bmc messsysteme gmbh

Installations- und Programmierhandbuch

Version 4

www.bmcm.de

LibadX

Programmer's Guide to
Drivers' Programming Interface

Inhaltsverzeichnis

1 Überblick

1.1 Einleitung

LibadX ist eine einheitliche Programmierschnittstelle zu allen Messsystemen von BMC Messsysteme GmbH. Diese Schnittstelle ist u. a. von C++®, Visual C++®, Delphi®, als auch von Visual Basic® aus erreichbar.

- **Bitte beachten Sie, dass alle Beispielcodes in diesem Handbuch aus Gründen der Einfachheit bewusst auf eine Fehlerbehandlung verzichten. Selbstverständlich muss diese in selbst geschriebenen Programmen realisiert werden.**
- **Benutzer der ehemaligen Programmierschnittstelle BMCSAD, die ausschließlich die File Schnittstelle verwenden, können ihre Anwendungen zu 100% übertragen.**

1.3 Urheberrechte

Die Programmierschnittstelle **LibadX** mit allen Erweiterungen wurde mit größtmöglicher Sorgfalt erstellt und geprüft. Die BMC Messsysteme GmbH gibt keine Garantien, weder in Bezug auf dieses Handbuch noch in Bezug auf die in diesem Buch beschriebene Hard- und Software, ihre Qualität, Durchführbarkeit oder Verwendbarkeit für einen bestimmten Zweck. Die BMC Messsysteme GmbH haftet in keinem Fall für direkt oder indirekt verursachte oder erfolgte Schäden, die entweder aus unsachgemäßer Bedienung oder aus irgendwelchen Fehlern am System resultieren. Änderungen, die dem technischen Fortschritt dienen, bleiben uns vorbehalten.

Stand: 01. November 2006

BMC Messsysteme GmbH

Hauptstraße 21
82216 Maisach
DEUTSCHLAND

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

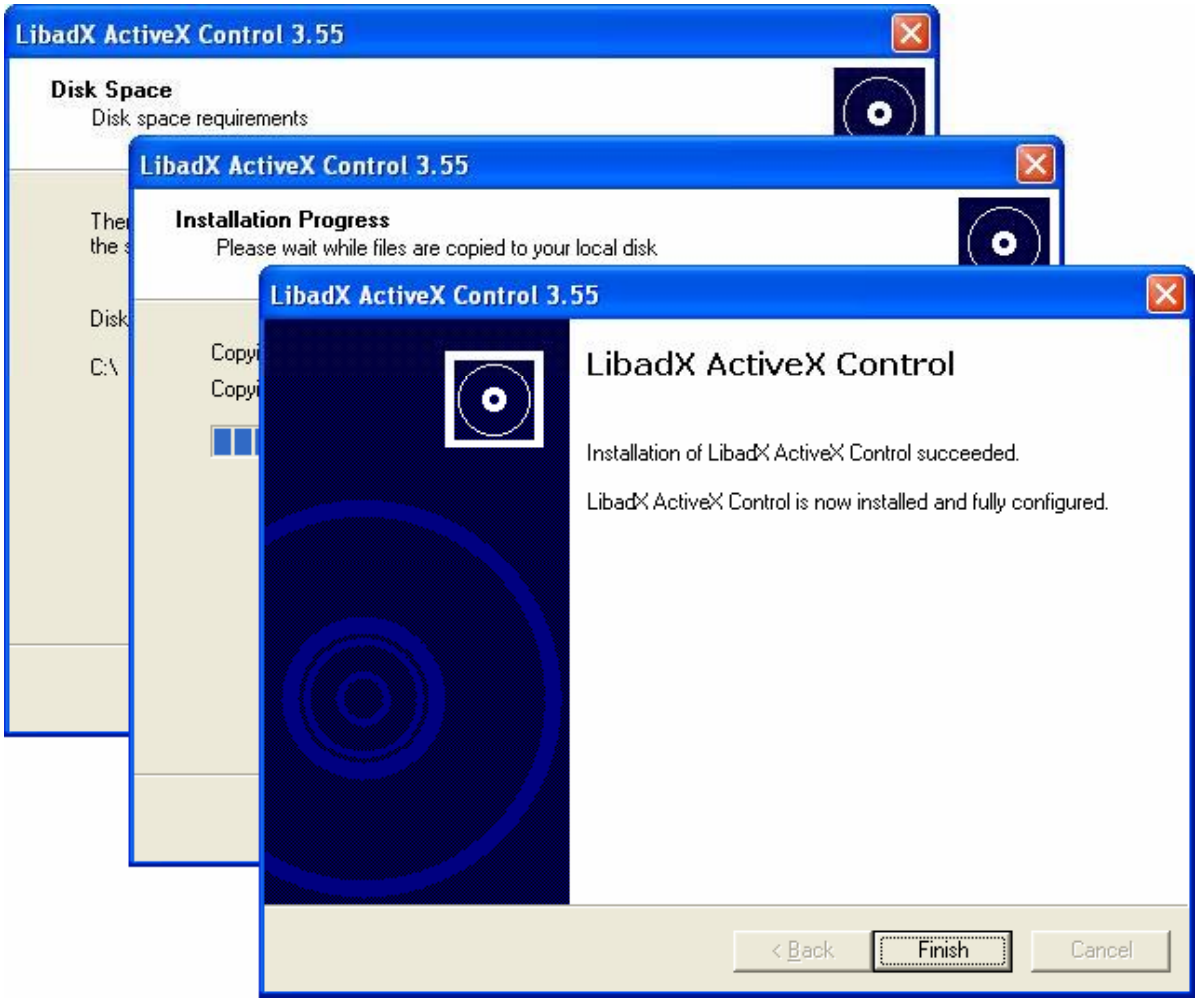

Abbildung 3

2.3 Anbindung an Visual Basic® 4.0 - 6.0

Starten Sie Visual Basic® und wählen Sie die Option "Standard EXE" aus dem Anfangsdialog (bzw. Menüpunkt "Datei / Neues Projekt").

