It allows the plugin to cleanup any resources it may have acquired for the calculations.

4.3 Implement IPluginFunction

A plugin function must be implemented by a class that derives from IPluginFunction. A function has always one result of type double and can have multiple arguments of type double.

The IPluginFunction interface definition looks like this in C# syntax:

```
public interface IPluginFunction
{
    void Initialize(Int64 xBegin, Int64 xBase, double[] inputs);
    double Calculate(Int64 x, double[] inputs);
}
```

The interface definition looks like this in VB.NET syntax:

```
Public Interface IPluginFunction
    Sub Initialize(ByVal xBegin As Int64, ByVal xBase As Int64,
         ByVal inputs As Double())
    Function Calculate(ByVal x As Int64, ByVal inputs As Double())
         As Double
End Interface
```

4.3.1 Function Initialize()

This function is called one time by *ibaPDA* at the start of the acquisition.

- *xBegin* is the timestamp of the first sample. It is expressed as a number of 100 ns ticks.
- *xBase* is the time base of all the inputs and of the output of the function. It is again expressed as a number of 100 ns ticks.
- The *inputs* array contains values of the first sample of each argument of the function. The length of the inputs array can be smaller than the maximum number of arguments. In this case there are some missing optional arguments and you should initialize them to the correct default value.

This function should do all initializations required for the calculation.

4.3.2 Function Calculate()

This function is called for every sample. It gets the values of the arguments as input and it must calculate the result of the function.

- The timestamp *x* represents the current timestamp (as always expressed in 100 ns ticks).
- The *inputs* array contains the values of each argument at the current timestamp *x*.

This function is called a lot during the acquisition of *ibaPDA* so it must be well optimized. *ibaPDA* receives new data for signals in blocks of about 50 ms. This means that if the timebase of the signals is 1 ms, then it gets 50 samples of a signal at once. *ibaPDA* then calls the Calculate function 50 times quickly after each other. Then *ibaPDA* waits x ms until a new block of data arrives and then the Calculate function is called 50 times quickly after each other again. So you have to take this timing into account when you implement your plugin.

4.4 Implement IPluginFunctionWithText

If you want to handle text values in your plugin, then you have to implement the IPluginFunctionWithText interface instead of the IPluginFunction interface.

The IPluginFunctionWithText interface definition looks like this in C# syntax:

```
public interface IPluginFunctionWithText : IPluginFunction
{
   void Initialize(Int64 xBegin, Int64 xBase, double[] inputs,
         string[] textInputs);
   double Calculate(Int64 x, double[] inputs, string[] textInputs);
}
```

The interface definition looks like this in VB.NET syntax:

```
Public Interface IPluginFunctionWithText Inherits IPluginFunction
   Sub Initialize(ByVal xBegin As Int64, ByVal xBase As Int64,
        ByVal inputs As Double(),ByVal textInputs As String())
   Function Calculate(ByVal x As Int64, ByVal inputs As Double(),
        ByVal textInputs As String()) As Double
End Interface
```

You only have to implement the Initialize and Calculate functions with the textInputs arguments. The inherited Initialize and Calculate functions of the IPluginFunction interface can be left empty because *ibaPDA* does not call them. The textInputs array contains the values of the text arguments in the order they appear in the function prototype. Text arguments and normal arguments can be mixed.

4.5 Implement IPluginFunctionWithTextResult

If you want to return text values in your plugin, then you have to implement the IPluginFunctionWithTextResult interface instead of the IPluginFunction or IPluginFunctionWithText.

The IPluginFunctionWithTextResult interface definition looks like this in C# syntax:

```
public interface IPluginFunctionWithTextResult : IPluginFunction
{
   void Initialize(Int64 xBegin, Int64 xBase, double[] inputs,
         string[] textInputs);
   string Calculate(Int64 x, double[] inputs, string[] textInputs);
}
```

The interface definition looks like this in VB.NET syntax:

```
Public Interface IPluginFunctionWithTextResult Inherits IPluginFunction
   Sub Initialize(ByVal xBegin As Int64, ByVal xBase As Int64,
       ByVal inputs As Double(),ByVal textInputs As String())
   Function Calculate(ByVal x As Int64, ByVal inputs As Double(),
       ByVal textInputs As String()) As String
End Interface
```

You only have to implement the Initialize and Calculate functions with the textInputs arguments and the text result. The inherited Initialize and Calculate functions of the IPluginFunction interface can be left empty because *ibaPDA* does not call them. The textInputs array contains the values of the text arguments in the order they appear in the function prototype. Text arguments and normal arguments can be mixed.

