bmcm

bmc messsysteme gmbh

LIBADX

Programming Interface (ActiveX Control) for
bmcm DAQ system drivers

Installation and Programming Guide

Version 4.6

www.bmcm.de

bavarian measurement company munich

Contents

1 Overview

1.1 Introduction

LibadX is a common programming interface to all data acquisition systems from BMC Messsysteme GmbH. This interface can be accessed by all programming environments in which ActiveX components can be loaded (e.g. C++®, Visual C++®, Visual C#®, Visual Basic®, Visual Basic® .NET, Delphi®).

- **LibadX is a 32-bit interface. If programming on a 64-bit system, the application must be created as a 32-bit application.**
- **Please note that these code extracts as well as all the other examples in this manual consciously skip any error handling to simplify matters. Of course, this has to be realized in self written programs.**
- **The integration of an ActiveX Control is done by the programming environment used. Because every programming environment realizes the integration in a different way, this manual can only give an overview about how to use the LibadX in different programming environments. For more information about the integration of ActiveX components, please see the documentation of your programming environment.**

Normally, the programming environment imports the ActiveX components and generates the source code for a utility class used to call the functions of the component. This utility class eventually defines the proper calling convention of the functions.

Depending on the programming environment, the functions described in this manual may be available under another name or with slightly changed parameters. For this reason, the documentation of the relevant programming environment should be consulted to get information about the respective conventions when importing ActiveX components.

1.2 BMC Messsysteme GmbH

BMC Messsysteme GmbH stands for innovative measuring technology made in Germany. We provide all components required for the measuring chain, from sensor to software.

Our hardware and software components are perfectly tuned with each other to produce an extremely user-friendly integrated system. We put great emphasis on observing current industrial standards, which facilitate the interaction of many components.

Products by BMC Messsysteme are applied in industrial large-scale enterprises, in research and development and in private applications. We produce in compliance with ISO-9000-standards because standards and reliability are of paramount importance to us - for your profit and success.

Please visit us on the web (http://www.bmcm.de/) for detailed information and latest news.

1.3 Copyrights

The programming interface **LibadX** with all extensions has been developed and tested with utmost care. BMC Messsysteme GmbH does not provide any guarantee in respect of this manual, the hard- and software described in it, its quality, its performance or fitness for a particular purpose. BMC Messsysteme GmbH is not liable in any case for direct or indirect damages or consequential damages, which may arise from improper operation or any faults whatsoever of the system. The system is subject to changes and alterations which serve the purpose of technical improvement.

The programming interface **LibadX**, the manual provided with it and all names, brands, pictures, other expressions and symbols are protected by law as well as by national and international contracts. The rights established therefrom, in particular those for translation, reprint, extraction of depictions, broadcasting, photomechanical or similar way of reproduction - no matter if used in part or in whole - are reserved. Reproduction of the programs and the manual as well as passing them on to others is not permitted. Illegal use or other legal impairment will be prosecuted by criminal and civil law and may lead to severe sanctions.

Updated: 12/09/2014

BMC Messsysteme GmbH

Hauptstrasse 21
82216 Maisach
GERMANY

Phone: +49 8141/404180-1
Fax: +49 8141/404180-9
E-mail: info@bmcm.de

1.4 Quickstart

Install the hardware as described in your documentation before installing the LibadX and verify in the Windows® device manager if the hardware is recognized by the PC.

- To check the correct installation of the measurement hardware, open the Windows® Device Manager displaying the current PC configuration:
  - **Windows® 8**: Start / Control Panel / System and Security / System / Device Manager
  - **Windows® 7**: Right-click screen corner bottom left (keyboard "Windows+X") / Device Manager
  - **Windows® XP**: Start / Control Panel / System / TAB "Hardware" / button "Device Manager"
- If the installation was successful (data acquisition system must be connected and operational!), the newly installed hardware has been added to the entry "Data Acquisition (BMC Messsysteme GmbH)". A double-click on the device shows its properties and any existing conflicts.
- If the hardware is recognized by the PC and working properly, install the **LibadX** by means of the included "Software Collection" CD. Change to the product page of the bmcm hardware used ("Products / <Product name>") and click the item "STR-LIBADX" in the section "API (Programming)" for programming on Windows®.
- The installation can be opened directly. If your browser does not allow this, please first save the file **`libad-actx.exe`** on hard disk and then start the installation by clicking the icon.
- You only need to enter the directory path before the available storage capacity is calculated and files are copied to disk. The required ActiveX component is copied to the Windows® system directory.
- After installation, the **LibadX** ActiveX Control is available to be used in own programs. The integration may be different depending on the programming environment (see "Integration in Programming Languages", p. 15).

2 Installation and Integration

2.1 General

The hardware driver must be installed before installing LibadX!

For installation, insert the bmcm "Software Collection" CD included with delivery into your CD-ROM drive.

The programming interface **LibadX** is implemented as an *ActiveX Control* , which is registered in the system by the installation program. This, however, is not sufficient for the **LibadX** functions to be available in most of the programming environments. The following chapters give an overview about the necessary integration for some selected programming environments. For detailed information about integrating an ActiveX Control, please see the documentation of your programming environment.

2.2 LibadX Installation

When inserting the "Software Collection" CD, a CD starter is opened. If the AutoPlay function of your CD-ROM is not selected, please open the file **`openhtml.exe`**.

Change to the product overview of the bmcm hardware by selecting the category "Products" and then the data acquisition system used. For programming on Windows® XP/7/8, click the item "STR-LIBADX" in the section "API (Programming)" to start the installation.

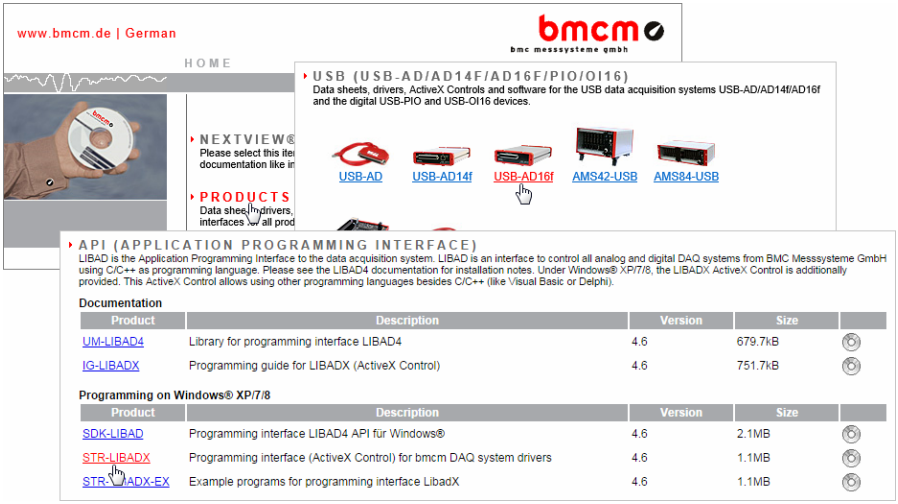

Figure 1

If using the CD starter in HTML format, you can decide to directly open the installation program or to save it to disk. Both options are possible.

Some browsers require saving the installation program to hard disk before. In this case, you must start the installation program **`libad-actx.exe`** explicitly after copying.

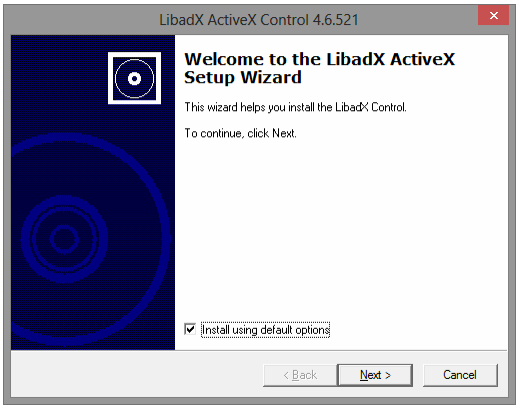

Fig. 1

An installation wizard will guide you through the installation step by step. The button "Next" will lead you to the next dialog box, with "Back" you go one step backwards. The installation can always be stopped early without saving anything by pressing "Cancel".

If you do not want to use default settings, you can uncheck this option in the first window of the installation program and decide where to install the driver package.

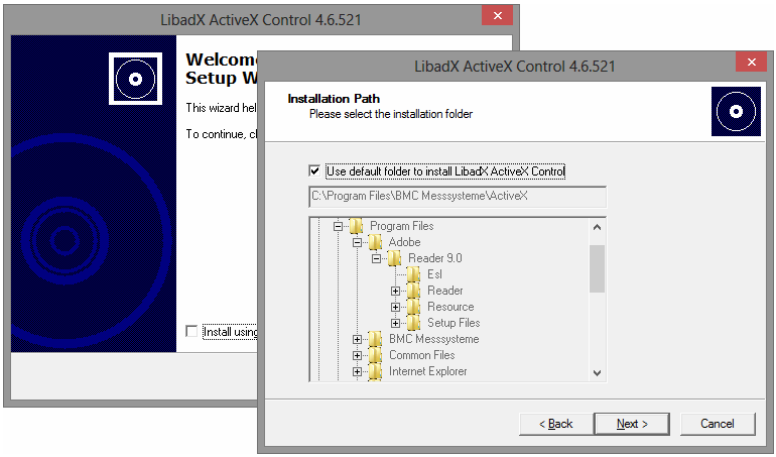

Fig. 2

The suggested default directory path can be modified as desired, of course. To switch to another installation directory, uncheck the checkbox for to activate the boxes below.

After all information is given, the size of the available disk space is determined and the files required to install the **LibadX** ActiveX Control are copied to disk.

Restart your computer if necessary for these changes to take effect..

