BSA3 Processor Driver

Reference Manual

9040U2997

PUBLICATION NO.: 9040U2997

Table of Content

Table of Content i

1 Introduction vii

1.1 Preface vii

1.2 Purpose vii

1.3 Reading this Manual viii

1.4 Laser Safety 9

1.4.1 Danger of Laser Light 9

1.4.2 Precautions 9

1.4.3 Laser Safety Poster 10

1.5 Software License Agreement 10

1.6 Using the BSA Processor Driver 13

1.6.1 Installation 13

1.6.2 How to Install 14

1.6.3 BSA Flow Software Script Generator 15

1.6.4 Considerations Using 64-bit Windows 16

1.6.5 Migrating Code to BSA3 Processor Driver 17

1.6.6 Quick Start 20

1.6.7 Quick Examples 23

2 BSA Control Function Reference 35

2.1 Connecting 35

2.2 Configuring 36

2.3 Setting-up 36

2.4 Acquiring 40

2.5 Connecting 40

2.5.1 IBSA::Connect 40

2.5.2 IBSA::ConnectEx 42

2.5.3 IBSA::Disconnect 42

2.5.4 IBSA::GetConnectState 43

2.6 Configuring 43

2.6.1 IBSA::SetProcessor 43

2.6.2 IBSA::SetConfiguration 44

2.6.3 IBSA::SetProperties 47

2.6.4 IBSA::ActivateSysMon 51

2.6.5 IBSA::DeactivateSysMon 52

2.6.6 IBSA::SetLog 52

2.7 Acquisition Settings 53

2.7.1 IBSA::SetHighVoltageActivation 53

2.7.2 IBSA::GetHighVoltageActivation 54

2.7.3 IBSA::SetAnodeCurrentWarningLevel 54

2.7.4 IBSA::GetAnodeCurrentWarningLevel 54
2.7.5 IBSA::SetDataCollectionMode 55
2.7.6 IBSA::GetDataCollectionMode 56
2.7.7 IBSA::SetDutyCycle 56
2.7.8 IBSA::GetDutyCycle 56
2.7.9 IBSA::SetDeadTime 57
2.7.10 IBSA::GetDeadTime 57
2.8 Frequency Shift Settings 58
2.8.1 IBSA::SetBraggCellOutput 58
2.8.2 IBSA::GetBraggCellOutput 58
2.8.3 IBSA::SetBraggCellOutputFreq 59
2.8.4 BraggCellOutputFreq 59
2.9 Synchronization Input Signals 60
2.9.1 Dependencies 60
2.9.2 Programmable Inputs 60
2.9.3 IBSA::SetSync1 61
2.9.4 IBSA::GetSync1 62
2.9.5 IBSA::SetSync1Edge 62
2.9.6 IBSA::GetSync1Edge 63
2.9.7 IBSA::SetSync2 63
2.9.8 IBSA::GetSync2 64
2.9.9 IBSA::SetSync2Edge 64
2.9.10 IBSA::GetSync2Edge 64
2.9.11 IBSA::SetEncoderReset 65
2.9.12 IBSA::GetEncoderReset 65
2.9.13 IBSA::SetEncoderResetEdge 65
2.9.14 IBSA::GetEncoderResetEdge 66
2.9.15 IBSA::SetEncoder 66
2.9.16 IBSA::GetEncoder 66
2.9.17 IBSA::SetEncoderEdge 67
2.9.18 IBSA::GetEncoderEdge 67
2.9.19 IBSA::SetStartMeasurement 68
2.9.20 IBSA::GetStartMeasurement 68
2.9.21 IBSA::SetStartMeasurementEdge 68
2.9.22 IBSA::GetStartMeasurementEdge 68
2.9.23 IBSA::SetStopMeasurement 69
2.9.24 IBSA::GetStopMeasurement 69
2.9.25 IBSA::SetStopMeasurementEdge 69
2.9.26 IBSA::GetStopMeasurementEdge 70
2.9.27 IBSA::SetBurstDetectorEnable 70
2.9.28 IBSA::GetBurstDetectorEnable 71
2.9.29 IBSA::SetBurstDetectorEnableLevel 71

