

BSA Processor Driver

Reference Manual

This reference manual may not be copied, photocopied, translated, modified, or reduced to any electronic medium or machine-readable form, in whole, or in part, without the prior written consent of Dantec Dynamics A/S.

Publication no.: 9040U2993. Date: May, 2008.

©2005-2008 by Dantec Dynamics A/S, P.O. Box 121, Tonsbakken 18, DK-2740 Skovlunde, Denmark. All rights reserved.

All trademarks referred to in this document are registered by their owners.

1. Table of Contents

1.	Table of Contents.....	3
2.	Dantec Dynamics License Agreement.....	7
2.1	Grant of License	7
2.2	Upgrades.....	7
2.3	Copyright.....	7
2.4	Limited Warranty	8
2.5	Limitation of Liability	8
2.6	Product Liability.....	8
2.7	Governing Law and Proper Forum.....	9
2.8	Questions.....	9
3.	Laser Safety	9
3.1	Danger laser light	9
3.2	Precautions	10
4.	Introduction.....	11
4.1	Preface	11
4.2	Purpose	11
4.3	Reading this Manual.....	12
5.	Using the BSA Processor Driver	13
5.1	Installation.....	13
5.2	BSA Flow Software Script Generator	15
5.3	Quick Start.....	16
5.4	Quick Examples	19
5.4.1	Visual Basic Quick Example	20
5.4.2	LabVIEW Quick Example.....	24
5.4.3	C++ Quick Example	30
6.	BSA.Ctrl Function Reference	31
6.1	Connecting.....	35
6.1.1	IBSA::Connect.....	35
6.1.2	IBSA::Disconnect	36
6.1.3	IBSA::GetConnectState	37
6.2	Configuring	38
6.2.1	IBSA::SetProcessor	38
6.2.2	IBSA::SetConfiguration	38
6.2.3	IBSA::SetProperties.....	42
6.2.4	IBSA::ActivateSysMon	46
6.2.5	IBSA::DeactivateSysMon.....	47
6.2.6	IBSA::GetStatus	48
6.3	Advanced acquisition settings.....	49
6.3.1	IBSA::SetHighVoltageActivation	49
6.3.2	IBSA::GetHighVoltageActivation.....	49
6.3.3	IBSA::SetAnodeCurrentWarningLevel	51
6.3.4	IBSA::GetAnodeCurrentWarningLevel	51
6.3.5	IBSA::SetDataCollectionMode	52
6.3.6	IBSA::GetDataCollectionMode.....	53
6.3.7	IBSA::SetDutyCycle.....	54
6.3.8	IBSA::GetDutyCycle.....	54
6.3.9	IBSA::SetDeadTime	55
6.3.10	IBSA::GetDeadTime	55
6.4	Frequency shift.....	56

6.4.1	IBSA::Set40MHzFreqShiftOutput.....	57
6.4.2	IBSA::Get40MHzFreqShiftOutput.....	57
6.4.3	IBSA::SetVarFreqShiftOutput.....	58
6.4.4	IBSA::GetVarFreqShiftOutput.....	58
6.4.5	IBSA::SetVarShiftFreq.....	59
6.4.6	IBSA::GetVarShiftFreq.....	59
6.5	Synchronization Signals Settings.....	60
6.5.1	Dependencies.....	60
6.5.2	Output Signals settings.....	60
6.5.3	IBSA::SetSync1.....	62
6.5.4	IBSA::GetSync1.....	62
6.5.5	IBSA::SetSync2.....	64
6.5.6	IBSA::GetSync2.....	64
6.5.7	IBSA::SetSync1Edge.....	65
6.5.8	IBSA::GetSync1Edge.....	65
6.5.9	IBSA::SetSync2Edge.....	66
6.5.10	IBSA::GetSync2Edge.....	66
6.5.11	IBSA::SetEncoderReset.....	67
6.5.12	IBSA::GetEncoderReset.....	67
6.5.13	IBSA::SetEncoder.....	68
6.5.14	IBSA::GetEncoder.....	68
6.5.15	IBSA::SetStartMeasurement.....	69
6.5.16	IBSA::GetStartMeasurement.....	69
6.5.17	IBSA::SetStopMeasurement.....	70
6.5.18	IBSA::GetStopMeasurement.....	70
6.5.19	IBSA::SetBurstDetectorEnable.....	71
6.5.20	IBSA::GetBurstDetectorEnable.....	71
6.5.21	IBSA::SetRefClk.....	72
6.5.22	IBSA::GetRefClk.....	72
6.6	Sync Output signals.....	74
6.6.1	IBSA::SetBNC.....	74
6.6.2	IBSA::GetBNC.....	75
6.6.3	IBSA::SetDSUB1.....	76
6.6.4	IBSA::GetDSUB1.....	76
6.6.5	IBSA::SetDSUB2 and IBSA::SetDSUB3.....	76
6.6.6	IBSA::GetDSUB2 and IBSA::GetDSUB3.....	76
6.6.7	IBSA::SetShutter.....	77
6.6.8	IBSA::GetShutter.....	77
6.7	Group Settings.....	78
6.7.1	IBSA::SetMaxSamples.....	78
6.7.2	IBSA::GetMaxSamples.....	78
6.7.3	IBSA::SetMaxAcqTime.....	80
6.7.4	IBSA::GetMaxAcqTime.....	80
6.7.5	IBSA::SetFilterMethod.....	82
6.7.6	IBSA::GetFilterMethod.....	82
6.7.7	IBSA::SetBurstWindow.....	84
6.7.8	IBSA::GetBurstWindow.....	84
6.7.9	IBSA::SetScopeDisplay.....	85
6.7.10	IBSA::GetScopeDisplay.....	85
6.7.11	IBSA::SetScopeTrigger.....	87
6.7.12	IBSA::GetScopeTrigger.....	87
6.7.13	IBSA::SetScopeZoom.....	89
6.7.14	IBSA::GetScopeZoom.....	89
6.8	LDA Settings.....	90

6.8.1	IBSA::SetCenterFreq	90
6.8.2	IBSA::GetCenterFreq	91
6.8.3	IBSA::SetBandwidth	92
6.8.4	IBSA::GetBandwidth	92
6.8.5	IBSA::SetRecordLengthMode	94
6.8.6	IBSA::GetRecordLengthMode	94
6.8.7	IBSA::SetRecordLength	96
6.8.8	IBSA::GetRecordLength	96
6.8.9	IBSA::SetMaxRecordLength	98
6.8.10	IBSA::GetMaxRecordLength	98
6.8.11	IBSA::SetHighVoltage	100
6.8.12	IBSA::GetHighVoltage	100
6.8.13	IBSA::SetSignalGain	102
6.8.14	IBSA::GetSignalGain	102
6.8.15	IBSA::SetBurstDetectorSNR	103
6.8.16	IBSA::GetBurstDetectorSNR	103
6.8.17	IBSA::SetLevelValidationRatio	104
6.8.18	IBSA::GetLevelValidationRatio	104
6.9	PDA Methods	106
6.9.1	IBSA::SetPDACalibrationMode	106
6.9.2	IBSA::GetPDACalibrationMode	107
6.9.3	IBSA::SetPDAHighVoltage	108
6.9.4	IBSA::GetPDAHighVoltage	108
6.9.5	IBSA::SetPDAAutoBalancing	109
6.9.6	IBSA::GetPDAAutoBalancing	110
6.10	Advanced	111
6.10.1	IBSA::SetAnodeCurrentLimit	111
6.10.2	IBSA::GetAnodeCurrentLimit	111
6.11	Acquiring	113
6.11.1	Memory Usage	115
6.11.2	IBSA::StartAcq	118
6.11.3	IBSA::StopAcq	118
6.11.4	IBSA::PreAllocateData	119
6.11.5	IBSA::ReadDataLength	120
6.11.6	IBSA::ReadArrivalTimeData	121
6.11.7	IBSA::ReadTransitTimeData	122
6.11.8	IBSA::ReadVelocityData	122
6.11.9	IBSA::ReadDiameterData	123
6.11.10	IBSA::ReadEncoderData	124
6.11.11	IBSA::ReadSyncDataLength	124
6.11.12	IBSA::ReadSyncArrivalTimeData	125
6.11.13	IBSA::ReadSyncData	125
6.12	Optics Methods	127
6.12.1	IBSA::SetOpticsShift	128
6.12.2	IBSA::GetOpticsShift	129
6.12.3	IBSA::SetOpticsConversionFactor	129
6.12.4	IBSA::GetOpticsConversionFactor	130
6.12.5	IBSA::SetOpticsPhaseFactor	132
6.12.6	IBSA::GetOpticsPhaseFactor	133
6.12.7	IBSA::SetOpticsValidationBand	134
6.12.8	IBSA::GetOpticsValidationBand	134
6.12.9	IBSA::SetOpticsFringeDirection	135
6.12.10	IBSA::GetOpticsFringeDirection	135
6.13	Event Handling	136

6.14 Error Handling.....	138
6.14.1 Method Errors.....	139
6.14.2 Processor Errors	141
7. BSA System Monitor Function Reference	143
7.1 Connecting.....	143
7.1.1 IBSASysMon::Connect	143
7.1.2 IBSASysMon::Disconnect.....	144
7.2 Using the System Monitor.....	144
8. BSA.Ctrl Constants.....	146
9. Examples.....	149
9.1 VBScript Example.....	149
9.2 JScript Example.....	151
9.3 MS Excel VBA Example.....	152
9.4 HTML Example	154
9.5 LabVIEW Example	157
9.6 Visual Basic Example	161
9.7 MATLAB Example.....	165
9.8 C# .NET Example	168

2. Dantec Dynamics License Agreement

This License Agreement is concluded between you (either an individual or a corporate entity) and Dantec Dynamics A/S (“Dantec”). Please read all terms and conditions of this License Agreement before opening the disk package. When you break the seal of the software package and/or use the software enclosed, you agree to be bound by the terms of this License Agreement.

2.1 Grant of License

This License Agreement permits you to use one copy of the Dantec software product supplied to you (the “Software”) including documentation in written or electronic form. The Software is licensed for use on a single computer and/or electronic signal processor, and use on any additional computer and/or electronic signal processor requires the purchase of one or more additional license(s). The Software's component parts may not be separated for use on more than one computer or more than one electronic signal processor at any time. The primary user of the computer on which the Software is installed may also use the Software on a portable or home computer. This license is your proof of license to exercise the rights herein and must be retained by you. You may not rent or lease the Software.

2.2 Upgrades

This License Agreement shall apply to any and all upgrades, new releases, bug fixes, etc. of the Software, which are supplied to you. Any such upgrade etc. may be used only in conjunction with the version of the Software you have already installed, unless such upgrade etc. replaces that former version in its entirety and such former version is destroyed.

2.3 Copyright

The Software (including text, illustrations and images incorporated into the Software) and all proprietary rights therein are owned by Dantec or Dantec's suppliers, and are protected by the Danish Copyright Act and applicable international law. You may not reverse assemble, decompile, or otherwise modify the Software except to the extent specifically permitted by applicable law without the possibility of contractual waiver. You are not entitled to copy the Software or any part thereof. However you may either (a) make a copy of the Software solely for backup or archival purposes, or (b) transfer the Software to a single hard disk, provided you keep the original solely for backup purposes. You may not copy the User's Guide accompanying the Software, nor print copies of any user documentation provided in “on-line” or electronic form, without Dantec's prior written permission.

2.4 Limited Warranty

You are obliged to examine and test the Software immediately upon your receipt thereof. Until 30 days after delivery of the Software, Dantec will deliver a new copy of the Software if the medium on which the Software was supplied (e.g. a diskette or a CD-ROM) is not legible. A defect in the Software shall be regarded as material only if it has an effect on the proper functioning of the Software as a whole, or if it prevents operation of the Software. If until 90 days after the delivery of the Software, it is established that there is a material defect in the Software, Dantec shall, at Dantec's discretion, *either* deliver a new version of the Software without the material defect, *or* remedy the defect free of charge *or* terminate this License Agreement and repay the license fee received against the return of the Software. In any of these events the parties shall have no further claims against each other. Dantec shall be entitled to remedy any defect by indicating procedures, methods or uses ("workarounds") which result in the defect not having a significant effect on the use of the Software.

The Software was tested prior to delivery to you. However, software is inherently complex and the possibility remains that the Software contains bugs, defects and inexpediences, which are not covered by the warranty, set out immediately above. Such bugs etc. shall not constitute due ground for termination and shall not entitle you to any remedial action. Dantec will endeavor to correct all bugs etc. in subsequent releases of the Software.

The Software is licensed "as is" and without any warranty, obligation to take remedial action or the like thereof in the event of breach other than as stipulated above. It is therefore not warranted that the operation of the Software will be without interruptions, free of defects, or that defects can or will be remedied.

2.5 Limitation of Liability

Neither Dantec nor its distributors shall be liable for any indirect damages including without limitation loss of profits, or any incidental, special or other consequential damages, even if Dantec is informed of their possibility. In no event shall Dantec's total liability hereunder exceed the license fee paid by you for the Software.

2.6 Product Liability

Dantec shall be liable for injury to persons or damage to objects caused by the Software in accordance with those rules of the Danish Product Liability Act, which cannot be contractually waived. Dantec disclaims any liability in excess thereof.

2.7 Governing Law and Proper Forum

This License Agreement shall be governed by and construed in accordance with Danish law. The sole and proper forum for the settlement of disputes hereunder shall be The Maritime and Commercial Court of Copenhagen (Sø- og Handelsretten i København).

2.8 Questions

Should you have any questions concerning this License Agreement, or should you have any questions relating to the installation or operation of the Software, please contact the authorized Dantec distributor serving your country. You can find a list of current Dantec distributors on our web site: www.dantecdynamics.com

Dantec Dynamics A/S
Tonsbakken 16 – 18
DK - 2740 Skovlunde
Denmark

Tel: +45 44 57 80 00

Fax: + 45 44 57 80 01

Web site: www.dantecdynamics.com

3. Laser Safety

All equipment using lasers must be labeled with the safety information. Dantec Dynamics FiberFlow, FlowLite and FlowExplorer systems use class III and class IV lasers whose beams are safety hazards. Please read the laser safety sections of the documentation for the laser and LDA optics carefully. Furthermore, you must instigate appropriate laser safety measures and abide by local laser safety legislation. Use protective eye wears when the laser is running.

3.1 Danger laser light

Laser light is a safety hazard and may harm eyes and skin. Do not aim a laser beam at anybody. Do not stare into a laser beam. Do not wear watches, jewelry or other blank objects during alignment and use of the laser. Avoid reflections into eyes. Avoid accidental exposure to seculars beam reflections. Avoid all reflections when using a high-power laser.

Screen laser beams and reflections whenever possible. Follow your local safety regulations. You must wear appropriate laser safety goggles during laser alignment and operation.

During alignment, run the laser at low power whenever possible. Since this document is concerned with the BSA Flow Software, there are no direct instructions in this document regarding laser use. Therefore, before connecting any laser to the system, you must consult the laser safety section of the laser instruction manual.

3.2 Precautions

As general precautions to avoid eye damage, carefully shield any reflections so that they do not exit from the measurement area.



4. Introduction

4.1 Preface

The BSA Processor Driver is building on the widespread Component Object Model (COM), which is a binary standard. The COM or ActiveX is a set of services and specifications that allow the user to create modular, object-oriented, customizable and upgrade able, distributed applications using a number of programming languages.

This provides the user with the ability to communicate with a Dantec Dynamics BSA Processor by implementing the BSA Processor Driver in a script- or programming languages that can integrate COM objects. A part of the list is shown below:

- LabVIEW
- C/C++
- Java/JScript
- Visual Basic/VBScript/VBA

The BSA Processor Driver supports a dual interface. That is you can communicate with BSA Processor Driver in two ways. The first way is static invocation, which means there is a contract between the server and the client. The client knows exactly which interfaces there are, and what methods they have. The other way to communicate is dynamic invocation where the client can ask the server which interfaces and methods it has through its automation interface. A lot of programming tools demands the automation interface. But because The BSA Processor Driver supports dual interface the user can choose him self what to use.

4.2 Purpose

So why use the BSA Processor Driver? Often customers have a very special need of different kinds of user interfaces when making measurements, or maybe some customers need to make their own applications communicating with a Dantec Dynamics BSA Processor.

Therefore we have developed this binary block of code, which has an interface to communicate with our processors. Most programming tools have built-in tools to make use of COM very easy. In this way the customer can just implement the BSA Processor Driver in his application without thinking on the code, which lies behind the communication.

This BSA Processor Driver can be used in many situations, just to mention a few:

- Medium- to large-scale wind tunnels where the BSA Processor is only one probe amongst others all controlled by a single computer running e.g. LabVIEW.
- Processes control milieus where the BSA Processor it to perform the same tasks repeatedly using simple batch runs.
- Diagnostic systems where the BSA Processor supervises process environments, looking for changes and act in response to these.
- Institutions who are interested in developing special dedicated applications using our BSA Processor.

4.3 Reading this Manual

This manual primarily acts as a reference manual for the BSA Processor Drivers interface methods. It is intended to provide you with a quick entry to how to use the driver along with an in dept description of all methods and their function.

The driver itself is more or less self-explanatory; all method names are easy understandable and all methods are supplied with online help. Most modern programming- and script languages provide automatic method listings and help texts within the development environment GUI.

This manual provides numerous simple examples accompanied by more complete ready to run example projects on the installation CD-ROM. The examples are primarily shown in script languages since these languages are available in all MS Windows versions through the build-in Windows Script Host (WSH) component. The script languages also resemble macro descriptions of the functionality, – very easy to convert to other languages. All examples are marked with the language type.

If you are a skilled programmer the examples will guide you to a quick understanding of the interface between the BSA Processor Driver and the BSA Processor. If you are new to COM objects we recommend you to seek information in the online help your programming environment or on the web. In all circumstances it will be a good start to have the development environment started next to reading this manual when you get curious looking at the examples.

All methods are described with a prototype of the method call in IDL/C++ and Visual Basic/Script syntax, a parameter listing including parameter direction, and in some cases a remark section for further information. When required the supported processor types are explained.

For all get methods the BSA Processor default value is noted. This is the value that the property initially has at start-up. It is only needed to change the value of properties that are not default correct at start-up.

5. Using the BSA Processor Driver

5.1 Installation

The BSA Processor Driver is available from a CD-ROM or from download from the Dantec Dynamics web download section.

Requirements

- BSA F- or P-series Processor.
- PC with a Pentium (or similar) 1 GHz minimum processor.
- Microsoft® Windows© XP with latest updates.
- Microsoft® Internet Explorer 6 or later with latest security updates.
- 512 MB of RAM minimum.
- 20 MB of available hard-disk space minimum; 100 MB for running acquisitions.
- CD-ROM drive.
- 10/100 Base-T Ethernet adapters with RJ-45 connector.
- Super VGA (800x600) or higher resolution monitor.
- Mouse or compatible pointing device.

Versions

BSA Processor Driver v4.00 and above requires BSA Processor Software v3.00 or newer.

Previous versions of the BSA Processor Driver can be used from BSA Processor Software v2.42.

Installation

Place the CD-ROM in the drive or start the setup.exe and follow the instructions on the screen. During installation you will be asked for the installation location of the driver. We recommend using the default location.

Since the BSA Processor Driver is made of COM objects the disk location is registered in the Windows Registry Database and the components can be accessed through their object **ProgID** or **ClassID**.

The BSA Processor Driver consists of two COM objects: the **BSA.Ctrl** and the **BSA.SysMon.Ctrl**.

ProgID	ClassID
BSA.Ctrl	{83E16B05-CDDD-11D0-89E3-0000C0C27240}

BSA.SysMon.Ctrl	{83E16B06-CDDD-11D0-89E3-0000C0C27240}
-----------------	--

An additional .NET wrapper object is created based on the **BSA.Ctrl** COM object.

Namespace
Dantec.BSA.Driver

The BSA.Ctrl is used to communicate with the BSA Processor. Through this component the processor can be configured, the properties can be changed and data can be acquired.

The BSA.SysMon.Ctrl is a visual control that must be embedded into a COM container or another window. The component displays online information from the BSA Processor.

When the BSA Processor Driver is installed on your machine you can start using the object. It is done in three steps:

1. Create an instance of the components.
2. Get access to the components interfaces.
3. Call the methods on the interfaces.

The next section will describe a quick example of how to use the driver. For more examples please look in the

Examples section in this manual.

5.2 BSA Flow Software Script Generator

This section and the Quick Start and Quick Examples sections give a quick introduction to how to use the driver.

Normally, when starting to use the driver, the best approach is to start getting a quintet with the system, the BSA Processor is through the application software BSA Flow Software. Again pointing out that it is not necessary to use BSA Flow Software when running the driver.

When setting up an experiment BSA Flow Software give a large variety of possibilities, which are out of the scope of this manual, but are described in the BSA Flow Software Installation and User's Guide.

One feature in BSA Flow Software is linking the application to the BSA Processor Driver, and that is the Script Generator. The Script Generator is a tool for automatically generating ready-to-run Visual Basic or Java Scripts for the BSA Processor Driver. The scripts are generated based on the current configuration in BSA Flow Software project.

To access the script generator select the Script option in the output properties of the BSA F/P Application object.

Figure 1 Selecting to run Script Generator for BSA Processor Driver from within BSA Flow Software.

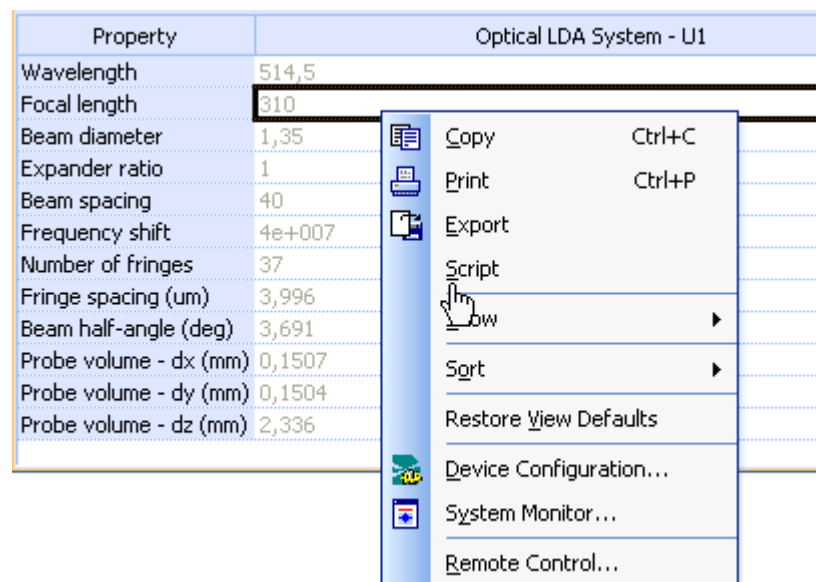
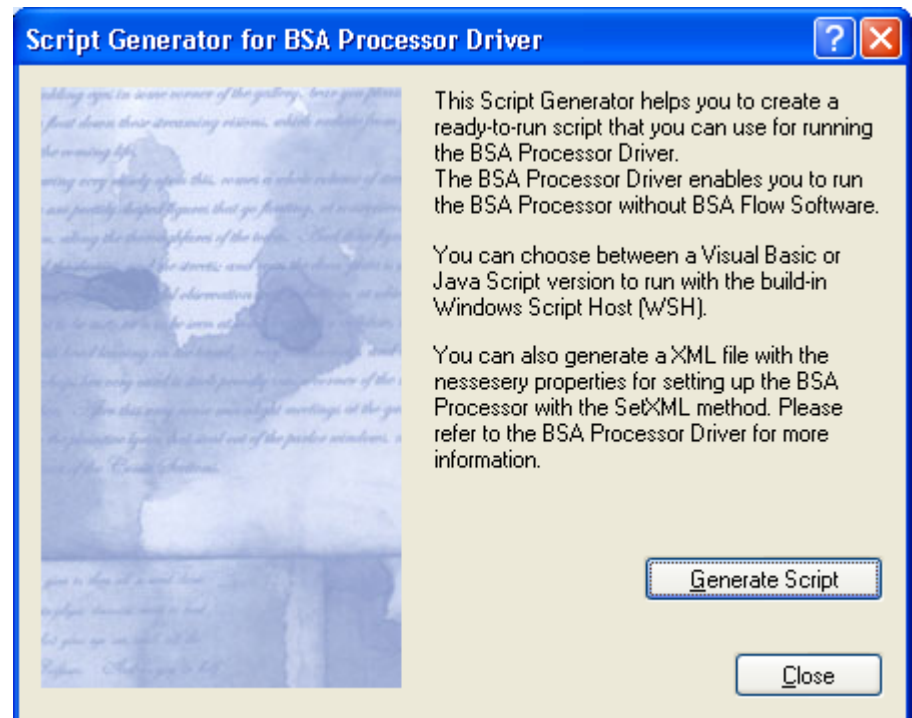


Figure 2 BSA Flow Software Script Generator for BSA Processor Driver.



Press Generate Script to select between Visual Basic Script (VBScript) or Java Script (JScript) generation. The scripts are saved with vbs or js extensions and can be directly run on Windows including the Windows Script Host component, by double clicking on the script file.

A third possibility is to generate a XML file of the current configuration. This file can be directly used in the **SetProperties** method for the driver. By saving different configurations in XML files you can quickly change between configurations and measurements.

5.3 Quick Start

In this section we provide a quick step-by-step approach to use the BSA.Ctrl, and BSA.SysMon.Ctrl.

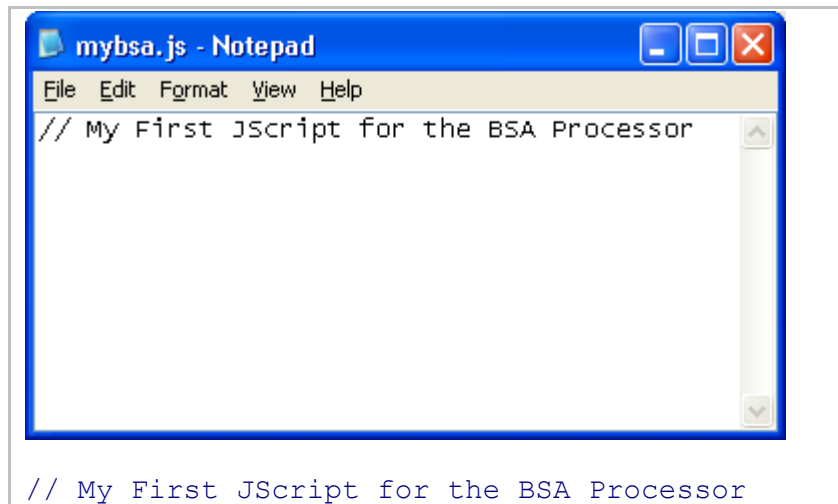
Initial Steps

1. Ensure that you have a BSA Processor with a known IP-address connected to your network or in a peer-to-peer configuration with your controlling PC.
2. The flow signal to the processor should, for this test, be a stable flow with well-known characteristics, like frequency and variance.
3. For this quick example you need to have the Windows Script Hosting (WSH) component installed. You can get the WSH from the Microsoft Download Center www.microsoft.com/downloads, if not already located on your PC.

4. Install the BSA.Ctrl component on your PC, from the installation CD-ROM or from the Dantec Dynamics Download section.

Writing a Script

We will now start writing a small Java Script (JScript) for communicating with the BSA Processor. Open Notepad and write the following line:



Select **Save As...** choose your location on your disk and type the file name “**mybsa.js**”.

First we need to create an instance of the BSA.Ctrl object. We will use this object to access the methods to call the BSA Processor. All the available methods can be seen in the BSA.Ctrl Function Reference section.

```
// My First JScript for the BSA Processor  
var MyBSA = new ActiveXObject("BSA.Ctrl");
```

Now we have an instance of the BSA.Ctrl object called **MyBSA**, and then we need to connect to the processor. We do this through the **MyBSA** object by calling the **Connect** method. This method takes the IP-address of the processor. The second parameter we start by setting to **false**, indicating that an information GUI dialog should indicate the status of the connection. We add the **WScript.Echo** command in the end of the script to indicate when the script is ended. Remember to change the IP-address to the address of your processor.

```
// My first JScript for the BSA Processor  
var MyBSA = new ActiveXObject("BSA.Ctrl");  
MyBSA.Connect("10.10.100.100", false);  
WScript.Echo("My First JScript Ended");
```

Save the file, and go to the Windows Explorer and execute the file. **.js** files should be assigned the WSH component.