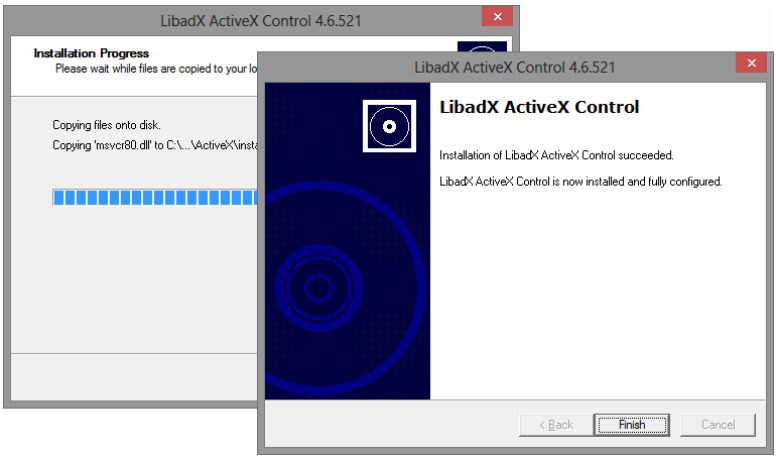

Figure 2

2.3 Integration in Programming Languages

2.3.1 Integration in Visual Basic® 4.0 - 6.0

Start Visual Basic® and click the option "Standard EXE" in the start screen (or menu item "File / New Project").

Like any other ActiveX Control, the **LibadX** is integrated in Visual Basic® by selecting the entry "Components" of the "Project" menu. In the following dialog box "Components", check the item "LibadX Object Library 4.0".

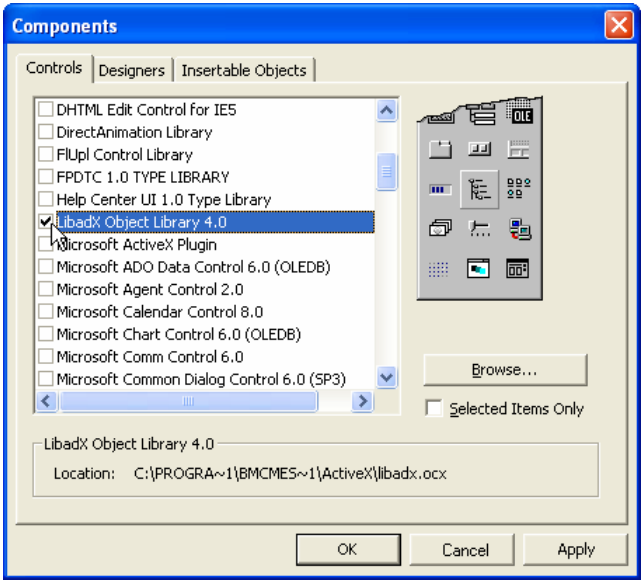

Figure 3

The **LibadX** icon is included in the toolbar of Visual Basic® now and available to be integrated in a form. Like the timer control, it is invisible while the program is executed.

Figure 4

Click the icon as usual and draw a frame on the form where the hardware is to be used. After adding the object, this frame is reduced to its original icon size.

Create the following routine **`Form_Load()`** in the code window of the project:

VB	`Private Sub Form_Load()`
	`  LIBADX1.AboutBox`
	`End Sub`

To make sure the **LibadX** is correctly installed and available in Visual Basic®, we recommend to start this program. It must display the form without any errors on the screen.

- **For compatibility reasons, the icon of the former programming interface BMCSAD is also integrated in the toolbar. LibadX users do not need this icon or the former programming interface.**

- **Please note that these code extracts as well as all the other examples in this manual consciously skip any error handling to simplify matters. Of course, this has to be realized in self written programs.**
- **Other example programs (see "Example Programs", p. 23") with source code can be installed from the LibadX product page of the "Software Collection" CD.**

2.3.2 Integration in Delphi® 3.01 - 5.0

Start Delphi® and open a new project (menu item "File / New project").

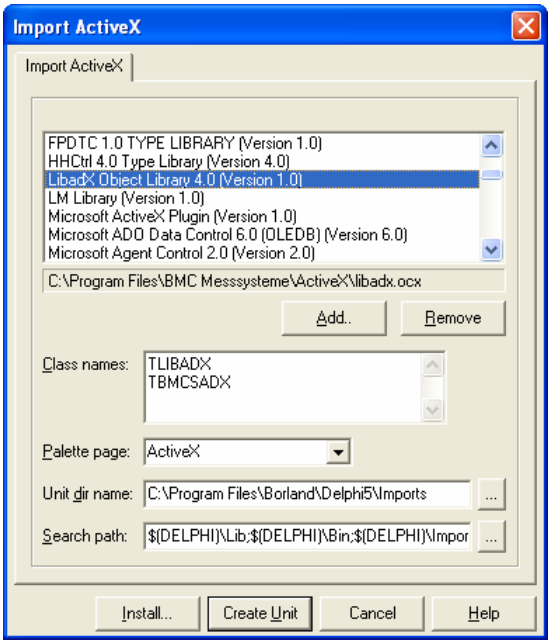

Figure 5

In the "Components" menu, call the command "Import ActiveX...". Then select "LibadX Object Library 4.0" in the displayed dialog box. Press the button "Install..." to import the **LibadX** in Delphi® and register the ActiveX control as a component.

In the following dialog, choose the package which the new component is to be installed in and confirm with OK.

The selected package is rebuilt and installed to integrate the information about the new ActiveX Control. When compilation is finished, the changes done are reported.

Figure 6

The **LibadX** icon is provided in the tab "ActiveX" of the Delphi® toolbar now. Add the object to the form of the new project.

Create an event handler for the **`OnCreate()`** event of the form and proceed as follows:

Delphi	`procedure TForm1.FormCreate(Sender: TObject);`
	`begin`
	`  LibadX.AboutBox ();`
	`end;`

To make sure the **LibadX** is correctly installed and available in Delphi®, we recommend to start this program. It must display the form without any errors on the screen.

- **For compatibility reasons, the icon of the former programming interface BMCSAD is also integrated in the toolbar. LibadX users do not need this icon or the former programming interface.**

- **Please note that these code extracts as well as all the other examples in this manual consciously skip any error handling to simplify matters. Of course, this has to be realized in self-written programs.**
- **Other example programs (see "Example Programs", p. 23") with source code can be installed from the LibadX product page of the "Software Collection" CD.**

2.3.3 Integration in Visual C++® 5.0/6.0

By means of the preprocessor command **`#import`** Visual C++® 5.0/6.0 provides for the possibility to integrate COM interfaces into a C++® program. The following code examples demonstrate this procedure:

C++

```
#include <windows.h>
#import "c:\LibadX\LibadX.ocx"

LIBADX::_DLibadXPtr libadx;

int
main (int argc, char **argv)
{
  HRESULT result = CoInitialize (NULL);
  if (FAILED (result))
    return FALSE;

  libadx.CreateInstance (__uuidof(LIBADX::LIBADX));
  libadx->AboutBox ();

  return 0;
}
```

- **For further details about `#import`, `__uuidof()` and the compiler support classes for COM see the article "Microsoft Visual C++® Compiler Native COM Support" from Microsoft® as well as the relating Microsoft® compiler documentation.**

- **Please note that these code extracts as well as all the other examples in this manual consciously skip any error handling to simplify matters. Of course, this has to be realized in self written programs.**
- **Other example programs (see "Example Programs", p. 23") with source code can be installed from the LibadX product page of the "Software Collection" CD.**

2.3.4 Integration in Visual C#®.NET

The "managed code" of a .NET program does not contain any direct support to call ActiveX Controls. For this reason, a DLL serving as a "bridge" between "managed code" and ActiveX Control must be generated before using the ActiveX Control. In this case, only a reference to this "bridge" is passed to the Visual C#® program.

Although Visual Studio® supports the automatic import of ActiveX Controls, this procedure involves restrictions (see MSDN documentation). It is recommended to generate the respective bridge by calling the program **`tlbimp`** of the .NET SDK.

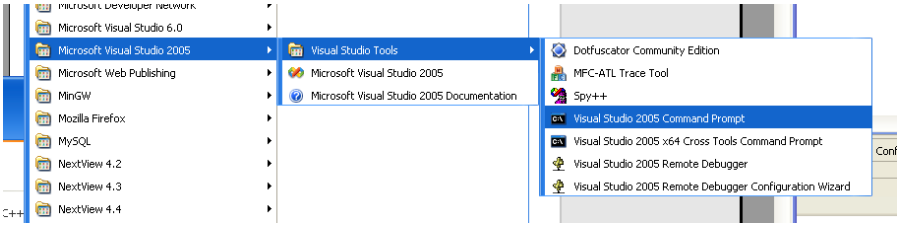

Figure 7

Start a "Microsoft Visual Studio Command Prompt" and enter the following command (make sure to replace **`libadx.ocx`** by the complete path to the ActiveX component).

```
tlbimp libadx.ocx /out:libadxTypeLib.dll /namespace:LIBADX
```

Then a reference to **`libadxTyleLib.dll`** can be added to each .NET program and the functionality of the ActiveX Control is available to Visual C#®.

A batch file to create the "bridge" and to compile a Visual C#® program is provided in the example programs for the LIBADX ActiveX Control (see "Example Programs, p. "23).

The proper calling conventions of the generated "bridge" DLL can be looked up with the Visual Studio® Object Browser.