LibadX wird wie jedes ActiveX Control in Visual Basic® über das Menü "Projekt" und den Eintrag "Komponenten" eingebunden. Im darauf folgenden Dialog "Komponenten" muss der Eintrag "LibadX Object Library 4.0" aktiviert werden.

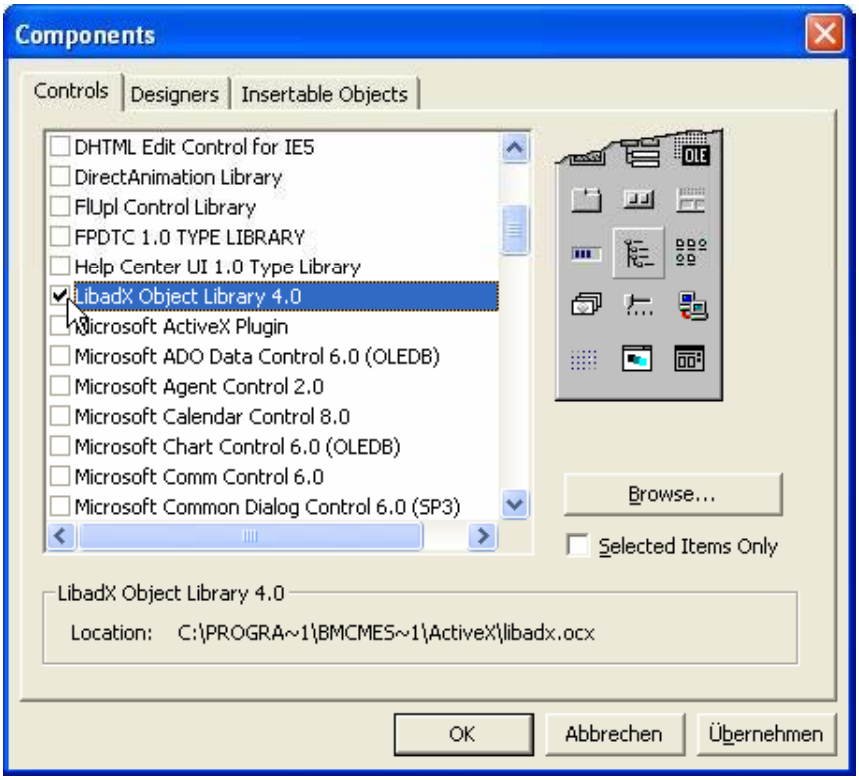

Abbildung 4

Abbildung 5

Danach steht das **LibadX** Icon in der Werkzeugleiste von Visual Basic® zur Verfügung und kann in eine Form eingefügt werden. Wie der Timer ist es während der Ausführung des Programms unsichtbar.

Klicken Sie wie gewohnt auf das Icon und ziehen Sie einen Rahmen in der Form des neu angelegten Projekts auf. Dieser Rahmen wird nach Einfügen des Objekts wieder auf die feste Größe des Icons reduziert.

Erstellen Sie dann die Routine **`Form_Load()`** im Codefenster des Projekts wie folgt:

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

Um die korrekte Installation der **LibadX** und ihre Erreichbarkeit in Visual Basic® zu überprüfen, empfiehlt es sich dieses Programm zu starten. Das Programm muss die Form ohne Fehlermeldung auf dem Bildschirm anzeigen.

- **Aus Kompatibilitätsgründen befindet sich das Icon der ehemaligen Programmierschnittstelle BMCSAD ebenfalls in der Toolbar. Als Benutzer der LibadX benötigen Sie dieses Icon und die ehemalige Programmierschnittstelle nicht.**
- **Bitte beachten Sie, dass alle Beispielcodes hier aus Gründen der Einfachheit bewusst auf eine Fehlerbehandlung verzichten. Selbstverständlich muss diese in selbst geschriebenen Programmen realisiert werden.**
- **Weitere Beispielprogramme mit Sourcecode können von der LibadX Produktseite der "Software Collection"-CD oder der "NextView®4"-CD installiert werden.**

2.5 Anbindung an Visual C++® 5.0/6.0

Visual C++® 5.0/6.0 bietet mit Hilfe der Preprozessoranweisung **`#import`** die Möglichkeit einfach COM-Schnittstellen in ein C++ Programm zu integrieren. Folgender Beispielcode demonstriert das Vorgehen:

```
#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;
}
```

- **Nähere Hinweise zu `#import`, `__uuidof()` und den Compiler Support Klassen für COM entnehmen Sie bitte dem Artikel "Microsoft Visual C++® Compiler Native COM Support" von Microsoft®, sowie der entsprechenden Compilerdokumentation von Microsoft®.**
- **Bitte beachten Sie, dass alle Beispielcodes hier aus Gründen der Einfachheit bewusst auf eine Fehlerbehandlung verzichten. Selbstverständlich muss diese in selbst geschriebenen Programmen realisiert werden.**
- **Weitere Beispielprogramme mit Sourcecode können von der LibadX Produktseite der "Software Collection"-CD oder der "NextView®4"-CD installiert werden.**

2.6 Beispielprogramme

Von der **LibadX** Produktseite der "Software Collection"-CD oder der "NextView®4"-CD können Beispielprogamme mit SourceCode für die Programmiersprachen Visual Basic®, Delphi® und Visual C++® installiert werden. Die **LibadX** Produktseite erreicht man durch Auswählen der Einträge "Programmierung" und "LibadX ActiveX Control".