The result of this function is a text signal. Text signals are always non-equidistant. This means that you do not have to generate a text in every Calculate call. You may return null to not generate a text sample for timestamp x. If the function flag *OnlyEvaluateWhenInputsChanged* is set for this function, then the Calculate function is only called when at least one of the inputs has changed.

4.6 Implement IRententiveFunction or IRetentiveStateFunction

A retentive function stores the calculation result at the stop of the acquisition and uses it again at the start of the acquisition. In order to implement such a function your class must derive from IRetentiveFunction or IRetentiveStateFunction, and the IsRetentive function flag should be set in the PluginFunctionInfo.

The IRetentiveFunction interface allows saving and restoring the last calculation result of the function. Its definition looks like this in C# syntax:

```
public interface IRetentiveFunction
{
   double RetentiveValue { get; set; }
}
```

The IRetentiveStateFunction interface allows saving and restoring the last state of the function. The last state is a string. You can decide freely on the structure of this string, e.g. as a concatenation of floating point values separated by a comma or a semicolon. It is important that this string is culture invariant. The definition of the interface looks like this in C# syntax:

```
public interface IRetentiveStateFunction
{
    string RetentiveState { get; set; }
}
```

The RetentiveState setter must be implemented in a robust way. It is possible that the incoming string is not formatted as expected. Note that you can change the RetentiveState string manually in the *ibaPDA* I/O Manager in the Virtual retentive module.

At the stop of the acquisition, *ibaPDA* reads the value of the RetentiveValue property. *ibaPDA* then stores this value as the default value of the retentive virtual signal. At the next start of the acquisition, *ibaPDA* sets the saved value to the RetentiveValue property before the Initialize function is called.

Note

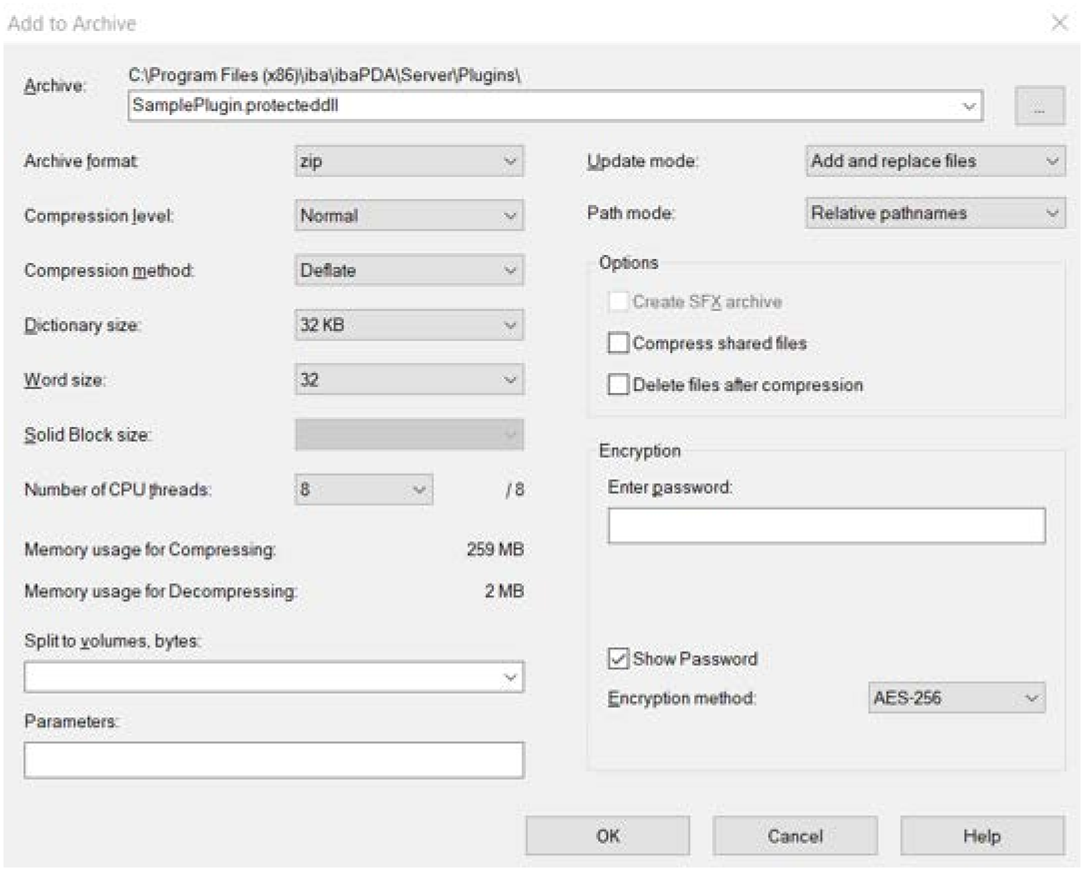
If the function implements both IRetentiveFunction and IRetentiveStateFunction, the implementation of IRetentiveFunction is ignored.