2.3.5 Integration in VB.NET (Microsoft®)

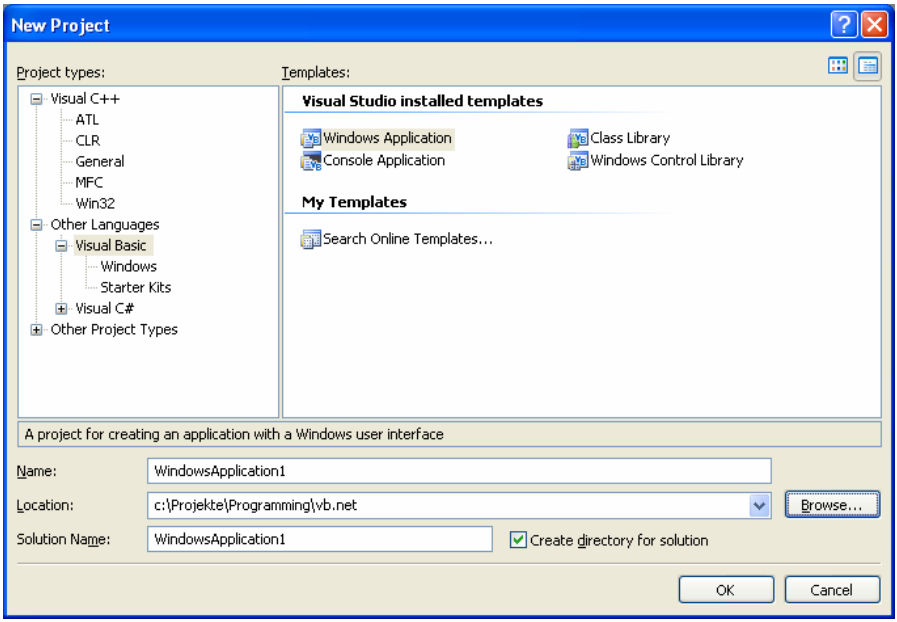

Figure 8

Start Visual Studio® and create a new project in Visual Basic® (e.g. click menu item "File / New Project") as a Windows® Application (s. Figure 8).

Open the context menu of the tool box with a right click and select the command "Choose Items…".

Check the COM component "LibadX Object Library 4.0" to integrate the **LibadX** ActiveX Control in the programming environment.

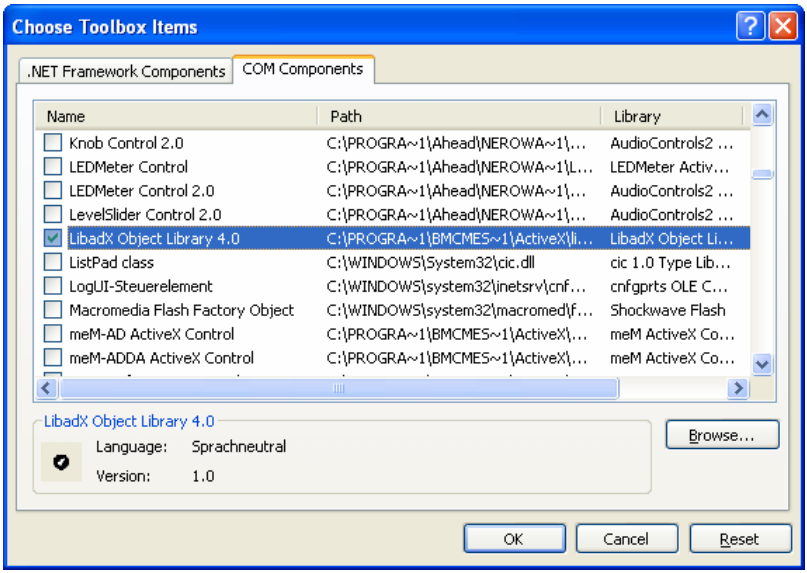

Figure 9

The **LibadX** icon is included in the toolbar of Visual Basic® now and available to be integrated in a form.

Click the icon as usual and draw a frame on the form where the hardware is to be used. After adding the object, this frame is reduced to its original icon size.

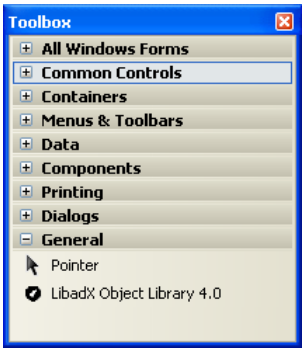

Figure 10

2.4 Example Programs

The "Software Collection" CD provides example programs demonstrating how to use the **LibadX** ActiveX Control. They can be installed from the respective product page of the DAQ system used.

To start the installation program, select the item "STR-LIBADX-EX" in the section "Programming on Windows® XP/7/8" listed under "API (Application Programming Interface)" on the product page.

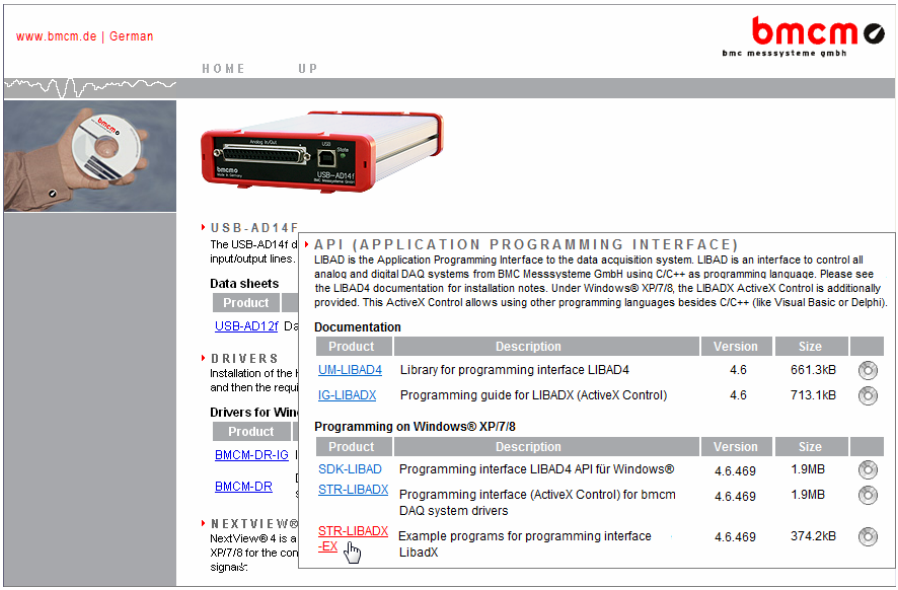

Figure 11

The example programs are provided in the directory chosen during installation (e.g. "Programs \ BMC Messsysteme \ ActiveX \ LibadX Examples") differentiated by programming language.

Programming language	Folder
Visual Basic®	`vb`
Delphi®	`delphi`
Visual C++®	`vc5`
Visual C#®	`.net`

Please note that all example programs are intended to be very simple and do not contain any error handling. Therefore, they cannot be considered a full application.

3 Basics

3.1 General

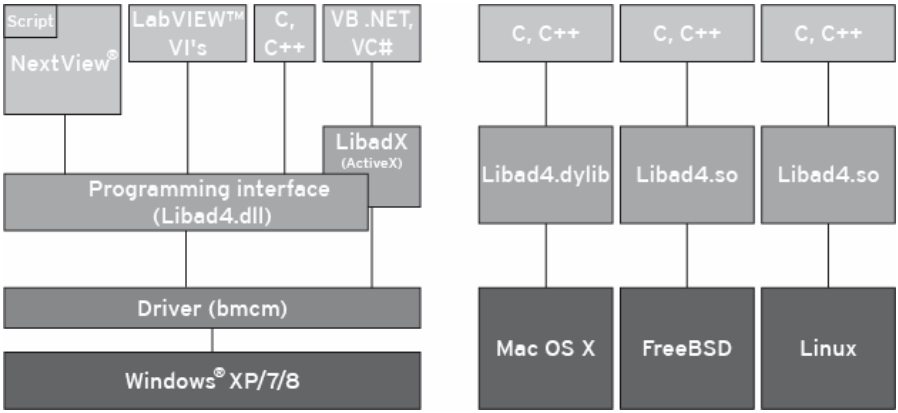

Figure 12

The **LibadX** ActiveX control is the programming interface to the **LIBAD4** library, which is an interface to all data acquisition systems of BMC Messsysteme GmbH to read and write single values, read in an analog channel or set a value of an analog output.

In addition to the input and output of single values, a scan can be carried out with the **LibadX**. Scanning of the input channels takes place in the corresponding driver so that it is time decoupled from the application allowing for the input channels to be scanned fast and without any loss of measuring values.

Besides that, you have got access to the measuring files of the data acquisition and analysis software **NextView®4**.

3.2 Connect to the Data Acquisition System

The **LibadX** ActiveX control provides two functions for opening or closing the connection to a data acquisition system.

With the **`Open()`** function a data acquisition system is opened, with **`Close()`** the connection is closed. The following example demonstrates the basic procedure:

```
if (LIBADX1.Open ("usb-pio"))
   ...
   LIBADX1.Close
else
   MsgBox "Could not open USB-PIO device"
```

The name of the data acquisition system is passed to the function **`Open()`**. This string is not case-sensitive, i.e. "usb-pio" and "USB-PIO" both open a USB-PIO / USB-PIO. If a connection to a data acquisition system has been opened, **`Open()`** returns the value **`TRUE`**, and **`FALSE`** if an error occurs.

It is not possible, to use one object for opening several devices at the same time. However, several (different) data acquisition systems can be opened with several objects. The following example opens a PCIe-BASE / PCI-BASEII/300/1000 / PCI-PIO and a USB-PIO / USB-PIO:

```
if (LIBADX1.Open ("pcibase")
    AND LIBADX2.Open ("usb-pio"))
    ...
endif
```

3.2.1 Channel Numbers and Measuring Ranges

In **LibadX**, input and output channels are identified by their channel number. The channel number depends on the data acquisition system used and is explained in the relating chapters. The first analog input of a USB-AD14f / USB-AD12f, for example, is channel 1.

In addition to the channel number, analog channels require information about the measuring range (or output range) used to scan (or to output). Like the channel

number, the measuring range depends on the data acquisition system and is documented in the following chapters.