2.9.30 IBSA::GetBurstDetectorEnableLevel 71
2.9.31 IBSA::SetRefClk 72
2.9.32 IBSA::GetRefClk 72
2.10 Synchronization Output Signals 73
2.10.1 Dependencies 73
2.10.2 Programmable Outputs 73
2.10.3 IBSA::SetOutput 73
2.10.4 IBSA::GetOutput 74
2.11 Group Settings 75
2.11.1 IBSA::SetMaxSamples 75
2.11.2 IBSA::GetMaxSamples 75
2.11.3 IBSA::SetMaxAcqTime 76
2.11.4 IBSA::GetMaxAcqTime 76
2.11.5 IBSA::SetFilterMethod 77
2.11.6 IBSA::GetFilterMethod 77
2.11.7 IBSA::SetBurstWindow 78
2.11.8 IBSA::GetBurstWindow 78
2.11.9 IBSA::SetScopeDisplay 79
2.11.10 IBSA::GetScopeDisplay 79
2.11.11 IBSA::SetScopeTrigger 80
2.11.12 IBSA::GetScopeTrigger 80
2.11.13 IBSA::SetScopeZoom 81
2.11.14 IBSA::GetScopeZoom 81
2.11.15 IBSA::SetScopeYScale 82
2.11.16 IBSA::GetScopeYScale 82
2.11.17 IBSA::SetScopeAveraging 82
2.11.18 IBSA::GetScopeAveraging 83
2.12 LDA Settings 83
2.12.1 IBSA::SetCenterFreq 84
2.12.2 IBSA::GetCenterFreq 84
2.12.3 IBSA::SetBandwidth 85
2.12.4 IBSA::GetBandwidth 86
2.12.5 IBSA::SetRecordLengthMode 86
2.12.6 IBSA::GetRecordLengthMode 87
2.12.7 IBSA::SetRecordLength 88
2.12.8 IBSA::GetRecordLength 88
2.12.9 IBSA::SetMaxRecordLength 89
2.12.10 IBSA::GetMaxRecordLength 89
2.12.11 IBSA::SetHighVoltage 90
2.12.12 IBSA::GetHighVoltage 90
2.12.13 IBSA::SetSignalGain 91
2.12.14 IBSA::GetSignalGain 91

2.12.15 IBSA::SetBurstDetectorSNR 91
2.12.16 IBSA::GetBurstDetectorSNR 92
2.12.17 IBSA::SetLevelValidationRatio 92
2.12.18 IBSA::GetLevelValidationRatio 93
2.13 PDA Settings 94
2.13.1 IBSA::SetPDACalibrationMode 94
2.13.2 IBSA::GetPDACalibrationMode 94
2.13.3 IBSA::SetPDAHighVoltage 95
2.13.4 IBSA::GetPDAHighVoltage 95
2.13.5 IBSA::SetPDAAutoBalancing 95
2.13.6 IBSA::GetPDAAutoBalancing 96
2.13.7 IBSA::CalibratePDA 97
2.14 Optics Settings 97
2.14.1 IBSA::SetOpticsShift 98
2.14.2 IBSA::GetOpticsShift 99
2.14.3 IBSA::SetOpticsConversionFactor 99
2.14.4 IBSA::GetOpticsConversionFactor 100
2.14.5 IBSA::SetOpticsPhaseFactor 101
2.14.6 IBSA::GetOpticsPhaseFactor 102
2.14.7 IBSA::SetOpticsValidationBand 102
2.14.8 IBSA::GetOpticsValidationBand 103
2.14.9 IBSA::SetOpticsFringeDirection 103
2.14.10 IBSA::GetOpticsFringeDirection 104
2.15 Advanced Settings 104
2.15.1 IBSA::SetAnodeCurrentLimit 104
2.15.2 IBSA::GetAnodeCurrentLimit 104
2.15.3 IBSA::SetTransientData 105
2.15.4 IBSA::GetTransientData 105
2.16 Acquiring 106
2.16.1 Memory Usage 108
2.16.2 IBSA::StartAcq 109
2.16.3 IBSA::StopAcq 109
2.16.4 IBSA::GetAcqState 110
2.16.5 IBSA::SetBlockSize 110
2.16.6 IBSA::PreAllocateData 111
2.16.7 IBSA::ReadDataLength 112
2.16.8 IBSA::ReadArrivalTimeData 113
2.16.9 IBSA::ReadTransitTimeData 113
2.16.10 IBSA::ReadVelocityData 114
2.16.11 IBSA::ReadDiameterData 114
2.16.12 IBSA::ReadEncoderData 115
2.16.13 IBSA::ReadSyncDataLength 115