4.7 Create a protected plugin dll

You can protect a plugin dll so that the plugin can only be used on an *ibaPDA* system with a specific license container. To protect a plugin dll you need a password, which you can obtain from iba AG and is license container specific. An *ibaPDA* system running with the correct license container unzips the protected dll in memory and then loads the plugin functions.

The product *ibaPDA-Plugin Password Generation* (Order no. 60.000015) is necessary to obtain a password for protection. You have to provide the license number of the target *ibaPDA* system as well.

1. Using e.g. 7zip, create a ZIP archive that includes your plugin dll.
2. Set the following parameters:
   - Archive format: zip (compression level = normal; compression method = deflate)
   - Encryption method: AES-256


Add to Archive

Archive: C:\Program Files (x86)\iba\ibaPDA\Server\Plugins\
SamplePlugin.protecteddll

Archive format: zip
Compression level: Normal
Compression method: Deflate
Dictionary size: 32 KB
Word size: 32
Solid Block size:
Number of CPU threads: 8 / 8
Memory usage for Compressing: 259 MB
Memory usage for Decompressing: 2 MB
Split to volumes, bytes:
Parameters:

Update mode: Add and replace files
Path mode: Relative pathnames

Options
- [ ] Create SFX archive
- [ ] Compress shared files
- [ ] Delete files after compression

Encryption
Enter password:
- [x] Show Password
Encryption method: AES-256

3. Enter the password you have obtained from iba AG.

   You can use the password to protect multiple plugin DLL files used on the same *ibaPDA* system.

4. After you have created the ZIP file, rename the extension of this file from **.zip* to **.protectedDLL*.

5. Move the zipped plugin dll in the plugin directory `C:\ProgramData\iba\ibaPDA\Plugins\Server\`.

   This directory only needs the protected plugin dll.

4.8 Install plugin

Installing the plugin for *ibaPDA* is an easy procedure.

1. Copy the plugin dll to the Plugins subdirectory under the *ibaPDA* server directory.
   Default `C:\ProgramData\iba\ibaPda\Plugins\Server\`

2. Also copy all dlls that your plugin requires to this directory.

   Do not copy `ibaPdaPluginInterface.dll` to the Plugins\Server directory because it is already present in the server directory.

3. All dlls located in the Plugins directory are loaded by *ibaPDA* when the service is started.
   If you copy the dll to the directory when the *ibaPDA* service is already running, restart the *ibaPDA* service to load the new plugin.

5 How to debug a plugin

There are two ways to debug a plugin. You can either write a small test program that hosts the plugin and calls its functions. The sample solution in this package contains such an example test program. Or you can also use *ibaPDA* to host the plugin and debug it by attaching to the *ibaPDA* service process.

5.1 Debug using ibaPDA

You can debug the plugin with Visual Studio when it is loaded by *ibaPDA*.

1. Stop the *ibaPDA* service.
2. Copy the plugin dll and the plugin pdb file to the Plugins directory.
3. Start the *ibaPDA* service.

→ *ibaPDA* loads the plugin.

Tip

You can automate this procedure by writing a batch file. You can find a sample batch file in the sample solution (`CopyPlugin.bat`).

4. When the *ibaPDA* service is running, you can attach to the process with Visual Studio. For this, run Visual Studio as administrator.
5. In Visual Studio open the *Debug* menu and select *Attach to Process*.
6. Find the `ibaPdaService.exe` process and attach to it.

   If you cannot find the `ibaPdaService.exe` process in the list, then check the *Show process from all users* checkbox.
7. In the field *Attach to* select the option *Manage (.NET 4x code)*. Do not select *Native*.
8. Now set breakpoints in your plugin and start debugging it.

   If the breakpoints are currently not hit, check whether the plugin dll and pdb files in the *ibaPDA* plugins directory match with the dll and pdb files in the bin directory of the Visual Studio project.

The advantage of this method is that you are now debugging in the real target environment for your plugin. The disadvantage is that it is not so fast.

5.2 Debug using a test program

You can debug the plugin with a test program.

1. Add a test program to your solution. This test program can directly reference the plugin or it can host it via a PluginManager (see sample solution in this package).
2. Call the CreateFunctionObject function on IPlugin.
3. Call Initialize on the returned IPluginFunction object.
4. Finally, call Calculate for every sample.