3.2.2 iM-AD25a / iM-AD25 / iM3250T / iM3250

To open the iM-AD25a, iM-AD25, iM3250T or iM2350 with the **LibadX**, the string "**`im:<ip-addr>`**" must be passed to **`Open()`**. Here **`<ip-addr>`** must be replaced by the relating IP address. The string "**`im:192.168.1.1`**", for example, opens the iM device with the IP address 192.168.1.1. When opening the driver, no difference is made between different iM device types.

DAQ syst.	Analog	Channel number	Meas. range	Range	Digital
iM-AD25a	16 inputs	1..16	±10.24V ±5.12V	1 0	1: output (bit 0..3)
iM-AD25	16 inputs	1..16	±5.12V	0	1: output (bit 0..3)
iM3250T	32 inputs	17..48	±5.12V	0	-
iM3250	32 inputs	AIn 1..16: 1..16 (with 1 BPL) 17..32 (with 2 BPL) AIn 17..32: 33..48	±5.00V	0	-

Please note that MAL measuring amplifiers installed in the iM3250T might change the measuring range of the corresponding channels.

3.2.3 LAN-AD16fx / LAN-AD16f

Open the LAN-AD16f(x) (also: AMS42/84-LAN16f, AMS42/84-LAN16fx) with the **LIBAD4** by passing the string "**`lanbase:<ip-addr>`**" to **`Open()`**. Here **`<ip-addr>`** must be replaced by the relating IP address. The string "**`lanbase:192.168.1.1`**", for example, opens the LAN device with the IP address 192.168.1.1.

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
LAN-AD16fx	16 inputs 2 outputs	1..16 1 .. 2	0 (±1.024V) 1 (±2.048V) 2 (±5.120V) 3 (±10.240V)	0 (±10.24V)	2 ports (16 bit each)	1: port A 2: port B
LAN-AD16f	16 inputs 2 outputs	1..16 1 .. 2	0 (±1.024V) 1 (±2.048V) 2 (±5.120V) 3 (±10.240V)	0 (±10.24V)	2 ports (16 bit each)	1: input (bit 0..15) 2: output (bit 0..15)

The 16 analog inputs of a LAN-AD16f(x) are addressed via the channel numbers 1-16. The 2 analog outputs are reached via channel numbers 1 and 2.

The LAN-AD16f(x) provides two 16-bit digital ports. The digital ports of the LAN-AD16fx are bidirectional (see "**`DigitalDirection`**", p. 55) and are configured in groups of 8, the lines of the LAN-AD16f, in contrast, are hard-wired. After boot-up, 16 lines of the first port (DIO1, channel number: 1) are set to input, the 16 lines of the second port (DIO2, channel number: 1) to output.

The counters of the LAN-AD16f(x) can only be programmed with the LIBAD4 SDK.

3.2.4 PCIe-BASE / PCI-BASEII/300/1000 / PCI-PIO

To open the PCIe-BASE, PCI-BASEII, PCI-BASE300, PCI-BASE1000 or PCI-PIO with the **LibadX**, the string "**`pcibase`**" (or "**`pci300`**") must be passed to **`Open()`**. When opening the driver, no difference is made between different versions of the PCI(e) data acquisition card.

To distinguish between several cards, the card number is explicitly used (1. card with "**`pcibase:0`**", 2. card with "**`pcibase:1`**", etc.).

A DAQ card is also directly accessible via its serial number. The card with the serial number 157 can be addressed with "**`pcibase:@157`**", for example.

3.2.4.1 Digital Ports and Counters

The PCIe-BASE / PCI-BASEII/300/1000 / PCI-PIO features two 16-bit digital ports.

The digital lines of the PCIe-BASE, PCI-BASEII und PCI-PIO are bidirectional and are configured in groups of 8. Their direction can be changed in groups of 8. After boot-up, the default direction of the first port is input and output of the second.

The ports of the PCI-BASE300/1000 are hard-wired. The first port is set to input, the second port to output.

In addition, some versions (PCIe-BASE, PCI-BASEII, PCI-PIO) are provided with three 32-bit counters.

The counters of the PCIe-BASE, PCI-BASEII, and PCI-PIO can only be programmed with the LIBAD4 SDK.

3.2.4.2 MAD12/12a/12b/12f/16/16a/16b/16f

The first analog input channel of a MAD12/12a/12b/12f/16/16a/16b/16f starts with 1. If there is a second analog module on the PCI(e) multi-function card (not: PCI-PIO), the first input of the second module is addressed by the number 257 (0x100+1).

Of course, one input module can be operated in differential (not MAD12b/16b) and the other in single-ended mode, thus providing for 24 input channels.

The measuring ranges of the input channels depend on the module. If different analog input modules are plugged on the PCI(e) data acquisition card (not PCI-PIO), the measuring ranges of the channel 1..16 may differ from the measuring ranges of the channels 17..32.

Module	Analog	Channel number	Meas. range	Range
MAD12, MAD16	16 inputs (single-ended) 8 inputs (differential)	1..16 (se) 17..24 (diff)	±1.024V ±2.048V ±5.120V ±10.240V 0.06V..5.06V	0 1 2 3 4
MAD12a, MAD12f, MAD16a, MAD16f	16 inputs (single-ended) 8 inputs (differential)	1..16 (se) 17..24 (diff)	±1.024V ±2.048V ±5.120V ±10.240V	0 1 2 3
MAD12b, MAD16b	16 inputs (single-ended)	1..16	±1.024V ±2.048V ±5.120V ±10.240V	0 1 2 3

3.2.4.3 MADDA16/16n

The first analog input or output channel of a MADDA16/16n starts with 1. If there is a second analog module on the PCI(e) multi-function card (not: PCI-PIO), the first input of the second module is addressed by the number 257 (0x100+1).

The measuring ranges of the input channels depend on the module. If different analog input modules are plugged on the PCI(e) data acquisition card (not PCI-PIO), the measuring ranges of the channel 1..16 may differ from the measuring ranges of the channels 17..32.

Module	Analog	Channel number	Meas. range	Output range
MADDA16, MADDA16n	16 inputs 2 outputs	1..16 1..2	0 (±1.024V) 1 (±2.048V) 2 (±5.120V) 3 (±10.240V)	0 (±10.24V)

3.2.4.4 MDA12/12-4/16/16-2i/16-4i/16-8i

Corresponding to the MAD12/12a/12b/12f/16/16a/16b/16f, the channels of a second analog output module are accessible from number 257 (0x100+1) on.

Module	Analog	Channel number	Output range	Range
MDA12, MDA16	2 outputs	1..2	±10.24V ±5.12V	0 1
MDA12-4	4 outputs	1..4	±10.24V ±5.12V	0 1
MDA16-2i	2 outputs	1..2	±10.24V	0
MDA16-4i	4 outputs	1..4	±10.24V	0
MDA16-8i	8 outputs	1..8	±10.24V	0

The output ranges of the output modules MDA12/MDA12-4 and MDA16 are configured on the hardware. The user must ensure that the passed measuring range complies with the configuration set on the module.

3.2.5 meM-AD /-ADDA /-ADf / -ADfo

Open the meM-AD/-ADDA/-ADf/-ADfo with the **LibadX** by passing the string "**memadusb**" (meM-AD), "**memaddausb**" (meM-ADDA), "**memadfusb**" (meM-ADf) or "**memadfpusb**" (meM-ADfo) to **Open()**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1st device with "**memadusb:0**", 2nd device with "**memadusb:1**", etc.). The device order results from the order of connecting.

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected meM-ADDA devices is removed, the remaining meM-ADDA devices are addressed with "**memaddausb:0**" and "**memaddausb:2**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The device with the serial number 157 can be addressed with "**memadfpusb:@157**", for example.

DAQ system	Analog	Channel number	Input/Output range	Range	Digital	Channel number
meM-AD	16 inputs	1..16	±5.12V	0	-	-
meM-ADDA, meM-ADf	16 inputs 1 output	1..16 1	±5.12V	0	2 ports (4 bit each)	1: input (bit 0..3) 2: output (bit 0..3)
meM-ADfo	16 inputs 1 output	1..16 1	±5.12V	0	2 ports (8 bit each)	1: input (bit 0..7) 2: output (bit 0..7)

The 16 analog inputs of a meM-AD/-ADDA/-ADf/-ADfo are addressed via the channel numbers 1-16. The analog output is reached via channel number 1.

The direction of the digital ports is hard-wired. The 4 (meM-ADfo: 8) lines of the first port (DIO1, channel number: 1) are set to input, the 4 (meM-ADfo: 8) lines of the second port (DIO2, channel number: 2) to output.

3.2.6 meM-PIO / meM-PIO-OEM

Open the meM-PIO/meM-PIO-OEM with the **LibadX** by passing the string "**`mempiousb`**" to **`Open()`**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1st device with "**`mempiousb:0`**", 2nd device with "**`mempiousb:1`**", etc.). The device order results from the order of connecting.

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected meM-PIO devices is removed, the remaining meM-PIO devices are addressed with "**`mempiousb:0`**" and "**`mempiousb:2`**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The device with the serial number 157 can be addressed with "**`mempiousb:@157`**", for example.

DAQ system	Digital	Channel number
meM-PIO, meM-PIO-OEM	3 ports (8 bit each)	1..3 (bit 0..7)

The line direction is set for each port separately in groups of eight (see "**`DigitalDirection`**", S. 55). The first port (DIO1) has channel number 1, the second port (DIO2) channel number 2 and the third port (DIO3) channel number 3.

3.2.7 USB-AD

Open the USB-AD with the **LibadX** by passing the string "**usb-ad**" to **Open()**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1. device with "**usb-ad:0**", 2. device with "**usb-ad:1**", etc.). The device order results from the order of connecting.:

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected USB-AD devices is removed, the remaining USB-AD devices are addressed with "**usb-ad:0**" and "**usb-ad:2**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The device with the serial number 157 can be addressed with "**usb-ad:@157**", for example.