During the connection you should now be able to see if the BSA.Ctrl connects to the BSA Processor successfully. If this is not the case please check the processor status, the network connection and the IP-address.

When the BSA.Ctrl connects successfully to the processor we can change the code to connect silently by changing the second parameter in the **Connect** method to **true**.

```
// My first JScript for the BSA Processor
var MyBSA = new ActiveXObject("BSA.Ctrl");
MyBSA.Connect("10.10.100.100", true);
WScript.Echo("My First JScript Ended");
```

Now the configuration for the processor must be set. This must always be done after a connection is established. Here we set the configuration to a 1D LDA on a F-series BSA Processor for simplicity. Calling the **SetProcessor** and **SetConfiguration** methods both with the parameter 0 does this.

```
// My First JScript for the BSA Processor
MyBSA = new ActiveXObject("BSA.Ctrl");
MyBSA.Connect("10.10.100.100", true);
MyBSA.SetProcessor(0);
MyBSA.SetConfiguration(0);
WScript.Echo("My First JScript Ended");
```

Now we are ready to set properties in the processor. The processor wakes up in a default state so you need to change all properties that are specific for your flow. You must now set the bandwidth and center frequency to match your flow. In this sample we set the center frequency to 0.0 MHz and the bandwidth to 11.5 MHz. The first parameter is 0 indicating that these properties will be set for LDA channel 1.

```
// My First JScript for the BSA Processor
MyBSA = new ActiveXObject("BSA.Ctrl");
MyBSA.Connect("10.10.100.100", true);
MyBSA.SetProcessor(0);
MyBSA.SetConfiguration(0);
MyBSA.SetCenterFreq(0, 0.0);
MyBSA.SetBandwidth(0, 11500000);
WScript.Echo("My First JScript Ended");
```

Please refer to the BSA.Ctrl Function Reference section for information about all properties.

Now you can start to prepare an acquisition. If the properties are set correctly you should now be able to acquire data from your processor. In this example we will set the acquisition to acquire 2.000 samples or to run in 10 seconds. The first parameter 0 in these calls now applies to the first coincidence group.

```
// My First JScript for the BSA Processor
MyBSA = new ActiveXObject("BSA.Ctrl");
MyBSA.Connect("10.10.100.100", true);
MyBSA.SetProcessor(0);
MyBSA.SetConfiguration(0);
MyBSA.SetCenterFreq(0, 0.0);
MyBSA.SetBandwidth(0, 11500000);
MyBSA.SetMaxSamples(0, 2000);
MyBSA.SetMaxAcqTime(0, 10);
WScript.Echo("My First JScript Ended");
```

Now you can start the acquisition. The parameter true in the **StartAcq** method indicates that the method should wait for all data to be acquired before continuing. After the acquisition the number of acquired samples is displayed.

```
// My First JScript for the BSA Processor
MyBSA = new ActiveXObject("BSA.Ctrl");
MyBSA.Connect("10.10.100.100", true);
MyBSA.SetProcessor(0);
MyBSA.SetConfiguration(0);
MyBSA.SetCenterFreq(0, 0.0);
MyBSA.SetBandwidth(0, 11500000);
MyBSA.SetMaxSamples(0, 2000);
MyBSA.SetMaxAcqTime(0, 10);
MyBSA.StartAcq(true);
var numSamples = MyBSA.ReadDataLength(0);
WScript.Echo(numSamples);
WScript.Echo("My First JScript Ended");
```

Save the file, and go to the Windows Explorer and execute the file.

Final Step

You have now written your first JScript for controlling the BSA Processor. You can extend the sample with all the functionality of the processor by adding methods from the BSA.Ctrl Function Reference. This quick example does not include the BSA.SysMon.Ctrl to display the System Monitor data. We advise you to look in the samples in the end of this manual, on the installation CD-ROM, and for the latest samples and information on the Dantec Dynamics download section.

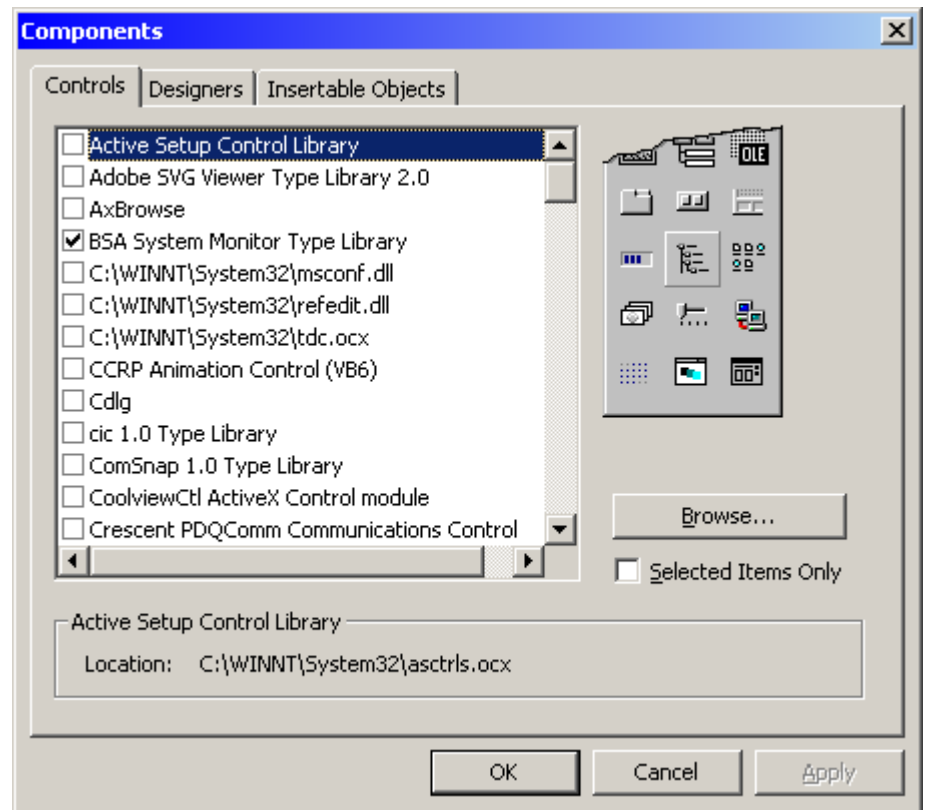
5.4 Quick Examples

We will now go through two different examples that describe how to start using the components. We have chosen the most commonly used development environments used by the community; one in Visual Basic and another one in LabVIEW. It is a requirement that you have installed the component on your PC correctly.

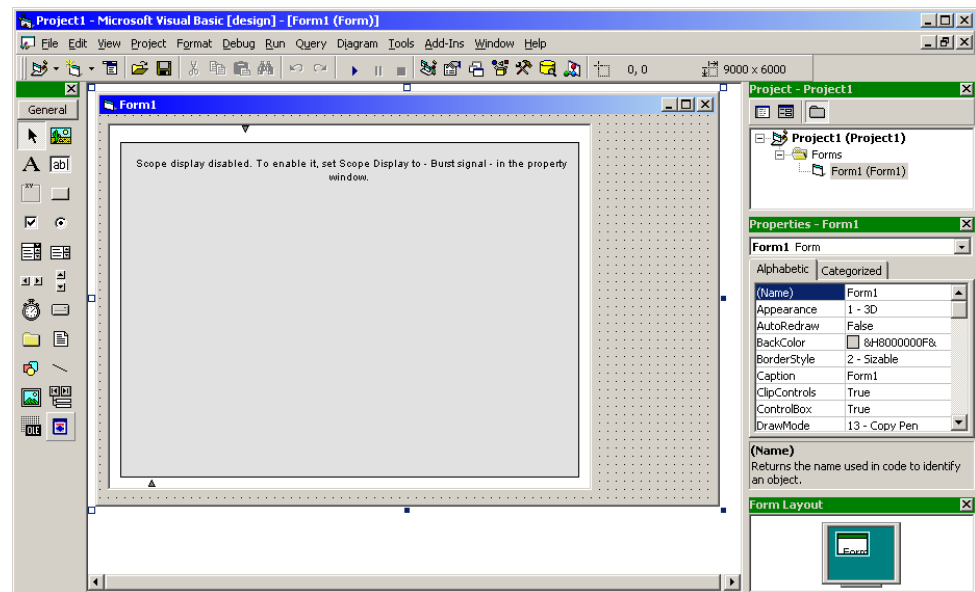
5.4.1 Visual Basic Quick Example

This simple Visual Basic 6 quick example shows how to select the BSA.SysMon.Ctrl in the Visual Basic environment.

1. Create a new blank project.
2. To create a reference to the components open the Component dialog. If the driver is correctly installed the BSA System Monitor control should be visible in the list.

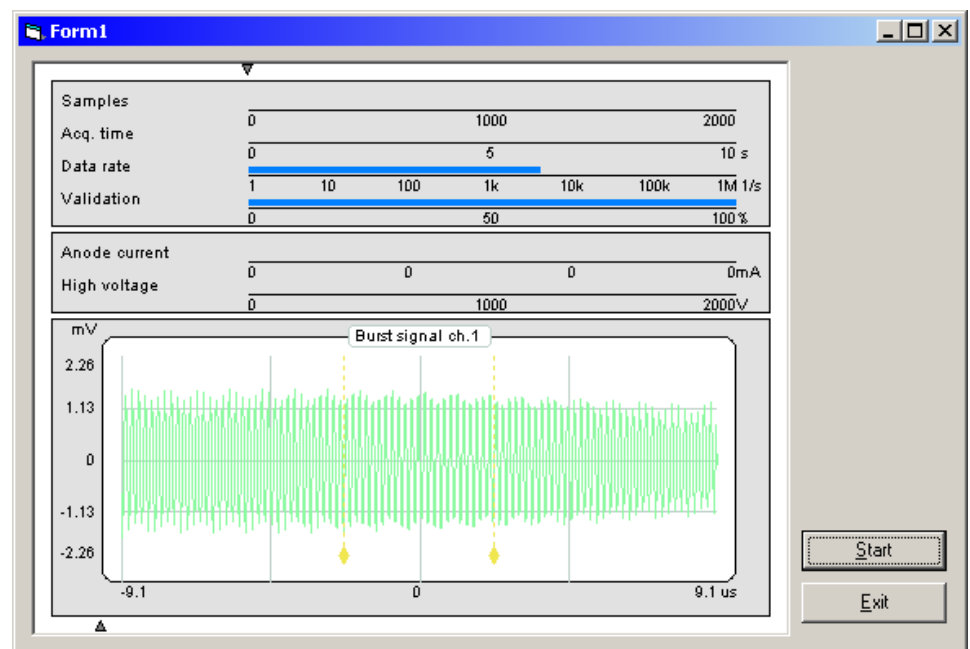


- Place the control onto a form in Visual Basic. And drag the component to the desired size.



- Call the Connect method of the control with the correct ip-address as parameter, and run the program.

The result should look like this:



[Visual Basic]

```
Private Sub cmdStartSysmon_Click()  
SysMon1.Connect ("10.10.100.100")  
End sub
```

5. To reference the BSA.Ctrl component go to the code view and define a variable as following

[Visual Basic]

```
Public MyBSA As New BSACtrl.BSA
```

6. Now you have an object to the interface. With this object you can call all the methods in the component. As an example you can call connect as shown below:

[Visual Basic]

```
MyBSA.Connect "10.10.100.100", 0
```

You should substitute the default IP address "10.10.100.100" with the one used for your BSA processor.

You are now ready to build up your application controlling the BSA Processor. For more information see the enclosed code example.

```
`-----`
` This code example assumes that the BSACtrl
` component has been installed.
` It will then initiate a dataacquisition in
` the BSA processor. Wait until data has been
` acquired and then read data into the array
` variable Velocity (type Variant). Finally
` data is displayed in a listbox.
`-----`

Private Sub cmdGetVelocityData_Click()
    Dim NumberOfSamples as long
    Dim SampleIdx as long
    Dim Velocity()

    `Start acquisition of data
    Call MyBSA.StartAcq(0)

    Do

        `Read group 0
        NumberOfSamples =
            MyBSA.ReadDataLength(0)

        `Wait for samples to become available
    Loop While
        NumberOfSamples < REQUESTED_SAMPLES

    ` Read data from processor...
    `-----`

    ` Display data in a listbox
    List1.Clear

    ` Group=0; LDA_Ch = 0
    Velocity = MyBSA.ReadVelocityData(0,0)

    ` Loop for all samples
    For SampleIdx = 0 To NumberOfSamples-1

        List1.AddItem "Idx: " &
            Format(SampleIdx + 1) &
            "; Velocity: " &
            Format(Velocity(SampleIdx))

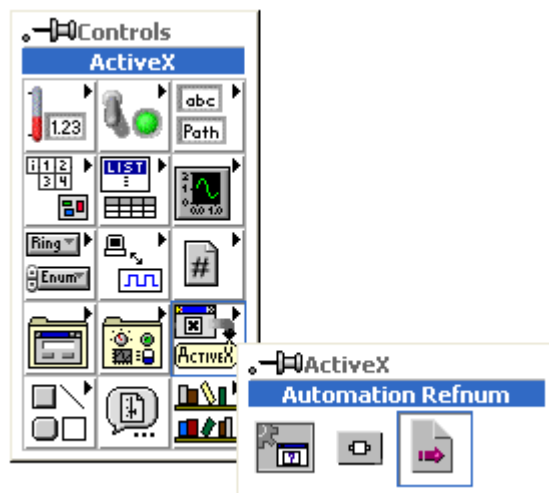
    Next SampleIdx
End Sub
```

5.4.2 LabVIEW Quick Example

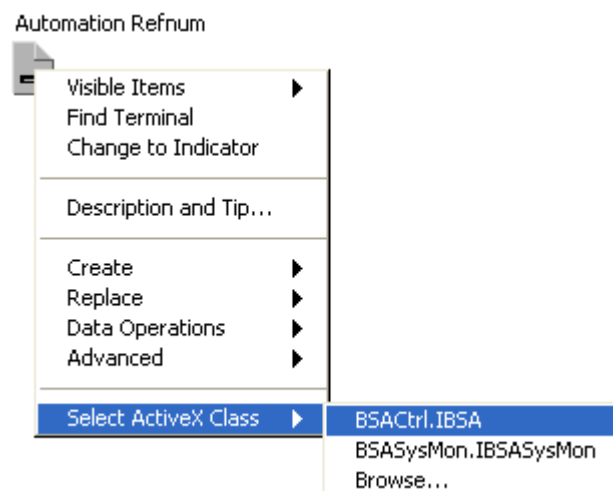
This quick example will provide you with the very basic steps to use the BSA.Ctrl and BSA.SysMon.Ctrl with the LabVIEW graphical development environment. LabVIEW supports automation interfaces; we recommend that version 6 or newer is used with our controls.

Start up LabVIEW and make a new project.

1. From the front panel we will add a reference to the BSA.Ctrl. Right click and select the *Controls| ActiveX| Automation Refnum* palette list.

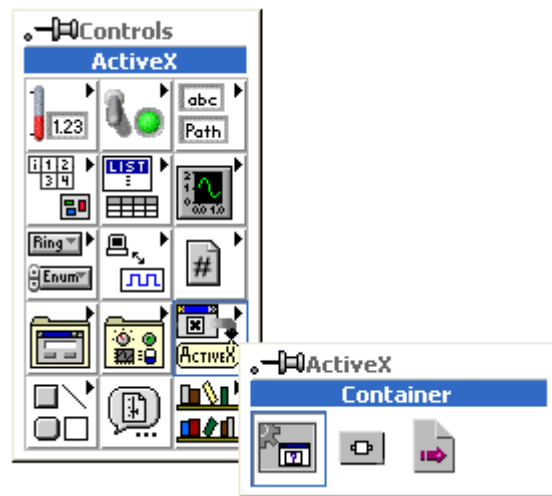


Place the control on the front panel and right-click again for selecting the ActiveX class. If the control does not show directly browse for the dll.

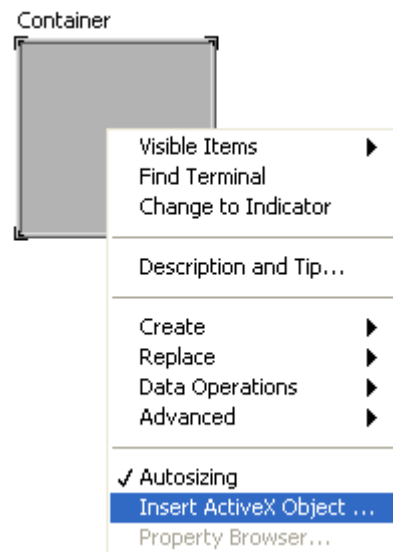


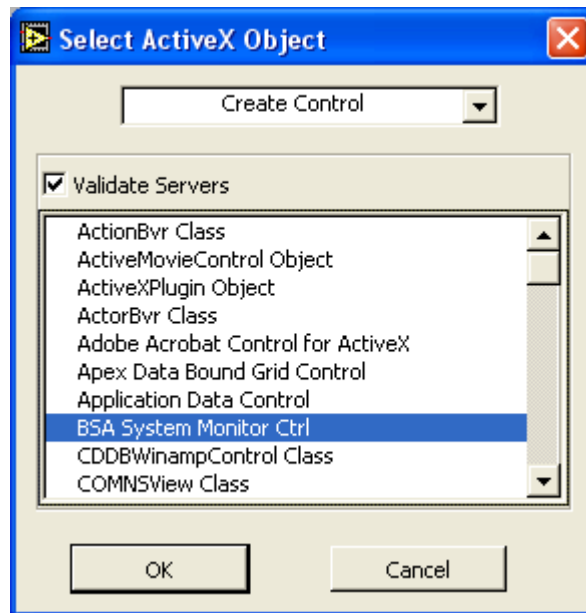
Now you have a reference to the BSA.Ctrl.

2. From the front panel we will now add a reference to the BSA.SysMon.Ctrl. Right click and select the *Controls|ActiveX|Container* palette list.



Place the container control on the front panel and right-click again for inserting the ActiveX object.

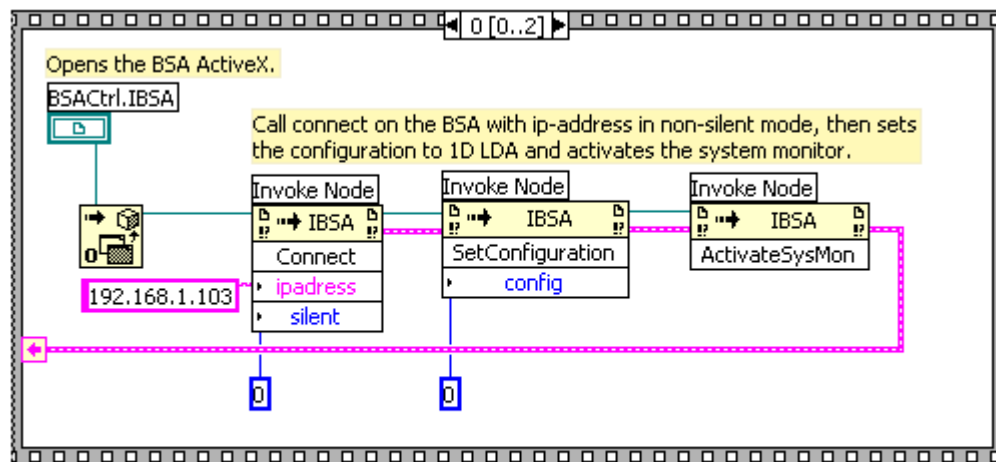




Now you have a reference to the BSA.SysMon.Ctrl.

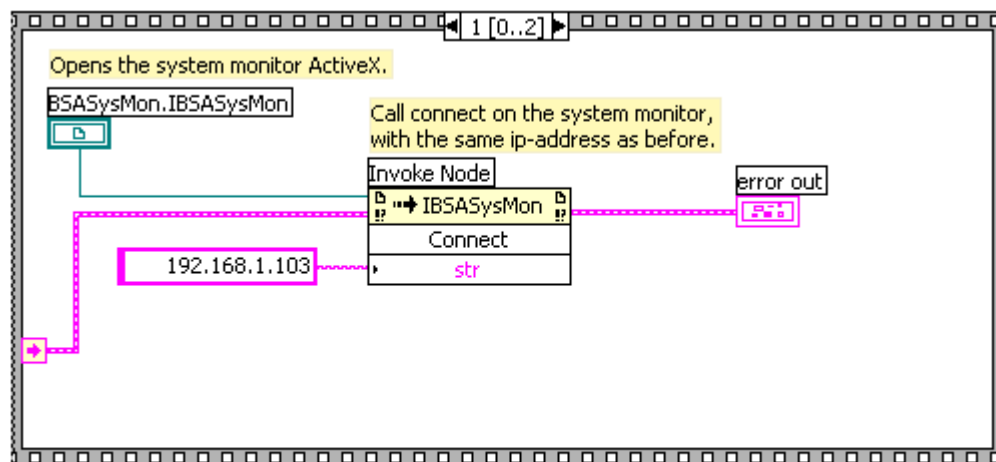
3. In the same way add a Stop button on the front panel that we can use later.
4. Now change to the diagram window.
5. Add three sequence-frames that we can use for scheduling the calls to the components.
6. In the first sequence add the reference to the BSA.Ctrl. Add an Automation Open function along with three Invoke Nodes calling **Connect**, **SetConfiguration** and **ActivateSysMon**. For the Connect method define the IP-address for your BSA Processor, and set the connection mode to non-silent, so that you can see the connection progress, and set the configuration to 1D LDA for simplicity.

Wire up the different nodes like this:



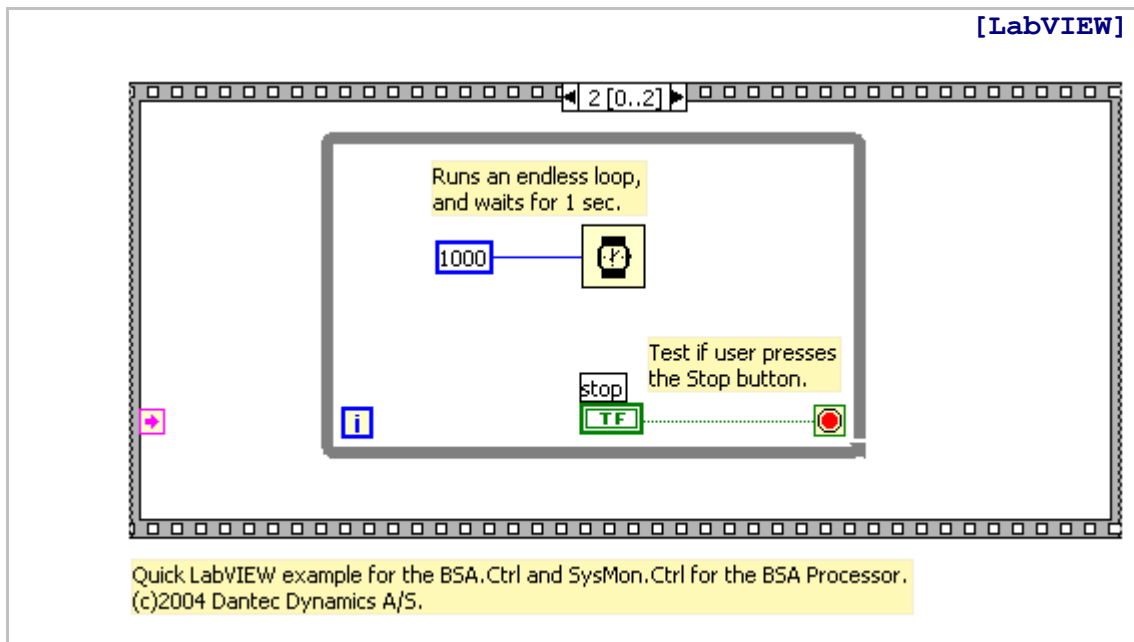
Quick LabVIEW example for the BSA.Ctrl and SysMon.Ctrl for the BSA Processor.
(c)2004 Dantec Dynamics A/S.

7. In the second sequence add the reference to the BSA.SysMon.Ctrl. Add one Invoke Node for the connect method, and specify the same IP-address as before.

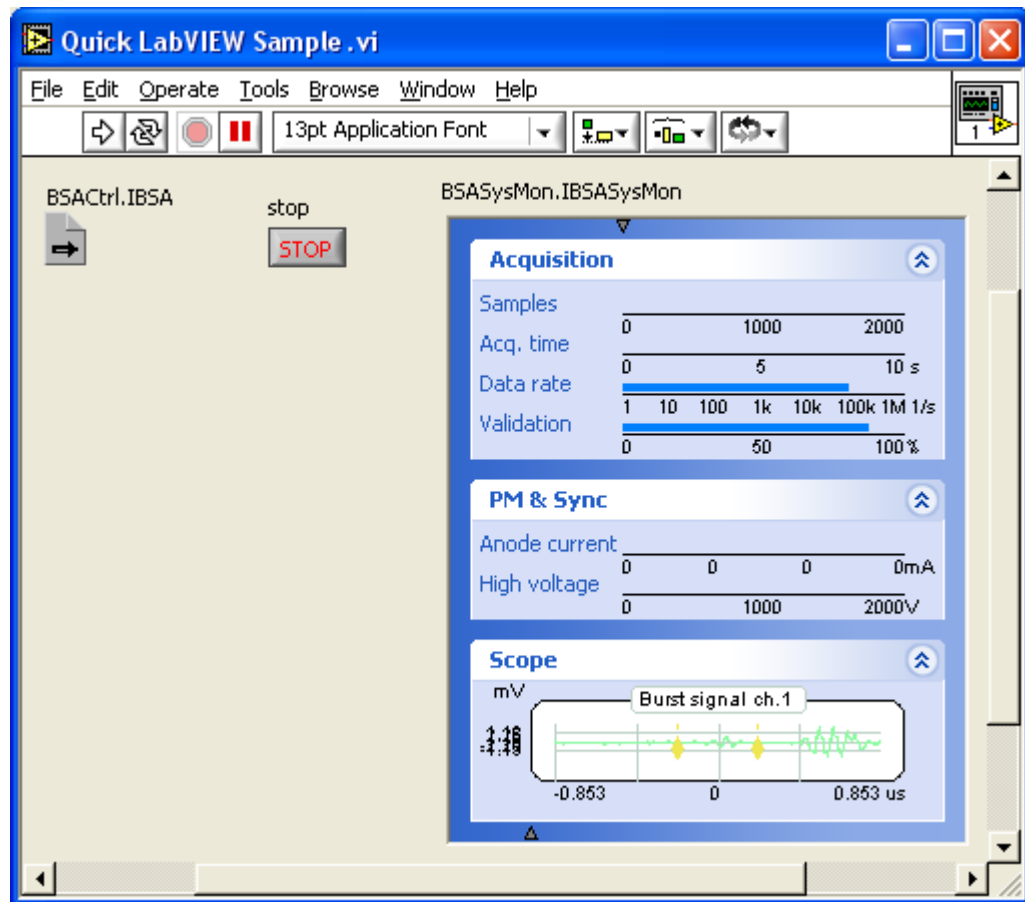


Quick LabVIEW example for the BSA.Ctrl and SysMon.Ctrl for the BSA Processor.
(c)2004 Dantec Dynamics A/S.

8. The purpose of the third sequence is to keep the program alive until we press the Stop button added earlier. Add a while loop inside the sequence and attach it to the criteria connector. Also add a timer, which will ensure that the system pauses the execution, making other programs running in the meanwhile.



9. Now you are ready to run the VI you just made. One nice feature of LabVIEW is the ability to highlight executions as the progress through the VI. Another good idea is to display any errors occurring by connecting the error in and out connectors of the controls and add an error control.



This example can be found together with the installation of the driver. Please refer to the LabVIEW documentation for any questions related to the LabVIEW environment. Also look at the LabVIEW example in the end of this manual, and on the download section on our web site.

5.4.3 C++ Quick Example

This example describes how to implement the component in a C++ project. We are in this case only implementing the raw interface.

In your header file add the following line to your code:

[C++ Header]

```
#import "<path>\BSAProcessorCtrl.dll"  
    named_guids  
using namespace BSACtrl;
```

<path> is the full path to the component, the default location is *C:\Program Files\Dantec Dynamics\BSA Processor Driver\BSAProcessorCtrl.dll*.