Die Beispielprogramme befinden sich im während der Installation ausgewählten Verzeichnis (z. B. "Programme \ BMC Messsysteme") im Ordner "Examples \ LibadX", jeweils nach Programmiersprache getrennt.

Programmiersprache	Verzeichnis
Visual Basic®	**examples\vb**
Delphi®	**examples\delphi**
Visual C++®	**examples\vc5**

In jedem dieser Inhaltsverzeichnisse sind folgende Demoprogramme enthalten:

Name	Beschreibung
scan1	Demoprogramm für Konfiguration, Start und Speicherung einer Messung
scan2	wie **scan1**, demonstriert aber andere Speicherungsmöglichkeit, sowie Datei- und Signalinterface
sigview	Programm zur Anzeige einer gemessenen Kurve: eine Messdatei wird geöffnet und das erste Signal der Datei angezeigt
sigcalc	Programm zur Verrechnung gespeicherter Signale: eine Messdatei wird geöffnet, das erste Signal der Datei mit dem Faktor 10.0 multipliziert und das Ergebnis in einer anderen Datei gespeichert

Bitte beachten Sie, dass alle Beispielprogramme möglichst einfach ausgeführt worden sind, keine Fehlerbehandlung enthalten und aus diesem Grund keine vollständige Applikation darstellen.

3 Grundlagen

3.1 Allgemeines

Das **LibadX** ActiveX Control ist die Programmierschnittstelle zur **LIBAD4** Bibliothek. Diese ist eine Schnittstelle zu allen Messsystemen der BMC Messsysteme GmbH. Sie erlaubt das Lesen und Schreiben von Einzelwerten, wie das Einlesen eines Analogeingangs oder das Ausgeben eines Werts an einen Analogausgang.

Neben der Ein-/Ausgabe von Einzelwerten kann mit der LibadX eine Messung durchgeführt werden. Das Scannen der Eingangskanäle findet im entsprechenden Treiber statt und ist aus diesem Grund zeitlich von der Applikation entkoppelt. Schnell und ohne jeglichen Verlust von Messwerten lassen sich damit die Eingangskanäle abtasten.

Zusätzlich kann man mit der LibadX Programmierschnittstelle auf Messdateien der Messdatenerfassungs- und Analysesoftware **NextView®4** zugreifen.

3.2 Verbindung zum Messsystem

Das **LibadX** ActiveX Control stellt zwei Funktionen zur Verfügung, mit denen eine Verbindung zu einem Messsystem geöffnet bzw. wieder geschlossen werden kann.

Mit der Funktion **`Open()`** wird ein Messsystem geöffnet, mit **`Close()`** wieder geschlossen. Folgendes Beispiel demonstriert das prinzipielle Vorgehen:

```
if (LIBADX1.Open ("mempiousb"))
   ...
   LIBADX1.Close
else
   MsgBox "Konnte meM-PIO Gerät nicht öffnen"
```

Der Funktion **Open()** wird der Name des Messsystems übergeben. Der übergebene String wird ohne Berücksichtigung von Groß- und Kleinschreibung verwendet, das heißt “mempiousb" und “MEMPIOUSB" öffnen beide die meM-PIO. Wurde eine Verbindung mit dem Messsystem geöffnet, gibt **Open()** den Wert **TRUE** zurück, beim Auftritt eines Fehlers **FALSE**.

Dabei ist es nicht möglich mehrere Geräte mit einem Objekt gleichzeitig zu öffnen. Es ist aber durchaus möglich mehrere (verschiedene) Messsysteme zu öffnen, wenn mehrere Objekte benutzt werden. Folgendes Beispiel öffnet eine PC16TR / PC20TR und eine PCI-BASE300/1000:

```
if (LIBADX1.Open ("pci300")
    AND LIBADX2.Open ("pc20"))
    ...
endif
```

3.2.1 Kanalnummern und Messbereiche

Ein- bzw. Ausgangskanäle werden in **LibadX** durch Kanalnummern spezifiziert. Die verwendeten Kanalnummern sind abhängig vom eingesetzten Messsystem und in den entsprechenden Kapiteln dokumentiert. Beispielsweise ist der erste Analogeingang einer PCI-BASE300/1000 der Kanal 1.

Analoge Kanäle erwarten neben der Kanalnummer noch die Angabe eines Messbereichs (bzw. eines Ausgabebereichs), in dem gemessen (bzw. ausgegeben) werden soll. Dieser Messbereich ist wie die Kanalnummer vom Messsystem abhängig und in den folgenden Kapiteln dokumentiert.

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

Um ein iM-AD25a, iM-AD25, iM3250T oder iM2350 mit **LibadX** zu öffnen, muss an **Open()** der String "**im:<ip-addr>**" übergeben werden. Dabei muss **<ip-addr>** durch die entsprechende IP-Adresse ersetzt werden. Beispielsweise öffnet der String "**im:192.168.1.1**" das iM-Gerät mit der IP Adresse 192.168.1.1. Beim Öffnen des Treibers wird nicht zwischen den iM-Gerätetypen unterschieden.

3.2.5 PC20NHDL / PC20NVL / P1000TR/ P1000NV

Um eine PC20NHDL/PC20NVL/P1000TR/P1000NV mit der **LibadX** zu öffnen, muss an **`Open()`** der String "**`p1000`**" übergeben werden. Es wird beim Öffnen des Treibers nicht zwischen den einzelnen Kartenversionen unterschieden.

Mehrere Karten lassen sich durch Angabe der Kartennummer unterscheiden (1. Karte mit "**`p1000:0`**", 2. Karte mit "**`p1000:1`**", usw.).