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
USB-AD	16 inputs 1 output	1..16 1	0 (±5.12V)	0 (±5.12V)	2 ports (4 bit each)	1: input (bit 0..3) 2: output (bit 0..3)

The 16 analog inputs of a USB-AD are addressed via the channel numbers 1-16. The analog output is reached via channel number 1.

For compatibility reasons, the measuring range 33 can be used for analog inputs and the output range 1 for the analog output.

The direction of the digital ports is hard-wired. The 4 lines of the first port (DIO1, channel number: 1) are set to input, the 4 lines of the second port (DIO2, channel number: 2) to output.

Example:

VB

```
If LIBADX1.Open("usb-ad:0") Then

    Dim tmp As Integer
    tmp = LIBADX1.DigitalIn(1)

    Dim bool As Boolean
    ' reads the state of the first line of port 1
    bool = LIBADX1.DigitalInLine(1, 0)

    ' delete all lines
    LIBADX1.DigitalOut(2) = 0
    ' line 2 high
    LIBADX1.DigitalOutLine(2, 1) = True

    Dim val As Double
    ' reads the value of Analog In 1 with measuring range 0
    val = LIBADX1.AnalogIn(1, 0)

    ' set Analog Out 1 to 4.5 Volt
    LIBADX1.AnalogOut(1, 0) = 4.5

    LIBADX1.Close
  End If
```

3.2.8 USB-AD14f / USB-AD12f

Open the USB-AD14f / USB-AD12f with the **LibadX** by passing the string "**usbad14f**" or "**usbad12f**" to **Open()**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1st USB-AD14f with "**usbad14f:0**", 2nd USB-AD14f with "**usbad14f:1**", etc.). The device order results from the order of connecting.

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected USB-AD14f devices is removed, the remaining USB-AD14f devices are addressed with "**usbad14f:0**" and "**usbad14f:2**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The USB-AD14f with the serial number 157 can be addressed with "**usbad14f:@157**", for example.

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
USB-AD14f	16 inputs 1 output	1..16 1	0 (±10.24V)	0 (±5.12V)	2 ports (8 bit each)	1: input (bit 0..7) 2: output (bit 0..7)
USB-AD12f	16 inputs 1 output	1..16 1	0 (±10.24V)	0 (±5.12V)	2 ports (4 bit each)	1: input (bit 0..3) 2: output (bit 0..3)

The 16 analog inputs of a USB-AD14f / USB-AD12f are addressed via the channel numbers 1-16. The analog output is reached via channel number 1.

The direction of the digital ports is hard-wired. The 8 (USB-AD14f) or 4 (USB-AD12f) lines of the first port (DIO1) are set to input, the 8 (USB-AD14f) or 4 (USB-AD12f) lines of the second port (DIO2) to output..

14f12fThe first digital input (bit 1) can be used as a 16-bit counter. It is treated like an analog channel by the **LibadX**. In this case, the channel number of the counter must be extended by the counter channel type (**hex 0x08000000**) in the analog functions **AnalogIn** (see p. 52), **AnalogOut** (see p. 52) and **ScanAnalogIn**

(see p. 46) so that the counter has channel number **0x08000001** in hexadecimal notation. The range parameter to be passed is always '0'. Passing the value 0 with the command **AnalogOut** resets the counter.

3.2.9 USB-AD16f

Open the USB-AD16f (also: AMS42-USB, AMS84-USB) with the **LibadX** by passing the string "**usbbase**" to **Open()**. To distinguish between several USB-AD16f data acquisition systems, the device number is explicitly used (1. device with "**usbbase:0**", 2. device with "**usbbase:1**", etc.). The device order results from the order of connecting.AD16f:O

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected USB-AD16f devices is removed, the remaining USB-AD16f devices are addressed with "**usbbase:0**" and "**usbbase:2**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The device with the serial number 157 can be addressed with "**usbbase:@157**", for example.

DAQ system	Analog	Channel number	Measuring range	Output range	Digital	Direction
USB-AD16f	16 inputs 2 outputs	1..16 1 .. 2	0 (±1.024V) 1 (±2.048V) 2 (±5.120V) 3 (±10.240V)	0 (±10.24V)	2 ports (4 bit each)	1: input (bit 0..3) 2: output (bit 0..3)

The 16 analog inputs of a USB-AD16f are addressed via the channel numbers 1-16. The 2 analog outputs are reached via channel number 1 and 2.

The direction of the ports is hard-wired. The 4 lines of the first port (DIO1, channel number: 1) are set to input, the 4 lines of the second port (DIO2, channel number: 1) to output.

The USB-AD16f additionally features a counter input, which is treated like an analog channel by the **LibadX**. In this case, the channel number of the counter must be extended by the counter channel type (**hex 0x08000000**) in the analog

functions **AnalogIn** (see p. 52), **AnalogOut** (see p. 52) and **ScanAnalogIn** (see p. 46) so that the counter has channel number **0x08000001** in hexadecimal notation. The range parameter to be passed is always '0'. Passing the value 0 with the command **AnalogOut** resets the counter.

3.2.10 USB-OI16

Open the USB-OI16 with the **LibadX** by passing the string "**usb-oi16**" to **Open()**. To distinguish between several USB devices, the device number is explicitly used (e.g. 1st device with "**usb-oi16:0**", 2nd device with "**usb-oi16:1**", etc.). The device order results from the order of connecting.

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected USB-OI16 devices is removed, the remaining USB-OI16 devices are addressed with "**usb-oi16:0**" and "**usb-oi16:2**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The device with the serial number 157 can be addressed with "**usb-oi16:@157**", for example.

DAQ system	Digital	Channel number
USB-OI16	2 ports (16 bit each)	1: input 2: output

The USB-OI16 provides two 16-bit digital ports. The direction of the digital ports is hard-wired. The 16 lines of the first port (DIO1) are set to input, the 16 lines of the second port (DIO2) to output.

The counters of the USB-OI16 can only be programmed with the LIBAD4 SDK.

3.2.11 USB-PIO / USB-PIO-OEM

Open the USB-PIO / USB-PIO-OEM with the **LibadX** by passing the string "**`usb-pio`**" to **`Open()`**. To distinguish between several USB data acquisition systems, the device number is explicitly used (e.g. 1. device with "**`usb-pio:0`**", 2. device with "**`usb-pio:1`**", etc.). The device order results from the order of connecting.

As USB data acquisition systems can be plugged and unplugged during operation, it may happen that the device numbers are not assigned consecutively. For example, if the second of three connected USB-PIO / USB-PIO-OEM devices is removed, the remaining USB-PIO / USB-PIO-OEM devices are addressed with "**`usb-pio:0`**" and "**`usb-pio:2`**".

To avoid managing the order of connecting, a device is also accessible via its serial number. The device with the serial number 157 can be addressed with "**`usb-pio:@157`**", for example.

DAQ system	Digital	Channel number
USB-PIO, USB-PIO-OEM	3 ports (8 bit each)	1..3 (bit 0..7)

The line direction is set for each port separately in groups of eight (see "**`DigitalDirection`**", S. 55). The first port (DIO1) has channel number 1, the second port (DIO2) channel number 2 and the third port (DIO3) channel number 3.

Example:

VB

```
If LIBADX1.Open("usb-pio:0") Then
  LIBADX1.DigitalDirection(1) = &H0   ' all output
  LIBADX1.DigitalDirection(2) = &HFF  ' all input
  LIBADX1.DigitalDirection(3) = &H0   ' all output

  Dim tmp As Integer
  ' reads the state of all lines of port 2
  tmp = LIBADX1.DigitalIn(2)

  Dim bool As Boolean
  ' reads the state of the first line of port 2
  bool = LIBADX1.DigitalInLine(2, 0)

  ' delete all lines
  LIBADX1.DigitalOut(1) = 0
  ' line 8 of port 1 high
  LIBADX1.DigitalOutLine(1, 7) = True

  ' set port 3 to &H15 = line 1, 3, 5 high
  LIBADX1.DigitalOut(3) = &H15

  LIBADX1.Close
End If
```

4 Interfaces and Functions

4.1 The LibadX Interface

The **LibadX** interface is directly imported by the LibadX ActiveX Control. It provides the connection to the measurement data server.

4.1.1 Overview

Function	Description
`Open`	opens the connection to a data acquisition system
`Close`	closes the connection to a data acquisition system
`GetVersion`	returns the version number of the **`LIBAD4.dll`**
`LastError`	returns the last error code
`LastErrorString`	returns a description of the last error
`ScanPrepare`	prepares a scan
`ScanAnalogIn`	adds an analog input to the scan list
`ScanDigitalIn`	adds a digital input to the scan list
`Scan`	starts a prepared scan
`ScanSave`	saves a performed scan
`FileOpen`	creates a file object used to get access to stored measuring files
`FileCreatePrepare`	prepares the creation of a scan file
`FileCreateAnalogIn`	adds an analog input to the channel list
`FileCreateDigital`	adds a digital input to the channel list
`FileCreate`	creates a prepared scan file
`AnalogIn`	returns the current value of an analog input
`AnalogOut`	returns the current value of an analog output
`DigitalIn`	returns the current value of a digital input channel
`DigitalOut`	returns the current value of a digital output channel

DigitalInLine	returns the current value of a digital input line
DigitalOutLine	returns the current value of a digital output line
DigitalDirection	set/returns the direction of a digital channel
Sample	reads the value of a sample in a scan
AboutBox	displays the AboutBox of **LibadX**

4.1.2 Open

C++	`VARIANT_BOOL Open (_bstr_t path)`

BASIC	`Function Open (path As String) As Boolean`

Delphi	`function Open (const path: WideString): WordBool`

The **Open()** function provides a connection to the data acquisition system by passing the name of the data acquisition system. The passed string is not case-sensitive, i.e. "**pcibase**" and "**PCIBASE**" both open the PCIe-BASE / PCI-BASEII/300/1000 / PCI-PIO.