Now we can declare a pointer to the interface on the component.

[C++ Header]

```
IBSA* pMyBSA;
```

Before we can call COM library functions we need to initialize the COM library, and create an instance of our object. This is done as following

[C++]

```
::CoInitialize(NULL);  
  
::CoCreateInstance(  
    CLSID_BSA, NULL,  
    CLSCTX_INPROC_SERVER, IID_IBSA,  
    reinterpret_cast<void**>(&pMyBSA));
```

Now we have a pointer pointing to our interface, and its time to build up your application. If you want to connect to the processor you can add the following line to your code.

[C++]

```
pMyBSA->Connect("10.10.100.100", false);
```

For more examples see section 0 and on the installation CD.

6. BSA.Ctrl Function Reference

The purpose of this chapter is to give an understanding of the functions in the BSA Processor Driver. For each function there will be a short introduction followed by a description of the parameters to the given function. For practical examples see the code examples on the CD in the documentation folder. Here you can find a LabVIEW, Visual Basic and MFC project.

The use of the component can be divided into four sections.

1. Connecting
2. Configuring
3. Setting-up
4. Acquiring

The component connects to the BSA processor through a network IP-address. Connecting to the component can be done in two different modes: A normal mode that can be used in GUI applications indicating the connection progress and status showing a dialog box; and a silent mode that must be used in non-GUI applications, scripts and console applications.

After a successful connection a configuration must be defined. The configuration defines the number of channels and coincidence groups. The maximum configuration is defined by the hardware in the BSA Processor. **Currently only LDA configurations are available.**

All the BSA Processor properties can be controlled by the component. Properties can be set and retrieved through simple access functions. Channel methods require a channel index, and group properties require a group index, both starting with 0.

The acquisition is controlled by the **StartAcq** and **StopAcq** methods. The acquisition can be run in a synchronous mode returning the function when data is acquired. The outer limits of the acquisition are determined by the **MaxAcqTime** and **MaxAcqSamples** methods.

Connecting Methods	Description
Connect	Establishing a connection to the BSA Processor using a network IP-address.
Disconnect	Closes the connection to the BSA Processor.
GetConnectState	Detects if the driver is connected to the processor or not.

Configuring Methods	Description

SetProcessor	Specifies the BSA Processor model.
SetConfiguration	Specifies a LDA configuration in the BSA Processor.
SetProperties	Set multiple properties in the BSA Processor using one call to a configuration file.
ActivateSysMon	Starts the System Monitor.
DeactivateSysMon	Stops the System Monitor.
GetStatus	Retrieves the current BSA Processor status.

Acquisition Methods	Description
SetHighVoltageActivation	Determines how the high voltage is handled during acquisition.
SetAnodeCurrentWarningLevel	Specifies the warning level for the anode current during a measurement.
SetDataCollectionMode	Specifies the method in which data is collected.
SetDutyCycle	Sets the length of a continuous signal.
SetDeadTime	Sets the minimum time-window between samples.

Frequency Shift Methods	Description
Set40MHzFreqShiftOutput	Enables the 40 MHz frequency output, on the back of the BSA Processor.
SetVarFreqShiftOutput	Enables the variable frequency output, on the back of the BSA Processor.
SetVarShiftFreq	Sets the variable frequency output.

Synchronization Methods	Description
SetSync1	Assigns input for the sync. 1 signal.
SetSync2	Assigns input for the sync. 2 signal.
SetSync1Edge	Sets sync. 1 trigger edge direction.
SetSync2Edge	Sets sync. 2 trigger edge direction.
SetEncoderReset	Assigns input for the encoder reset signal.
SetEncoder	Assigns input for the encoder clock signal.
SetStartMeasurement	Assigns input for the start measurement signal.
SetStopMeasurement	Assigns input for the stop measurement signal.

SetBurstDetectorEnable	Assigns trigger input for enabling the burst detector.
SetRefClk	Assigns input for external reference clock.
SetBNC	Assigns a signal to the BNC output connector.
SetDSUB	Assigns a signal to one of the DSUB output connectors.
SetShutter	Assigns a signal to the shutter output connector.

Group Methods	Description
SetMaxSamples	Specifies the maximum number of samples to acquire for the group.
SetMaxAcqTime	Specifies the maximum time for acquisition for the group.
SetFilterMethod	Specifies the coincidence filtering method.
SetBurstWindow	Sets the coincidence time window.
SetScopeDisplay	Specifies the scope display type in the System Monitor.
SetScopeTrigger	Sets the scope trigger method or channel in the System Monitor.
SetScopeZoom	Sets the zoom factor of the scope display in the System Monitor.

LDA Methods	Description
SetCenterFreq	Specifies the center frequency of the LDA channel.
SetBandwidth	Specifies the bandwidth of the LDA channel.
SetRecordLengthMode	Sets the record length mode.
SetRecordLength	Specifies the record length.
SetMaxRecordLength	Specifies the maximum record length.
SetHighVoltage	Sets the high voltage level of the LDA channel.
SetSignalGain	Sets the gain for the LDA channel.
SetBurstDetectorSNR	Defines the SNR threshold in the burst detector.
SetLevelValidationRatio	Defines the level of peak validation for the channel.
SetAnodeCurrentLimit	Sets the maximum allowed anode current.

PDA Methods	Description
SetPDACalibration Mode	Set the PDA phase calibration mode.
SetPDAAutoBalancing	Set automatic high voltage balancing for PDA channels.
SetPDAAutoHighVoltage	Sets the high voltage level of the PDA channel.

Data Read Methods	Description
StartAcq	Starts an acquisition.
StopAcq	Aborts a running acquisition.
PreAllocatedData	Allocates necessary data memory.
ReadDataLength	Gets the number of acquired samples.
ReadArrivalTimeData	Reads the arrival time data.
ReadTransitTimeData	Reads the transit time data.
ReadVelocityData	Reads the velocity data.
ReadEncoderData	Reads the encoder data.
ReadSyncDataLength	Gets the number of acquired sync. samples.
ReadSyncArrivalTime Data	Reads the sync. arrival time data.
ReadSyncData	Reads the sync. event data.
ReadDiameterData	Reads PDA diameter data.

Optics Methods	Description
SetOpticsShift	Sets the frequency shift for a channel.
SetOpticsConversion Factor	Sets the conversion factors for the channels.
SetOpticsPhaseFactor	Sets the PDA phase factors.
SetOpticsValidation Band	Sets the PDA validation band.
SetOpticsFringe Direction	Sets the fringe direction.

6.1 Connecting

This section describes the methods that are involved in connecting to the BSA Processor. During operation the BSA Ctrl. component must be connected to the BSA Processor, no other connections can be established at the same time.

6.1.1 IBSA::Connect

Call this method for to connect to the processor with the IP-address given by the parameter. It is optional to run in silent mode or not. You must have a connection to a BSA Processor before trying to call any other function.

```
[C++,IDL]
HRESULT Connect(
    /*[in]*/ BSTR IPAddress,
    /*[in]*/ int Silent);

[Visual Basic 6]
Sub Connect(
    IPAddress As String,
    Silent As Long)
```

Parameters

IPAddress

[in] The IP-address of the processor as it is seen on the network. The syntax consists of a common dotted octet IP-address, for example 10.10.100.100.

Silent

[in] Determines if the connection dialogs should be displayed during the connection process. If true (≠0) the dialogs are displayed otherwise the connection will take place silently.

Remarks

The connection process can take up till several seconds.

If the *Silent* flag is set to **false** error messages and status information will be displayed during connection. When called from within a Win32 console application or from a script language set the silent mode to **true**.

Normally the processor will keep a fixed IP-address. The default IP-address is 10.10.100.100, but can be changed, please refer to the BSA Flow Software Installation and User's Guide. It is also possible to assign the processor a dynamic IP-address if the network supports DHCP, please refer to your network administrator. In this case the IP-address must be the name of the processor, please refer to the BSA Flow Software Installation and User's Guide.

The following sample in VBScript shows how to call connect in silent mode and how to catch any potential connection errors.

[VBScript]

```
' connects to the BSA Processor, resuming
' to the next error handling line on error
On Error Resume Next
MyBSA.Connect "10.10.100.100", True

' handling in case of connection error
If Err.Number<>0 Then
    MsgBox "Error: " & Err.Description
    Err.Clear
End If
```

Using multiple network adapters

When the BSA.Ctrl is used on a Windows PC with multiple network adapters installed the driver must know which adapter to use for communication. Identify the IP-address of the network adapter connected to the BSA Processor from the Network Connections overview in the Windows Control Panel. This IP-address must be specified in the Windows Registry Database at the current location:

[Windows Registry]

```
HKEY_CURRENT_USER\Software\Dantec
Dynamics\BSA Flow Software\B2K
Processor\NICIP = "0.0.0.0".
```

“0.0.0.0” identifies the default adapter on the PC, changing this value to another network adapter IP-address will tell the driver to use this for connecting to the BSA Processor. If you are not comfortable with changing values in the Windows Registry Database please ask your local network administrator for advice.

6.1.2 IBSA::Disconnect

Before shutting down the application remember to disconnect from the processor. This is done calling this function.

[C++,IDL]

```
HRESULT Disconnect();
```

[Visual Basic 6]

```
Sub Disconnect()
```

Parameters

This method has no parameters.

Remarks

When disconnected to the processor, the **GetConnectState** method will return 0.

6.1.3 IBSA::GetConnectState

In order to determine the connection state of the BSA Processor you can call the **GetConnectState** method. It is recommended to call this function after a call to **Connect** to ensure that the connection is established.

```
[C++,IDL]
HRESULT GetConnectState(
    /*[out, retval]*/ int* pConnected);

[Visual Basic 6]
Function GetConnectState(
    ) As Long
```

Parameters

pConnected

[out, retval] A pointer that holds the connected state. *pConnected* is set to the value 1 if there is a connection to the processor, and 0 if there is no connection.

Remarks

The **GetConnectState** method is useful to test a connection., see the **Connect** method.

It is also advisable in a larger application to call this method before any other call to the driver. Calling a get or set method while not connected will generate an error exception.

6.2 Configuring

This section describes the methods necessary to call to configure the BSA Processor prior to setting up the processor.

6.2.1 IBSA::SetProcessor

The BSA Processor comes in two models. BSA Flow Processor, known as the F-series processor, is a LDA processor capable of measuring velocities. BSA Flow and Particle Processor, known as the P-series processor, is a PDA processor measuring both velocity and particle size. Please refer to Table 1 Processor Model.

```
[C++,IDL]
HRESULT SetProcessor(
    /*[in]*/ int Processor)

[Visual Basic 6]
Sub SetProcessor(
    Processor As Long)
```

Parameters

Processor

[in] Specifies the processor type used with the configuration.

Remark

The Processor can be set to one of the following configuration modes:

Table 1 Processor Models

Processor Model	Value
bsaLDA	0
bsaPDA	1

Good programming requires that the **SetProcessor** method must be called before calling **SetConfiguration**. However because of backward compatibility issues the default processor model is set to bsaLDA, and it is in that case not necessary to call this method before **SetConfiguration**.

Defaults

The default processor model is set to bsaLDA = 0.

6.2.2 IBSA::SetConfiguration

Due to different sensitivity of the various receivers, a seeding particle passing through the outskirts of the measuring volume may sometimes generate a velocity sample on one channel, without simultaneously generating a sample on the other(s).

Most calculations involving two or more velocity components require that the velocity samples in question are coincident (i.e. simultaneous).

Although you are not using two or three channels, you must define at least one group holding one channel for a 1D configuration.

The **1D configuration** defines one group holding only channel 1. Velocity samples can arrive at any time.

The **2D coincident configuration** defines one group holding channel 1 and channel 2. Velocity samples must arrive at the same time to each channel in order to give a result.

The **3D coincident configuration** defines one group holding channel 1, channel 2 and channel 3. Velocity samples must arrive at the same time to each channel in order to give a result.

The **2D non-coincident configuration** defines two groups. Group 1 holds channel 1 and group 2 holds channel 2. Velocity samples can arrive at any time to each channel.

The **3D non-coincident configuration** defines three groups. Group 1 holds channel 1, group 2 holds channel 2 and group 3 holds channel 3. Velocity samples can arrive at any time to each channel.

The **3D semi-coincident configuration** defines two groups. Channel 1 and channel 2 are in group 1 and channel 3 is in group two.

Velocity samples from channel 1 and channel 2 must arrive at the same time to each channel in order to give a result in group 1. In group 2 velocity samples can arrive at any time to channel 3.

The BSA processor can be run in different modes. For velocity measurements only the F- and P-series processors can be run in LDA mode. For velocity and particle size measurements only P-series processors can be used in PDA or DualPDA modes. Please refer to Table 2 Configuration Modes.

```
[C++,IDL]
HRESULT SetConfiguration(
    /*[in]*/ int Configuration)

[Visual Basic 6]
Sub SetConfiguration(
    Configuration As Long)
```

Parameters

Configuration

[in] Specifies the configuration for the processor.

Remark

The configuration must always be set after a connection to the processor. You should call the **GetStatus** method after a setting the configuration for handling any errors.

It is necessary to specify the processor mode along with the configuration. BSA Flow Processor (F-series) does not include any PDA phase channels. BSA Particle and Flow Processors (P-series) can be run in either LDA mode, standard PDA mode or in the enhanced DualPDA mode, based on the optical configuration.

DualPDA configurations can only be run with 2 or more velocity channels.

If a PDA or DualPDA configuration is selected, the **SetOpticsPhaseFactors** method must be called prior to running an acquisition.

In case of PDA measurements the **SetProcessor** method must be called with the bsaPDA processor model before calling **SetConfiguration**.

You will not be able to run an acquisition without specifying a valid configuration.

The configuration modes can be set to one of the following configuration modes, depending on the processor model:

Table 2 Configuration Modes

Configuration Mode	Value	Groups × Channels	LDA	PDA
bsa1DLDA	0	1 × 1	X	X
bsa2DLDACOINC	1	1 × 2	X	X
bsa3DLDACOINC	2	1 × 3	X	X
bsa2DL DANONCOINC	3	2 × 1	X	X
bsa3DL DANONCOINC	4	3 × 1	X	X
bsa3DL DASEMICOINC	5	1 × 2 and 1 × 1	X	X
bsa1DPDA	6	1 × 1		X
bsa2DPDACOINC	7	1 × 2		X
bsa3DPDACOINC	8	1 × 3		X
bsa2DP DANONCOINC	9	2 × 1		X
bsa3DP DANONCOINC	10	3 × 1		X
bsa3DP DASEMICOINC	11	1 × 2 and 1 × 1		X
bsa2DDUALPDACOINC	12	1 × 2		X
bsa3DDUALPDACOINC	13	1 × 3		X
bsa3DDUALPDASEMICOINC	14	1 × 2 and 1 × 1		X

When setting and retrieving properties for the channels and groups the index must be applied to the method called. For setting the bandwidth on channel 1 to 90 MHz, the index 0 must be used in the call to specify the channel, and the same applies to group properties.

This VBScript sample sets the configuration to 2D LDA non-coincidence, resulting in two coincidence groups. The group indices are thereafter used to specify the maximum number of samples that are to be acquired in each group.

[VBScript]

```
' set configuration to 2D LDA non-coin.
MyBSA.SetProcessor 0
MyBSA.SetConfiguration 3
' set max. number of samples to 20.000
' in the first group,
MyBSA.SetMaxSamples 0, 20000
' and the max. number of samples to 5.000
' in the second group
MyBSA.SetMaxSamples 1, 5000
```

This VBScript sample sets that center frequency and bandwidth for each channel in a 3D LDA coincident setup.

[VBScript]

```
' set configuration to 3D LDA coin.
MyBSA.SetProcessor 0
MyBSA.SetConfiguration 2
' set the bandwidth to 45MHz on ch. 1
MyBSA.SetBandwidth 0, 45000000
' set the bandwidth to 45MHz on ch. 2
MyBSA.SetBandwidth 1, 45000000
' set the bandwidth to 90MHz on ch. 3
MyBSA.SetBandwidth 2, 90000000
```

This VBScript sample sets that center frequency and bandwidth for each channel in a 1D PDA coincident setup.

[VBScript]

```
' set configuration to 1D PDA coin.
MyBSA.SetProcessor 1
MyBSA.SetConfiguration 6
' set PDA optical phase factors
MyBSA.SetOpticsPhaseFactors 3.1, 2.8
' set the bandwidth to 45MHz on ch. 1
MyBSA.SetBandwidth 0, 45000000
```

Defaults

Since this function must be called before setting any other parameters in the processor, no default values are specified.

6.2.3 IBSA::SetProperties

This method makes it possible to set multiple properties with one single method call. The properties must be defined and saved into a XML file structure.

This XML file structure can be generated directly by the BSA Flow Software based on the current configuration in the application; which is especially convenient when you want to run and test your configuration in BSA Flow Software before applying it to the BSA Processor Driver, see section BSA Flow Software Script Generator. Or it can be hand-coded following the rules of the XML file structure describe in this section.

```
[C++,IDL]
HRESULT SetProperties(
    /*[in]*/ BSTR XMLFile);

[Visual Basic 6]
Sub SetProperties(
    XMLFile As String)
```

Parameters

XMLFile

[in] Specifies the path and filename of the XML file containing the properties.

Remarks

The XML file structure contains tags (known from HTML) describing sections, properties and values. The structure is divided into three parts:

<information> [optional]

This section is optional. It contains information about from were the properties originate, who created them, and when. Since this section is optional the driver does not read this section, and it is only for traceability and information.

<setup> [optional]

This section is optional. If this section is not used the configuration of the processor must be specified using the **SetProcessor** and **SetConfiguration** methods prior to calling this method. However, if this section exists the driver will use the setup information to re-configure the processor. The section informs the driver about the processor type, and measurement configuration.

```
<processor>[processor model]</processor>
<configuration>[configuration
    mode]</configuration>
<dimension>[number of channels]</dimension>
```

processor

The processor element specifies what kind of BSA Processor to use for the current setup. The valid processor models can be found in Table 1 Processor Model. If this element is not specified LDA processor model is assumed.

configuration

The configuration element specifies the configuration model for the setup. If this element is not specified in the setup no configuration will be set in the processor. The valid configuration modes can be found in Table 2 Configuration Modes.

dimension [not used]

The dimension element specifies the number of LDA channels defined in the configuration.

<properties>

The properties section contains all the properties of the processor. The section must contain one or more property tags. Each property includes three string attributes; a name, an instance, and a value.

```
<property name="[propertyname]"
  instance="[channel/group index]"
  value="[propertyvalue]" />
```

name

The name attribute specifies which property to set. The names are the same as the names of the methods for the driver, without the Set or Get prefix. E.g. the center frequency can be set using the "SetCenterFreq" method; hence the property name is "CenterFreq".

instance

The instance attribute indicates the index of the property. This can have different meanings, most often either the channel index or the group index. E.g. to set a property for the first LDA channel the index must be "0", for the second channel "1" etc., and the same applies to PDA channels and coincidence groups.

value

The value attribute gives the value of the property. Since all the attributes are string types, the values must be formatted accordingly. The values use the same units as the corresponding methods. E.g. the bandwidth property value must be specified in Hz.

<optics>

The optics section contains the optical properties that must be known by the processor. The section must contain one or more lda or pda tags. Each lda's or pda's includes three string attributes; a name, an instance, and a value.

```
<lda name="[opticsname]"
```

```

instance="[optical system index]"
value="[opticsvalue]" />
<pda name="[opticsname]"
instance="[phase index]"
value="[opticsvalue]" />

```

name

The name attribute specifies which optical property to set. The names are the same as the names of the methods for the driver, without the SetOptics prefix. E.g. the optical shift can be set using the "SetOpticsShift" method; hence the optical property name is "Shift".

A list of possible lda and pda names are:

<lda name="ConversionFactor" ...

<lda name="Shift" ...

The instance defines the index of the optical system, 0 for first beam pair, 1 for the second etc. The value gives the value.

<pda name="PhaseFactor" ...

The instance defines the index of the phase channel. The phase channels are placed in index 1 and 2. The value gives the value.

<pda name="ValidationBand" ...

<pda name="FringeDirection" ...

The instance is not used. The value gives the value.

instance

As seen above the instance can have different meanings; either it specifies the index of the optical system (beam pairs) or it specifies the phase channel index. E.g. to set a property for the first LDA channel the index must be "0", for the second channel "1" etc., and the same applies to groups.

value

The value attribute gives the value of the optical property. Since all the attributes are string types, the values must be formatted accordingly. The values use the same units as the corresponding methods. E.g. the optical shift frequency value must be specified in Hz.

The following is an example XML file generated by BSA Flow Software:

```

<?xml version="1.0" encoding="windows-1252"
    standalone="yes" ?>
- <!-- XML Driver Setup from BSA Flow
    Software -->
- <driver
    xmlns="http://www.dantecdynamics.com/
    bsa/driver">
- <projectproperties>
    <version>04.00.00.00</version>
    <username>MyName</username>
    <userinitials>MyInitials</userinitials>
    <projectpath>C:\DOCUMENTS AND SETTINGS\
        MY\MY DOCUMENTS\MY FLOW PROJECTS\
        XML.lda</projectpath>
    <projectname>My Project</projectname>
    <projecttitle />
    <projectcomments />
</projectproperties>
- <setup>
    <processor>1</processor>
    <configuration>2</configuration>
    <dimension>3</dimension>
    <setup>
- <properties>
    <property name="MaxSamples"
        instance="0" value="5000" />
    <property name="MaxAcqTime"
        instance="0" value="30.0000" />
    <property name="CenterFreq"
        instance="0" value="-115000.0000" />
    <property name="Bandwidth"
        instance="0" value="937500.0000" />
    <property name=" CenterFreq "
        instance="1" value="115000.0000" />
    <property name="Bandwidth"
        instance="1" value="937500.0000" />
    <property name=" CenterFreq "
        instance="2" value="-465000.0000" />
    <property name="Bandwidth"
        instance="2" value="1875000.0000" />
</properties>
- <optics>
    <lda name="ConversionFactor"
        instance="0" value="5.76e-006" />
    <lda name="Shift"
        instance="0" value="4e+007" />
    <lda name=" ConversionFactor "
        instance="1" value="5.335e-006" />
    <lda name=" Shift"
        instance="1" value="4e+007" />
    <lda name=" ConversionFactor "
        instance="2" value="2.55e-006" />
    <lda name=" Shift"

```

```
        instance="2" value="4e+007" />
    </optics>
</driver>
```

Another hand-coded example shows how simple it is to define multiple processor properties:

```
[XML]

<?xml version="1.0" ?>
- <driver>
- <properties>
  <property name="MaxSamples"
    instance="0" value="10000" />
  <property name="MaxAcqTime"
    instance="0" value="10" />
</properties>
</driver>
```

6.2.4 IBSA::ActivateSysMon

The System Monitor requires that the processor is in a special acquisition mode where only the System Monitor is updated with data. When calling this function this special acquisition mode is activated.

This function is closely related to the **ISysMon** interface, please refer to the chapter:

```
[C++,IDL]  
HRESULT ActivateSysMon();
```

```
[Visual Basic 6]  
Sub ActivateSysMon()
```

Parameters

This method has no parameters.

Remarks

This sample in a HTML VBScript block shows how to connect to the BSA Processor and start the System Monitor. This block of code can be added to a HTML page that also contains the BSA.SysMon.Ctrl. to automatically start the System Monitor when the page is loaded.

[HTML VBScript]

```
<script language="vbscript">  
Set MyBSA = CreateObject("BSA.Ctrl")  
  
' connects to the BSA Processor  
MyBSA.Connect "10.10.100.100", True  
  
' specify a 1D LDA configuration  
MyBSA.SetProcessor 0  
MyBSA.SetConfiguration 0  
  
' activates the system monitor  
MyBSA.ActivateSysMon  
</script>
```

6.2.5 IBSA::DeactivateSysMon

When finished with the System Monitor, remember to deactivate the data update of System Monitor; calling this method does this.

```
[C++,IDL]  
HRESULT DeactivateSysMon();
```

```
[Visual Basic 6]  
Sub DeactivateSysMon()
```

Parameters

This method has no parameters.

Remarks

After this function is called the System Monitor will not be able to display any more online data.

6.2.6 IBSA::GetStatus

This function is used to retrieve the last errors or events send from the processor.

```
[C++,IDL]
HRESULT GetStatus(
    /*[out, retval]*/ BSTR *pStatus);

[Visual Basic 6]
Function GetStatus(
    ) As String
```

Parameters

pStatus

[out, retval] Description of last event or error received from the processor.

Remarks

Use this method to check for status information send from the processor during setup and/or acquisition. Calling this method consecutively will return more status information if available.

If you are using events, you should call this function every time the **onStatus** event occurs.

For correct error handling you should always call the **GetStatus** method after calls to the **SetConfiguration** and **StartAcq** methods.

Please refer to the

Event Handling section for information about how to use this method.

6.3 Advanced acquisition settings

The functions described in this section are either used to control the High Voltage supply to the photo multipliers or the way of collecting data.

6.3.1 IBSA::SetHighVoltageActivation

High voltage activation can be set to Automatic or Manual. In the former case, the photo-multiplier high voltage is switched on when data acquisition is taking place. Automatic is recommended.

```
[C++,IDL]
HRESULT SetHighVoltageActivation(
    /*[in]*/ int Activation);

[Visual Basic 6]
Sub SetHighVoltageActivation(
    Activation As Long)
```

Parameters

Activation

[in] The high voltage activation state to be set.

Remarks

The *Activation* parameter can be set to one of the following constants:

Table 3 Voltage Activation Constants

Constant	Value
bsaOFF	0
bsaAUTOMATIC	1
bsaHV	2

If set to bsaAUTOMATIC the high voltage will also automatically be turned on and off when the System Monitor is started.

6.3.2 IBSA::GetHighVoltageActivation

This method gets the current state of the high voltage activation setting, set by the **SetHighVoltageActivation**.

```
[C++,IDL]
HRESULT GetHighVoltageActivation(
    /*[out, retval]*/ int* pActivation);

[Visual Basic 6]
Function GetHighVoltageActivation(
    ) As Long
```

Parameters

pActivation

[out, retval] A pointer to an int that is to hold voltage activation state. Refer to Table 3 Voltage Activation Constants for possible return values.

Default

The default high voltage activation mode is automatic
bsaAUTOMATIC = 1.

6.3.3 IBSA::SetAnodeCurrentWarningLevel

Anode warning level can be set from 50 to 150 %. The setting determines at what level the System monitor indicates a warning about the photo-multiplier anode current. A protection circuit reduces the photo-multiplier high voltage if the mean anode current exceeds 100 % of the Anode current limit (defined under LDA settings). The peak anode current can exceed this limit during short intervals; hence the warning level can be set to exceed 100 %.

```
[C++,IDL]
HRESULT SetAnodeCurrentWarningLevel (
    /*[in]*/ int Level);

[Visual Basic 6]
Sub SetAnodeCurrentWarningLevel (
    Level As Long)
```

Parameters

Level

[in] The anode current warning level to be set. *Level* can be set in the range from 50 to 150 %.

6.3.4 IBSA::GetAnodeCurrentWarningLevel

This method gets the current anode current warning level.

```
[C++,IDL]
HRESULT GetAnodeCurrentWarningLevel (
    /*[out, retval]*/ int* pLevel);

[Visual Basic 6]
Function GetAnodeCurrentWarningLevel (
    ) As Long
```

Parameters

pLevel

[out, retval] A pointer to an int that is to hold the anode current warning level. *pLevel* is in the range from 50 to 150 %.

Default

The default anode current warning level is 90 %.

6.3.5 IBSA::SetDataCollectionMode

The BSA Processor supports up to four different data collection modes: burst mode, continuous mode, dead-time mode and external burst triggering mode.

During normal use **burst mode** or **dead-time mode** is used, – briefly explained: **Burst mode** is used when high temporal resolution is required, e.g. if the required result is auto-correlation or turbulence spectrum data, or in connection with rotating machinery. **Dead-time mode** can be used to avoid over-sampling the velocity information, and thereby eliminate velocity bias. It is typically used when the required result is the moments of the velocity distribution; mean, RMS, skewness, or flatness. It has the additional advantage of reducing the amount of data, compared to the other modes.

In **dead-time mode**, the time axis is divided into intervals. In each interval, only the first Doppler burst is processed. The interval is specified in the dead-time property.