Mess-system	Analog	Kanal-nummer	Digital	Kanal-nummer	range (Messber.)	range (Aus-gabebereich)
PC20NHDL, PC20NVL, P1000TR, P1000NV	16 Eing. 2 Ausg.	1..16 1 .. 2	2 Ports (je 16Bit)	1..2	0 (±10V) 1 (±5V) 2 (±2V) 3 (±1V)	0 (±10V) 1 (±5V)

Die Richtung der Portleitungen ist in 8-er Gruppen umschaltbar (s. **`DigitalDirection()`**, S. 45).

Die Ausgabebereiche der beiden analogen Ausgangskanäle werden hardwaremäßig auf der Messkarte konfiguriert. Der Aufrufer muss sicherstellen, dass der übergebene Messbereich mit dem konfigurierten Messbereich des Ausgangs übereinstimmt.

3.2.6 PIO24II / PIO48II

Um eine PIO24II oder PIO48II mit **LibadX** zu öffnen, muss an **`Open()`** der String "**`pioii`**" übergeben werden. Es wird beim Öffnen des Treibers nicht zwischen PIO24II und PIO48II unterschieden.

Mehrere Karten lassen sich durch Angabe der Kartennummer öffnen (1. Karte mit "**`pioii:0`**", 2. Karte mit "**`pioii:1`**", usw.).

Messsystem	Digital	Kanalnummer
PIO48II	6 Ports (je 8Bit)	1..6
PIO24II	3 Ports (je 8Bit)	1..3

Messsystem	Analog	Kanal-nummer	Digital	Kanal-nummer	Eing.-/Ausg.-bereich	range
USB-AD	16 Eingänge 1 Ausgang	1..16 1	2 Ports (je 4Bit)	1 .. 4 1 .. 4	±5.12V	33

Der erste analoge Eingangskanal eines USB-AD beginnt bei 1. Damit ergeben sich für die 16 Analogeingänge folgende Konstanten:

```
C
#define AI1   (AD_CHA_TYPE_ANALOG_IN|0x0001)
#define AI2   (AD_CHA_TYPE_ANALOG_IN|0x0002)
...
#define AI16  (AD_CHA_TYPE_ANALOG_IN|0x0010)
```

Der analoge Ausgangskanal eines USB-AD erhält die folgende Konstante:

```
C
#define AO1   (AD_CHA_TYPE_ANALOG_OUT|0x0001)
```

Die Richtung der digitalen Portleitungen ist nicht umschaltbar. Dabei stehen die 4 Leitungen des ersten Ports (DIO1) auf Eingang, die 4 Leitungen des zweiten Ports (DIO2) auf Ausgang. Für die Kanäle ergeben sich folgende Konstanten:

```
C
#define DIO1   (AD_CHA_TYPE_DIGITAL_IO|0x0001)
#define DIO2   (AD_CHA_TYPE_DIGITAL_IO|0x0002)
```

3.2.10 USB-PIO

Um eine USB-PIO mit der **LIBAD4** zu öffnen, muss an **`Open()`** der String **`"usb-pio"`** übergeben werden. Mehrere USB Messsysteme lassen sich durch Angabe der Gerätenummer öffnen (1. Gerät mit **`"usb-pio:0"`**, 2. Gerät mit

"usb-pio:1", usw.). Die Reihenfolge der Geräte wird durch das Anstecken bestimmt.

Da die USB Messsysteme im laufenden Betrieb an- und wieder abgesteckt werden können, ist es möglich, dass die Gerätenummern nicht in aufsteigender Reihenfolge vergeben sind. Werden beispielsweise drei Geräte angesteckt und dann das zweite Gerät wieder abgesteckt, sind die beiden verbleibenden Geräte mit "**usb-pio:0**" und "**usb-pio:2**" anzusprechen.

Um unabhängig von dieser Ansteckreihenfolge zu sein, kann ein Gerät auch mit einer bestimmten Seriennummer geöffnet werden. Das Gerät mit der Seriennummer 157 lässt sich zum Beispiel durch Angabe von "**usb-pio:@157**" ansprechen.

Messsystem	Digital	Kanalnummer
USB-PIO	3 Ports (je 8Bit)	1..3

Die Richtung der Leitungen ist für jeden Port getrennt einstellbar. Die Umstellung erfolgt portweise (s. **DigitalDirection()**, S. 45).

C

```
#define DIO1   (AD_CHA_TYPE_DIGITAL_IO|0x0001)
#define DIO2   (AD_CHA_TYPE_DIGITAL_IO|0x0002)
#define DIO3   (AD_CHA_TYPE_DIGITAL_IO|0x0003)
```

4 Schnittstellen und Funktionsumfang

4.1 Die Schnittstelle LibadX

Die Schnittstelle **LibadX** wird direkt vom LibadX ActiveX Control exportiert. Sie dient dazu, die Verbindung zum Messdatenserver herzustellen.

4.1.1 Überblick

Funktion	Beschreibung
`Open`	öffnet eine Verbindung zu einem Messsystem
`Close`	schließt die Verbindung zu einem Messsystem
`GetVersion`	liefert die Versionsnummer der **`LIBAD4.dll`**
`LastError`	liefert den letzen Fehlercode zurück
`LastErrorString`	liefert eine Beschreibung des letzen Fehlers
`ScanPrepare`	bereitet einen Scan vor
`ScanAnalogIn`	fügt einen analogen Eingang der Scanliste hinzu
`ScanDigitalIn`	fügt einen digitalen Eingang der Scanliste hinzu
`Scan`	startet einen vorbereiteten Scan
`ScanSave`	speichert einen durchgeführten Scan
`FileOpen`	legt ein Dateiobjekt an, mit dem auf gespeicherte Messdateien zugegriffen werden kann
`FileCreatePrepare`	bereitet das Anlegen einer Scandatei vor
`FileCreateAnalog`	fügt der Kanalliste einen Analogkanal hinzu
`FileCreateDigital`	fügt der Kanalliste einen Digitalkanal hinzu
`FileCreate`	erzeugt eine vorbereitete Scandatei
`AnalogIn`	liefert den aktuellen Wert eines Analogeingangs