If the connection to the data acquisition has been opened, **Open** returns the value **TRUE**, and **FALSE** in case of an error. For a detailed description of the **Open()** command see chapter "Connect to the Data Acquisition System", p. 26.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.3 Close

C++	`HRESULT Close ()`
BASIC	`Sub Close ()`
Delphi	`procedure Close`

The **`Close()`** function shuts the connection to the data acquisition system.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.4 GetVersion

C++	`long GetVersion ()`
BASIC	`Function GetVersion () As Long`
Delphi	`function GetVersion: Integer`

The **`GetVersion()`** function returns the version of the LIBAD4.dll used by the **LibadX**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.5 LastError

C++	`long LastError ()`
BASIC	`Function LastError () As Long`
Delphi	`function LastError: Integer`

Returns the number of the last error. If no errors occurred, the function is **`0`**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.6 LastErrorString

C++	`_bstr_t LastErrorString ()`
BASIC	`Function LastErrorString () As String`
Delphi	`function LastErrorString: WideString`

Edits a description of the last error. If no errors occurred, the function returns "".

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.7 ScanPrepare

C++	`HRESULT ScanPrepare (float sample_rate, long samples)`

BASIC	`Sub ScanPrepare (sample_rate As Single, samples As Long)`

Delphi	`procedure ScanPrepare (sample_rate: Single;` `samples: Integer)`

Before starting a scan, **`ScanPrepare()`** must be called first. It prepares the **LibadX** for a scan and sets the sample rate to **`sample_rate`** and the number of values to be stored to **`samples`**.

To add a channel to the scan channel list, call **`ScanAnalogIn()`** or **`ScanDigitalIn()`**. The scan is started by calling the **`Scan()`** command.

The following Visual Basic® sample code demonstrates the procedure:

VB

```
' 1000 measuring values, 100Hz (0.01 sec.)
LIBADX1.ScanPrepare 0.01, 1000

' Save channel 1 & 2
LIBADX1.ScanAnalogIn 1, 0
LIBADX1.ScanAnalogIn 2, 0

' Save counter 1
LIBADX1.ScanDigitalIn &h08000001

' Save digital port 1
LIBADX1.ScanDigitalIn 1

' Start scan
LIBADX1.Scan

' Save scan
LIBADX1.ScanSave "scan.lfx"
```

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.8 ScanAnalogIn

C++	`HRESULT ScanAnalogIn (long index, long range)`
BASIC	`Sub ScanAnalogIn (index as Long, range as Long)`
Delphi	`procedure ScanAnalogIn (index, range: Integer)`

With **`ScanAnalogIn()`** the analog channel or counter with the number **`index`** and the range **`range`** is added to the scan channel list . The function throws an exception if the scan has not previously been prepared with **`ScanPrepare()`** (see p. 44).

- **Due to restrictions of most of the data acquisition cards, it is essential to add the input channels in ascending order to the channel list! If both analog inputs and counter or digital inputs are sampled, first the analog channels, then the counters and finally the digital channels must be specified!**
- **If using counters, the index number has to be extended by the counter channel type (hex 0x08000000). For example, the index number 0x08000001 in hexadecimal notation is assigned to counter 1.**

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.9 ScanDigitalIn

C++	`HRESULT ScanDigitalIn (long index)`
BASIC	`Sub ScanDigitalIn (index as Long)`
Delphi	`procedure ScanDigitalIn (index: Integer)`

With **`ScanDigitalIn()`** digital channel with the number **`index`** is added to the scan channel list. The function throws an exception if the scan has not previously been prepared with **`ScanPrepare()`** (see p. 44).

Due to restrictions of most of the data acquisition cards, it is essential to add the input channels in ascending order to the channel list! If both analog inputs and counter or digital inputs are sampled, first the analog channels, then the counters and finally the digital channels must be specified!

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.10 Scan

C++	`VARIANT_BOOL Scan ();`
BASIC	`Function Scan () As Boolean`
Delphi	`function Scan : WordBool`

With **`Scan()`** a scan prepared with **`ScanPrepare()`**, **`ScanAnalogIn()`** and **`ScanDigitalIn()`** is started. The execution is returned to the program not until the scan is finished.

The function throws an exception if the scan has not previously been prepared with **`ScanPrepare()`** (see p. 44).

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.11 ScanSave

C++	`VARIANT_BOOL ScanSave (_bstr_t path);`
BASIC	`Function ScanSave (path As String) As Boolean`
Delphi	`function ScanSave (const path: WideString): WordBool`

With **`ScanSave()`** a scan carried out with the **`Scan()`** function is saved.

The function throws an exception if a scan has not previously been performed with **`Scan()`**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.12 FileOpen

C++	`INvxFilePtr FileOpen (_bstr_t path)`
BASIC	`Function FileOpen (path As String) As INvxFile`
Delphi	`function FileOpen (const path: WideString): INvxFile`

Opens the specified measurement file. If the file does not exist or cannot be opened, the function throws an exception.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.13 FileCreatePrepare

C++	`HRESULT FileCreatePrepare (long samples)`

BASIC	`Sub FileCreatePrepare (samples As Long)`

Delphi	`procedure FileCreatePrepare (samples: Integer)`

The creation of a measurement file is the same as of a scan. First the **`FileCreatePrepare()`** function containing the number of values to be stored has to be called.

To add a channel to the file channel list, call **`FileCreateAnalogIn()`** or **`FileCreateDigital()`**. The file is then created by calling **`FileCreate()`**.

The following Visual Basic® sample code demonstrates the procedure:

VB

```
' 1000 measuring values
LIBADX1.FileCreatePrepare 1000

' 2 analog channels
LIBADX1.FileCreateAnalogIn
LIBADX1.FileCreateAnalogIn

' 1 counter
Const AD_CHA_TYPE_COUNTER as Integer = &h08000000
LIBADX1.FileCreateDigital AD_CHA_TYPE_COUNTER

' 1 digital channel with 16 lines
LIBADX1.FileCreateDigital 16

' create file
LIBADX1.FileCreate "scan.lfx"
```

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.14 FileCreateAnalogIn

C++	`long FileCreateAnalogIn ()`
BASIC	`Function FileCreateAnalogIn () As Long`
Delphi	`function FileCreateAnalogIn: Integer;`

With **`FileCreateAnalogIn()`** an analog channel or counter is added to the channel list of a file to be created. The return value is the channel index in the file. The function throws an exception if a measurement file has not previously been prepared with **`FileCreatePrepare()`** (see p. 48).

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.15 FileCreateDigital

C++	`long FileCreateDigital (long lines)`
BASIC	`Function FileCreateDigital (lines As Long) As Long`
Delphi	`function FileCreateDigital(lines: Integer): Integer;`

With **`FileCreateDigital()`** a digital channel is added to the channel list of a file to be created.

Concerning digital channels, **`lines`** is the number of lines to be stored and must not exceed 32. The return value is the channel index in the file.

Before writing data to the file, the signal parameters (see chapter "The INvxSignal", p. 61) "yMax" (see p. 68) and "yMin" (see p. 68) must be passed first. Otherwise the data might not be written correctly. The y-using (see chapter "ySetUsing", p. 71) should also be adjusted accordingly.

The function throws an exception if a measurement file has not previously been prepared with **`FileCreatePrepare()`** (see p. 48).

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.16 FileCreate

C++	`INvxFilePtr FileCreate (_bstr_t path)`
BASIC	`Function FileCreate (path As String) As INvxFile`
Delphi	`function FileCreate (const path: WideString): INvxFile`

`FileCreate()` creates a measurement file prepared with **`FileCreatePrepare()`**, **`FileCreateAnalogIn()`** and **`FileCreateDigital()`**.

The function throws an exception if a measurement file has not previously been prepared with **`FileCreatePrepare()`** (see p. 48) or if no channel has been added to the channel list.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.17 AnalogIn

C++	`__declspec(property(get=GetAnalogIn))` `  float AnalogIn[][]`
BASIC	`Property AnalogIn (index As Long, range as Long) As Single`
Delphi	`property AnalogIn [index, range: Integer]: Single readonly`

Returns the currently measured value of the analog input with the number **`index`** within the measuring range **`range`**. The value can only be read.

If using counters, the index number has to be extended by the counter channel type (hex 0x08000000). For example, the index number 0x08000001 in hexadecimal notation is assigned to counter 1.

The function throws an exception if the connection to a data acquisition system has not previously been established with **`Open()`**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.18 AnalogOut

C++	`__declspec(property(get=GetAnalogOut,put=PutAnalogOut))` `  float AnalogOut[][]`
BASIC	`Property AnalogOut (index As Long, range as Long) As Single`
Delphi	`property AnalogOut [index, range: Integer]: Single`

Sets or returns the current value of the output channel with the number **`index`** within the output range **`range`**.

If using counters, the index number has to be extended by the counter channel type (hex 0x08000000). For example, the index number 0x08000001 in hexadecimal notation is assigned to counter 1.

The function throws an exception if the connection to a data acquisition system has not previously been established with **`Open()`**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.19 DigitalIn

C++	`__declspec(property(get=GetDigitalIn))` `  long DigitalIn[]`
BASIC	`Property DigitalIn (index As Long) As Long`
Delphi	`property DigitalIn [index: Integer]: Integer readonly`

Returns the currently measured value of the digital input with the number **`index`**. The value of this property can only be read.

The function throws an exception if the connection to a data acquisition system has not previously been established with **`Open()`**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.20 DigitalOut

C++	`__declspec(property(get=GetDigitalOut,put=PutDigitalOut))` `  long DigitalOut[];`
BASIC	`Property DigitalOut (index As Long) As Long`
Delphi	`property DigitalOut [index: Integer]: Integer`

Sets or returns the current value of the digital output channel with the number **`index`**.