Burst mode is used when the photo-detector signal consists of discrete bursts from seeding particles, and when a high data rate is required. There is one measurement per detected burst, producing arrival time, transit time, and velocity information (and optional encoder data).

Continuous mode is used if the Doppler signals appear as quasi-continuous signals. This is typically the case when measuring the velocity of a solid surface. In continuous mode, the burst detector is not used, and the photo-detector signal is processed continuously. Several measurements can be taken during each burst. Auto-adaptive record length is not applicable in this mode.

During **external burst triggering mode** it is possible to use an external burst detector. The external device is connected at the rear panel to the connector specified in the synchronization input signal. Input Signals list under the Burst detector signal settings. TTL levels are used.

```
[C++,IDL]
HRESULT SetDataCollectionMode(
    /*[in]*/ int Mode);

[Visual Basic 6]
Sub SetDataCollectionMode(
    Mode As Long)
```

Parameters

Mode

[in] Specifies the data collection mode.

Remarks

Mode can be set to one of the following constants:

Table 4 Data Collection Mode Constants

Constant	Value
bsaBURST	0
bsaCONTINUOUS	1
bsaDEADTIME	2

Please note that only burst mode is available in PDA and DualPDA mode.

Dependencies

Depending of the data collection mode the dead-time or duty-cycle for the continuous mode must be set. Use the **SetDutyCycle** method to set the duty-cycle when continuous mode is set, and use the **SetDeadTime** method to set the values of the dead-time mode.

6.3.6 IBSA::GetDataCollectionMode

This method returns the current state of the data collection mode.

```
[C++,IDL]
HRESULT GetDataCollectionMode(
    /*[out, retval]*/ int* pMode);

[Visual Basic 6]
Function GetDataCollectionMode(
    ) As Long
```

Parameters

pMode

[out, retval] A pointer to an int that is to hold the current mode. Refer to Table 4 Data Collection Mode Constants for possible values.

Default

The default data collection mode is burst mode bsaBURST = 0.

6.3.7 IBSA::SetDutyCycle

Duty cycle is only applicable to continuous mode. The duty cycle is defined as the percentage of records transferred to FFT processing. It is used in experiments with a very high data rate to prevent an input buffer overflow.

```
[C++,IDL]
HRESULT SetDutyCycle(
    /*[in]*/ float Cycle);

[Visual Basic 6]
Sub SetDutyCycle(
    Cycle As Single)
```

Parameters

Cycle

[in] Specifies the duty cycle in percentage. *cycle* can be set to 3.25 % 6.5 % 12.5 % 25.0 % 50.0 % or 100.0 %.

Remarks

This setting only has effect when the data collection mode is set to continuous mode in the **SetDataCollectionMode** method.

6.3.8 IBSA::GetDutyCycle

This method returns the current value of the duty-cycle.

```
[C++,IDL]
HRESULT GetDutyCycle(
    /*[out, retval]*/ float* pCycle);

[Visual Basic 6]
Function GetDutyCycle(
    ) As Single
```

Parameters

pCycle

[out, retval] A pointer to a float that is to hold the current duty cycle percentage. On return *cycle* is set to 3.25 % 6.5 % 12.5 % 25.0 % 50.0 % or 100.0 %.

Default

The default duty-cycle is 100 %.

6.3.9 IBSA::SetDeadTime

Dead-time is only applicable to the dead-time data collection mode and defines the length of the time interval, from which to process the first burst.

```
[C++,IDL]
HRESULT SetDeadTime(
    /*[in]*/ float Time);

[Visual Basic 6]
Sub SetDeadTime(
    Time As Single)
```

Parameters

Time

[in] Specifies the dead time in seconds. *Time* can be set in the range from 10 μ sec. to 10 sec.

Remarks

This setting only has effect when the data collection mode is set to dead-time mode in the **SetDataCollectionMode** method.

6.3.10 IBSA::GetDeadTime

This method returns the current value of the dead-time.

```
[C++,IDL]
HRESULT GetDeadTime(
    /*[out, retval]*/ float* pTime);

[Visual Basic 6]
Function GetDeadTime(
    ) As Single
```

Parameters

pTime

[out, retval] A pointer to a float that is to hold the dead time setting. On return *pTime* is set in the range from 10 μ sec. to 10 sec.

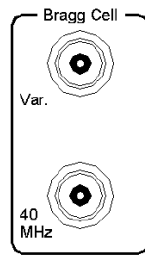
Default

The default dead-time is 100 μ sec.

6.4 Frequency shift

The functions in this section control the signals on the Bragg cell connectors at the rear panel of the BSA. These can be used individually or in combination.

The connector **40 MHz** is used with LDA optics using 40 MHz optical frequency shift, such as most Dantec Dynamics FiberFlow and FlowLite systems, and in dual Bragg cell systems with one cell operating at 40 MHz.



The connector **Var.** is used with LDA optics using other drive frequencies to the Bragg cell, like the 80 MHz FlowExplorer optics and in dual-Bragg cell systems with the first Bragg cell operating at 40 MHz.

6.4.1 IBSA::Set40MHzFreqShiftOutput

The 40 MHz frequency shift can be Enable (default) or Disable. Enable activates the BNC connector named Bragg cell 40 MHz. Disable disconnects the 40 MHz signal from the connector.

```
[C++,IDL]
HRESULT Set40MHzFreqShiftOutput(
    /*[in]*/ int Output);

[Visual Basic 6]
Sub Set40MHzFreqShiftOutput(
    Output As Long)
```

Parameters

Output

[in] Specifies the state of the 40 MHz output.

Remarks

Output can be set to one of the following constants:

Table 5 40 MHz Frequency Shift Output Constants

Constant	Value
bsaOFF	0
bsaON	1

6.4.2 IBSA::Get40MHzFreqShiftOutput

Gets the current state of the 40 MHz frequency output, set by the **Set40MHzFreqShiftOutput** method..

```
[C++,IDL]
HRESULT Get40MHzFreqShiftOutput(
    /*[out, retval]*/ int* pOutput);

[Visual Basic 6]
Function Get40MHzFreqShiftOutput(
    ) As Long
```

Parameters

pOutput

[out, retval] A pointer to an int that is to hold the 40 MHz output state. Refer to Table 5 40 MHz Frequency Shift Output Constants for possible values.

Default

The default setting of the 40 MHz shift frequency output is on bsaON = 1.

6.4.3 IBSA::SetVarFreqShiftOutput

Variable frequency shift directs a Bragg cell driver signal to the BNC connector named Bragg cell Var. The drive frequency is defined by the property Shift frequency.

```
[C++,IDL]
HRESULT SetVarFreqShiftOutput(
    /*[in]*/ int Output);

[Visual Basic 6]
Sub SetVarFreqShiftOutput(
    Output As Long)
```

Parameters

Output

[in] Specifies the state of the variable frequency output.

Remarks

Output can be set to one of the following constants:

Table 6 Variable Frequency Shift Output Constants

Constant	Value
bsaOFF	0
bsaON	1

6.4.4 IBSA::GetVarFreqShiftOutput

This method gets the state of variable frequency output, set by the **SetVarFreqShiftOutput** method.

```
[C++,IDL]
HRESULT GetVarFreqShiftOutput(
    /*[out, retval]*/ int* pOutput);

[Visual Basic 6]
Function GetVarFreqShiftOutput(
    ) As Long
```

Parameters

pOutput

[out, retval] A pointer to an int that is to hold the variable frequency output state. Refer to Table 6 Variable Frequency Shift Output Constants for possible return values.

Default

The default setting of the variable shift frequency output is off
bsaOFF = 0.

6.4.5 IBSA::SetVarShiftFreq

Variable shift frequency defines the frequency of the driver signal at the Bragg Cell Var. connector on the back of the BSA Processor. The variable frequency signal must be enabled using the **SetVarFreqShiftOutput** method.

```
[C++,IDL]
HRESULT SetVarShiftFreq(
    /*[in]*/ int Shift);

[Visual Basic 6]
Sub SetVarShiftFreq(
    Shift As Long)
```

Parameters

Shift

[in] Specifies the frequency shift in Hz. *Shift* can be set in the range from 30 MHz to 120 MHz.

6.4.6 IBSA::GetVarShiftFreq

This method gets the current variable frequency shift.

```
[C++,IDL]
HRESULT GetVarShiftFreq(
    /*[out, retval]*/ int* pShift);

[Visual Basic 6]
Function GetVarShiftFreq(
    ) As Long
```

Parameters

pShift

[out, retval] A pointer to an int that is to hold the variable frequency shift. On return *pShift* is set in the range from 30 MHz to 120 MHz.

Default

The default setting of the variable shift frequency output is 40 MHz.

6.5 Synchronization Signals Settings

The BSA has a number of input and output connectors for synchronization and handshake with external equipment. The software controls the routing of these signals to/from the connectors on the rear panel through the synchronization input signals.

6.5.1 Dependencies

All processors do not support the input and output synchronization options. The following table shows the dependencies.

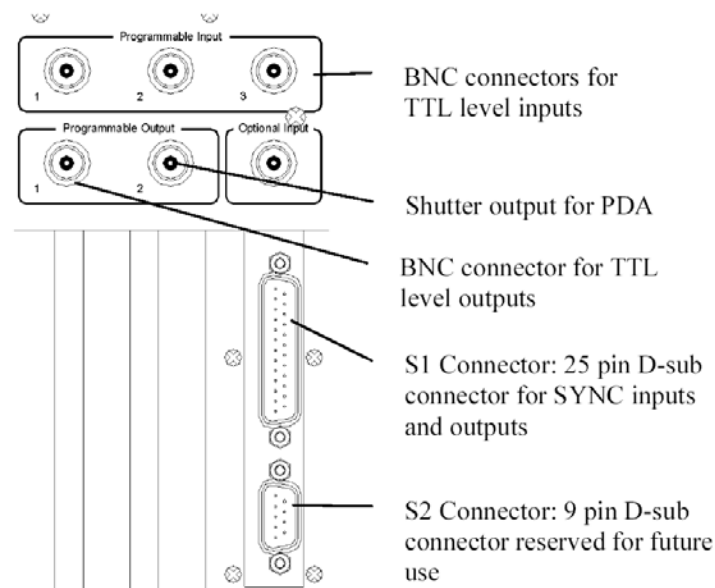
Table 7 Processor Models and Synchronization Capabilities

Processor Model	Remarks
F50, F60, F80	Only supported through the optional synchronization option.
F70	All synchronization is natively supported.

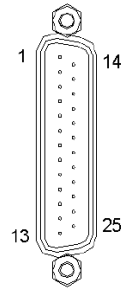
6.5.2 Output Signals settings

Note that some processors requires the 62N560 Synchronization option to make use of these inputs and outputs

The synchronization-input connectors are the three BNC connectors named Programmable Input 1, 2 and 3, and the 25-pin D-sub connector underneath these as shown on the figures below. The Optional Input connector is reserved for future extensions of functionality.



The pin numbering of the connector S1 is shown in the table below. Please note that some of the pins are assigned to synchronization outputs, which are described in the next section.



Pin	Signal	Pin	Signal
1	Out 1	14	Ground
2	Out 2	15	Ground
3	Out 3	16	Ground
4	DSUB 1	17	Ground
5	DSUB 2	18	Ground
6	DSUB 3	19	Ground
7	DSUB 4*	20	Ground
8	Differential 1 -	21	Ground
9	Differential 1 +	22	Ground
10	Differential 2 -	23	Ground
11	Differential 2 +	24	Ground
12	Differential 3 -	25	+ 5 Volt
13	Differential 3 +		

*DSUB 4 is currently not supported.

6.5.3 IBSA::SetSync1

The Sync1 signal can be used to mark an event during the acquisition. The Sync1 signal is normally used with cyclic events to mark the beginning of a cycle or revolution; thus it should be generated once per cycle. The reset pulse from an angular encoder can be connected to this input.

```
[C++,IDL]
HRESULT SetSync1(
    /*[in]*/ int Connector);

[Visual Basic 6]
Sub SetSync1(
    Connector As Long)
```

Parameters

Connector

[in] Specifies the connector that the Sync1 signal is connected to.

Requirements

This method only applies to models with synchronization option installed. Please refer to Table 7 Processor Models and Synchronization Capabilities.

Remarks

Type can be set to one of the following constants:

Table 8 Connector Constants

Constant	Value
bsaNONE	0
bsaBNC1	1
bsaBNC2	2
bsaBNC3	3
bsaDIFFERENTIAL1	4
bsaDIFFERENTIAL2	5
bsaDIFFERENTIAL3	6
bsaDSUB1	7
bsaDSUB2	8
bsaDSUB3	9

6.5.4 IBSA::GetSync1

Gets the input connector to which the Sync1 in connected.

```
[C++,IDL]
HRESULT GetSync1(
    /*[out, retval]*/ int* pConnector);

[Visual Basic 6]
Function GetSync1(
    ) As Long
```

Parameters

pConnector

[out, retval] A pointer to an int that is to hold the connector to which the Sync1 signal is connected. Refer to Table 8 Connector Constants for possible return values.

Default

The default setting of the input signal is bsaNONE = 0.

6.5.5 IBSA::SetSync2

The Sync2 signal can be used to mark an event during the acquisition. The Sync2 signal can be used for encoder pulses. This signal is connected to a counter, which is read when a Doppler burst is detected. The counter is reset whenever a Sync1 pulse is received. The counter reading is used to relate the velocity measurements to the angular position of the encoder.

```
[C++,IDL]
HRESULT SetSync2(
    /*[in]*/ int Connector);

[Visual Basic 6]
Sub SetSync2(
    Connector As Long)
```

Parameters

Connector

[in] Specifies the connector that the Sync2 signal is connected. Refer to Table 8 Connector Constants for possible input values.

6.5.6 IBSA::GetSync2

Gets the input connector to which the Sync2 in connected.

```
[C++,IDL]
HRESULT GetSync2(
    /*[out, retval]*/ int* pConnector);

[Visual Basic 6]
Function GetSync2(
    ) As Long
```

Parameters

pConnector

[out, retval] A pointer to an int that is to hold the connector to which the Sync2 signal is connected. Refer to Table 8 Connector Constants for possible return values.

Default

The default setting of the input signal is bsaNONE = 0.

6.5.7 IBSA::SetSync1Edge

Trigger edges can take the values positive or negative, specifying whether the Sync1 input triggers on a positive going or negative going edge of a trigger pulse.

```
[C++,IDL]
HRESULT SetSync1Edge(
    /*[in]*/ int Edge);

[Visual Basic 6]
Sub SetSync1Edge(
    Edge As Long)
```

Parameters

Edge

[in] Specifies the edge on which to trigger Sync1.

Remarks

Type can be set to one of the following constants:

Table 9 Trigger Edge Constants

Constant	Value
bsaNEGATIVE	0
bsaPOSITIVE	1

6.5.8 IBSA::GetSync1Edge

This method returns the edge on which Sync1 is being triggered.

```
[C++,IDL]
HRESULT GetSync1Edge(
    /*[out, retval]*/ int* pEdge);

[Visual Basic 6]
Function GetSync1Edge(
    ) As Long
```

Parameters

pEdge

[out, retval] A pointer to an int that is to hold the edge on which sync1 is triggered. Refer to Table 9 Trigger Edge Constants for possible return values.

Default

The default setting of the edge direction is negative bsaNEGATIVE = 0.

6.5.9 IBSA::SetSync2Edge

Trigger edges can take the values positive or negative, specifying whether the Sync2 input triggers on a positive going or negative going edge of a trigger pulse.

```
[C++,IDL]
HRESULT SetSync2Edge(
    /*[in]*/ int Edge);

[Visual Basic 6]
Sub SetSync2Edge(
    Edge As Long)
```

Parameters

Edge

[in] Specifies the edge on which to trigger Sync2. Refer to Table 9 Trigger Edge Constants for possible input values.

6.5.10 IBSA::GetSync2Edge

This method returns the edge on which Sync2 is being triggered.

```
[C++,IDL]
HRESULT GetSync2Edge(
    /*[out, retval]*/ int* pEdge);

[Visual Basic 6]
Function GetSync2Edge(
    ) As Long
```

Parameters

pEdge

[out, retval] A pointer to an int that is to hold the edge on which sync2 is triggered. Refer to Table 9 Trigger Edge Constants for possible return values.

Default

The default setting of the edge direction is negative bsaNEGATIVE = 0.

6.5.11 IBSA::SetEncoderReset

Reset encoder used for once-per-revolution encoder pulses. Encoder reset signal is mapped to one of the selectable connectors.

```
[C++,IDL]
HRESULT SetEncoderReset(
    /*[in]*/ int Connector);

[Visual Basic 6]
Sub SetEncoderReset(
    Connector As Long)
```

Parameters

Connector

[in] Specifies the connector that the encoder reset signal is to be connected. Refer to Table 8 Connector Constants for possible input values.

Remarks

For the encoder to work signals must be applied as both encoder reset and encoder clock. The maximum internal encoder count is 65.536 corresponding to 16-bit resolution.

Remember to enable the encoder data output in one or more groups using the **SetEncoderData** method.

6.5.12 IBSA::GetEncoderReset

Gets the input connector to which the encoder reset is connected.

```
[C++,IDL]
HRESULT GetResetEncSignal(
    /*[out, retval]*/ int* pConnector);

[Visual Basic 6]
Function GetResetEncSignal(
    ) As Long
```

Parameters

pConnector

[out, retval] A pointer to an int that is to hold the connector to which the encoder reset signal is connected. Refer to Table 8 Connector Constants for possible return values.

Default

The default setting of the input signal is bsaNONE = 0.

6.5.13 IBSA::SetEncoder

Encoder used for encoder pulses. Encoder signal is mapped to one of the selectable connectors.

```
[C++,IDL]
HRESULT SetEncoder(
    /*[in]*/ int Connector);

[Visual Basic 6]
Sub SetEncoder(
    Connector As Long)
```

Parameters

Connector

[in] Specifies the connector that the encoder clock signal is to be connected. Refer to Table 8 Connector Constants for possible input values.

Remarks

For the encoder to work signals must be applied as both encoder reset and encoder clock.

Remember to enable the encoder data output in one or more groups using the **SetEncoderData** method.

6.5.14 IBSA::GetEncoder

Gets the input connector to which the Encoder is connected.

```
[C++,IDL]
HRESULT GetEncoder(
    /*[out, retval]*/ int* pConnector);

[Visual Basic 6]
Function GetEncoder(
    ) As Long
```

Parameters

pConnector

[out, retval] A pointer to an int that holds the connector to which the encoder clock signal is connected. Refer to Table 8 Connector Constants for possible return values.

Default

The default setting of the input signal is bsaNONE = 0.

6.5.15 IBSA::SetStartMeasurement

Start measurement is a trigger input, that will start data acquisition and hence the BSA Processors arrival time counter on the rising edge of a TTL pulse. Start measurement signal is mapped to one of the selectable connectors.

```
[C++,IDL]
HRESULT SetStartMeasurement(
    /*[in]*/ int Connector);

[Visual Basic 6]
Sub SetStartMeasurement(
    Connector As Long)
```

Parameters

Connector

[in] Specifies the connector that the start measurement signal is to be connected. Refer to Table 8 Connector Constants for possible input values.

6.5.16 IBSA::GetStartMeasurement

This method gets the input connector to which the start measurement is connected.

```
[C++,IDL]
HRESULT GetStartMeasurement(
    /*[out, retval]*/ int* pConnector);

[Visual Basic 6]
Function GetStartMeasurement(
    ) As Long
```

Parameters

pConnector

[out, retval] A pointer to an int that is to hold the connector to which the start measurement signal is connected. Refer to Table 8 Connector Constants for possible return values.

Default

The default setting of the input signal is bsaNONE = 0.

6.5.17 IBSA::SetStopMeasurement

Stop measurement is a trigger input that will stop data acquisition on the falling edge of a TTL pulse. Stop measurement signal is mapped to one of the selectable connectors.

```
[C++,IDL]
HRESULT SetStopMeasurement(
    /*[in]*/ int Connector);

[Visual Basic 6]
Sub SetStopMeasurement(
    Connector As Long)
```

Parameters

Connector

[in] Specifies the input, which stops measurement, is to be connected to. Refer to Table 8 Connector Constants for possible input values.

6.5.18 IBSA::GetStopMeasurement

Gets the input connector to which the stop measurement is connected.

```
[C++,IDL]
HRESULT GetStopMeasurement(
    /*[out, retval]*/ int* pConnector);

[Visual Basic 6]
Function GetStopMeasurement(
    ) As Long
```

Parameters

pConnector

[out, retval] A pointer to an int that is to hold the input- connector to witch stop Measurement is connected. Refer to Table 8 Connector Constants for possible return values.

Default

The default setting of the input signal is bsaNONE = 0.

6.5.19 IBSA::SetBurstDetectorEnable

Burst detector enable can be used as a gate signal to enable measurements only during the time that this signal is high (TTL level). The difference between this input and the start/stop inputs is that start/stop starts and stops the arrival time counter, whereas the Burst detector enable just enables or disables detection of bursts, while the arrival time counter keeps counting.

```
[C++,IDL]
HRESULT SetBurstDetectorEnable(
    /*[in]*/ int Connector);

[Visual Basic 6]
Sub SetBurstDetectorEnable(
    Connector As Long)
```

Parameters

Connector

[in] Specifies the connector that the burst detector enable signal is to be connected. Refer to Table 8 Connector Constants for possible input values.

6.5.20 IBSA::GetBurstDetectorEnable

Gets the input connector that the burst detector enable is connected.

```
[C++,IDL]
HRESULT GetBurstDetectorEnable(
    /*[out, retval]*/ int* pConnector);

[Visual Basic 6]
Function GetBurstDetectorEnable(
    ) As Long
```

Parameters

pConnector

[out, retval] A pointer to an int that is to hold the input connector to which the burst detector enable signal is connected. Refer to Table 8 Connector Constants for possible return values.

Default

The default setting of the input signal is bsaNONE = 0.

6.5.21 IBSA::SetRefClk

Reference clock mode can be set to internal or external. Default is internal. If two or more BSA Processors are used together, one of them should have this property set to internal, and the 10 MHz reference output signal BNC connector from this unit should be connected to the 10 MHz reference input on the other BSA, which should have this property set to external. The reference clock can also be used to synchronize the reference clocks among other third party equipment using a 10 MHz reference clock.

```
[C++,IDL]
HRESULT SetRefClk(
    /*[in]*/ int Clk);

[Visual Basic 6]
Sub SetRefClk(
    Clk As Long)
```

Parameters

Clk

[in] Specifies whether reference clock is internal or external.

Remarks

Clk can be set to one of the following constants:

Table 10 Reference Clock Constants

Constant	Value
bsaINTERNCLK	0
bsaEXTERNCLK	1

6.5.22 IBSA::GetRefClk

This method gets the setting of reference clock.

```
[C++,IDL]
HRESULT GetRefClk(
    /*[out, retval]*/ int* pClk);

[Visual Basic 6]
Function GetRefClk(
    ) As Long
```

Parameters

pClk

[out, retval] A pointer to an int that holds the reference clock setting. Refer to Table 10 Reference Clock Constants for possible return values.

Default

The default setting of the reference clock in internal bsaINTERN = 0.

6.6 Sync Output signals

Two types of TTL signals can be mapped to the connectors “Programmable Output” or “S1” on the bag of the processor: A Burst Detector signal from one of the velocity channels, or a Measurement Running signal.

The Burst Detector signal is a TTL signal, which is high for the duration of the burst. It is delayed by approx. 8 ms to the photo-multiplier signal. The measurement running signal is a TTL signal, which is high during acquisition. This can be used for gating external equipment during LDA data acquisition.

6.6.1 IBSA::SetBNC

This setting is used to specify the signal that is to output on the BNC 1 output connector.

```
[C++,IDL]
HRESULT SetBNC(
    /*[in]*/ int Signal);

[Visual Basic 6]
Sub SetBNC(
    Signal As Long)
```

Parameters

Signal

[in] Specifies the signal.

Remarks

Signal can be set to one of the following constants:

Table 11 Synchronization Signal Constants

Constant	Value
bsaUNUSED	0
bsaBURSTDetect1	1
bsaBURSTDetect2	2
bsaBURSTDetect3	3
bsaBURSTDetect4	4
bsaBURSTDetect5	5
bsaBURSTDetect6	6
bsaSTARTMEASUREMENT	7

6.6.2 IBSA::GetBNC

This method gets the current signal mapped to BNC1.

```
[C++,IDL]
HRESULT GetBNC(
    /*[out, retval]*/ int* pSignal);

[Visual Basic 6]
Function GetBNC(
    ) As Long
```

Parameters

pSignal

[out, retval] A pointer to an int that is to hold the selected signal for BNC1. Refer to Table 11 Synchronization Signal Constants for possible return values.

Default

The default setting of the output signal is bsaUNUSED = 0.

6.6.3 IBSA::SetDSUB1

This setting is used to specify the signal that is to output on the DSUB output connectors.

```
[C++,IDL]
HRESULT SetDSUB1(
    /*[in]*/ int Signal);

[Visual Basic 6]
Sub SetDSUB1(
    Signal As Long)
```

Parameters

Signal

[in] Specifies the signal

6.6.4 IBSA::GetDSUB1

This method gets the current signals mapped to the DSUB connectors set by the **SetDSUB1** method.

```
[C++,IDL]
HRESULT GetDSUB1(
    /*[out, retval]*/ int* pSignal);

[Visual Basic 6]
Function GetDSUB1(
    ) As Long
```

Parameters

pSignal

[out, retval] A pointer to the signal on the DSUB1 connector.

Default

The default setting of the output signals are bsaUNUSED = 0.

6.6.5 IBSA::SetDSUB2 and IBSA::SetDSUB3

See description about the method **SetDSUB1**.

6.6.6 IBSA::GetDSUB2 and IBSA::GetDSUB3

See description about the method **GetDSUB1**.

6.6.7 IBSA::SetShutter

This setting is used to specify the signal that is to output on the Shutter (BNC 2) output connector.

```
[C++,IDL]
HRESULT SetShutter(
    /*[in]*/ int Signal);

[Visual Basic 6]
Sub SetShutter(
    Signal As Long)
```

Parameters

Signal

[in] Specifies the signal. Refer to Table 11 Synchronization Signal Constants for possible input values.

6.6.8 IBSA::GetShutter

Gets the current signal mapped to the Shutter connector (BNC2).

```
[C++,IDL]
HRESULT GetShutter(
    /*[out, retval]*/ int* pSignal);

[Visual Basic 6]
Function GetShutter(
    ) As Long
```

Parameters

pSignal

[out, retval] A pointer to an int that is to hold the selected signal for BNC2. Refer to Table 11 Synchronization Signal Constants for possible return values.

Default

The default setting of the output signal is bsaUNUSED = 0.

6.7 Group Settings

Coincident data from velocity channels are obtained by placing those channels in a common group. The group settings are common for all channels in that group. If coincident data are not required, the channels can be placed in separate groups. See the **SetConfiguration** method.

6.7.1 IBSA::SetMaxSamples

Maximum number of samples; stop criterion for data acquisition for the selected group. Together with the **SetMaxAcqTime** method this constitutes the limits of the acquisition for the group.

```
[C++,IDL]
HRESULT SetMaxSamples(
    /*[in]*/ int Group,
    /*[in]*/ int Samples);

[Visual Basic 6]
Sub SetMaxSamples(
    Group As Long,
    Samples As Long)
```

Parameters

Group

[in] Specifies the group.

Samples

[in] Defines the maximum number of samples to acquire.

Remarks

The acquisition stops when the maximum number of samples is obtained, or when the maximum acquisition time is reached. If the acquisition stops on maximum acquisition time the number of samples acquired is properly less than the number specified in **SetMaxSamples**; the actual number of acquired samples can be found by calling the **ReadDataLength** method for the group.

6.7.2 IBSA::GetMaxSamples

This method gets the number of samples set by the method **SetMaxSamples**.

```
[C++,IDL]
HRESULT GetMaxSamples (
    /*[in]*/ int Group,
    /*[out, retval]*/ int* pSamples);

[Visual Basic 6]
Function GetMaxSamples (
    Group As Long
) As Long
```

Parameters

Group
[in] Specifies the group.

pGroup
[out, retval] A pointer to an int that holds the maximum number of samples setting.

Default

The default setting of the maximum number of samples is 2.000.