2.16.14 IBSA::ReadSyncArrivalTimeData 116
2.16.15 IBSA::ReadSyncData 116
2.16.16 IBSA::ReadStatValidation 117
2.16.17 IBSA::ReadStatSphericalValidation 118
2.17 Event Handling 118
2.17.1 BSA.Ctrl Component Events 119
2.17.2 Receiving Events 119
2.17.3 IBSAEvents::onStatus 120
2.17.4 IBSAEvents::onState 121
2.17.5 IBSAEvents::onAcqDone 122
2.17.6 IBSAEvents::onCalDone 123
2.17.7 IBSAEvents::onData 124
2.18 Error Handling 126
2.18.1 Method Errors 126
2.18.2 Processor Errors 128
2.19 BSA Control Constants 128
3 BSA System Monitor Control Function Reference 133
3.1 Connecting 133
3.2 Advanced 133
3.3 Using the BSA System Monitor Control 133
3.3.1 Resizing the BSA System Monitor Control 134
3.4 Connecting 134
3.4.1 IBSASysMon::Connect 135
3.4.2 IBSASysMon::ConnectEx 135
3.4.3 IBSASysMon::Disconnect 136
3.4.4 IBSASysMon::GetConnectState 136
3.5 Advanced Settings 136
3.5.1 IBSASysMon::ReadData 136
3.5.2 IBSASysMon::SetStyles 138
3.6 BSA System Monitor Control Constants 141
4 Examples 143
4.1 How to Get Started with the Examples 143
4.2 VBScript Example 143
4.3 JScript Example 144
4.4 MS Excel VBA Example 145
4.5 HTML Example 147
4.6 LabVIEW Example 151
4.7 Visual Basic Example 156
4.8 MATLAB Example 160
4.9 C# .NET Example 162
4.10 C++ MFC Example 163

1 Introduction

1.1 Preface

The BSA3 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 BSA3 Processor by implementing the BSA3 Processor Driver in a script- or programming languages that can integrate COM objects. A part of the list is shown below:

LabVIEW/MATLAB
C/C++/C#
Java/JScript
Visual Basic/VBScript/VBA

The BSA3 Processor Driver supports a dual interface. That is you can communicate with BSA3 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 BSA3 Processor Driver supports dual interface the user can choose him self what to use.

1.2 Purpose

So why use the BSA3 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 BSA3 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 BSA3 Processor Driver in his application without thinking on the code, which lies behind the communication.

This BSA3 Processor Driver can be used in many situations, just to mention a few:

Medium- to large-scale wind tunnels where the BSA3 Processor is only one probe amongst others all controlled by a single computer running e.g. LabVIEW.

Processes control milieus where the BSA3 Processor it to perform the same tasks repeatedly using simple batch runs.

Diagnostic systems where the BSA3 Processor supervises process environments, looking for changes and act in response to these.

Institutions who are interested in developing special dedicated applications using our BSA3 Processor.

1.3 Reading this Manual

This manual primarily acts as a reference manual for the BSA3 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 BSA3 Processor Driver and the BSA3 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 BSA3 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.

1.4 Laser Safety

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

Appropriate laser safety measures must be implemented when aligning and using lasers and illumination systems. You are therefore urged to follow the precautions below, which are general safety precautions to be observed by anyone working with illumination systems to be used with a laser. Again, before starting, it is recommended that you read the laser safety notices in all documents provided by the laser manufacturer and illumination system supplier and follow these as well as your local safety procedures.

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

AVOID EXPOSURE - LASER RADIATION
IS EMITTED FROM THIS APERTURE

1.4.1 Danger of 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, jewellery or other blank objects during alignment and use of the laser. Avoid reflections into eyes. Avoid accidental exposure to specular 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 is 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.

1.4.2 Precautions

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

1.4.3 Laser Safety Poster

Safety first with lasers
When working with lasers, always ensure a safe working environment for visitors, colleagues and yourself! - think before you act!
SWITCH OFF
DANTEC DYNAMICS

Displays the essential precautions for ensuring a safe laboratory environment when using lasers in your experiments. (Wall poster 70 x 100 cm). Available from http://www.d-antecdynamics.com, or through Dantec Dynamics' sales offices and representatives.

1.5 Software License Agreement

This software end user license agreement ("License Agreement") is concluded between you (either an individual or a corporate entity) and Dantec Dynamics A/S ("Dantec Dynamics"). Please read all terms and conditions of this License Agreement before installing the Software. When you install the Software, you agree to be bound by the terms of this License Agreement. If you cannot agree to the terms of this License Agreement you may not install or use the software in any manner.