The function throws an exception if the connection to a data acquisition system has not previously been established with **`Open()`**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.21 DigitalInLine

C++	`__declspec(property(get=GetDigitalInLine))` `  VARIANT_BOOL DigitalInLine[][];`
BASIC	`Property DigitalInLine (index As Long, line As Long)` `  As Boolean`
Delphi	`property DigitalInLine [index, line: Integer]:` `  WordBool readonly`

Returns the currently measured value of the line number **`line`** of the digital input channel with the number **`index`**. The value of this property can only be read.

The function throws an exception if the connection to a data acquisition system has not previously been established with **`Open()`**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.22 DigitalOutLine

C++	`__declspec(property(get=GetDigitalOutLine, put=PutDigitalOutLine)) VARIANT_BOOL DigitalOutLine[][];`
BASIC	`Property DigitalOutLine (index As Long, line As Long) As Boolean`
Delphi	`property DigitalOutLine [index, line: Integer]: WordBool`

Sets or returns the current value of the line number **`line`** of the digital output channel with the number **`index`**.

The function throws an exception if the connection to a data acquisition system has not previously been established with **`Open()`**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.23 DigitalDirection

C++	`__declspec(property(get=GetDigitalDirection, put=PutDigitalDirection)) long DigitalDirection[];`
BASIC	`Property DigitalDirection (index As Long) As Long`
Delphi	`property DigitalDirection [index: Integer]: Integer`

Sets or returns the direction (input/output) of the digital channel with the number **`index`**. This property passes a bitmask describing the direction of the digital line. A high bit ("1") represents an input line, a low bit ("0") an output line. Bit #0 defines the direction of the first line of the digital port.

The function throws an exception if the connection to a data acquisition system has not previously been established with **`Open()`**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.24 Sample

C++	`__declspec(property(get=GetSample,put=PutSample))` `  float Sample[][]`
BASIC	`Property Sample (index As Long, pos As Long) As Single`
Delphi	`property Sample [index, pos: Integer]: Single`

Sets of returns the sample of the channel **`index`** at the position **`pos`** of the executed scan.

The function throws an exception if no scan has previously been run of if **`index`** or **`pos`** are not valid.

Due to single floating point use, high counter values of 32-bit counters get lost. Only values within the range of +/-16777216 are available.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.1.25 AboutBox

C++	`HRESULT AboutBox ()`
BASIC	`Sub AboutBox ()`
Delphi	`procedure AboutBox`

Displays the AboutBox of **LibadX**.

A list of all possible commands is provided in chapter "Overview", p. 41.

4.2 The INvxFile Interface

The **`INvxFile`** provides for the access to saved measurement data.

4.2.1 Overview

Function	Description
`Open`	opens a measurement file
`Create`	creates a new measurement file
`Close`	closes a measurement file
`SignalCount`	returns the number of signals in the measurement file
`Signal`	returns the interface of a signal in the measurement file

4.2.2 Open

C++	`HRESULT Open(_bstr_t fileName);`
BASIC	`Sub Open(fileName As String)`
Delphi	`procedure Open(const fileName: WideString);`

Opens the specified measurement file. If the file does not exist or cannot be opened, the function throws an exception.

A list of all possible commands is provided in chapter "Overview", p. 58.

4.2.3 Create

C++	`HRESULT Create(_bstr_t fileName,`
	`long signalCount,`
	`long sampleCount);`

BASIC	`Sub Create(fileName As String,`
	`signalCount As Long,`
	`sampleCount As Long)`

Delphi	`procedure Create(const fileName: WideString;`
	`signalCount: Integer;`
	`sampleCount: Integer);`

Creates a new measurement file. **`SignalCount`** signals are generated in the file. Each signal can save **`SampleCount`** measurement values.

A list of all possible commands is provided in chapter "Overview", p. 58.

4.2.4 Close

C++	`HRESULT Close();`
BASIC	`Sub Close()`
Delphi	`procedure Close;`

Closes a measurement file previously been opened with **`Open()`** or **`Create()`**.

A list of all possible commands is provided in chapter "Overview", p. 58.

4.2.5 SignalCount

C++	`long SignalCount();`
BASIC	`Function SignalCount() As Long`
Delphi	`function SignalCount: Integer;`

Returns the number of signals in a measurement file. The function throws an exception if no measurement file has previously been opened with **`Open()`** or created with **`Create()`**.

A list of all possible commands is provided in chapter "Overview", p. 58.

4.2.6 Signal

C++	`INvxSignalPtr Signal(long index);`
BASIC	`Function Signal(index As Long) As INvxSignal`
Delphi	`function Signal(index: Integer): INvxSignal;`

Returns a signal from the measurement file. The first signal in the file has the index number 1.

A list of all possible commands is provided in chapter "Overview", p. 58.

4.3 The INvxSignal Interface

The **`INvxSignal`** interface allows the access to a single signal of a measurement file.

4.3.1 Overview

Function	Description
`Name`	name of the signal
`GroupName`	group name of the signal
`Comment`	comment of the signal
`xStart`	starting time of the signal
`xEnd`	end time of the signal
`xDelta`	scan time of the signal
`xUnit`	unit of the x-axis
`xSetUsing`	sets the using of the x-axis
`xGetUsing`	returns the using of the x-axis
`yMin`	lower limit of the measuring range
`yMax`	upper limit of the measuring range
`yDefaultMin`	lower limit of the default range
`yDefaultMax`	upper limit of the default range
`yDelta`	resolution of the signal
`yUnit`	unit of the x-axis
`ySetUsing`	sets the using of the y-axis
`yGetUsing`	returns the using of the y-axis
`ScanStart`	date at the beginning of the scan
`SampleCount`	number of measuring values of the signal
`ScaleX`	scaling of the x-axis
`ScaleY`	scaling of the y-axis
`ResetDataPosition`	reset the internal signal counter
`GetNextScaled`	returns the next scaled pair of values

GetNextScaledDigital	returns the next scaled pair of values of a digital signal
Unscale	removes the scaling of the signal
NextSample	returns the next sample at the current position of the signal
NextDigitalSample	returns the next sample at the current position of the digital signal
GetSampleAt	returns a sample at a certain signal position
GetSampleAtOffset	returns a sample at a certain offset in the signal
IsAnalog	verifies if the signal contains analog measuring values
IsDigital	verifies if the signal contains digital or counter values

4.3.2 Name

C++	`__declspec(property(get=GetName,put=PutName))` `  _bstr_t Name;`

BASIC	`Property Name As String`

Delphi	`property Name: WideString` `  read Get_Name write Set_Name;`

Returns the name of the signal.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.3 GroupName

C++	`__declspec(property(get=GetName,put=PutName))` `  _bstr_t Name;`
BASIC	`Property GroupName As String`
Delphi	`property GroupName: WideString` `  read Get_GroupName write Set_GroupName;`

Returns the group name of the signal.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.4 Comment

C++	`__declspec(property(get=GetComment,put=PutComment))` `  _bstr_t Comment;`
BASIC	`Property Comment As String`
Delphi	`property Comment: WideString` `  read Get_Comment write Set_Comment;`

Returns the comment of the signal.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.5 xStart

C++	`__declspec(property(get=GetxStart,put=PutxStart))` `  double xStart;`

BASIC	`Property xStart As Double`

Delphi	`property xStart: Double` `  read Get_xStart write Set_xStart;`

Returns the starting time of the signal in seconds. This value is usually 0.0s. Only for scans using a trigger a negative value is returned indicating the length of the prehistory.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.6 xEnd

C++	`__declspec(property(get=GetxEnd,put=PutxEnd))` `  double xEnd;`

BASIC	`Property xEnd As Double`

Delphi	`property xEnd: Double` `  read Get_xEnd write Set_xEnd;`

Returns the end time of the signal. Please note that the total time of the signal can be different from the end time. The total time is **`xEnd-xStart`**.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.7 xDelta

C++	`__declspec(property(get=GetxDelta,put=PutxDelta))` `  double xDelta;`
BASIC	`Property xDelta As Double`
Delphi	`property xDelta: Double` `  read Get_xDelta write Set_xDelta;`

Returns the scan time of the signal in seconds.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.8 xUnit

C++	`__declspec(property(get=GetxUnit,put=PutxUnit))` `  _bstr_t xUnit;`
BASIC	`Property xUnit As String`
Delphi	`property xUnit: WideString` `  read Get_xUnit write Set_xUnit;`

Returns the unit of the x-axis.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.9 xSetUsing

C++	`HRESULT xSetUsing(long format,`
	`long width,`
	`long frac,`
	`long opt);`

BASIC	`Sub xSetUsing(format As Long,`
	`width As Long,`
	`frac As Long,`
	`opt As Long)`

Delphi	`procedure xSetUsing(format: Integer;`
	`width: Integer;`
	`frac: Integer;`
	`opt: Integer);`

Sets the using for the values of the x-axis used for the signal. **`format`** defines the output format, **`width`** the number of total characters of a value and **`frac`** the number of digits after the decimal place. The argument **`opt`** is only used for the scientific format specifying the decimal power used as base (see following table).