6.7.3 IBSA::SetMaxAcqTime

The maximum acquisition time is one of the stop criteria for data acquisition for the selected coincidence group. Together with the **SetMaxSamples** method this constitutes the limits of the acquisition for the group.

```
[C++,IDL]
HRESULT SetMaxAcqTime (
    /*[in]*/ int Group,
    /*[in]*/ float Time);

[Visual Basic 6]
Sub SetMaxAcqTime (
    Group As Long,
    Time As Single)
```

Parameters

Group

[in] Specifies the group.

Time

[in] Specifies the maximum acquisition time in seconds.

Remarks

The acquisition stops when the maximum number of samples is obtained, or when the maximum acquisition time is reached. If more than one group is defined in the configuration the maximum duration of the acquisition is the largest value defined for all the groups.

6.7.4 IBSA::GetMaxAcqTime

This method gets the maximum acquisition time set by the method **SetMaxAcqTime**.

```
[C++,IDL]
HRESULT GetMaxAcqTime (
    /*[in]*/ int Group,
    /*[out, retval]*/ float* pTime);

[Visual Basic 6]
Function GetMaxAcqTime (
    Group As Long
) As Single
```

Parameters

Group

[in] Specifies the group.

pTime

[out, retval] A pointer to a float that holds the maximum acquisition time in seconds.

Default

The default setting of the maximum acquisition time is 10 sec.

6.7.5 IBSA::SetFilterMethod

Filter method can be set to burst overlapped or windowing. This determines how coincidence filtering is made.

Burst overlapped mode demands that bursts from coincident channels are partly overlapped.

Windowing mode requires that bursts from coincident channels are detected within a time window, defined by the **SetBurstWindow** method. This setting may be useful when measuring through curved window surfaces, or other situations where burst overlap may not be possible.

```
[C++,IDL]
HRESULT SetFilterMethod(
    /*[in]*/ int Group,
    /*[in]*/ int Filter);

[Visual Basic 6]
Sub SetFilterMethod(
    Group As Long,
    Filter As Long)
```

Parameters

Group

[in] Specifies the group.

Filter

[out, retval] Specifies the filter method.

Remarks

Filter can be set to one of the following constants:

Table 12 Filter Method Constants

Constant	Value
bsaWINDOW	0
bsaOVERLAP	1

6.7.6 IBSA::GetFilterMethod

This method gets the filter method set by the method **SetFilterMethod**.

```

[C++,IDL]
HRESULT GetFilterMethod(
    /*[in]*/ int Group,
    /*[out, retval]*/ int* pFilter);

[Visual Basic 6]
Function GetFilterMethod(
    Group As Long
) As Long

```

Parameters

Group

[in] Specifies the group.

pFilter

[out, retval] A pointer to an int that is to hold the filter method. Refer to Table 12 Filter Method Constants for possible return values.

Default

The default setting of the coincidence filtering mode is burst overlapping bsaOVERLAP = 1.

6.7.7 IBSA::SetBurstWindow

Burst window defines the time window for coincidence filtering by the windowing mode set by the **SetFilterMethod** method.

```
[C++,IDL]
HRESULT SetBurstWindow(
    /*[in]*/ int Group,
    /*[in]*/ float Window);

[Visual Basic 6]
Sub SetBurstWindow(
    Group As Long,
    Window As Single)
```

Parameters

Group

[in] Specifies the group.

Window

[in] Specifies the time length of window. *Window* can be set in the range from 10 μ sec to 1 sec.

6.7.8 IBSA::GetBurstWindow

This method gets the time set by the method **SetBurstWindow**.

```
[C++,IDL]
HRESULT GetBurstWindow(
    /*[in]*/ int Group,
    /*[out, retval]*/ float* pWindow);

[Visual Basic 6]
Function GetBurstWindow(
    Group As Long
) As Single
```

Parameters

Group

[in] Specifies the group.

pWindow

[out, retval] A pointer to a float that is to hold the window time length. On return *pWindow* is set in the range from 10 μ sec to 1 sec set by the method **SetBurstWindow**.

Default

The default setting of the burst windowing width is 10 μ sec.

6.7.9 IBSA::SetScopeDisplay

Enables or disables data output for scope display in the System Monitor.

```
[C++,IDL]
HRESULT SetScopeDisplay(
    /*[in]*/ int Group,
    /*[in]*/ int Display);

[Visual Basic 6]
Sub SetScopeDisplay(
    Group As Long,
    Display As Long)
```

Parameter

Group
[in] Specifies the group.

Display
[in] Enable or disable.

Remarks

Display can be set to one of the following constants:

Table 13 Scope Monitor Constants

Constant	Value
bsaOFF	0
bsaBURSTSIGNAL	1
bsaBURSTSPECTRUM	2

6.7.10 IBSA::GetScopeDisplay

This method gets the view state of the scope display in the System Monitor.

```
[C++,IDL]
HRESULT GetScopeDisplay(
    /*[in]*/ int Group,
    /*[out, retval]*/ int* pDisplay);

[Visual Basic 6]
Function GetScopeDisplay(
    Group As Long
) As Long
```

Parameter

Group
[in] Specifies the group.

pDisplay

[out, retval] Enable or disable. Refer to Table 13 Scope Monitor Constants for possible return values.

Default

The default setting of the scope monitor signal is burst signal
bsaBURSTSIGNAL = 1.

6.7.11 IBSA::SetScopeTrigger

The scope monitor is digitally triggered at the center of a burst except in free-run. In normal operation, the trigger channel should be set to the same channel as the burst channel, meaning that the burst triggers individual by itself. If dependency between velocity channels is investigated (e.g. to check coincidence), the trigger channel could be another velocity channel. The free-run can be used to get an impression of the data rate or background noise.

```
[C++,IDL]
HRESULT SetScopeTrigger(
    /*[in]*/ int Group,
    /*[in]*/ int Trigger);

[Visual Basic 6]
Sub SetScopeTrigger(
    Group As Long,
    Trigger As Long)
```

Parameter

Group

[in] Specifies the group.

Trigger

[in] Trigger channel.

Remarks

Trigger is the source of the trigger signal as specified by the constants:

Table 14 Trigger Channel Constants

Constant	Value
bsaFREERUN	-1
bsaINDIVIDUAL	0
bsaCHANNEL1	1
bsaCHANNEL2	2
bsaCHANNEL3	3
bsaCHANNEL4	4
bsaCHANNEL5	5
bsaCHANNEL6	6

6.7.12 IBSA::GetScopeTrigger

This method gets the setting of the scope monitor trigger for a specified group.

```

[C++,IDL]
HRESULT GetScopeTrigger(
    /*[in]*/ int Group,
    /*[out, retval]*/ int* pTrigger);

[Visual Basic 6]
Function GetScopeTrigger(
    Group As Long
) As Long

```

Parameter

Group
[in] Specifies the group.

pTrigger
[out, retval] On return the value of this parameter specifies the trigger channel. Refer to Table 14 Trigger Channel Constants for possible return values.

Default

The default setting of the scope monitor trigger signal is triggering individual on the same channel as itself `bsaINDIVIDUAL = 0`.

6.7.13 IBSA::SetScopeZoom

Scope zoom determines the scaling of the scope display time axis in the System Monitor.

```
[C++,IDL]
HRESULT SetScopeZoom(
    /*[in]*/ int Group,
    /*[in]*/ int Zoom);

[Visual Basic 6]
Sub SetScopeZoom(
    Group As Long,
    Zoom As Long)
```

Parameters

Group

[in] Specifies the group.

Zoom

[in] Specifies the percentage of zoom. *Zoom* can be set in the range from 1 % to 3.200 %.

6.7.14 IBSA::GetScopeZoom

This method gets the zoom value set by the **SetScopeZoom** method.

```
[C++,IDL]
HRESULT GetScopeZoom(
    /*[in]*/ int Group,
    /*[out, retval]*/ int* pZoom);

[Visual Basic 6]
Function GetScopeZoom(
    Group As Long
) As Long
```

Parameters

Group

[in] Specifies the group.

pZoom

[out, retval] Specifies the percentage of zoom. On return *pZoom* is set in the range from 1 % to 3.200 %.

Default

The default setting of the scope zoom is 400 %.

6.8 LDA Settings

The settings in this section define the behavior of the individual channels.

The following table defines the valid frequency inputs, bandwidths and center frequency settings, based on the BSA Processor model.

Table 15 BSA Processor models and frequency inputs, bandwidths and center frequencies.

Parameter	BSA F/P30	BSA F/P50	BSA F/P60	BSA F/P70	BSA F/P80
Max. input frequency (MHz)	90	80	100	160	180
Min. input frequency (MHz)	25	15	15	15	15
Max. bandwidth (MHz)	5,625	30	30	120	120
Min. bandwidth (MHz)	0,011	0,117	0,015	0,011	0,011
Max. center frequency (MHz)	85	75	95	140	160
Min. center frequency (MHz)	30	30	30	30	30
No. of bandwidths	7	9	12	28	28

6.8.1 IBSA::SetCenterFreq

This method is used to set the center frequency of a particular channel.

```
[C++,IDL]
HRESULT SetCenterFreq(
    /*[in]*/ int Channel,
    /*[in]*/ float Freq);

[Visual Basic 6]
Sub SetCenterFreq(
    Channel As Long,
    Freq As Single)
```

Parameters

Channel

[in] Specifies the input channel.

Freq

[in] Frequency in Hz. *Freq* is the parameter that specifies the center frequency. The center frequency is set in 5 kHz steps.

Remarks

The center frequency and the bandwidth of the channel are closely related. Whenever the center frequency is changed the value of the bandwidth for a channel should be checked to see if it has changed using the **GetBandwidth** method.

6.8.2 IBSA::GetCenterFreq

This method gets the center frequency from a specified channel. The center frequency can be set using the **SetCenterFreq** method or it can have been changed because the bandwidth has been changed using the **SetBandwidth** method.

```
[C++,IDL]
HRESULT GetCenterFreq(
    /*[in]*/ int Channel,
    /*[out, retval]*/ float* pFreq);

[Visual Basic 6]
Function GetCenterFreq(
    Channel As Long
) As Single
```

Parameters

Channel

[in] Specifies the input channel.

pFreq

[out, retval] A pointer to a float that is to hold the center frequency in Hz.

Default

The default setting of the center frequency is 0 Hz. Remember that this setting is dependent of the bandwidth setting.

6.8.3 IBSA::SetBandwidth

This method sets the bandwidth for a specified channel. The bandwidth defines the width of the velocity range around the center frequency. Bandwidth should be set according to the expected range of fluctuations in flow velocity.

If in doubt, set a large bandwidth to begin with, make an acquisition, and zoom in to proper center frequency and bandwidth afterwards.

```
[C++,IDL]
HRESULT SetBandwidth(
    /*[in]*/ int Channel,
    /*[in]*/ float Bandwidth);

[Visual Basic 6]
Sub SetBandwidth(
    Channel As Long,
    Bandwidth As Single)
```

Parameters

Channel

[in] Specifies the input channel.

Bandwidth

[in] Bandwidth in Hz.

Remarks

The center frequency and the bandwidth of the channel are closely related. Whenever the bandwidth is changed the value of the center frequency of the channel should be checked to see if it has changed using the **GetCenterFreq** method.

The resulting frequency is also dependent of the optical shift (Bragg cell) frequency set by the **SetOpticsShift** method.

6.8.4 IBSA::GetBandwidth

Gets the center frequency from a specified channel.

```
[C++,IDL]
HRESULT GetBandwidth(
    /*[in]*/ int Channel,
    /*[out, retval]*/ float* pBandwidth);

[Visual Basic 6]
Function GetBandwidth(
    Channel As Long,
    pBandwidth) As Single
```

Parameters

Channel

[in] Specifies the input channel.

pBandwidth

[out, retval] A pointer to a float that is to hold the bandwidth. The bandwidth is returned in Hz.

Default

The default setting of the bandwidth is 11.25 MHz. Remember that this setting is dependent of the center frequency setting.

6.8.5 IBSA::SetRecordLengthMode

Length mode can be set to auto-adaptive or fixed modes.

In **auto-adaptive record length mode**, the processor will adapt the record lengths individually to each burst, within the limits specified by the record length and the maximum record length values.

In **fixed record length mode**, the record length value is always used.

To estimate the required record length, use the Doppler monitor display.

```
[C++,IDL]
HRESULT SetRecordLengthMode(
    /*[in]*/ int Channel,
    /*[in]*/ int Mode);

[Visual Basic 6]
Sub SetRecordLengthMode(
    Channel As Long,
    Mode As Long)
```

Parameters

Channel

[in] Specifies the input channel.

Mode

[in] The record length mode.

Remarks

Mode can be set to one of following constants:

Table 16 Record Length Mode Constants

Constant	Value
bsaFIXED	0
bsaAUTO	1

6.8.6 IBSA::GetRecordLengthMode

This method gets the record length mode from a specified channel.

```

[C++,IDL]
HRESULT GetRecordLengthMode(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pMode);

[Visual Basic 6]
Function GetRecordLengthMode(
    Channel As Long
) As Long

```

Parameters

Channel

[in] Specifies the input channel.

pMode

[out, retval] A pointer to an int that holds the value of the record length. Refer to Table 16 Record Length Mode Constants for possible return values.

Default

The default setting of the record length mode is fixed bsaFIXED = 0.

6.8.7 IBSA::SetRecordLength

This method sets the record length for a specified channel, which is passed to the FFT processor to determine the Doppler frequency. It sets the number of samples of the shortest record length in auto-adaptive mode, or the applied record length in fixed mode.

```
[C++,IDL]
HRESULT SetRecordLength(
    /*[in]*/ int Channel,
    /*[in]*/ int Length);

[Visual Basic 6]
Sub SetRecordLength(
    Channel As Long,
    Length As Long)
```

Parameters

Channel

[in] Specifies the input channel.

Length

[in] The record length.

Remarks

Length specifies the record length, can be set to one of following constants:

Table 17 Record Length Constants

Constant	Value
bsaREC16	16
bsaREC32	32
bsaREC64	64
bsaREC128	128
bsaREC256	256

6.8.8 IBSA::GetRecordLength

This method gets the record length from a specified channel.


```

[C++,IDL]
HRESULT GetRecordLength(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pLength);

[Visual Basic 6]
Function GetRecordLength(
    Channel As Long
) As Long

```

Parameters

Channel

[in] Specifies the input channel.

pLength

[out, retval] A pointer to an int that is to hold the record length. Refer to Table 17 Record Length Constants for possible return values.

Default

The default setting of the record length is `bsaREC128 = 128`. Remember that this setting can be dependent of the maximum record length setting, since the maximum record length setting must be larger or equal to the record length setting.

6.8.9 IBSA::SetMaxRecordLength

Maximum record length sets the number of samples of the longest record length in bsaAUTO mode. Not used in bsaFIXED mode.

```
[C++,IDL]
HRESULT SetMaxRecordLength(
    /*[in]*/ int Channel,
    /*[in]*/ int Length);

[Visual Basic 6]
Sub SetMaxRecordLength(
    Channel As Long,
    Length As Long)
```

Parameters

Channel

[in] Specifies the input channel.

Length

[in] The maximum record length. Refer to Table 17 Record Length Constants for possible input values.

Remarks

The maximum record length can only be specified in auto-adaptive record length mode, see **SetRecordLengthMode**. In this case the record length and the maximum record length will be related since the maximum record length setting must be larger or equal to the record length setting.

6.8.10 IBSA::GetMaxRecordLength

This method gets the maximum record length from a specified channel.

```
[C++,IDL]
HRESULT GetMaxRecordLength(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pLength);

[Visual Basic 6]
Function GetMaxRecordLength(
    Channel As Long
) As Long
```

Parameters

Channel

[in] Specifies the input channel.

pLength

[out, retval] A pointer to an int that is to hold the maximum record length. Refer to Table 17 Record Length Constants for possible return values.

Default

The default setting of the record length is $\text{bsaREC128} = 128$. Remember that this setting can be dependent of the maximum record length setting, since the maximum record length setting must be larger or equal to the record length setting.

6.8.11 IBSA::SetHighVoltage

This method sets the high voltage to the photo-multiplier for a specified LDA channel. The recommended starting level is around 1.000 V. Depending on the size of the particles, the laser power, the optics and the position of the measurement volume in the flow, the optimum may be higher or lower. Optimization should take both this property and the signal gain, set by the **SetSignalGain** method, into account. The criterion may be the validation rate or the data rate or some compromise between them, depending on the application. The validation rate and data rates are displayed in the System Monitor window.

```
[C++,IDL]
HRESULT SetHighVoltage(
    /*[in]*/ int Channel,
    /*[in]*/ int Volt);

[Visual Basic 6]
Sub SetHighVoltage(
    Channel As Long,
    Volt As Long)
```

Parameters

Channel

[in] Specifies the input LDA channel.

Volt

[in] Specifies the voltage level. *Volt* specifies the voltage level can be set in the range from 0 to 1.800 Volts.

6.8.12 IBSA::GetHighVoltage

This method gets the high voltage level from a specified LDA channel from the processor.

```
[C++,IDL]
HRESULT GetHighVoltage(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pVolt);

[Visual Basic 6]
Function GetHighVoltage(
    Channel As Long
) As Long
```

Parameters

Channel

[in] Specifies the input LDA channel.

pVolt

[out, retval] A pointer to an int that is to hold the voltage level.

Default

The default setting of the high voltage level is 1.000 Volts.

6.8.13 IBSA::SetSignalGain

This method sets the gain of the photo-multiplier signal amplifier. Recommended starting level is around 24 dB. This parameter should be optimized together with the high voltage as described above.

```
[C++,IDL]
HRESULT SetSignalGain(
    /*[in]*/ int Channel,
    /*[in]*/ int Gain);
```

```
[Visual Basic 6]
Sub SetSignalGain(
    Channel As Long,
    Gain As Long)
```

Parameters

Channel

[in] Specifies the input channel.

Gain

[in] Specifies the gain in dB. *Gain* can be set to 0, 2, 4, ..., 30 dB

6.8.14 IBSA::GetSignalGain

This method gets the signal gain for a specified channel from the processor.

```
[C++,IDL]
HRESULT GetSignalGain(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pGain);
```

```
[Visual Basic 6]
Function GetSignalGain(
    Channel As Long
) As Long
```

Parameters

Channel

[in] Specifies the input channel.

pGain

[out, retval] A pointer to an int that is to hold the signal gain.

Default

The default setting of the signal gain is 26 dB.

6.8.15 IBSA::SetBurstDetectorSNR

This method sets the burst detectors signal-to-noise (SNR) threshold level for a specified channel. The default value is 0 dB. High values will reject more noisy bursts, leading to higher validation and lower data rate, and vice versa.

```
[C++,IDL]
HRESULT SetBurstDetectorSNR(
    /*[in]*/ int Channel,
    /*[in]*/ int SNR);
```

```
[Visual Basic 6]
Sub SetBurstDetectorSNR(
    Channel As Long,
    SNR As Long)
```

Parameter

Channel

[in] Specifies the input channel.

SNR

[in] The burst detectors SNR threshold level in dB. *SNR* can be set in the range from -6 dB to 5 dB.

6.8.16 IBSA::GetBurstDetectorSNR

This method gets the burst detectors signal-to-noise (SNR) threshold level for a specified channel from the processor.

```
[C++,IDL]
HRESULT GetBurstDetectorSNR(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pSNR);
```

```
[Visual Basic 6]
Function GetBurstDetectorSNR(
    Channel As Long
) As Long
```

Parameter

Channel

[in] Specifies the input channel.

pSNR

[out, retval] Pointer to an int that is to hold the burst detector SNR threshold level.

Default

The default setting of the burst detector SNR threshold level is 0 dB.

6.8.17 IBSA::SetLevelValidationRatio

Level validation looks at the ratio between the two highest peaks in the burst spectrum. If the ratio is higher than the specified value the burst is validated, otherwise it is rejected. Only validated samples will contribute to the number of bursts counted and be stored.

```
[C++,IDL]
HRESULT SetLevelValidationRatio(
    /*[in]*/ int Channel,
    /*[in]*/ int Ratio);
```

```
[Visual Basic 6]
Sub SetLevelValidationRatio(
    Channel As Long,
    Ratio As Long)
```

Parameter

Channel

[in] Specifies the input channel.

Ratio

[in] Specifies the validation ratio. The ratio can be set to one of the following numbers: 2, 4, 8, 10, 12 or 16.

Remarks

Table 18 Level Validation Constants

Constant	Value
bsaLEVEL2	2
bsaLEVEL4	4
bsaLEVEL8	8
bsaLEVEL10	10
bsaLEVEL12	12
bsaLEVEL16	16

Limitations

The level validation property can only be set on BSA F/P70 and F/P80 Processor models. All other models use the default level validation ratio of bsaLEVEL4 = 4.

6.8.18 IBSA::GetLevelValidationRatio

This method gets the validation ratio from the processor set by the **SetLevelValidationRatio** method.


```

[C++,IDL]
HRESULT GetLevelValidationRatio(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pRatio);

[Visual Basic 6]
Function GetLevelValidationRatio(
    Channel As Long
) As Long

```

Parameter

Channel

[in] Specifies the input channel.

pRatio

[out, retval] A pointer to an int that holds the level validation ratio. Refer to Table 18 Level Validation Constants for possible return values.

Default

The default setting of the level validation is bsaLEVEL4 = 4.

6.9 PDA Methods

PDA methods are specific to BSA P-series Processors. The methods involve the automatic phase calibration in the BSA Processor, and phase and diameter validation.

The methods are only valid in PDA measurements.

6.9.1 IBSA::SetPDACalibrationMode

Defines the optical PDA phase calibration mode. PDA phase calibration is necessary for keeping relation between processor settings and signal strengths. Properties' influencing the phase calibration includes: high voltage, gain, center frequency, bandwidth, and time (recalibration after 15 minutes of operation).

```
[C++,IDL]
HRESULT SetPDACalibrationMode(
    /*[in]*/ int Mode);

[Visual Basic 6]
Sub SetPDACalibrationMode(
    Mode As Long)
```

Parameter

Mode

[in] Specifies the PDA calibration mode.

Remarks

The phase calibration modes determine when the PDA Processor performs a calibration. The calibration will be performed before an acquisition is started.

The default setting is automatic calibration bsaAUTOCALIBRATE. This will calibrate whenever one of the involved properties are changed. Another possibility is to force the PDA processor to make a calibration before every acquisition bsaALWAYSCALIBRATE . This is especially useful when running long acquisitions. The last calibration mode is to skip phase calibration of the processor. This setting can be used to speed up the acquisition, **but may result in unreliable phase and hence particle size results.**

Table 19 PDA Calibration Modes

Constant	Value
bsaAUTOCALIBRATE	0
bsaALWAYSCALIBRATE	1
bsaNEVERCALIBRATE	2

6.9.2 IBSA::GetPDACalibrationMode

This method gets the calibration mode from the processor set by the **SetPDACalibrationMode** method.

```
[C++,IDL]
HRESULT GetPDACalibrationMode(
    /*[out, retval]*/ int* pMode);

[Visual Basic 6]
Function GetPDACalibrationMode(
    ) As Long
```

Parameter

pMode

[out, retval] A pointer to the PDA calibration mode. See Table 19 PDA Calibration Modes for possible return values.

Default

The default value is bsaAUTOCALIBRATE.

6.9.3 IBSA::SetPDAHighVoltage

This method sets the high voltage to the photo-multiplier for a specified PDA channel. Please refer to the description of the **SetHighVoltage** method.

```
[C++,IDL]
HRESULT SetPDAHighVoltage(
    /*[in]*/ int Channel,
    /*[in]*/ int Volt);
```

```
[Visual Basic 6]
Sub SetPDAHighVoltage(
    Channel As Long,
    Volt As Long)
```

Parameters

Channel

[in] Specifies the input PDA channel.

Volt

[in] Specifies the voltage level. *Volt* specifies the voltage level can be set in the range from 0 to 1.800 Volts.

6.9.4 IBSA::GetPDAHighVoltage

This method gets the high voltage level from a specified PDA channel from the processor.

```
[C++,IDL]
HRESULT GetPDAHighVoltage(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pVolt);
```

```
[Visual Basic 6]
Function GetPDAHighVoltage(
    Channel As Long
) As Long
```

Parameters

Channel

[in] Specifies the input PDA channel.

pVolt

[out, retval] A pointer to an int that is to hold the voltage level.

Default

The default setting of the high voltage level is 1.000 Volts.

6.9.5 IBSA::SetPDAAutoBalancing

Sets automatic balancing of high voltage between controlling velocity channel and the corresponding phase channel.

Locking the relative changes in high voltage values makes the PDA setup easier, since changes in sensitivity will follow relatively making the likelihood of a successful phase calibration higher.

```
[C++,IDL]
HRESULT SetPDAAutoBalancing(
    /*[in]*/ int Channel,
    /*[in]*/ int Balancing);

[Visual Basic 6]
Sub SetPDAAutoBalancing(
    Channel As Long,
    Balancing As Long)
```

Parameters

Channel

[in] Specifies the phase channel.

Balancing

[in] Enables or disables the automatic high voltage balancing.

Remarks

The phase channel will always either be channel 2 or 3 in the BSA Processor, and therefore either bsaPHASECHANNEL1 = bsaCHANNEL2 or bsaPHASECHANNEL1 = bsaCHANNEL3 must be used as the phase channel.

The automatic balancing will lock the relative changes to the high voltage property of the dependent PDA channel to the controlling LDA channel.

In PDA processor mode bsaPDA, set by the **SetConfiguration** method, the first LDA channel is controlling both PDA channels. In DualPDA configuration, the first LDA channel is controlling the first PDA channel, and the second LDA channel is controlling the second PDA channel.

Table 20 Auto Balancing Modes

Constant	Value
bsaON	1
bsaOFF	0

6.9.6 IBSA::GetPDAAutoBalancing

This method gets the automatic balancing mode for a specific phase channel set by the **SetPDAAutoBalancing** method.

```
[C++,IDL]
HRESULT GetPDAAutoBalancing(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pBalancing);

[Visual Basic 6]
Function GetPDAAutoBalancing(
    Channel As Long
) As Long
```

Parameter

pBalancing

[out, retval] A pointer to the automatic balancing mode. See Table 20 Auto Balancing Modes for possible return values.

Default

The default value is bsaOFF.

6.10 Advanced

6.10.1 IBSA::SetAnodeCurrentLimit

To protect the photo-multiplier tube e.g. when approaching walls, the BSA Processor includes a current limiter circuit which reduces the high voltage if the photo-multiplier anode current exceeds the set limit. For Dantec standard photo-multipliers 57X08 and those used in FlowLite optical systems, the limit should be set to 1,5 mA. For other photo-multipliers, please consult the suppliers' technical specifications.

```
[C++,IDL]
HRESULT SetAnodeCurrentLimit(
    /*[in]*/ int Channel,
    /*[in]*/ int Limit);

[Visual Basic 6]
Sub SetAnodeCurrentLimit(
    Channel As Long,
    Limit As Long)
```

Parameter

Channel

[in] Specifies the input channel.

Limit

[in] The anode current limit in μA . The limit can be set in the range from 100 μA to 3.000 μA .

6.10.2 IBSA::GetAnodeCurrentLimit

This method gets the anode current limit set for the specified channel.

```
[C++,IDL]
HRESULT GetAnodeCurrentLimit(
    /*[in]*/ int Channel,
    /*[out, retval]*/ int* pLimit);

[Visual Basic 6]
Function GetAnodeCurrentLimit(
    Channel As Long
) As Long
```

Parameter

Channel

[in] Specifies the input channel.

pLimit

[out, retval] A pointer to an int that is to hold the anode current value.

Default

The default setting of the anode current limit is 1,5 mA.

6.11 Acquiring

This section describes methods for acquiring data from the BSA Processor. Dependent of the configuration you can acquire arrival time, transit time and velocity data from one or more LDA channels, along with encoder and/or synchronization data.

The data is formatted as arrays of variants. Most application supporting COM and ActiveX automation technology understands variant and safearray types. The variant types can be integers, floating points and double precision floating points (doubles). It can be necessary for some languages to transform the variant data.

The first sample shows how to acquire and read the acquired data in VBScript.