Grant of License

This License Agreement together with the Software package including eventual media, user's guide and documentation and the invoice constitutes your proof of license and the right to exercise the rights herein and must be retained by you.

One license permits you to use one installation at a time of the Dantec Dynamics software product supplied to you (the "Software") including documentation in written or electronic form and solely for your own internal business purposes.

You may not rent, lease, sublicense or otherwise distribute, assign, transfer or make the Software available to any third party without the express consent of Dantec Dynamics except to the extent specifically permitted by mandatory applicable law.

Updates

Updates, new releases, bug fixes, etc. of the Software which are supplied to you (if any), may be used only in conjunction with versions of the Software legally acquired and licensed by you that you have already installed, unless such update etc. replaces that former version in its entirety and such former version is destroyed.

Copyright

The Software (including text, illustrations and images incorporated into the Software) and all proprietary rights therein are owned by Dantec Dynamics or Dantec Dynamics' 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 mandatory applicable law. You are not entitled to copy the Software or any part thereof except as otherwise expressly set out above. However you may make a copy of the Software solely for backup or archival purposes. You may not copy the user's guide accompanying the Software, nor distribute copies of any user documentation provided in “on-line” or electronic form, without Dantec Dynamics' prior written permission.

License and Maintenance Fees

You must pay any and all licensee and maintenance fees in accordance with the then-current payment terms established by Dantec Dynamics.

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 Dynamics will deliver a new copy of the Software if the medium on which the Software was supplied is not legible.

A defect in the Software shall be regarded as material only if it has a material effect on the proper functioning of the Software as a whole, or if it prevents operation of the Software in its entirety. If until 90 days after the delivery of the Software, it is established that there is a material defect in the Software, Dantec Dynamics shall, at Dantec Dynamics' 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 Dynamics shall be entitled to remedy any defect by indicating procedures, methods or uses ("work-arounds") which result in the defect not having a significant effect on the use of the Software.

Software is inherently complex and the possibility remains that the Software contains bugs, defects and inexpediencies which are not covered by the warranty set out immediately above. Such bugs, defects and inexpediencies etc. shall not constitute due ground for termination and shall not entitle you to any remedial action including refund of fees or payment of damages or costs. Dantec Dynamics will endeavour to correct bugs, defects 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 bugs or defects, or that bugs or defects can or will be remedied.

Indemnification

Dantec Dynamics will indemnify you against any claim by an unaffiliated third party that the Software infringes such unaffiliated third party's intellectual property rights and shall pay to you the amount awarded to such unaffiliated third party in a final judgment (or settlement to which Dantec Dynamics has consented) always subject however to the limitations and exclusions set out in this paragraph.

You must notify Dantec Dynamics promptly in writing of any such claim and allow Dantec Dynamics to take sole control over its defense. You must provide Dantec Dynamics with all reasonable assistance in defending the claim.

Dantec Dynamics' obligation to indemnify you shall not apply to the extent that any claim comprised by this paragraph is based in whole or in part on (i) any materials provided directly or indirectly by you; (ii) your exploitation of the Software for other purposes than those expressly contemplated in this License Agreement; and/or (iii) combining of the Software with third party products. You shall reimburse Dantec Dynamics for any costs or damages that result from such actions.

If Dantec Dynamics receives information of an alleged infringement of third party intellectual property rights or a final adverse judgment is passed by a competent court or a final settlement consented to by Dantec Dynamics is reached regarding an intellectual property right infringement claim related to the Software, Dantec Dynamics may (but shall not under any obligation to do so), either (i) procure for you the right to continue to use the Software as contemplated in this License Agreement; or (ii) modify the Software to make the Software non infringing; (iii) replace the relevant portion of the Software with a non infringing functional equivalent; or (iv) terminate with immediate effect your right to install and use the Software against a refund of the license fees paid by you prior to termination. You acknowledge and agree that Dantec Dynamics is entitled to exercise either of the aforesaid options in Dantec Dynamics' sole discretion and that this constitutes your sole and exclusive remedy in the event of any infringement of third party intellectual property rights.

Limitation of Liability

Neither Dantec Dynamics nor its distributors shall be liable for any indirect damages including without limitation loss of profits and loss of data or restoration hereof, or any other incidental, special or other consequential damages, and even if Dantec Dynamics has been informed of their possibility. Further, Dantec Dynamics disclaims and excludes any and all liability based on Dantec Dynamics' simple or gross negligent acts or omissions. In addition to any other limitations and exclusions of liability, Dantec Dynamics' total aggregate liability to pay any damages and costs to you shall in all events be limited to a total aggregated amount equal to the license fee paid by you for the Software.