The following values can be passed for **`format`**, all others lead to the error code **`E_INVALIDARG`**:

Value	Description	Example: 17336.78
0	uses integer values	17336
3	value is written as a decimal value with **frac** digits after the decimal place	17336.780
4	exponential notation **E+xxx**	1.734E+004
5	scientific format: The representation of values is optimized by automatically using metric units for the decimal power: p (10^{-12}), n (10^{-9}), µ (10^{-6}), m (10^{-3}), k (10^{3}), M (10^{6}), G (10^{9})	17.337k
6	Fixed scientific notation: The decimal power is preset by the parameter **opt**. The following values can be chosen for **opt**: 0: p (10^{-12}) 3: m (10^{-3}) 0: p (10^{-12}) 1: n (10^{-9}) 4: (10^{0}) 1: n (10^{-9}) 2: µ (10^{-6}) 5: k (10^{3}) 2: µ (10^{-6})	0.017M 3: m (10^{-3}) 4: (10^{0}) 5: k (10^{3})

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.10 xGetUsing

C++
```
HRESULT xGetUsing(long *format,
                  long *width,
                  long *frac,
                  long *opt);
```

BASIC
```
Sub xGetUsing(format As Long,
              width As Long,
              frac As Long,
              opt As Long)
```

Delphi
```
procedure xGetUsing(var format: Integer;
                    var width: Integer;
                    var frac: Integer;
                    var opt: Integer);
```

Returns the settings used for the values at the x-axis of the signal. The meaning of the individual parameters is described in chapter "xSetUsing", p. 66.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.11 yMin

C++	`__declspec(property(get=GetyMin,put=PutyMin))` `  double yMin;`

BASIC	`Property yMin As Double`

Delphi	`property yMin: Double` `  read Get_yMin write Set_yMin;`

Returns the lower limit of the measuring range the signal has been recorded with.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.12 yMax

C++	`__declspec(property(get=GetyMax,put=PutyMax))` `  double yMax;`

BASIC	`Property yMax As Double`

Delphi	`property yMax: Double` `  read Get_yMax write Set_yMax;`

Returns the upper limit of the measuring range the signal has been recorded with.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.13 yDefaultMin

C++	`__declspec(property(get=GetyDefaultMin,put=PutyDefaultMin))` `  double yDefaultMin;`
BASIC	`Property yDefaultMin As Double`
Delphi	`property yDefaultMin: Double` `  read Get_yDefaultMin write Set_yDefaultMin;`

Returns the lower limit of the default range setting for displaying the signal.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.14 yDefaultMax

C++	`__declspec(property(get=GetyDefaultMax,put=PutyDefaultMax))` `  double yDefaultMax;`
BASIC	`Property yDefaultMax As Double`
Delphi	`property yDefaultMax: Double` `  read Get_yDefaultMax write Set_yDefaultMax;`

Returns the upper limit of the default range setting for displaying the signal.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.15 yDelta

C++	`__declspec(property(get=GetyDelta,put=PutyDelta))` `  double yDelta;`

BASIC	`Property yDelta As Double`

Delphi	`property yDelta: Double` `  read Get_yDelta write Set_yDelta;`

Returns the resolution of the signal values.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.16 yUnit

C++	`__declspec(property(get=GetyUnit,put=PutyUnit))` `  _bstr_t yUnit;`

BASIC	`Property yUnit As String`

Delphi	`property yUnit: WideString` `  read Get_yUnit write Set_yUnit;`

Returns the unit of the y-axis.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.17 ySetUsing

C++	`HRESULT ySetUsing(long format,`
	`long width,`
	`long frac,`
	`long opt);`

BASIC	`Sub ySetUsing(format As Long,`
	`width As Long,`
	`frac As Long,`
	`opt As Long)`

Delphi	`procedure ySetUsing(format: Integer;`
	`width: Integer;`
	`frac: Integer;`
	`opt: Integer);`

Sets the using for the values of the y-axis used for the signal. **`format`** defines the output format, **`width`** the number of total characters of a value and **`frac`** the number of digits after the decimal place. The argument **`opt`** is only used for the scientific format specifying the decimal power used as base (see following table).

The following values can be passed for **`format`**, all others lead to the error code **`E_INVALIDARG`**:

Value	Description	Example: 17336.78
0	uses integer values	17336
3	value is written as a decimal value with **frac** digits after the decimal place	17336.780
4	exponential notation **E+xxx**	1.734E+004
5	scientific format: The representation of values is optimized by automatically using metric units for the decimal power: p (10^{-12}), n (10^{-9}), µ (10^{-6}), m (10^{-3}), k (10^{3}), M (10^{6}), G (10^{9})	17.337k
6	Fixed scientific notation: The decimal power is preset by the parameter **opt**. The following values can be chosen for **opt**: 0: p (10^{-12}) 3: m (10^{-3}) 0: p (10^{-12}) 1: n (10^{-9}) 4: (10^{0}) 1: n (10^{-9}) 2: µ (10^{-6}) 5: k (10^{3}) 2: µ (10^{-6})	0.017M 3: m (10^{-3}) 4: (10^{0}) 5: k (10^{3})

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.18 yGetUsing

C++
```
HRESULT yGetUsing(long *format,
                  long *width,
                  long *frac,
                  long *opt);
```

BASIC
```
Sub yGetUsing(format As Long,
              width As Long,
              frac As Long,
              opt As Long)
```

Delphi
```
procedure yGetUsing(var format: Integer;
                    var width: Integer;
                    var frac: Integer;
                    var opt: Integer);
```

Returns the settings used for the values at the y-axis of the signal. The meaning of the individual parameters is described in chapter "ySetUsing", S. 71.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.19 ScanStart

C++	`__declspec(property(get=GetScanStart,put=PutScanStart))` `  double ScanStart;`
BASIC	`Property ScanStart As Double`
Delphi	`property ScanStart: Double` `  read Get_ScanStart write Set_ScanStart;`

Returns the data of the scan start (i.e. time which the first signal sample has been recorded at). The date is passed in seconds since January 1st, 1970.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.20 SampleCount

C++	`long SampleCount();`
BASIC	`Function SampleCount() As Long`
Delphi	`function SampleCount: Integer;`

Returns the number of signal samples .

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.21 ScaleX

C++	`HRESULT ScaleX(double xStart,`
	`double xEnd,`
	`long px);`

BASIC	`Sub ScaleX(xStart As Double,`
	`xEnd As Double,`
	`px As Long)`

Delphi	`procedure ScaleX(xStart: Double;`
	`xEnd: Double;`
	`px: Integer);`

Scales the x-range of the signal in such a way, that the samples between **`xStart`** und **`xEnd`** are passed by **`GetNextScaled`**. The function **`GetNextScaled`** must be called **`px`**-times, to get the complete graph.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.22 ScaleY

C++	`HRESULT ScaleY(double yMin,`
	`double yMax,`
	`long py);`

BASIC	`Sub ScaleY(yMin As Double,`
	`yMax As Double,`
	`py As Long)`

Delphi	`procedure ScaleY(yMin: Double;`
	`yMax: Double;`
	`py: Integer);`

Scales the y-range of the signal in such a way, that the samples between **`yMin`** and **`yMax`** are displayed to the integer values **`0`** to **`py`**.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.23 ResetDataPosition

C++	`HRESULT ResetDataPosition();`
BASIC	`Sub ResetDataPosition()`
Delphi	`procedure ResetDataPosition;`

Resets the internal signal counter so that the next call of **`GetNextScaled`** will return the first minimum/maximum pair (or **`NextSample`** will return the first signal sample).

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.24 GetNextScaled

C++	`VARIANT_BOOL GetNextScaled(long *min, long *max);`
BASIC	`Function GetNextScaled(min As Long, max As Long) As Boolean`
Delphi	`function GetNextScaled(out min: Integer;` `  out max: Integer): WordBool;`

Returns the next minimum/maximum pair of the signal according to the scaling defined by **`ScaleX()`** and **`ScaleY()`**.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.25 GetNextScaledDigital

C++	`VARIANT_BOOL GetNextScaledDigital(long *min, long *max);`
BASIC	`Function GetNextScaledDigital(min As Long, max As Long) As Boolean`
Delphi	`function GetNextScaledDigital(out min: Integer; out max: Integer): WordBool;`

Returns the next minimum/maximum pair of the signal according to the scaling defined by **`ScaleX()`** as a digital value. This function does not regard the settings of **`ScaleY()`**.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.26 Unscale

C++	`HRESULT Unscale();`
BASIC	`Sub Unscale()`
Delphi	`procedure Unscale;`

Removes the signal scaling so that all signal samples can be retrieved by means of the function **`NextSample()`**.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.27 NextSample

C++	`__declspec(property(get=GetNextSample,put=PutNextSample)` `  double NextSample;`
BASIC	`Property NextSample As Double`
Delphi	`property NextSample: Double` `  read Get_NextSample write Set_NextSample`

Returns the next signal sample. This function only returns meaningful values if the signal scaling has previously been turned off with **`Unscale()`**.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.28 NextDigitalSample

C++	`__declspec(property(get=GetNextDigitalSample,`
	`  put=PutNextDigitalSample)`
	`  long NextDigitalSample;`

BASIC	`Property NextDigitalSample As Long`

Delphi	`property NextDigitalSample: Long`
	`  read Get_NextDigitalSample`
	`  write Set_NextDigitalSample`

Returns the next value of a digital signal.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.29 GetSampleAt

C++	`double GetSampleAt(double time);`
BASIC	`Function GetSampleAt(time As Double) As Double`
Delphi	`function GetSampleAt(time: Double): Double;`

Returns a measuring value at a certain point of time in the signal.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.30 GetSampleAtOffset

C++	`double GetSampleAtOffset(long offset);`
BASIC	`Function GetSampleAtOffset(offset As Long) As Double`
Delphi	`function GetSampleAt(offset: Integer): Double;`

Returns a measuring value at a certain offset in the signal. The parameter **`offset`** must be between **`0`** and **`SampleCount`**.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.31 IsAnalog

C++	`VARIANT_BOOL IsAnalog();`
BASIC	`Function IsAnalog() As Boolean`
Delphi	`function IsAnalog: WordBool;`

Returns **TRUE** if the signal contains analog values.

A list of all possible commands is provided in chapter "Overview", p. 61.

4.3.32 IsDigital

C++	`VARIANT_BOOL IsDigital();`
BASIC	`Function IsDigital() As Boolean`
Delphi	`function IsDigital: WordBool;`

Returns **TRUE** if the signal contains digital values.

A list of all possible commands is provided in chapter "Overview", p. 61.

5 Index

F

G

I

L

M

N

O

P

R

S

T

U

V

X

Y