[VBScript]

```
dim i,numSamples,safeArray,sData
' start acquisition
MyBSA.StartAcq true
' read acquired data
numSamples = MyBSA.ReadDataLength(0)
safeArray = MyBSA.ReadArrivalTimeData(0)
' run through acquired data
for i=0 to numSamples - 1
    sData = cstr(safeArray(i))
next
```

The second sample shows how to acquire and read the acquired data in JavaScript. Since JavaScript doesn't understand variant and safearray types, a build-in object *VBArray* extends the library with the capability to understand and convert variant and safearray types.

[JScript]

```
// start acquisition
MyBSA.StartAcq(true);

// read acquired data
var numSamples = MyBSA.ReadDataLength(0);
var safeArray =
    MyBSA.ReadArrivalTimeData(0);

// convert safearray from vb style to java
var vbArray = new VBArray(safeArray);
var jArray = vbArray.toArray();

// run through acquired data
var sData;
for (var i=0;i<numSamples - 1;i++)
{
    sData = jArray[i].toString();
}
```

The data will be returned in different formats depending of the type of data. The list below shows the sizes of data from all the data read methods.

ReadArrivalTimeData	64-bit double precision floating point, VT_R8, double
ReadTransitTimeData	32-bit floating point, VT_R4, float
ReadVelocityData	32-bit floating point, VT_R4, float
ReadDiameterData	32-bit floating point, VT_R4, float
ReadEncoderData	32-bit integer, VT_I4, int
ReadSyncArrivalTimeData	64-bit double precision floating point, VT_R8, double
ReadSyncData	32-bit integer, VT_I4, int

In script languages the reading of data is simple since the conversions are made automatically, please refer to the samples above.

In C++ we can access the data as a **_variant_t** type. Since the data is arranged in an array we need to access the data through the **parray->pvData** member.

[C++]

```
_variant_t vdataArray = pMyBSA->
    ReadArrivalTimeData(0);
int nLength = pMyBSA->ReadDataLength(0);
for (int i=0;i<nLength;i++)
{
    _variant_t vData(((VARIANT*)
        vdataArray.parray->pvData)[i]);
    char str[256] = {0};
    sprintf(str,"%4g",vData.dblVal);
}
```

LabVIEW provides a build-in method for converting variant data into a format that can be displayed or processed in LabVIEW.

[LabVIEW]

Variant To Data

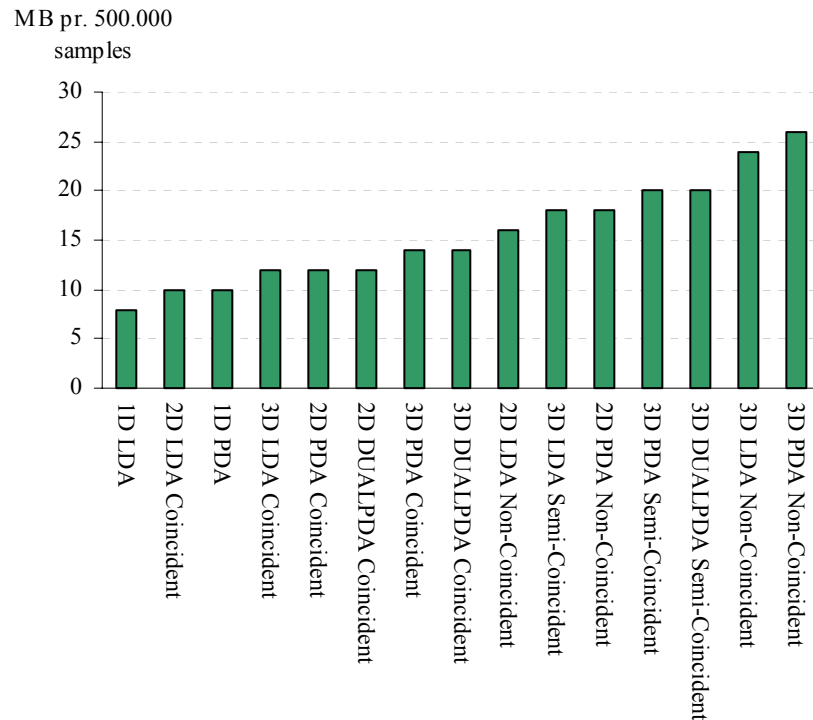


6.11.1 Memory Usage

Some applications require or benefits from a controlled memory household, keeping track of the resources of the computer running the driver. Other applications will run effectively using the default settings when memory and resources are not an important issue.

Since this driver consumes data from the processor and stores the data temporary in PC memory, until it is read by the application calling the driver. For acquisition with a high data rate the number of samples and hence memory consumption is increasing rapidly. The overall limit is controlled by the **SetMaxSamples** and **SetMaxAcqTime** methods.

The following graph shows the memory consumption by the driver for different processor models and configurations:



The graph shows the memory consumption in MB for an acquisition of 500.000 samples for the given configuration. Based on the knowledge of the data we can approximately calculate the effect on the memory of each data channel.

We know that the Arrival Time data channel is double precision, and all other channels single precision.

For the 1D LDA configuration the calculation looks like: AT + TT + Vel = 4 + 2 + 2 MB pr. 500.000 samples giving approximately 8 + 4 + 4 Bytes = 16 Bytes, as expected.

Using the **PreAllocateData** method the memory footprint can be reduced. The effect is as shown above dependent on the data channel and configuration.

For the 3D PDA Non-Coincident we can reduce the memory consumption from 26 MB to 20 MB pr. 500.000 samples by ignoring the Transit Time data channel:

```
PreAllocateData (bsaDATAALL&~ (bsaDATATT) )
```

If only the correlation between velocity and diameter is to be investigated in a 3D PDA Coincident configuration, the memory consumption can be reduced from 14 MB to 10 MB pr. 500.000 samples by specifying the data channels explicitly:

```
PreAllocateData (bsaDATAVEL|bsaDATADIA)
```

As shown in the examples above considerable amount of memory can be saved keeping track of the memory household using the **PreAllocateData** method.

6.11.2 IBSA::StartAcq

This method starts an acquisition. The acquisition will run until either the number of samples has reach the maximum samples set by the **SetMaxSamples** method or the duration of the acquisition is equal to the maximum acquisition time set by calling **SetMaxAcqTime**. Data can after the acquisition is ended be read using the data access functions **ReadArrivalTimeData**, **ReadVelocityData** etc.

```
[C++,IDL]
HRESULT StartAcq(
    /*[in]*/ int Wait);

[Visual Basic 6]
Sub StartAcq(
    Wait As Long)
```

Parameter

Wait

[in] Indicates wither the call to acquisition start should return immediately (0 or False) or wait for the acquisition to end (1 or True).

Remarks

Wait is to be set to true if the function should wait for acquisition to end before continuing. This is useful in a non-GUI application and script applications that have synchronous execution. If the function should not wait it will return immediately after the initialization of the acquisition is done.

Calling the **ReadDataLength** for each configured group, method after the acquisition is ended, will return the number of samples acquired.

If the *Wait* parameter is set to **false** you can stop the acquisition by calling the **StopAcq** method.

You should call the **GetStatus** method after ending an acquisition for handling any errors.

6.11.3 IBSA::StopAcq

Forces an acquisition to abort before it ends. Normally the acquisition ends when the maximum number of samples set by the **SetMaxSamples** or the maximum acquisition time set by **SetMaxAcqTime** is reached

```
[C++,IDL]
HRESULT StopAcq();

[Visual Basic 6]
Sub StopAcq()
```

Parameters

This method has no parameters.

Remarks

This method only applies to an acquisition that is started with *Wait* parameter in the **StartAcq** method set to **false**.

When the acquisition is ended the **onAcqDone** event is fired.

6.11.4 IBSA::PreAllocateData

During normal use all possible data is received from the BSA Processor. But some applications require the smallest possible memory overhead to run effectively, reducing the resources of the computer. With this method it is possible to pre-allocate only the necessary amount of memory needed.

```
[C++,IDL]
HRESULT PreAllocateData(
    /*[in]*/ int Flags);

[Visual Basic 6]
Sub PreAllocateData(
    Flags As Long)
```

Parameters

Flags

[in] Specifies the pre-allocation data options.

Remarks

The memory footprint and hence the data available can be controlled by combining the flags in Table 21 Pre-allocation data flags.

For example if you only need velocity information you can call the method with bsaDATAVEL as the only parameter.

If only particle information is needed you can specify the bsaDATAAT and bsaDATADIA flags with the bitwise **Or** operator (|) like this, bsaDATAAT | bsaDATADIA.

If everything but transit time is requested you can combine the bsaDATAALL parameter with the bsaDATATT using the bitwise **And** operator (&) in combination with the bitwise **Not** operator (~) like this, bsaDATAALL & ~bsaDATATT.

Table 21 Pre-allocation data flags.

Constant	Value
bsaDATAAT	0x001
bsaDATATT	0x002
bsaDATAVEL	0x004
bsaDATAENC	0x008
bsaDATADIA	0x020
bsaDATASYNC	0x040
bsaDATAALL	0xFFFF

Default

By default all possible data is received from the BSA Processor
bsaDATAALL = 0xFFFF.

6.11.5 IBSA::ReadDataLength

When an acquisition is done the number of acquired samples collected for a specified group can be found by calling this method.

Since the **SetMaxSamples** and **SetMaxAcqTime** methods control the data-collection for each group, the **ReadDataLength** will return the actual number of acquired samples limited by these methods. The **ReadDataLength** cannot be larger than the value specified in **SetMaxSamples** for the group.

```
[C++,IDL]
HRESULT ReadDataLength(
    /*[in]*/ int Group,
    /*[out, retval]*/ int* pLength);

[Visual Basic 6]
Function ReadDataLength(
    Group As Long
) As Long
```

Parameters

Group

[in] Specifies the group.

pLength

[out, retval] A pointer to an int that is to holds the number of samples collected during the last acquisition.

Remarks

The *pLength* can be used to determine the length of the acquired data arrays. The data lengths can also be determined from the data arrays themselves. See the following samples

[VBScript]

```
dim numSamples1,numSamples2,safeArray
' determine the length of the acquired data
numSamples1 = MyBSA.ReadDataLength(0)
' numSamples1 specifies the number of samples
safeArray = MyBSA.ReadArrivalTimeData(0)
numSamples2 = ubound(safeArray)
' numSamples2 specifies the upper bound,
' one less than the number of samples
' returned in numSamples1
```

[JScript]

```
// determine the length of the acquired data
var numSamples1 = MyBSA.ReadDataLength(0);
// numSamples1 specifies the number of
// samples
var safeArray = MyBSA.ReadArrivalTimeData(0);
// convert safearray from style to java
var vbArray = new VBArray(safeArray);
var jArray = vbArray1.toArray();
var numSamples2 = jArray.length;
var numSamples3 = vbArray.ubound();
// numSamples1 and numSamples2 will be equal
// numSamples3 specifies the upper bound and
// will be one less
```

6.11.6 IBSA::ReadArrivalTimeData

When an acquisition has ended, calling this method can retrieve the arrival time data for a specified group.

[C++,IDL]

```
HRESULT ReadArrivalTimeData(
    /*[in]*/ int Group,
    /*[out, retval]*/ VARIANT* pData);
```

[Visual Basic 6]

```
Function ReadArrivalTimeData(
    Group As Long
) As Variant
```

Parameters

Group

[in] Specifies the group.

pData

[out, retval] A pointer to a VARIANT data that is to hold the arrival times.

Remarks

The data buffer is a VARIANT type of VT_ARRAY | VT_VARIANT, holding an array of VT_R8 64-bit double-precision floating points. The data must be read as a variant array with each data element as a double.

6.11.7 IBSA::ReadTransitTimeData

When an acquisition is ended the transit time of each sample can be found calling this method.

```
[C++,IDL]
HRESULT ReadTransitTimeData(
    /*[in]*/ int Group,
    /*[out, retval]*/ VARIANT* pData);

[Visual Basic 6]
Function ReadTransitTimeData(
    Group As Long
) As Variant
```

Parameters

Group

[in] Specifies the group.

pData

[out, retval] A pointer to a VARIANT data that is to hold the transit times.

Remarks

The data buffer is a VARIANT type of VT_ARRAY | VT_VARIANT, holding an array of VT_R4 32-bit floating points. The data must be read as a variant array with each data element as a float.

The transit time for groups holding more than one LDA channel is calculated as the mean transit time for all the LDA channels in the group.

6.11.8 IBSA::ReadVelocityData

When an acquisition has ended the velocity in form of a frequency of each sample for each channel can be found calling this method.

```

[C++,IDL]
HRESULT ReadVelocityData(
    /*[in]*/ int Group,
    /*[in]*/ int Channel,
    /*[out, retval]*/ VARIANT* pData);

[Visual Basic 6]
Function ReadVelocityData(
    Group As Long,
    Channel As Long
) As Variant

```

Parameters

Group

[in] Specifies the group.

Channel

[in] Specifies the input channel.

pData

[out, retval] A pointer to a VARIANT data that is to hold the velocities.

Remarks

The data buffer is a VARIANT type of VT_ARRAY | VT_VARIANT, holding an array of VT_R4 32-bit floating points. The data must be read as a variant array with each data element as a float.

Velocities will be returned as the frequency in Hz. To convert the frequency in Hz to a velocity in m/s or ft/s you need to multiply with a factor specified by your optics.

6.11.9 IBSA::ReadDiameterData

When a PDA or DualPDA acquisition has ended the particle diameter in mm of each sample can be found calling this method. Diameters are only available if the processor is set to run a PDA measurement, this function will fail when called after a LDA measurement.

```

[C++,IDL]
HRESULT ReadDiameterData(
    /*[out, retval]*/ VARIANT* pData);

[Visual Basic 6]
Function ReadDiameterData(
) As Variant

```

Parameters

None

Remarks

The data buffer is a VARIANT type of VT_ARRAY | VT_VARIANT, holding an array of VT_R4 32-bit floating points. The data must be read as a variant array with each data element as a float.

If the diameter data is set to include invalid particles by the **SetSphericalValidation** method

Diameter values will be returned in micrometers ($\mu\text{m} = 10^{-6} \text{ m}$).

6.11.10 IBSA::ReadEncoderData

When an acquisition has ended and encoder data is included using the **SetEncoderData** method, along with supplying the signals to the encoder clock and the encoder reset assigned by the methods **SetEncSignal** and **SetResetEncSignal**; the encoder counts can be found calling this method.

```
[C++,IDL]
HRESULT ReadEncoderData(
    /*[in]*/ int Group,
    /*[out, retval]*/ VARIANT* pData);

[Visual Basic 6]
Function ReadEncoderData(
    Group As Long
) As Variant
```

Parameters

Group

[in] Specifies the group.

pData

[out, retval] A pointer to a VARIANT data that is to hold the encoder counts.

Remarks

The data buffer is a VARIANT type of VT_ARRAY | VT_VARIANT, holding an array of VT_I4 32-bit integers. The data must be read as a variant array with each data element as an integer.

6.11.11 IBSA::ReadSyncDataLength

When synchronization is enabled with the **SetSync1** and/or **SetSync2** methods, synchronization data will be acquired along with LDA data. This function returns the number of acquired synchronization samples.

```
[C++,IDL]
HRESULT ReadSyncDataLength(
    /*[out, retval]*/ int* pLength);
```

```
[Visual Basic 6]
Function ReadSyncDataLength(
    ) As Long
```

Remarks

This number is not the same as the number of acquired data found with the **ReadDataLength** method. Synchronization data is acquired independently from normal LDA data; why the synchronization data can be treated as events in the flow or during the acquisition. The synchronization data is also acquired as common data for all groups and channels.

Calling the **ReadSyncArrivalTimeData** and **ReadSyncData** subsequent to a call to this method will return the synchronization data events.

6.11.12 IBSA::ReadSyncArrivalTimeData

When an acquisition has ended, calling this method can retrieve the synchronization arrival time data.

```
[C++,IDL]
HRESULT ReadSyncArrivalTimeData(
    /*[out, retval]*/ VARIANT* pData);
```

```
[Visual Basic 6]
Function ReadSyncArrivalTimeData(
    ) As Variant
```

Parameters

pData

[out, retval] A pointer to a VARIANT data that is to hold the synchronization arrival times.

Remarks

The data buffer is a VARIANT type of VT_ARRAY | VT_VARIANT, holding an array of VT_R8 64-bit double-precision floating points. The data must be read as a variant array with each data element as a double.

6.11.13 IBSA::ReadSyncData

When an acquisition has ended, calling this method can retrieve the synchronization data.

```

[C++,IDL]
HRESULT ReadSyncData(
    /*[out, retval]*/ VARIANT* pData);

[Visual Basic 6]
Function ReadSyncData(
    ) As Variant

```

Parameters

pData

[out, retval] A pointer to a VARIANT data that is to hold the synchronization values.

Remarks

The data buffer is a VARIANT type of VT_ARRAY | VT_VARIANT, holding an array of VT_I4 32-bit integers. The data must be read as a variant array with each data element as an integer.

The synchronization values are defined as:

Table 22 Synchronization Event Values

Value	Synchronization Event
0x1	Sync1
0x2	Sync2
0x3	Sync1 and Sync2 at the same time

6.12 Optics Methods

The BSA Processor supports three different optical systems: LDA Optics, PDA Optics and DualPDA Optics.

To be able to measure zero velocities and to remove the directional ambiguity the BSA Processor supports frequency shifted beam pairs. The optical shift between the beams defines the measurement range and is typically 40 MHz for Dantec Dynamics optical systems.

The LDA Optical system defines the fringe spacing for each pair of beams (dimensions) leading to the conversion factors, converting measured frequency to velocity.

PDA and DualPDA Optical systems define the phase factors for converting phases into diameters. Along with the diameter calculation the phases are validated using spherical assumptions, and diameters are limited within a calculated range.

A practical way to determine the optical settings is to use BSA Flow Software to initially set up a system. The build-in optical calculations in BSA Flow Software can be used to find out the precise values of the optical properties. The optical output properties can be copied and exported from the BSA application, and then entered into your appropriate driver methods.

Figure 3 Optical Output Properties in BSA Flow Software.

Property	Optical PDA System - U1	Optical PDA System - U2
Frequency shift	4e+007	4e+007
Fringe spacing	3,996	3,79
Fringe direction	Positive	Positive
Phase factor P12	4,987	4,987
Phase factor P13	2,493	2,493

6.12.1 IBSA::SetOpticsShift

By introducing a shift of the frequencies between the two beams, the fringe pattern will move accordingly making it possible to determine the direction of the measured velocities. Since the measured frequency includes this shift frequency, you must take this optical shift into account when calculating the velocities.

```
[C++,IDL]
HRESULT SetOpticsShift(
    /*[in]*/ int Channel,
    /*[in]*/ float Shift);

[Visual Basic 6]
Sub SetOpticsShift(
    Channel As Long,
    Shift As Single)
```

Parameters

Channel

[in] Specifies the input channel

Shift

[in] Frequency shift in Hz for the give channel.

Remarks

Changing the optical frequency shift affects the center frequency of the signal and hence the possible selection of bandwidths. It is recommended to call this function before the **SetBandwidth** and **SetCenterFreq** methods.

Dantec Dynamics standard optics includes a Bragg cell using an optical shift frequency of 40 MHz, the FlowExplorer optics is using 80 MHz; other optical configurations may use other frequencies, please refer to the manual of the optics.

Specifying positive or negative values as the optical shift frequency dictates the direction of the frequency shift. A positive value indicated an optical up-shift, and a negative values an optical downshift.

The BSA Processor provides two frequency outputs than can be used as inputs to your Bragg cell in your optical system. Please refer to the **Set40MHzFreqShiftOutput** and **SetVarFreqShiftOutput** methods.

Default

Default frequency shift is 40.000.000 Hz = 40 MHz up-shift. This value is the standard shift frequency used by most Dantec Dynamics optical systems.

6.12.2 IBSA::GetOpticsShift

This method gets the optical shift frequency for a given channel earlier set by the method **SetOpticsShift**.

```
[C++,IDL]
HRESULT GetOpticsShift(
    /*[in]*/ int Channel,
    /*[out, retval]*/ float* pShift);

[Visual Basic 6]
Sub GetOpticsShift(
    Channel As Long
) As Single
```

Parameters

Channel

[in] Specifies the input channel

pShift

[out, retval] A pointer to the frequency shift in Hz for the specified channel.

Remarks

The optical parameters are stored in the driver and not in the processor. This means, that the optical parameters are reset when the driver is restarted.

The optical shift frequency will have the value 40.000.000 Hz = 40 MHz, when the driver is restarted.

6.12.3 IBSA::SetOpticsConversionFactor

Specifies the optical conversion factor for each LDA channel. The optical conversion factors are used in the transformation of frequencies to velocities, also known as the fringe spacing in the optics. If the conversion factors are defined the returned velocities read by calling the **ReadVelocityData** method, will be returned using the conversion factor.

```
[C++,IDL]
HRESULT SetOpticsConversionFactor(
    /*[in]*/ int Channel,
    /*[in]*/ float Factor);

[Visual Basic 6]
Sub SetOpticsConversionFactor(
    Channel As Long,
    Factor As Single)
```

Parameters

Channel

[in] Specifies the input channel

Factor

[in] Conversion factor for converting frequency to velocity.

Remarks

The conversion factors (also referred to as fringe factors) can be calculated using BSA Flow Software based on a specified optical PDA configuration. Please refer to the BSA Flow Software Installation and User's Guide for more information. For calibrated probes the conversion factor is directly indicated on the probe.

The unit of the conversion factor is normally meters (m). Since the velocities are measured in Hz, multiplying with the fringe factor will result in a velocity in meters pr. second (m/s). You can use another unit for the conversion factor to get the velocity results in the format you prefer.

This example shows how to read the velocity data in two different formats. The first format is m/s and by changing the conversion factor in ft/s.

[VBScript]

```
Dim aVel
' read velocity in Hz
aVel = MyBSA.ReadVelocityData(0, 0)
' set conversion factor Hz -> m/s
MyBSA.SetOpticsConversionFactor 0, 1.00e-006
' read velocity in m/s
aVel = MyBSA.ReadVelocityData(0, 0)
' set conversion factor Hz -> ft/s
MyBSA.SetOpticsConversionFactor 0, 3.28e-006
' read velocity in m/s
aVel = MyBSA.ReadVelocityData(0, 0)
```

Default

Default conversion factor is 1, meaning that all returned velocities are in frequency Hz.

6.12.4 IBSA::GetOpticsConversionFactor

This method gets the optical conversion factor for a given channel earlier set by the method **SetOpticsConversionFactor**.

[C++,IDL]

```
HRESULT GetOpticsConversionFactor(
    /*[in]*/ int Channel,
    /*[out, retval]*/ float* pFactor);
```

[Visual Basic 6]

```
Sub GetOpticsConversionFactor(
    Channel As Long
) As Single
```

Parameters

Channel

[in] Specifies the input channel

pFactor

[out, retval] A pointer to the conversion factor for the specified channel.

Remarks

The optical parameters are stored in the driver and not in the processor. This means, that the optical parameters are reset when the driver is restarted.

The optical conversion factor will have the value 1, when the driver is restarted.

6.12.5 IBSA::SetOpticsPhaseFactor

Specifies the optical phase factors for each phase channel. The optical phase factors are used in the transformation of measures PDA phase differences to a corresponding particle diameter.

If a PDA or DualPDA configuration is selected through the **SetConfiguration** method, this must be called prior to running an acquisition.

```
[C++,IDL]
HRESULT SetOpticsPhaseFactor(
    /*[in]*/ int Channel,
    /*[in]*/ float Phase);

[Visual Basic 6]
Sub SetOpticsPhaseFactor(
    Channel As Long,
    Phase As Single)
```

Parameters

Channel

[in] Specifies the phase channel.

Phase

[in] Specifies the phase factor.

Remarks

The phase factors can be calculated using BSA Flow Software based on a specified optical PDA configuration. Please refer to the BSA Flow Software Installation and User's Guide for more information. Alternatively the BSA Flow Software Script Generator can be used, please refer to section 5.2.

Channel 1 is representing P12 and channel 2 is representing P13.

The unit of the phase factors are in degrees pr. micrometer (deg / μm = 10^6 deg / m).

The maximum diameter is calculated based on the second phase channel (P13), using the formula: 260 deg / phase. The maximum diameter is used for validating the diameter data.

Table 23 PDA Phase Channels

Constant	Value
bsaPHASECHANNEL1	1
bsaPHASECHANNEL2	2

Default

Since this function must be called before running any PDA acquisition, no default values are specified.

6.12.6 IBSA::GetOpticsPhaseFactor

This method gets the optical phase factor for a given phase channel earlier set by the method **SetOpticsPhaseFactor**.

```
[C++,IDL]
HRESULT GetOpticsPhaseFactor(
    /*[in]*/ int Channel,
    /*[out, retval]*/ float* pPhase);

[Visual Basic 6]
Sub GetOpticsPhaseFactor(
    Channel As Long
) As Single
```

Parameters

Channel

[in] Specifies the phase channel

pPhase

[out, retval] A pointer to the phase factor for the specified phase channel in degrees pr. micrometer.

Remarks

The optical parameters are stored in the driver and not in the processor. This means, that the optical parameters are reset when the driver is restarted.

6.12.7 IBSA::SetOpticsValidationBand

Specifies the optical validation band. The optical validation band is used for PDA configurations in the determination of valid particles or diameters.

```
[C++,IDL]
HRESULT SetOpticsValidationBand(
    /*[in]*/ float Band);

[Visual Basic 6]
Sub SetOpticsValidationBand(
    Band As Single)
```

Parameter

Band

[in] Specifies the validation band in percent.

Default

The default value is 5 %.

6.12.8 IBSA::GetOpticsValidationBand

This method gets the optical validation band earlier set by the method **SetOpticsValidationBand**.

```
[C++,IDL]
HRESULT GetOpticsValidationBand(
    /*[out, retval]*/ float* pBand);

[Visual Basic 6]
Sub GetOpticsValidationBand(
    ) As Single
```

Parameters

pBand

[out, retval] A pointer to the validation band in %.

Remarks

The optical parameters are stored in the driver and not in the processor. This means, that the optical parameters are reset when the driver is restarted.

The optical validation band will have the value 5 %, when the driver is restarted.

6.12.9 IBSA::SetOpticsFringeDirection

Specifies the optical fringe shift direction. The optical fringe shift direction is used for PDA configurations in the determination of valid particles or diameters.

```
[C++,IDL]
HRESULT SetOpticsFringeDirection(
    /*[in]*/ int Direction);

[Visual Basic 6]
Sub SetOpticsFringeDirection(
    Direction As Long)
```

Parameter

Direction

[in] Specifies the direction of the fringes. A value larger than 0 means positive fringe direction, and a value less than 0 means negative fringe direction.

Default

The default value is 1 for positive direction.

6.12.10 IBSA::GetOpticsFringeDirection

This method gets the optical fringe direction earlier set by the method **SetOpticsFringeDirection**.

```
[C++,IDL]
HRESULT GetOpticsFringeDirection(
    /*[out, retval]*/ float* pDirection);

[Visual Basic 6]
Sub GetOpticsFringeDirection(
    ) As Single
```

Parameters

pDirection

[out, retval] A pointer to the fringe direction indicator.

Remarks

The optical parameters are stored in the driver and not in the processor. This means, that the optical parameters are reset when the driver is restarted.

The optical fringe direction will have the value 1, when the driver is restarted.

6.13 Event Handling

The BSA.Ctrl is able to fire events to the calling application or script. Events can be treated as asynchronous callbacks to your application from the driver.

Events are especially useful for returning status, errors, and to inform about states. The BSA.Ctrl supports two events; **onStatus** and **onAcqDone**.

```
[C++,IDL]
HRESULT onStatus();

[Visual Basic 6]
Event onStatus()
```

Parameters

This method has no parameters.

```
[C++,IDL]
HRESULT onAcqDone(
    /*[out, retval]*/ int* pFlag);

[Visual Basic 6]
Event onAcqDone(
    ) As Long
```

Parameters

pFlag

[out, retval] A pointer to a flag indicating why the acquisition is stopped. Refer to Table 24 Acquisition Done Flags for definitions.

Table 24 Acquisition Done Flags

Value	Synchronization Event
0	The acquisition stopped normally, on maximum number of samples or time, specified by the functions: SetMaxSamples or SetMaxAcqTime .
1	The user aborted the acquisition by calling the StopAcq function.
2	A timeout error occurred during acquisition. No data was received for a too long period of time. This timeout period can vary from system to system, but normally it is approximately the maximum acquisition time plus 1 sec.
3	An unknown error occurred during

	acquisition.
--	--------------

Exclusively for WSH there exists a special **ConnectObject** method for relating an object to an event method prefix. The event methods are hereafter a combination of the prefix and the event name, **prefix_eventname**.