Product Liability

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

Assignment

Neither party shall be entitled to assign this License Agreement or any of its rights or obligations pursuant this Agreement to any third party without the prior written consent of the other party. Notwithstanding the aforesaid, Dantec Dynamics shall be entitled to assign this License Agreement in whole or in part without your consent to (i) a company affiliated with Dantec Dynamics or (ii) an unaffiliated third party to the extent that such assignment takes place in connection with a restructuring, divestiture, merger, acquisition or the like.

Term and Termination

Subject to and conditional upon your compliance with the terms and conditions of this License Agreement may install and use the Software as contemplated herein.

We may terminate this License Agreement for breach at any time with immediate effect by serving notice in writing to you, if you commit any material breach of any terms and conditions set out in this License Agreement. Without limiting the generality of the aforesaid, any failure to pay fees and amounts due to Dantec Dynamics and/or any infringement of Dantec Dynamics' intellectual property rights shall be regarded a material breach that entitles Dantec Dynamics to terminate this License Agreement for breach.

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 that of the venue of Dantec Dynamics. Notwithstanding the aforesaid, Dantec Dynamics shall forthwith be entitled to file any action to enforce Dantec Dynamics' rights including intellectual property rights, in any applicable jurisdiction using any applicable legal remedies.

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 Dynamics distributor serving your country. You can find a list of current Dantec Dynamics distributors on our web site: www.dantecdynamics.com.

Dantec Dynamics A/S

Tonsbakken 16-18 - DK-2740 Skovlunde, Denmark

Tel: +45 4457 8000 - Fax: + 45 4457 8001

www.dantecdynamics.com

1.6 Using the BSA Processor Driver

1.6.1 Installation

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

Requirements

- BSA Processor
  - or -
  BSA3 Processor
- PC with a modern multi-core processor.
- Installation must be performed by an account with admin privileges.
- Microsoft® Windows© 10 x86/x64 with latest updates.
  - or -
  Microsoft® Windows© 8/8.1 x86/x64 with latest updates.
  - or -
  Microsoft® Windows© 7 x86/x64 with latest updates.

- or -
Microsoft® Windows© Vista x86/x64 with latest updates.

- Microsoft® Windows© Installer v3.0 or later.
- Microsoft® Internet Explorer 6 or later with latest security updates.
- 512 GB of RAM minimum.
- 100 GB of available hard-disk space minimum; 10 GB for running acquisitions.
- 100 Base-T Ethernet or faster adapter with RJ-45 connector.
- DVD/CD-ROM drive.
- Super VGA (800x600) or higher-resolution monitor with 65.000 colors or more.
- Mouse or compatible pointing device.

Versions

BSA Processor Driver v6.20 is designed for BSA Processor Software v5.03 and BSA3 Processor Firmware 403.

BSA Processor Driver v5.10 is designed for BSA Processor Software v5.03.

BSA Processor Driver v5.00 is designed for BSA Processor Software v5.00.

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.

1.6.2 How to Install

Place the DVD 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 driver control BSA.Ctrl/BSA3.Ctrl and the system monitor control BSA.SysMon.Ctrl/BSA3.SysMon.Ctrl.

ProgID	ClassID
BSA3.Ctrl	{83E16B05-CDAD-11D0-89E3-0000C0C27240}
BSA3.SysMon.Ctrl	{83E16B06-CDAD-11D0-89E3-0000C0C27240}

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

Namespace
Dantec.BSA3.Driver

The BSA3.Ctrl is used to communicate with the BSA3 Processor. Through this component the BSA3 Processor can be configured, the properties can be changed and data can be acquired. The BSA3.SysMon.Ctrl is a visual control that must be embedded into a COM container or another window. The component displays online information from the BSA3 Processor.

When the 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 BSA Processor Driver. For more examples please look in the "Examples" (on page 143) section in this manual.

1.6.3 BSA Flow Software Script Generator

This section and the "BSA Flow Software Script Generator" above and "BSA Flow Software Script Generator" above sections give a quick introduction to how to use the driver.