AnalogOut	liefert den aktuellen Wert eines Analogausgangs
DigitalIn	liefert den aktuellen Wert eines Digitaleingangs
DigitalOut	liefert den aktuellen Wert eines Digitalausgangs
DigitalInLine	liefert den aktuellen Wert einer digitalen Eingangsleitung
DigitalOutLine	liefert aktuellen Wert einer digitalen Ausgangsleitung
DigitalDirection	setzt/liefert die eingestellte Richtung eines Digitalkanals
Sample	liest den Wert eines Samples in einem Scan
AboutBox	zeigt die AboutBox von **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`

Die Funktion **`Open()`** stellt eine Verbindung zum Messsystem her. Es wird der Name des Messsystems übergeben. Der übergebene String wird ohne Berücksichtigung von Groß- und Kleinschreibung verwendet, das heißt "pci300" und "Pci300" öffnen beide die PCI-BASE300/1000.

Wurde eine Verbindung mit dem Messsystem geöffnet, gibt **`Open`** den Wert **`TRUE`** zurück, beim einem Fehler **`FALSE`**. Eine ausführliche Beschreibung des Befehls **`Open()`** finden Sie im Kapitel "Verbindung zum Messsystem", S. 21.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.3 Close

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

Die Funktion **`Close()`** schließt eine Verbindung zum Messsystem.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.4 GetVersion

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

Die Funktion **`GetVersion()`** liefert die Version der LIBAD4.dll zurück, die von **LibadX** benutzt wird.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.5 LastError

C++	`long LastError ()`
BASIC	`Function LastError () As Long`

Delphi	`function LastError: Integer`

Gibt die Nummer des zuletzt aufgetretenen Fehlers zurück. Ist kein Fehler aufgetreten, gibt die Funktion **0** zurück.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.6 LastErrorString

C++	`_bstr_t LastErrorString ()`

BASIC	`Function LastErrorString () As String`

Delphi	`function LastErrorString: WideString`

Gibt eine Beschreibung des zuletzt aufgetretenen Fehlers zurück. Ist kein Fehler aufgetreten, gibt die Funktion "" zurück.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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)`

Um einen Scan zu starten muss zuerst **`ScanPrepare()`** aufgerufen werden. Dadurch wird die **LibadX** auf einen Scan vorbereitet und die Abtastrate auf **`sample_rate`** und die Anzahl der zu speichernden Werte auf **`samples`** gestellt.

Um einen Kanal der Kanalliste für den Scan hinzuzufügen, ruft man **`ScanAnalogIn()`** bzw. **`ScanDigitalIn()`** auf. Der Start des Scans erfolgt durch einen Aufruf der Funktion **`Scan()`**.

Folgender Visual Basic® Beispielcode zeigt die Funktionsweise:

```
' 1000 Messwerte, 100Hz (0.01 Sek.)
LIBADX1.ScanPrepare 0.01, 1000

' kanal 1 & 2 speichern
LIBADX1.ScanAnalogIn 1, 0
LIBADX1.ScanAnalogIn 2, 0

' Scan durchführen
LIBADX1.Scan

' Scan speichern
LIBADX1.ScanSave "scan.lfx"
```

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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)`

Mit **`ScanAnalogIn()`** fügt man den analogen Kanal mit der Nummer **`index`** und der Range **`range`** der Kanalliste des Scans hinzu. Die Funktion löst eine Exception aus, wenn vorher kein Scan mit **`ScanPrepare()`** (s. S. 35) vorbereitet wurde.

Aufgrund von Einschränkungen bei den meisten Messkarten müssen die Eingangskanäle unbedingt in aufsteigender Reihenfolge der Kanalliste hinzugefügt werden! Werden sowohl analoge Eingänge als auch digitale Eingänge abgetastet, müssen zuerst alle analogen und danach die digitalen Kanäle angegeben werden!

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.9 ScanDigitalIn

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

Mit **`ScanDigitalIn()`** fügt man den digitalen Kanal mit der Nummer **`index`** und der Range **`range`** der Kanalliste des Scans hinzu. Die Funktion löst eine Exception aus, wenn vorher kein Scan mit **`ScanPrepare()`** (s. S. 35) vorbereitet wurde.

Aufgrund von Einschränkungen bei den meisten Messkarten müssen die Eingangskanäle unbedingt in aufsteigender Reihenfolge der Kanalliste hinzugefügt werden! Werden sowohl analoge Eingänge als auch digitale Eingänge abgetastet, müssen zuerst alle analogen und danach die digitalen Kanäle angegeben werden!

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.10 Scan

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

Mit **`Scan()`** wird ein mit **`ScanPrepare()`**, **`ScanAnalogIn()`** und **`ScanDigitalIn()`** vorbereiteter Scan gestartet. Während des Scans blockiert das aufrufende Programm bzw. der aufrufende Thread.

Die Funktion löst eine Exception aus, wenn vorher kein Scan mit **`ScanPrepare()`** (s. S. 35) vorbereitet oder kein Kanal der Kanalliste hinzugefügt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Mit **`ScanSave()`** wird ein mit **`Scan()`** durchgeführter Scan gespeichert.