[VBScript]

```
' creating the BSA.Ctrl object
dim MyBSA
set MyBSA = createobject("BSA.Ctrl")
' connects the object to an event handler
WScript.ConnectObject MyBSA, "MyBSA_"

' TODO code that generates an event

' event handler method for onStatus
sub MyBSA_onStatus()
    WScript.Echo "Status event received."
end sub

' event handler method for onAcqDone
sub MyBSA_onAcqDone(flag)
    WScript.Echo "Acquisition done event
    received."
end sub
```

Receiving events in HTML can be done using both JScript and VBScript through the **<object>** tag (supported from HTML 4.0). In the JScript example we load the object and assign the two events to the object using the **for-event** properties (these can be omitted in VBScript since the sub routines can be called directly). Again we use the **prefix_eventname** syntax for the event functions; witch in this case is automatically generated.

```
<head>
<!-- connecting the BSA.Ctrl to the
    MyBSA object -->
<object id="MyBSA"
    classid="clsid:83E16B05-CDDD-11D0-89E3-
        0000C0C27240">
</object>

<!-- assigning the MyBSA object to the
    event functions -->
<script language="javascript" for="MyBSA"
    event="onAcqDone(flag)">
    MyBSA_onAcqDone(flag)
</script>
<script language="javascript" for="MyBSA"
    event="onStatus()">
    MyBSA_onStatus(flag)
</script>
</head>

<body language="javascript"
    onload="onInitPage()">

<!-- called when HTML page is loaded -->
<script language="javascript">
function onInitPage()
{
    // TODO code that generates an event
}

<!-- event handler method for onStatus -->
function MyBSA_onStatus()
{
    alert("Status event received.");
}

<!-- event handler method for onAcqDone -->
function MyBSA_onAcqDone(flag)
{
    alert("Acquisition done event received.");
}
</script>

</body>
```

6.14 Error Handling

This section describes how to use the error handling build into the BSA.Ctrl. Most programming- and script languages provide error handling and error catching procedures.

General error handling requires knowledge of terms like error codes and exception handling; please seek for information on this elsewhere.

Errors from the BSA.Ctrl component can be divided into two types: **Method Errors** and **Processor Errors**.

6.14.1 Method Errors

Method errors are errors generated inside the method calls. These errors are generated as error return values or error exceptions. Since the component uses automation interfaces all errors will be generated and displayed as exceptions. In the following we will describe catching method errors in different languages.

We use a simple example where we call the **Disconnect** method without being connected. This will generate an error indicating that you must be connected to call this method. In all the examples we catch the error and display an error message dialog with the description of the error.

In VBScript error handling is limited to the **On Error** statement. Unlike **VBA** the **On Error GoTo** is not supported, therefore we use the **On Error Resume Next**, ignoring the error in the call itself for handling in the next statement. If **On Error** was omitted an internal exception handler in VBScript will catch the error and display a standard error message, but since we use the **Resume Next** the error handling can be left to us.

[VBScript]

```
' creates the object
dim MyBSA
set MyBSA = createobject("BSA.Ctrl")

' if an error occurs goto the next line
' instead of throwing an exception
on error resume next

' call disconnect without a connect call
' will generate an error
MyBSA.Disconnect

' catch the error, and display the error
' description in a message box
if Err.Number<>0 Then
    MsgBox "MyError: " & Err.Description
    Err.Clear
end if

' clean up
set MyBSA = nothing
```

In JScript exception handling can be performed with a **try-catch** statement. The method causing an error must be placed inside the **try** statement, and the error handling must be handled in the **catch** statement.

[JScript]

```
var MyBSA = new ActiveXObject("BSA.Ctrl");

// error handling in jscript is done through
// try-catch exception handling
try {
    MyBSA.Disconnect();
}
catch(e) {
    WScript.Echo("MyError: " + e.description);
}
```

In C++ this is done in the same way; using a **try-catch** statement. However we must use the **_com_error** C++ class to get the description information, since the C compiler select between different exception types.

[C++]

```
::CoInitialize(NULL);

IBSA* pMyBSA = NULL;
if (SUCCEEDED(::CoCreateInstance(
    __uuidof(BSA), NULL,
    CLSCTX_INPROC_SERVER, __uuidof(IBSA),
    reinterpret_cast<void*>(&pMyBSA)))
{
    try{
        pMyBSA->Disconnect();
    }
    catch(_com_error &e)
    {
        ::MessageBox(NULL,
            "MyError: " + e.Description(),
            "Dantec Dynamics BSA.Ctrl "//
            "Runtime Error", MB_OK);
    }
}
```

The above examples all displays a similar error dialog with the error description.

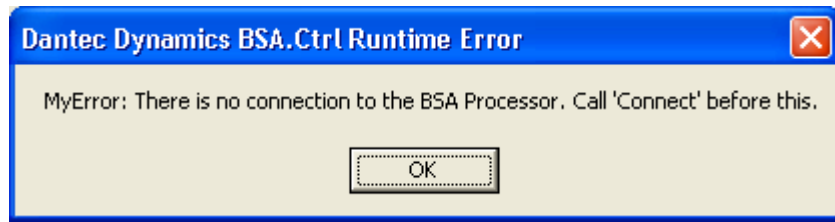


Figure 4 Sample run-time error dialog.

In special cases it is use full to provide your own error handling on methods. It is not necessary to handle all errors for all methods. It is only advisable to handle errors on the most critical methods. In most cases the error handling can be left to the build-in exception handler of the operating system.

6.14.2 Processor Errors

Processor status and errors are retrieved through the **GetStatus** method. Since the component does not support sending events this function must be called explicitly, when status is needed. Errors and other status information is most likely to be generated in the BSA Processor during configuration and acquisition. Consequently we recommend that calls to **GetStatus** are always performed after a call to the **SetConfiguration** and **StartAcq** methods.

As the BSA Processor can send more errors and information to the driver, the **GetStatus** call must be repeated until there is no return. In the following examples it is shown how the **GetStatus** can be called repeatable for different languages.

This example in VBScript shows how to use the **Do While** statement to display all the status messages from the processor.

[VBScript]

```
dim strErr
strErr = MyBSA.GetStatus
do while strErr<>""
    MsgBox "MyError: " + strErr
    strErr = MyBSA.GetStatus
loop
```

In JScript we can do the same with a single **While** statement.

[JScript]

```
var strErr = MyBSA.GetStatus();
while (strErr!="")
{
    WScript.Echo("MyError: " + strErr);
    strErr = MyBSA.GetStatus();
}
```

And the same can be done in C++.

[C++]

```
_bstr_t strErr = pMyBSA->GetStatus();
while (strErr.length() != 0)
{
    ::MessageBox(NULL,
        "MyError: " + strErr,
        "Dantec Dynamics BSA.Ctrl "//
        "Runtime Error", MB_OK);
    strErr = pMyBSA->GetStatus();
}
```

Here is an even simpler example using MFC.

[MFC]

```
CString strErr = pMyBSA->GetStatus();
while (!strErr.IsEmpty())
{
    AfxMessageBox(strErr);
    strErr = pMyBSA->GetStatus();
}
```

7. BSA System Monitor Function Reference

The purpose of this chapter is to give an understanding of the functions in the IBSASysMon interface for the BSA Processor Driver. For each function there will be a short introduction followed by a description of the parameters to the given function. For practical examples see the code examples in this reference manual and on the CD in the documentation folder. Here you can find examples written in LabVIEW, Visual Basic, MFC and HTML that shows how to use the BSA.SysMon.Ctrl component.

The component only uses two methods for connecting to and disconnecting from the BSA Processor. However the SysMon.Ctrl requires that the System Monitor is activated before displaying any information. To activate the System Monitor calls the BSA.Ctrl **ActivateSysMon** method or starts the System Monitor in the BSA Flow Software. Please refer to the HTML Example section.

Connecting Methods	Description
Connect	Connects the System Monitor to the BSA Processor.
Disconnect	Disconnects the System Monitor from the BSA Processor.

7.1 Connecting

This section describes the methods that are involved in connecting the System Monitor Ctrl to the BSA processor. While connected; the System Monitor Ctrl will display online information from the BSA Processor.

7.1.1 IBSASysMon::Connect

This method connects the System Monitor component to the BSA Processor through the IP-address of the processor. The System Monitor must be activated before showing any data.

```
[C++,IDL]
HRESULT Connect(
    /*[in]*/ BSTR IPAddress);

[Visual Basic 6]
Sub Connect(
    IPAddress As String)
```

Parameters

IPAddress

[in] The IP-address of the processor as it is seen on the network. The syntax consists of a common dotted octet IP-address, for example 10.10.100.100.

Remarks

It is possible to have multiple instances of the System Monitor component connected at the same time.

Normally the processor will keep a fixed IP-address. The default IP-address is 10.10.100.100, but can be changed, please refer to the BSA Flow Software Installation and User's Guide. It is also possible to assign the processor a dynamic IP-address if the network supports DHCP, please refer to your network administrator. In this case the IP-address must be the name of the processor, please refer to the BSA Flow Software Installation and User's Guide.

7.1.2 IBSASysMon::Disconnect

Calling this method disconnects the System Monitor component from the BSA Processor. Calling the IBSASysMon::Connect method will connect to the System Monitor.

```
[C++,IDL]
HRESULT Disconnect();

[Visual Basic 6]
Sub Disconnect()
```

Parameters

This method has no parameters.

Remarks

Disconnecting then System Monitor component will stop the System Monitor for receiving data from the BSA Processor. The System Monitor component will be removed from the BSA Processors list of listening System Monitor clients.

7.2 Using the System Monitor

The System Monitor is a valuable tool for monitoring the current online condition of the signal. You will instantaneously have an indication of the how your processor parameter settings match the signal in an oscilloscope like display.

In addition to the oscilloscope display it includes panes showing the photo-multiplier anode current and high voltage. Information about the validation of the signal is also provided.

During acquisition you will receive notifications about the progress and state of the acquired data. If the sync1 or sync2 inputs are used, the synchronization frequency is also displayed.

The two vertical dotted lines in the scope display indicate the record length that is passed to the FFT processor to determine the Doppler frequency. Recognizing a good Doppler burst requires some experience. The validation rate is a good indicator. You can see the validation rate in the System Monitor window. Optimize data rate and validation by adjusting the high voltage, signal gain and record length settings. The record interval should be shorter than or equal to the length of the displayed bursts.

8. BSA.Ctrl Constants

For convenience a number of constants are defined. These constants are located in the BSA.Ctrl module and can be accessed by languages that import the definitions.

Constant	Value
bsaNONE	0
bsaBNC1	1
bsaBNC2	2
bsaBNC3	3
bsaDIFFERENTIAL1	4
bsaDIFFERENTIAL2	5
bsaDIFFERENTIAL3	6
bsaDSUB1	7
bsaDSUB2	8
bsaDSUB3	9
bsa1DLDA	0
bsa2DLDACOINC	1
bsa3DLDACOINC	2
bsa2DL DANONCOINC	3
bsa3DL DANONCOINC	4
bsa3DL DASEMICOINC	5
bsa1DPDA	6
bsa2DPDACOINC	7
bsa3DPDACOINC	8
bsa2DPDANONCOINC	9
bsa3DPDANONCOINC	10
bsa3DPDASEMICOINC	11
bsa2DDUALPDACOINC	12
bsa3DDUALPDACOINC	13
bsa3DDUALPDASEMICOINC	14
bsaINTERNCLK	0
bsaEXTERNCLK	1
bsaNEGATIVE	0
bsaPOSITIVE	1
bsaAUTOMATIC	1
bsaHV	2
bsaOPTICAL	0
bsaINTERNAL	1
bsaZERO	2
bsaOFF	0
bsaON	1
bsaBURST	0
bsaCONTINUOUS	1
bsaDEADTIME	2
bsaNOSHIFT	0
bsa40MHZSHIFT	1
bsaVARSHIFT	2
bsaEXTSHIFT	3

bsaUP	0
bsaDOWN	1
bsaDISABLE	0
bsaBURSTDetect1	1
bsaBURSTDetect2	2
bsaBURSTDetect3	3
bsaBURSTDetect4	4
bsaBURSTDetect5	5
bsaBURSTDetect6	6
bsaSTARTMEASUREMENT	7
bsaUNUSED	0
bsaREC16	16
bsaREC32	32
bsaREC64	64
bsaREC128	128
bsaREC256	256
bsaFIXED	0
bsaAUTO	1
bsaBURSTSIGNAL	1
bsaBURSTSPECTRUM	2
bsaFREERUN	-1
bsaINDIVIDUAL	0
bsaGROUP1	0
bsaGROUP2	1
bsaGROUP3	2
bsaGROUP4	3
bsaCHANNEL1	0
bsaCHANNEL2	1
bsaCHANNEL3	2
bsaCHANNEL4	3
bsaPDACHANNEL1	0
bsaPHASECHANNEL1	1
bsaPHASECHANNEL2	2
bsaPDACHANNEL2	3
bsaPDACHANNEL3	4
bsaPDACHANNEL4	5
bsaWINDOW	0
bsaOVERLAP	1
bsaLEVEL2	2
bsaLEVEL4	4
bsaLEVEL8	8
bsaLEVEL10	10
bsaLEVEL12	12
bsaLEVEL16	16
bsaLDA	0
bsaPDA	1
bsaAUTOCALIBRATE	0
bsaALWAYSALIBRATE	1
bsaNEVERCALIBRATE	2
bsaDataAAT	0x001
bsaDATATT	0x002
bsaDATAVEL	0x004

bsaDATAENC	0x008
bsaDATADIA	0x020
bsaDATASYNC	0x040
bsaDataAALL	0xFFF

9. Examples

This chapter shows a number of examples calling the driver from different programming languages and script languages. These samples does not necessary provide a full functional application, but gives an impression of how to call the driver and what is necessary for calling the driver. Some examples are not fully described in source code in this chapter, but can be found on the installation CD in its full contents. Examples can also be downloaded from the Dantec Dynamics web site.

9.1 VBScript Example

```
Option Explicit

'=====
' VBScript Example
' This example is made for the Dantec Dynamics BSA
' Processor Driver
' Copyright Dantec Dynamics A/S, 2003 All Rights Reserved.
'
' This example will write time and velocity to a text file.
'=====

' creates objects for use
Const ForReading = 1, ForWriting = 2, ForAppending = 8
Dim MyBSA, FSO
Set MyBSA = CreateObject("BSA.Ctrl")
Set FSO = CreateObject("Scripting.FileSystemObject")

' connects to the BSA Processor
MyBSA.Connect "10.10.100.100", True

' specify 1D LDA configuration
MyBSA.SetProcessor 0
MyBSA.SetConfiguration 0

'=====
' TODO set and change properties for your processor
'=====

' set max. acquisition time/max. samples
Dim wFile,i,numSamples,safeArray1,safeArray2
MyBSA.SetMaxSamples 0, 200000
MyBSA.SetMaxAcqTime 0, 1

' start acquisition
MyBSA.StartAcq True

' read acquired data
numSamples = MyBSA.ReadDataLength(0)
safeArray1 = MyBSA.ReadArrivalTimeData(0)
safeArray2 = MyBSA.ReadVelocityData(0,0)

' write acquired data to file
```

```
// TODO change the file location if not appropriate
set wFile = FSO.CreateTextFile("c:\bsa_data.txt",
    ForWriting, True)
For i=0 To numSamples - 1
    wFile.WriteLine(CStr(safeArray1(i)) + Chr(9) +
        CStr(safeArray2(i)))
Next

' clean up
wFile.Close
Set FSO = Nothing
Set MyBSA = Nothing
```

9.2 JScript Example

```
//=====
// JScript Example
// This example is made for the Dantec Dynamics BSA
// Processor Driver
// Copyright Dantec Dynamics A/S, 2003 All Rights Reserved.
//
// This example will write time and velocity to a text file.
//=====

// creates objects for use
var ForReading = 1, ForWriting = 2, ForAppending = 8
var MyBSA = new ActiveXObject("BSA.Ctrl");
var FSO = new ActiveXObject("Scripting.FileSystemObject");

// connects to the BSA Processor
MyBSA.Connect("10.10.100.100", true);

// specify 1D LDA configuration
MyBSA.SetProcessor(0);
MyBSA.SetConfiguration(0);

// set max. acquisition time/max. samples
MyBSA.SetMaxSamples(0, 200000);
MyBSA.SetMaxAcqTime(0, 1);
//=====
// TODO set and change properties for your processor
//=====

// start acquisition
MyBSA.StartAcq(true);
// read acquired data
var numSamples = MyBSA.ReadDataLength(0);
var safeArray1 = MyBSA.ReadArrivalTimeData(0);
var safeArray2 = MyBSA.ReadVelocityData(0, 0)

// convert safearrays from vb style to java
// this is nessecery for java to read data in safearray format
var vbArray1 = new VbArray(safeArray1);
var jArray1 = vbArray1.toArray();
var vbArray2 = new VbArray(safeArray2);
var jArray2 = vbArray2.toArray();

// write acquired data to file
//=====
// TODO change the file location if not appropriate
//=====
var wfile = FSO.CreateTextFile("c:\\bsa_exjs.txt",
                               ForWriting, true);
for (var i=0;i<numSamples - 1;i++)
{
    wfile.WriteLine(jArray1[i].toString() + "\t" +
                    jArray2[i].toString());
}

// clean up
wfile.Close
```

```
delete FSO;
delete MyBSA;
```

9.3 MS Excel VBA Example

```
Option Explicit

'=====
' Excel VBA Example
' This example is made for the Dantec Dynamics BSA
' Processor Driver
' Copyright Dantec Dynamics A/S, 2003 All Rights Reserved.
'
' This example will display a time-series plot as an
' Excel Scatter Chart.
'=====

Sub BSACtrl()

' creates objects for use
Dim MyBSA
Set MyBSA = CreateObject("BSA.Ctrl")

'=====
' TODO change IP-address for your processor
'=====

' connects to the BSA Processor
MyBSA.Connect "10.10.100.100", True

Dim numSamples, safeArray1, safeArray2
' sets configuration to 1D LDA configuration
MyBSA.SetProcessor 0
MyBSA.SetConfiguration 0

' set max. acquisition time/max. samples
MyBSA.SetMaxSamples 0, 200000
MyBSA.SetMaxAcqTime 0, 1
'=====
' TODO set and change properties for your processor
'=====

' start acquisition
MyBSA.StartAcq True

' read acquired data
numSamples = MyBSA.ReadDataLength(0)
safeArray1 = MyBSA.ReadArrivalTimeData(0)
safeArray2 = MyBSA.ReadVelocityData(0)

Sheets("Sheet1").Activate
Application.ScreenUpdating = False

' write data out to Excel columns
range("A1").Activate
```



```

Dim Counter
For Counter = 0 To UBound(safeArray1)
    ActiveCell.Value = safeArray1(Counter)
    ActiveCell.Offset(1, 0).Activate
Next
range("B1").Activate
For Counter = 0 To UBound(safeArray2)
    ActiveCell.Value = safeArray2(Counter)
    ActiveCell.Offset(1, 0).Activate
Next

' adds a Excel scatter chart, and display data
Charts.Add
ActiveChart.ChartType = xlXYScatter
ActiveChart.Location Where:=xlLocationAsObject, NAME:="Sheet1"
ActiveChart.SetSourceData Source:=Sheets("Sheet1").range("A:B")
ActiveChart.PlotBy = xlColumns

Application.ScreenUpdating = True

End Sub

```

9.4 HTML Example

This example shows how to activate the System Monitor from the BSA.Ctrl and how to display the System Monitor embedded on a web page using the SysMon.Ctrl.

The example uses a script block in VBScript to create an instance of the BSA.Ctrl component and to call the initial functions. The script block is placed in the body part of the page, which means that it will be called every time the page is loaded.

The SysMon.Ctrl is placed in an object block, referencing the GUID of the component. The GUID of the SysMon.Ctrl component is not directly visible, but most HTML editors allow for components to be inserted directly through the GUI. As an example; in MS FrontPage you can select *Insert* | *Advanced* | *ActiveX Control* and get the dialog below:

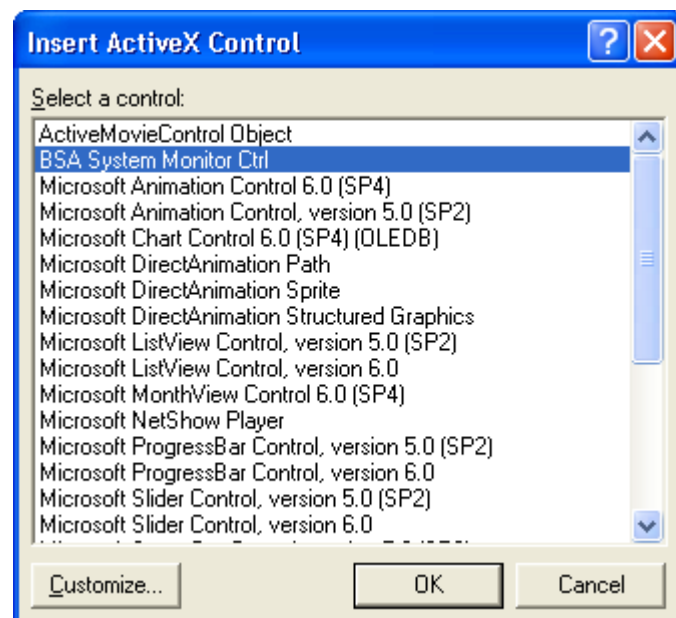


Figure 5 FrontPage dialog for inserting an ActiveX control.

Inserting the component will automatically generate the code.

[MS FrontPage Generated Code]

```
<object
  classid="clsid:83E16B06-CDDD-11D0-89E3-
  0000C0C27240"
  width="14" height="14">
</object>
```

The SysMon.Ctrl can be accessed through its ID parameter or object name, as can be seen in the two calls to START and STOP in the example.

```

<!--
HTML Example
This example is made for the Dantec Dynamics BSA
Processor Driver
Copyright Dantec Dynamics A/S, 2003 All Rights Reserved.

This example will activate the BSA System Monitor using
the BSA.Ctrl and display the System Monitor using the
BSA.SysMon.Ctrl
-->

<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
  charset=windows-1252">
<title>BSA System Monitor HTML Example</title>
</head>

<body>

<script language="vbscript">
Set MyBSA = CreateObject("BSA.Ctrl")
' connects to the BSA Processor
MyBSA.Connect "10.10.100.100", True
' activates the system monitor
MyBSA.ActivateSysMon
</script>

<object
  classid="clsid:83E16B06-CDDD-11D0-89E3-0000C0C27240"
  id="MySysMon" width="300" height="400">
</object>

<br>
<a href="javascript:MySysMon.Connect('10.10.100.100') ">
  START</a>
<br>
<a href="javascript:MySysMon.Disconnect() ">
  STOP</a>

</body>
</html>

```

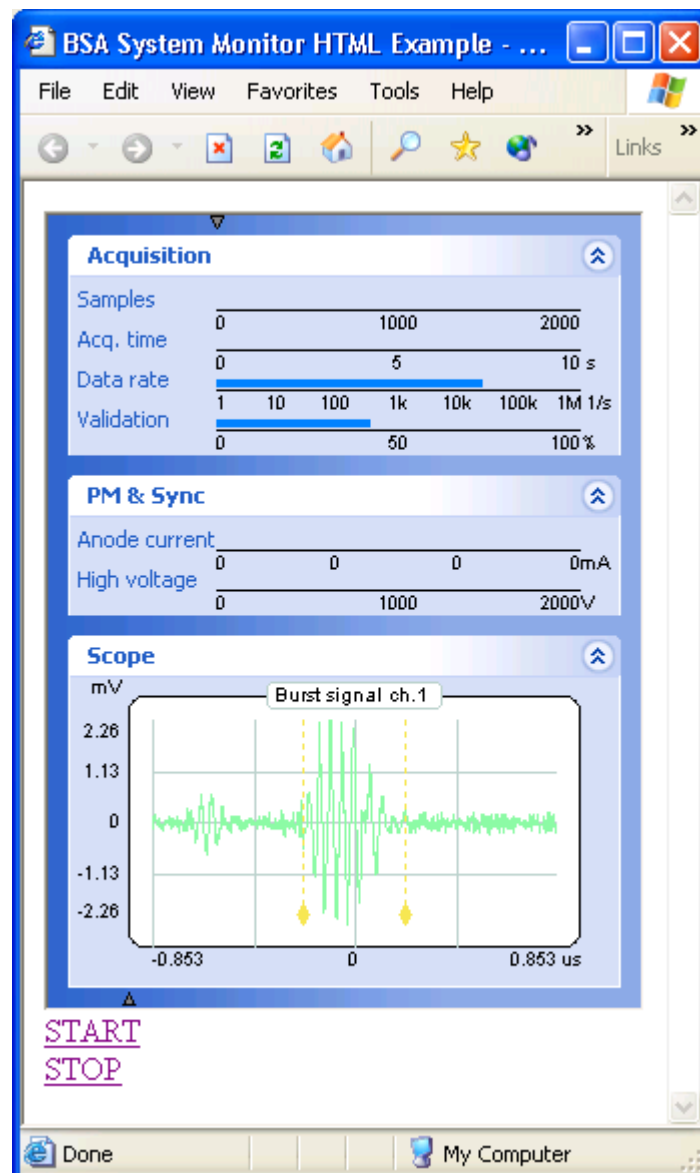




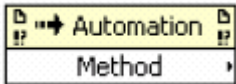

Figure 6 HTML example of embedding the System Monitor.

9.5 LabVIEW Example

To start ActiveX programming in LabVIEW, add an Automation Refnum to the frontpanel and select the proper ActiveX class. See section 6.3.2 how to do this.

Figure 7 shows LabVIEW's function palette to work with ActiveX. It has three relevant functions: Automation Open, Automation Close, and Invoke Node, that work with COM methods, and one relevant data conversion functions: Variant To Data, that work with COM data types. The Property Node and To Variant functions are not used by any of the components described in this document.

Table 25 Relevant LabVIEW functions.

Automation Function	Description
Automation Open 	Used to open an Automation refnum, which points to a specific ActiveX object.
Automation Close 	Used to close an Automation refnum. You should close an open Automation refnum when you no longer need it open.
Invoke Node 	Used to invoke a method or action on an ActiveX object.
Variant To Data 	Used to convert Variant Data to data that can be displayed or processed in LabVIEW.

To bring up these functions select the *Functions*| *Communications*| *ActiveX* palette list.

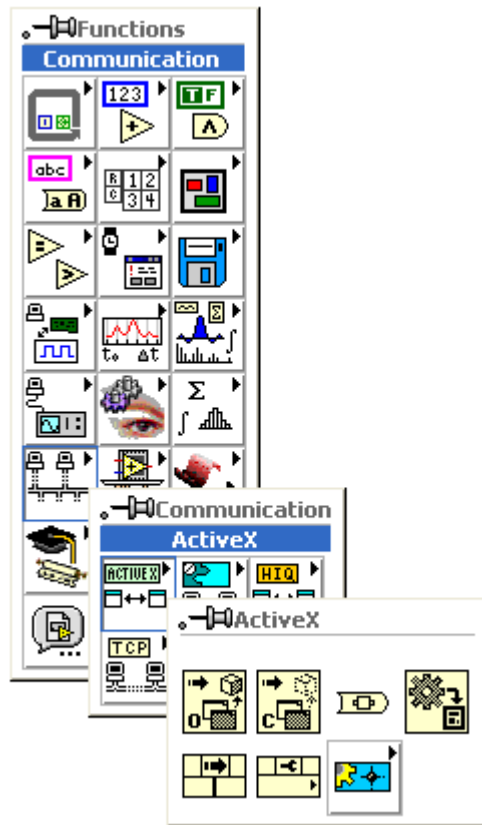


Figure 7 LabVIEW ActiveX palette.

Now we will use these LabVIEW functions to develop a sample VI, which will connect to the BSA.Ctrl, and read arrival time data and velocity data and display it as a time serie.

The following will provide a description of each part of the diagram to make this.