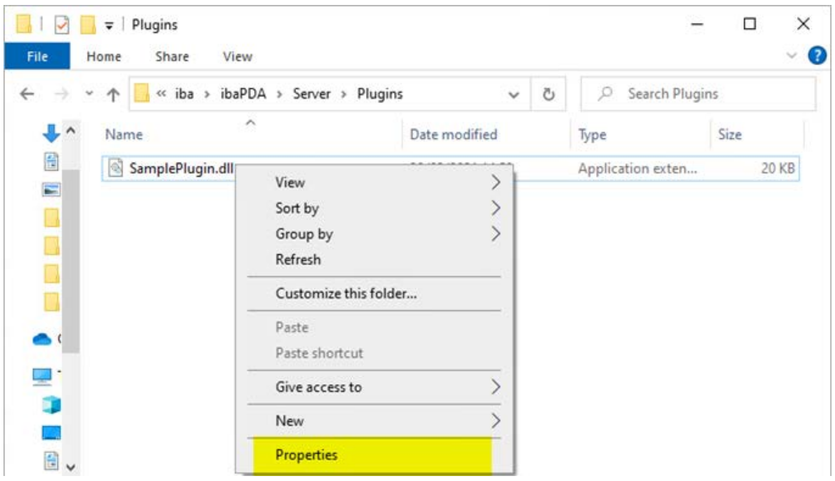
The advantage of this method is that it is fast. The disadvantage is that the calling context can be different than the calling context of the *ibaPDA* service.

6 Troubleshooting

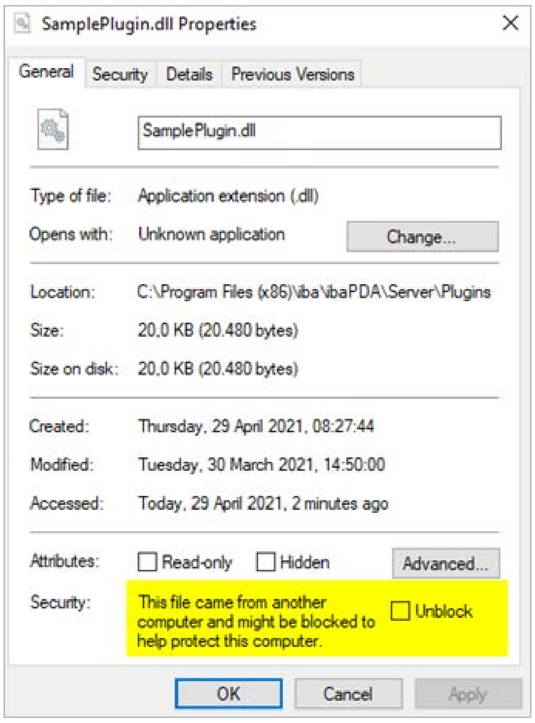
6.1 Plugin is not shown in the I/O Manager after copying the DLL

If the plugin is not shown in the I/O Manager although the plugin is in the right directory and the *ibaPDA* service has been restarted, it is possible that the dll is copied from a non-trustworthy place like from an e-mail.

1. Open the file properties of the dll in Windows explorer.
   Right-click on the file and select *Properties*.



2. If the *General* tab has a remark that the file is blocked, check the *Unblock* checkbox and click *Apply*.



SamplePlugin.dll

Type of file: Application extension (.dll)
Opens with: Unknown application

Location: C:\Program Files (x86)\iba\ibaPDA\Server\Plugins
Size: 20,0 KB (20.480 bytes)
Size on disk: 20,0 KB (20.480 bytes)

Created: Thursday, 29 April 2021, 08:27:44
Modified: Tuesday, 30 March 2021, 14:50:00
Accessed: Today, 29 April 2021, 2 minutes ago

Attributes: ☐ Read-only ☐ Hidden
Security: This file came from another computer and might be blocked to help protect this computer. ☐ Unblock

3. Restart the *ibaPDA* service.

→ The file should be loaded.

7 Support and contact

Support

Phone:	+49 911 97282-14
Fax:	+49 911 97282-33
Email:	support@iba-ag.com

Note

If you need support for software products, please state the number of the license container. For hardware products, please have the serial number of the device ready.

Contact

Headquarters

iba AG
Koenigswarterstrasse 44
90762 Fuerth
Germany

Phone:	+49 911 97282-0
Fax:	+49 911 97282-33
Email:	iba@iba-ag.com

Mailing address

iba AG
Postbox 1828
D-90708 Fuerth, Germany

Delivery address

iba AG
Gebhardtstrasse 10
90762 Fuerth, Germany

Regional and Worldwide

For contact data of your regional iba office or representative please refer to our web site:

www.iba-ag.com