Die Funktion löst eine Exception aus, wenn vorher kein Scan mit **`Scan()`** durchgeführt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Öffnet das angegebene Messdatenfile. Falls das File nicht existiert oder nicht geöffnet werden kann, löst die Funktion eine Exception aus.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.13 FileCreatePrepare

C++	`HRESULT FileCreatePrepare (long samples)`
BASIC	`Sub FileCreatePrepare (samples As Long)`
Delphi	`procedure FileCreatePrepare(samples: Integer)`

Die Erstellung einer Messdatei erfolgt wie bei einem Scan. Dazu muss zuerst **`FileCreatePrepare()`** mit der Anzahl der zu speichernden Werte aufgerufen werden.

Um der Kanalliste einer Datei einen Kanal hinzuzufügen, ruft man **`FileCreateAnalog()`** bzw. **`FileCreateDigital()`** auf. Die Datei wird durch einen Aufruf von **`FileCreate()`** schließlich erzeugt.

Folgender Visual Basic® Beispielcode demonstriert die Funktionsweise:

```
' 1000 Messwerte
LIBADX1.FileCreatePrepare 1000

' 2 Analoge Kanaele
LIBADX1.FileCreateAnalogIn
LIBADX1.FileCreateAnalogIn

' Datei anlegen
LIBADX1.FileCreate "scan.lfx"
```

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.14 FileCreateAnalog

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

Mit **`FileCreateAnalog()`** fügt man einen analogen Kanal der Kanalliste einer zu erstellenden Datei hinzu. Als Rückgabewert erhält man den Index des Kanals in der Datei. Die Funktion löst eine Exception aus, wenn vorher keine Messdatei mit **`FileCreatePrepare()`** (s. S. 39) vorbereitet wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.15 FileCreateDigital

C++	`long FileCreateDigital (long lines)`
BASIC	`Function FileCreateDigital (lines As Long) As Long`

Delphi	`function FileCreateDigital(lines: Integer): Integer;`

Mit **`FileCreateDigital()`** fügt man einen digitalen Kanal der Kanalliste einer zu erstellenden Datei hinzu. Dabei ist **`lines`** die Anzahl der zu speichernden Leitungen. Diese darf 32 nicht überschreiten. Als Rückgabewert erhält man den Index des Kanals in der Datei. Die Funktion löst eine Exception aus, wenn vorher keine Messdatei mit **`FileCreatePrepare()`** (s. S. 39) vorbereitet wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Mit **`FileCreate()`** wird ein mit **`FileCreatePrepare()`**, **`FileCreateAnalog()`** und **`FileCreateDigital()`** vorbereiteter Scan gestartet. Die Funktion löst eine Exception aus, wenn vorher keine Messdatei mit **`FileCreatePrepare()`** (s. S. 39) vorbereitet oder kein Kanal der Kanalliste hinzugefügt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Liefert für den analogen Eingangskanal mit der Nummer **`index`** den aktuell gemessenen Wert im Messbereich **`range`** dieses Kanals. Der Wert dieses Properties ist nur lesbar.

Die Funktion löst eine Exception aus, wenn vorher keine Verbindung mit einem Messsystem durch **`Open()`** hergestellt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Legt den für den Ausgangskanal mit der Nummer **`index`** aktuell gesetzten Wert im Ausgabebereichs **`range`** dieses Kanals fest oder gibt diesen zurück.

Die Funktion löst eine Exception aus, wenn vorher keine Verbindung mit einem Messsystem durch **`Open()`** hergestellt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Liefert für den digitalen Eingangskanal mit der Nummer **`index`** den aktuell gemessenen Wert. Der Wert dieses Properties ist nur lesbar.

Die Funktion löst eine Exception aus, wenn vorher keine Verbindung mit einem Messsystem durch **`Open()`** hergestellt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Legt den für den Ausgangskanal mit der Nummer **`index`** aktuell gesetzten Wert dieses Kanals fest oder gibt diesen zurück.

Die Funktion löst eine Exception aus, wenn vorher keine Verbindung mit einem Messsystem durch **`Open()`** hergestellt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Liefert für die Leitung Nummer **`line`** des digitalen Eingangskanals mit der Nummer **`index`** den aktuell gemessenen Wert. Der Wert dieses Properties ist nur lesbar.

Die Funktion löst eine Exception aus, wenn vorher keine Verbindung mit einem Messsystem durch **`Open()`** hergestellt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Legt für die Leitung Nummer **`line`** des digitalen Eingangskanals mit der Nummer **`index`** den aktuell gesetzten Wert fest oder gibt diesen zurück.

Die Funktion löst eine Exception aus, wenn vorher keine Verbindung mit einem Messsystem durch **Open()** hergestellt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Legt die Ein-/Ausgaberichtung aller Leitungen des Digitalkanals mit der Nummer **index** fest oder gibt diesen zurück. Dabei wird eine Bitmaske übergeben, die die Richtung der Digitalleitung beschreibt. Jedes gesetzte Bit definiert eine Eingangsleitung, jedes gelöschte Bit eine Ausgangsleitung. Das Bit #0 legt die Richtung der ersten Leitung des Digitalports fest.

Die Funktion löst eine Exception aus, wenn vorher keine Verbindung mit einem Messsystem durch **Open()** hergestellt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

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`

Legt das Sample des Kanals **index** an der Position **pos** des durchgeführten Scans fest oder gibt diesen zurück.

Die Funktion löst eine Exception aus, wenn vorher kein Scan durchgeführt wurde oder **index** bzw. **pos** ungültig sind.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.1.25 AboutBox

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

Zeigt die AboutBox von **LibadX**.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 32.

4.2 Die Schnittstelle INvxFile

Die Schnittstelle **`INvxFile`** ermöglicht den Zugriff auf gespeicherte Messdaten.