The diagram is divided into the following parts:

1. Connect to the BSA Processor
2. Disconnect from it
3. Get the connection state
4. Set one property at a time
5. Get a list of current properties
6. Start an acquisition
7. Stop it
8. Show data in a graph

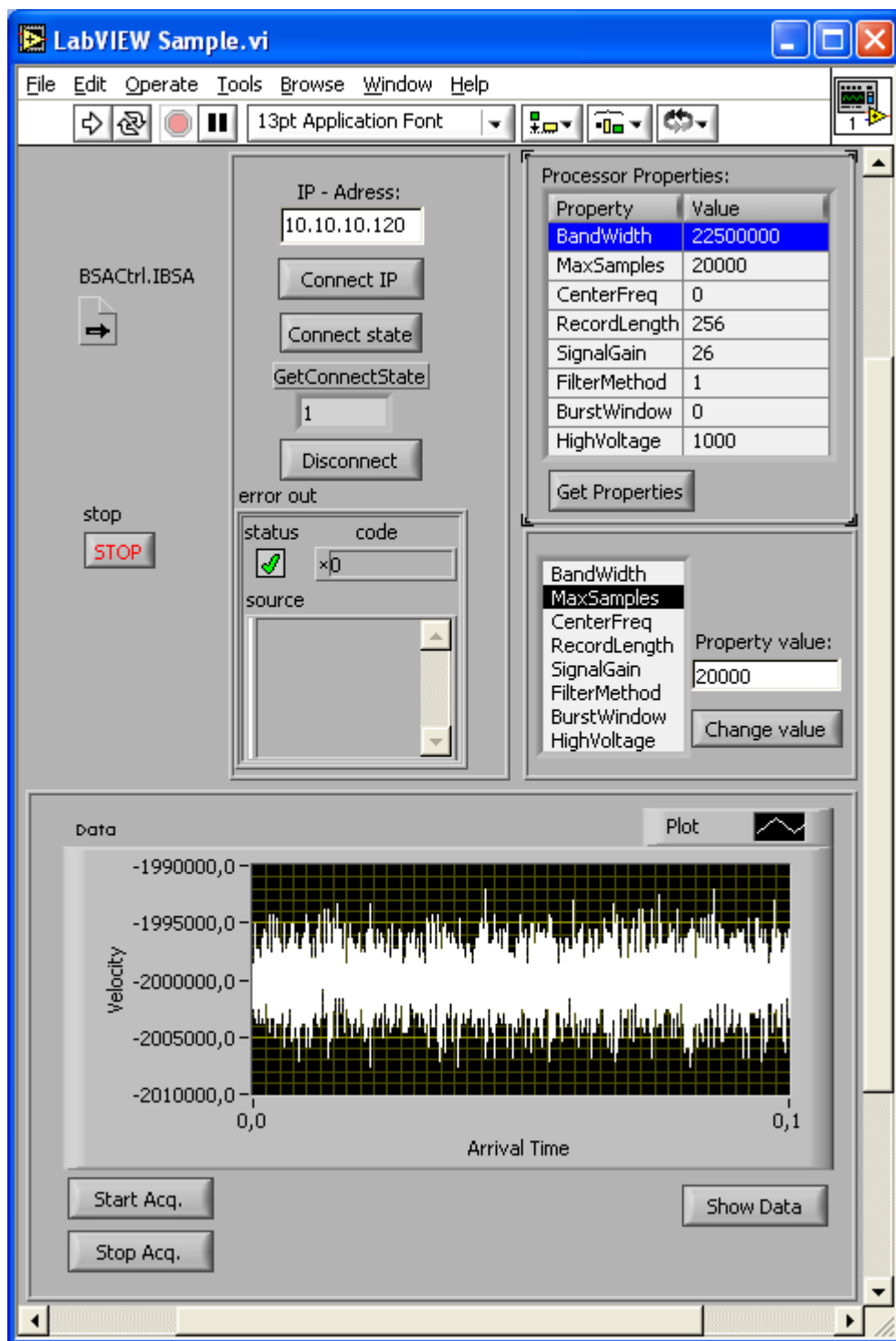


Figure 8 LabVIEW example control panel.

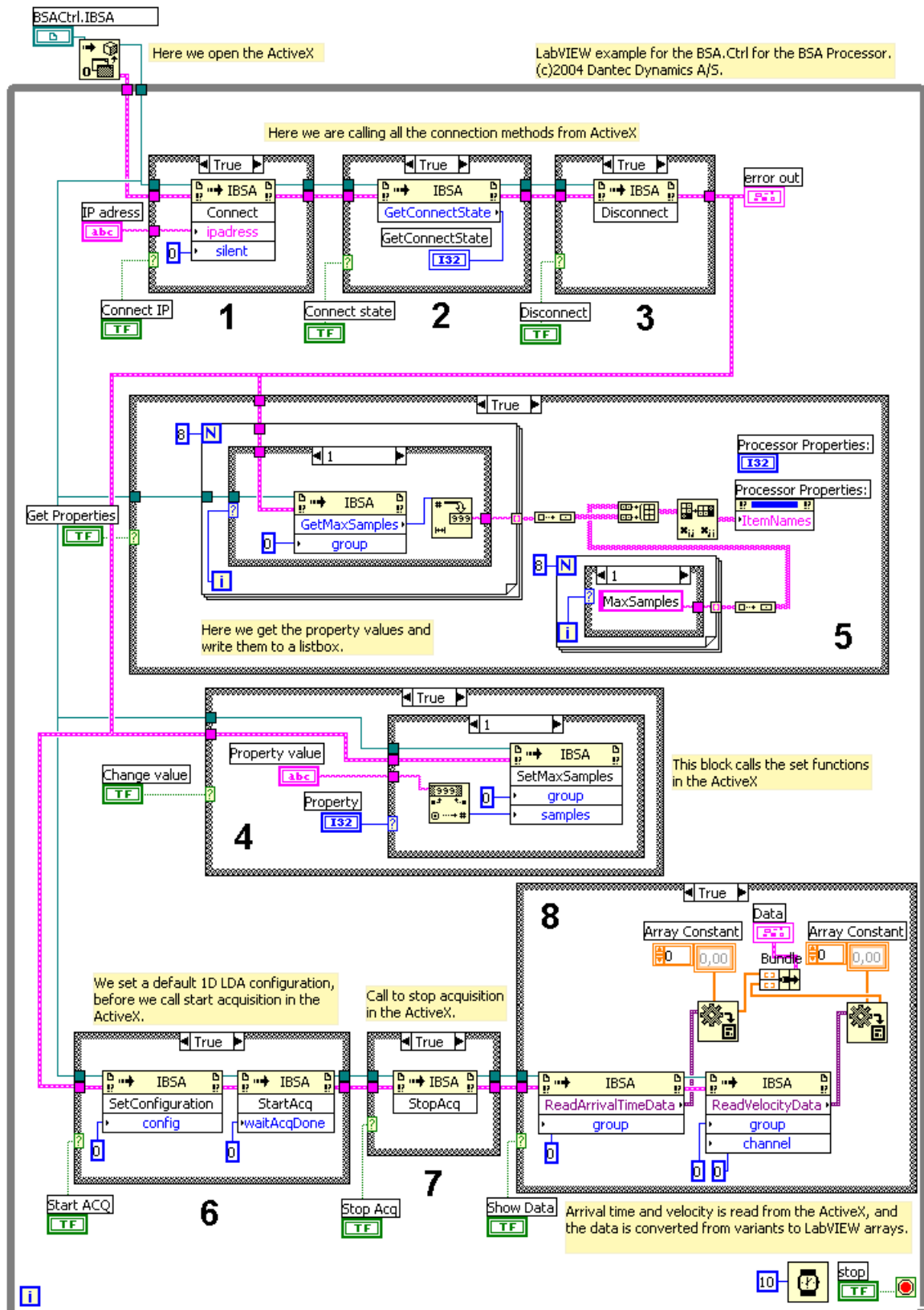


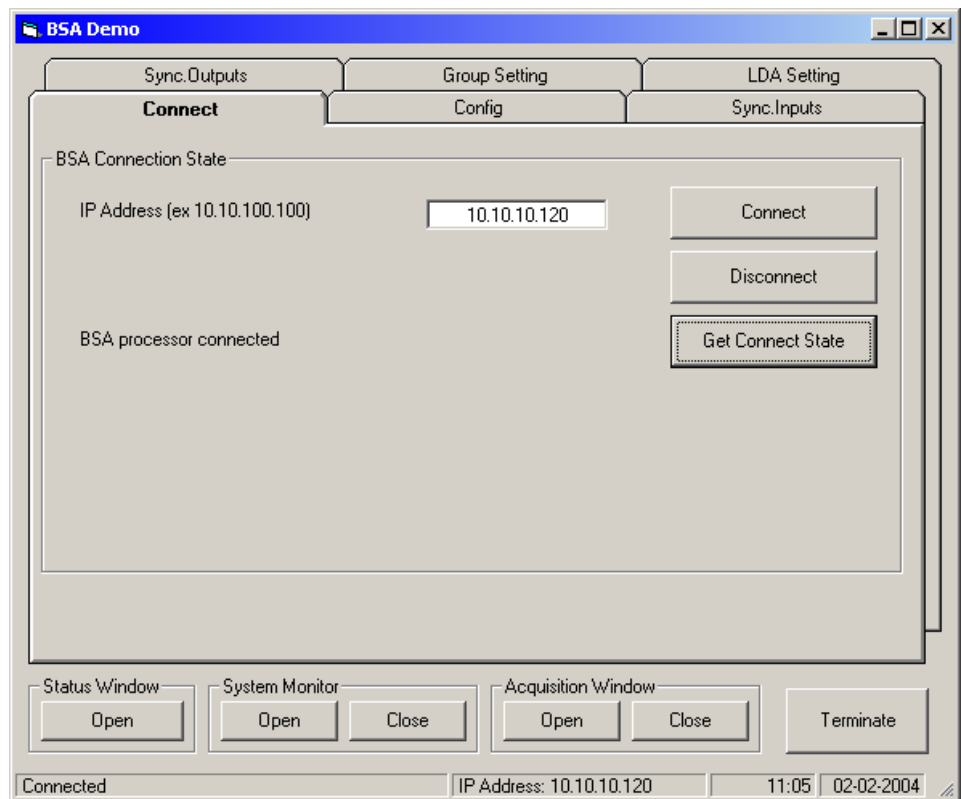
Figure 9 LabVIEW example diagram.

9.6 Visual Basic Example

On the CD a full Visual Basic 6 project is included. This program shows how to make an entire application using the BSA.Ctrl and BSA.SysMon.Ctrl. The program can be used to set and read all properties in the driver. It can start an acquisition and show online data in the System Monitor.

Please refer to the CD for more information. The following are screenshots from the program.

From the Connect page you can specify the IP-address of the BSA Processor. By pressing the buttons you can connect and disconnect from the processor and get the connection status.



The Config page sets the configuration along with global processor properties.

BSA Demo

Sync.Outputs Group Setting LDA Setting

Connect **Config** Sync.Inputs

BSA Configuration

☒ Activate System Monitor 1D LDA Set

Bragg Cell Output

☒ 40 MHz Frequency Get Set

☐ Variable Frequency = 40000000. Hz Get Set

Advanced Acquisition Setting

High Voltage Activation Automatic Get Set

Anode current Warning Level 90 % Get Set

Data Collection Mode Burst Get Set

Duty Cycle 100.00 Get Set

Dead Time 0.0001 Get Set

Status Window System Monitor Acquisition Window

Open Open Close Open Close Terminate

Connected IP Address: 10.10.10.120 11:06 02-02-2004

From the Group Settings page all properties related to coincidence groups can be set.

BSA Demo

Connect Config Sync.Inputs

Sync.Outputs **Group Setting** LDA Setting

Acquisition parameters

	Group 1 (setting)	Group 2 (setting)	Group 3 (setting)
Maximum samples to collect	2000	Unknown	Unknown
Maximum acquisition time	10	Unknown	Unknown

Coincidence scheme

Coincidence Filter Mode Burst Overlap Unknown Unknown

Size of burst window (%) 0.00001 Unknown Unknown

Scope display

ZOOM factor in %	400	Text1	Text1
Doppler Monitor Output Enabled	1	unknown	unknown
Doppler Monitor Triggered from Channel	0	unknown	unknown

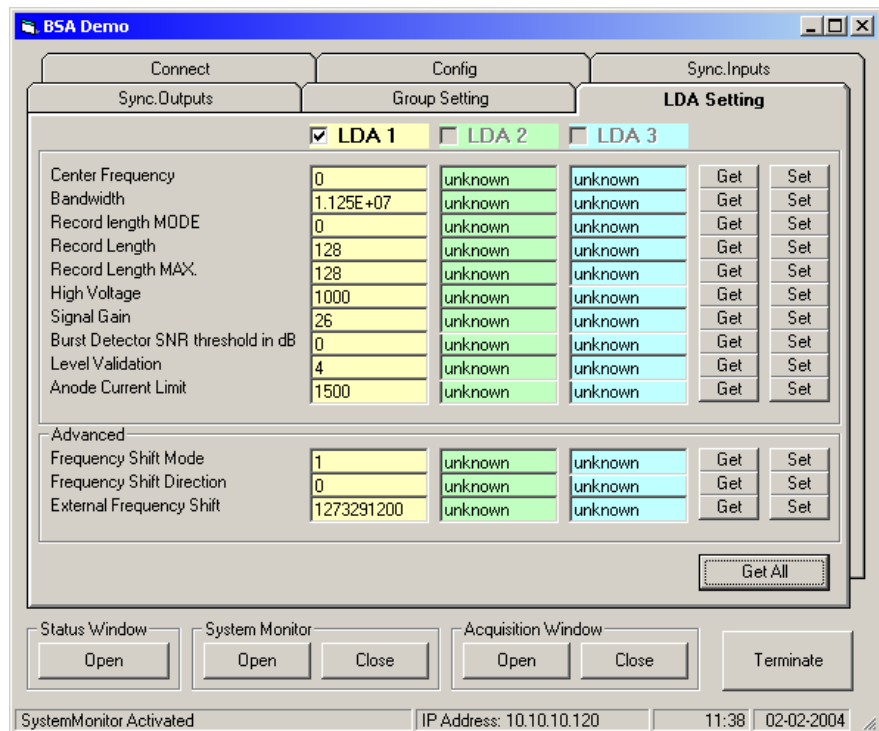
Get Set

Status Window System Monitor Acquisition Window

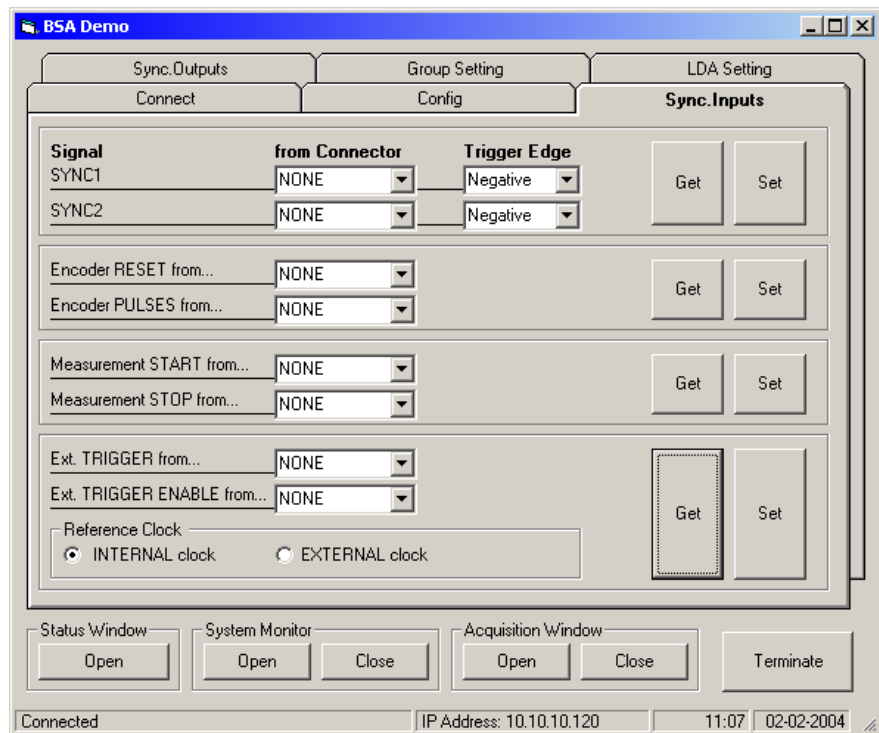
Open Open Close Open Close Terminate

Connected IP Address: 10.10.10.120 11:08 02-02-2004

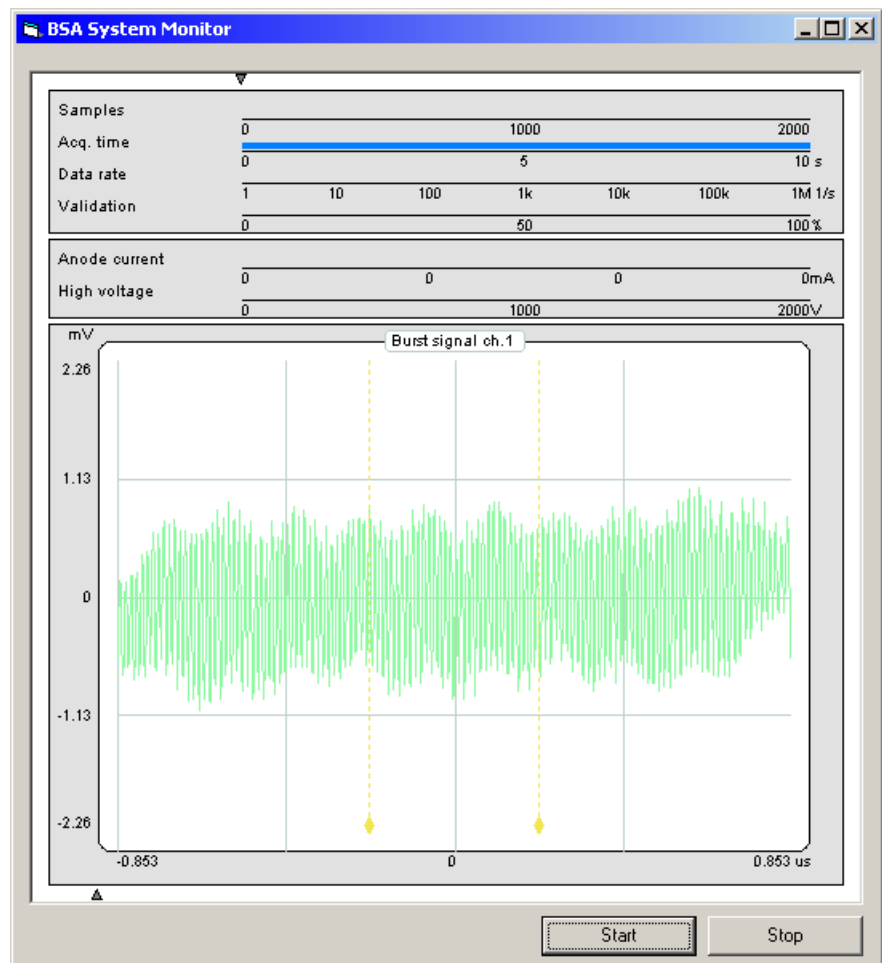
On the LDA Settings page all LDA channel properties can be set.



On the Sync. Inputs page all synchronization settings can be set.



When selecting System Monitor Open, the online system monitor window appears.



Before starting the system monitor, make sure the **Activate System Monitor** checkbox on the Config-TAB, is checked.

Then use the **start** button to run the system monitor. – Correct set-up and the presence of valid LDA signals are assumed.

9.7 MATLAB Example

MATLAB is able to call ActiveX components from v5.3. Newer versions of MATLAB includes better support for ActiveX components, this example is made using v5.3.

This example connects to a LDA processor with a 1D LDA configuration. The result is a MATLAB figure window displaying the System Monitor, and a time series and histogram plots of the measured velocities.

The API can be called from the command window or from an m-file. The following example consists of two m-files, one for creating the controls, connecting, measuring and plotting, and one for ending and cleaning up. Be sure to place the m-files in a path known by MATLAB; global paths can be found using the **path** command.

```
%=====
% MATLAB Example
% This example is made for the Dantec Dynamics BSA
% Processor Driver
% Copyright Dantec Dynamics A/S, 2005 All Rights Reserved.
%
% This example will run a 1D LDA measurement showing the
% System Monitor control inside a MATLAB figure, together
% with a time series and histogram of the velocities.
%=====

% create the BSA.Ctrl object
global MyBSA;
MyBSA = actxserver('BSA.Ctrl');

% create the BSA.SysMon.Ctrl object in a figure window
f = figure('pos',[300 300 500 500]);
global MySysMon;
MySysMon = actxcontrol('BSA.SysMon.Ctrl',[0 0 250 500],f);

% call connect the driver, and wait for the driver to connect
invoke(MyBSA,'Connect','10.10.10.130',0);
while invoke(MyBSA,'GetConnectState')==0
end

% call connect on the system monitor
invoke(MySysMon,'Connect','10.10.10.130');

% set configuration to LDA processor running 1D LDA
invoke(MyBSA,'SetProcessor',0);
invoke(MyBSA,'SetConfiguration',0);

% activates the system monitor
invoke(MyBSA,'ActivateSysMon');

%=====
% TODO set and change properties for your processor
%=====
```

```

% set max. acquisition time/max. samples
invoke(MyBSA, 'SetMaxSamples', 0, 2000);
invoke(MyBSA, 'SetMaxAcqTime', 0, 1.0);

% only allocate memory for arrival time and velocity
invoke(MyBSA, 'PreAllocateData', bitor(hex2dec('001'), ...
    hex2dec('004')));

% start acquisition
invoke(MyBSA, 'StartAcq', 1);

% read acquired data
Num = invoke(MyBSA, 'ReadDataLength', 0);
ATData = invoke(MyBSA, 'ReadArrivalTimeData', 0);
AT = cell2struct(ATData, {'Data'}, Num);
VelData = invoke(MyBSA, 'ReadVelocityData', 0, 0);
Vel = cell2struct(VelData, {'Data'}, Num);

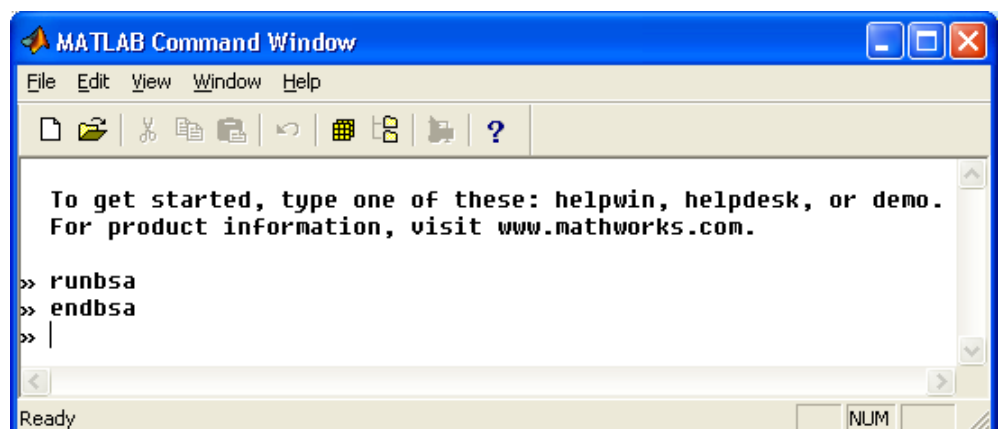
% display data in the current figure
subplot(2, 2, 2);
plot([AT.Data], [Vel.Data]);
axis([0 0.03 3500000 4500000]);
subplot(2, 2, 4);
hist([Vel.Data], 500);
axis([3500000 4500000 0 2000]);

% clean up
% call the endbsa.m routine

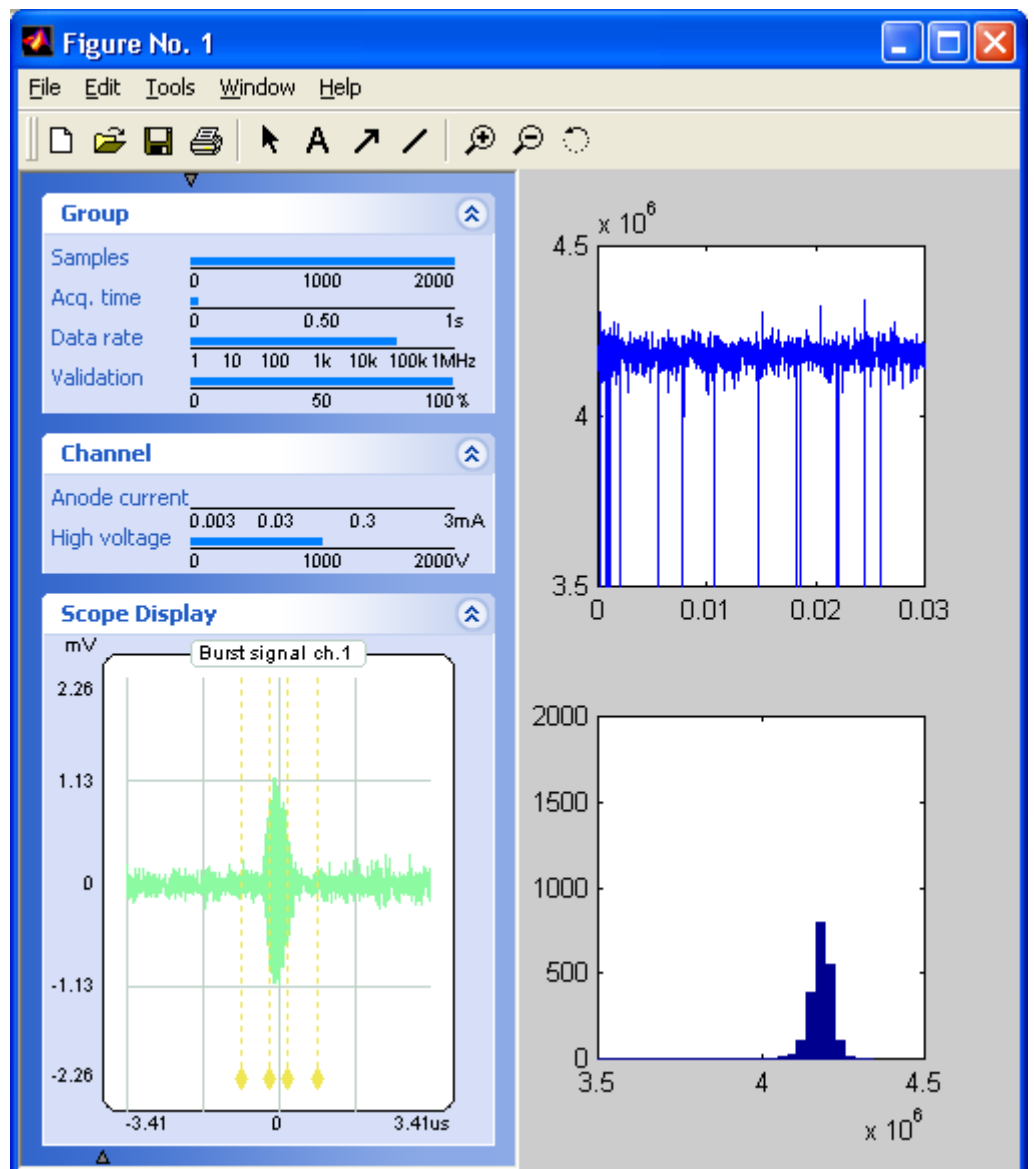
```

Call the function script directly from the MATLAB Command Windows using the m-file name **runbsa**, and when finished call **endbsa**.

Figure 10 MATLAB Command Windows calling script running the BSA.Ctrl and BSA.SysMon.Ctrl.

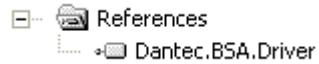


Running the sample produces the following figure, displaying a running System Monitor and a small acquisition.



9.8 C# .NET Example

When using the .NET version of the BSA Processor Driver a reference to the **Dantec.BSA.Driver** assembly must be added to the project.



Please note that the .NET samples included in the installation can have a broken reference path to the **Dantec.BSA.Driver** assembly. To correct this problem, browse for and reassign the assembly.

Here is an example of a small application using the **Dantec.BSA.Driver** assembly written in C# with Visual Studio .NET.