4.2.1 Überblick

Funktion	Beschreibung
`Open`	öffnet ein Messdatenfile
`Create`	legt ein neues Messdatenfile an
`Close`	schließt das Messdatenfile wieder
`SignalCount`	liefert die Anzahl der Signale im Messdatenfile
`Signal`	gibt die Schnittstelle zu einem Signal im Messdatenfile zurück

4.2.2 Open

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

Öffnet das angegebene Messdatenfile. Falls das File nicht existiert oder nicht geöffnet werden kann, löst die Funktion eine Exception aus.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 47.

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);`

Erzeugt ein neues Messdatenfile. In der Datei werden **`SignalCount`** Signale angelegt, wobei jedes Signal **`SignalCount`** Messwerte speichern kann.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 47.

4.2.4 Close

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

Schließt eine Messdatei, die mit **`Open()`** geöffnet oder **`Create()`** erzeugt wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 47.

4.2.5 SignalCount

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

Liefert die Signalanzahl im Messdatenfile. Die Funktion löst eine Exception aus, wenn vorher keine Messdatei mit **`Open()`** geöffnet oder mit **`Create()`** angelegt worden ist.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 47.

4.2.6 Signal

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

Liefert ein Signal aus dem Messdatenfile. Das erste Signal im File hat den Index 1.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 47.

4.3 Die Schnittstelle INvxSignal

Die Schnittstelle **`INvxSignal`** ermöglicht den Zugriff auf ein einzelnes Signal aus einem Messdatenfile.

4.3.1 Überblick

Funktion	Beschreibung
`Name`	Name des Signals
`GroupName`	Gruppenname des Signals
`Comment`	Kommentar des Signals
`xStart`	Startzeitpunkt des Signals
`xEnd`	Endzeitpunkt des Signals
`xDelta`	Abtastzeit des Signals
`xUnit`	Einheit der x-Achse
`xSetUsing`	legt das Zahlenformat der x-Achse fest
`xGetUsing`	liefert das Zahlenformat der x-Achse
`yMin`	untere Messbereichsgrenze
`yMax`	obere Messbereichsgrenze
`yDefaultMin`	untere Grenze des Standardanzeigebereichs
`yDefaultMax`	obere Grenze des Standardanzeigebereichs
`yDelta`	Auflösung des Signals
`yUnit`	y-Achseneinheit
`ySetUsing`	legt das Zahlenformat der y-Achse fest
`yGetUsing`	liefert das Zahlenformat der y-Achse
`ScanStart`	Datum zu Beginn der Messung
`SampleCount`	Anzahl der Messwerte im Signal
`ScaleX`	Skalierung der x-Achse
`ScaleY`	Skalierung der y-Achse
`ResetDataPosition`	Zurücksetzen der aktuellen Position im Signal

GetNextScaled	liefert das nächste skalierte Wertepaar
GetNextScaledDigital	liefert das nächste skalierte Wertepaar eines Digitalsignals
Unscale	hebt die Skalierung des Signals auf
NextSample	liefert den nächsten Messwert an der aktuellen Position im Signal
GetSampleAt	liefert einen Messwert an einer bestimmten zeitlichen Position im Signal
GetSampleAtOffset	liefert einen Messwert an einem bestimmten Offset im Signal
IsAnalog	überprüft, ob das Signal analoge Messwerte enthält
IsDigital	überprüft, ob das Signal digitale Messwerte enthält

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;`

Liefert den Namen des Signals.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

4.3.3 GroupName

C++	`__declspec(property(get=GetGroupName,put=PutGroupName))` `  _bstr_t GroupName;`

BASIC	`Property GroupName As String`

Delphi	`property GroupName: WideString` `  read Get_GroupName write Set_GroupName;`

Liefert den Gruppennamen des Signals.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert den Kommentar des Signals.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert den Startzeitpunkt des Signals in Sekunden. Dieser Wert steht im Normalfall auf 0.0s. Nur Messungen mit Trigger geben hier als negativen Wert die Dauer der Vorgeschichte zurück (als negativen Wert).

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert den Endzeitpunkt des Signals. Bitte beachten Sie, dass sich die Gesamtdauer des Signals vom Endzeitpunkt unterscheiden kann. Die Gesamtdauer lässt sich mit **`xEnd-xStart`** berechnen.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert die Abtastzeit des Signals in Sekunden.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert die Einheit der x-Achse.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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);`

Legt das Zahlenformat für die Ausgabe der Werte an der x-Achse dieses Signals fest. Dabei legt **`format`** das Ausgabeformat fest, **`width`** die Breite des

Zahlenfeldes und **frac** die Anzahl der Ziffern nach der Kommastelle. Das Argument **opt** wird nur für die feste wissenschaftliche Darstellung verwendet und gibt dort die verwendete Zehnerpotenz an (siehe nachfolgende Tabelle).

Für **format** können folgende Zahlenwerte übergeben werden, alle anderen führen zu dem Fehlercode **E_INVALIDARG**:

Wert	Bedeutung	Beispiel: 17336,78
0	gibt die Messwerte dezimal aus	17336
3	Der Messwert wird ohne Angabe einer Zehnerpotenz mit der Anzahl der Kommastellen laut **frac** ausgegeben.	17336,780
4	Angabe der Zehnerpotenz mit der Schreibweise **E+xxx**	1,734E+004
5	Wissenschaftliche Darstellung: Die Zehnerpotenz des Messwerts wird mit wissenschaftlichen Kürzeln dargestellt. p (10^{-12}), n (10^{-9}), µ (10^{-6}), m (10^{-3}), k (10^{3}), M (10^{6}), G (10^{9})	17,337k
6	Feste wissenschaftliche Darstellung: Die Zehnerpotenz wird fest über den Parameter **opt** vorgegeben. Dabei können für **opt** folgende Werte vorkommen: 0: p (10^{-12}) 3: m (10^{-3}) 6: M (10^{6}) 1: n (10^{-9}) 4: (10^{0}) 7: G (10^{9}) 2: µ (10^{-6}) 5: k (10^{3})	0,017M

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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);`

Liefert die Einstellungen für das aktuelle Zahlenformat der Werte an der x-Achse dieses Signals. Die Bedeutung der einzelnen Parameter ist im Kapitel "xSetUsing", S. 54 beschrieben.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert die untere Grenze des Messbereichs, in dem das Signal aufgezeichnet worden ist.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert die obere Messbereichsgrenze, in dem das Signal aufgezeichnet worden ist.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert die untere Grenze des Darstellungsbereichs, in dem das Signal standardmäßig gezeichnet werden soll.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert die obere Grenze des Darstellungsbereichs, in dem das Signal standard-mäßig gezeichnet werden soll.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert die Auflösung der Messwerte des Signals.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert die Einheit der y-Achse.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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);`

Legt das Zahlenformat für die Ausgabe der Messwerte des Signals fest. Dabei legt **`format`** das Ausgabeformat fest, **`width`** die Breite des Zahlenfeldes und **`frac`** die Anzahl Ziffern nach der Kommastelle. Das Argument **`opt`** wird nur für die feste wissenschaftliche Darstellung verwendet und gibt dort die verwendete Zehnerpotenz an (siehe nachfolgende Tabelle).

Für **`format`** können folgende Zahlenwerte übergeben werden, alle anderen führen zu dem Fehlercode **`E_INVALIDARG`**:

Wert	Bedeutung	Beispiel: 17336,78
0	gibt die Messwerte dezimal aus	17336
3	Der Messwert wird ohne Angabe einer Zehnerpotenz mit der Anzahl der Kommastellen laut **frac** ausgegeben.	17336,780
4	Messwerte werden unter Angabe der Zehnerpotenz in der Schreibweise **E+xxx** ausgegeben.	1,734E+004
5	Wissenschaftliche Darstellung: Die Zehnerpotenz des Messwerts wird mit wissenschaftlichen Kürzeln dargestellt p (10^{-12}), n (10^{-9}), µ (10^{-6}), m (10^{-3}), k (10^{3}), M (10^{6}), G (10^{9})	17,337k
6	Feste wissenschaftliche Darstellung: In diesem Fall wird die Zehnerpotenz fest über den Parameter **opt** vorgegeben. Dabei können für **opt** folgende Werte vorkommen: 0: p (10^{-12}) 3: m (10^{-3}) 6: M (10^{6}) 1: n (10^{-9}) 4: (10^{0}) 7: G (10^{9}) 2: µ (10^{-6}) 5: k (10^{3})	0,017M

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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);
```

Liefert die Einstellungen für das aktuelle Zahlenformat der Messwerte des Signals. Die Bedeutung der einzelnen Parameter ist im Kapitel "ySetUsing", S. 59 beschrieben.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert das Datum des Starts der Messung (d. h. den Zeitpunkt, zu dem der erste Messwert des Signals aufgenommen worden ist). Das Datum wird in Sekunden seit dem 1. Januar 1970 übergeben.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

4.3.20 SampleCount

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

Liefert die Anzahl der Messwerte des Signals.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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);`

Skaliert den x-Bereich eines Signals so, dass von **`GetNextScaled`** die Messwerte zwischen **`xStart`** und **`xEnd`** übergeben werden. Dabei muss die Routine **`GetNextScaled`** **`px`**-mal aufgerufen werden, um den gesamten Kurvenzug zu zeichnen.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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);`

Skaliert den y-Bereich eines Signals so, dass die Messwerte zwischen **yMinyMax** und **yMax** auf die Integerwerte **0** bis **py** abgebildet werden.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

4.3.23 ResetDataPosition

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

Setzt den internen Zähler des Signals zurück, so dass der nächste Aufruf von **GetNextScaled** das erste Minimum/Maximum-Paar zurückliefert (bzw. **NextSample** liefert den ersten Messwert des Signals).

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert das nächste Minimum/Maximum-Paar des Signals entsprechend den Skalierungen durch **ScaleX()** und **ScaleY()**.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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;`

Liefert das nächste Minimum/Maximum-Paar des Signals entsprechend den Skalierungen durch **`ScaleX()`** als Digitalwert. Diese Funktion berücksichtigt die Einstellungen von **`ScaleY()`** nicht.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

4.3.26 Unscale

C++	`HRESULT Unscale();`

BASIC	`Sub Unscale()`

Delphi	`Procedure Unscale;`

Schaltet die Skalierung des Signals ab, so dass alle Messwerte des Signals mit Hilfe der Funktion **`NextSample()`** abgefragt werden können.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

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`

Liefert den nächsten Messwert des Signals. Diese Funktion liefert nur dann sinnvolle Werte, wenn die Skalierung des Signals vorher mit **`Unscale()`** abgeschaltet wurde.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

4.3.28 GetSampleAt

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

Liefert einen Messwert zu einer bestimmten Zeit innerhalb des Signals.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

4.3.29 GetSampleAtOffset

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

Liefert einen Messwert an einem bestimmten Offset innerhalb des Signals. Dabei muss **`offset`** innerhalb von **`0`** und **`SampleCount`** liegen.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

4.3.30 IsAnalog

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

Liefert **`TRUE`** zurück, wenn im Signal analoge Werte gespeichert sind.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

4.3.31 IsDigital

C++	`VARIANT_BOOL IsDigital();`

BASIC	`Function IsDigital() As Boolean`

Delphi	`function IsDigital: WordBool;`

Liefert **TRUE** zurück, wenn im Signal digitale Werte gespeichert sind.

Eine Übersicht aller möglichen Befehle erhalten Sie im Kapitel "Überblick", S. 50.

5 Index

Z