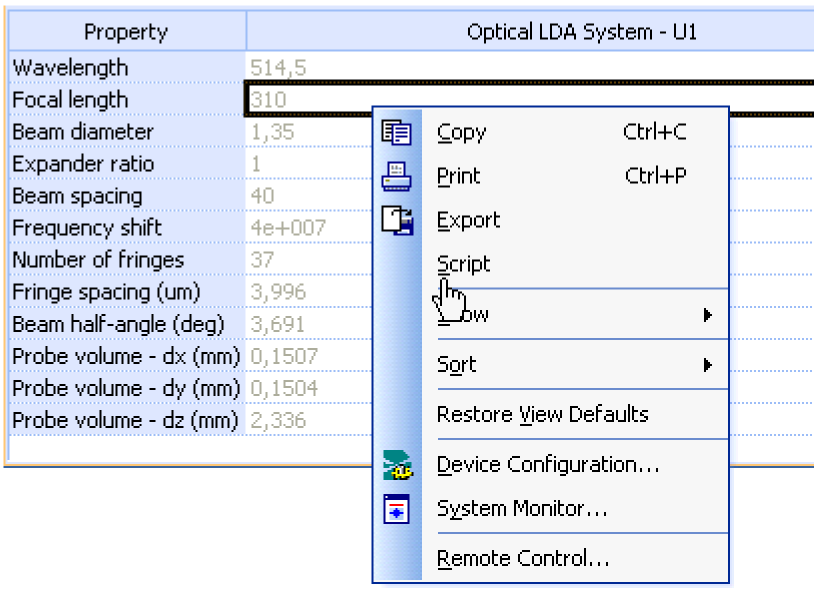
Normally, when starting to use the driver, the best approach is to start getting to know the BSA Processor Driver is through the application software BSA Flow Software. Again pointing out that it is not necessary to use BSA Flow Software when running the driver.

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.

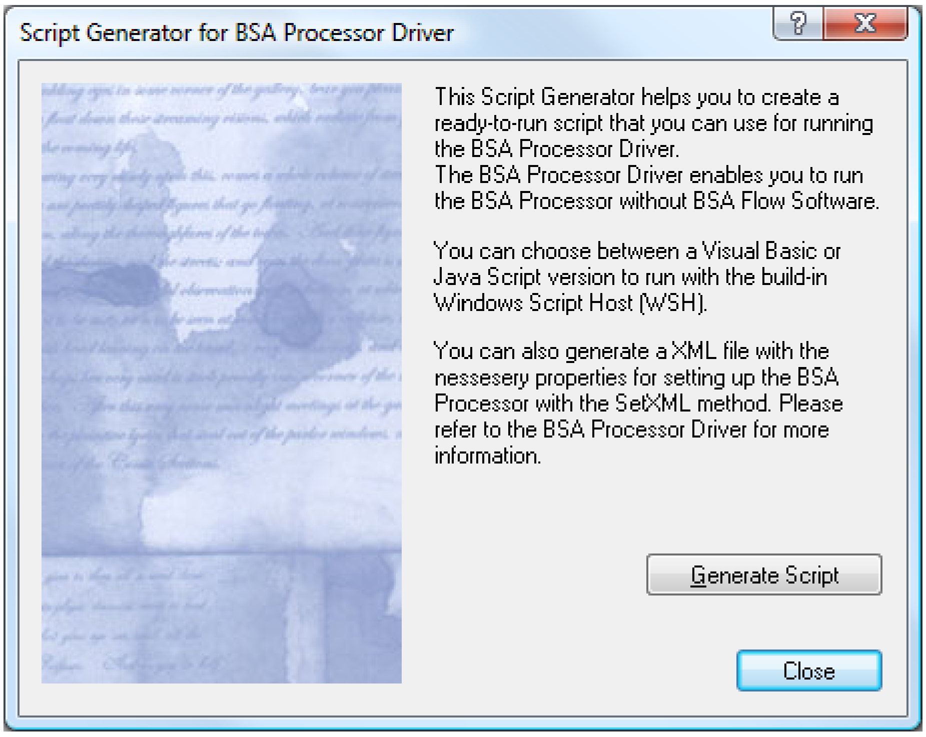
To access the script generator select the Script option in the output properties of the BSA F/P Application object.

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



Property	Optical LDA System - U1
Wavelength	514,5
Focal length	310
Beam diameter	1,35
Expander ratio	1
Beam spacing	40
Frequency shift	4e+007
Number of fringes	37
Fringe spacing (um)	3,996
Beam half-angle (deg)	3,691
Probe volume - dx (mm)	0,1507
Probe volume - dy (mm)	0,1504
Probe volume - dz (mm)	2,336

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


Script Generator for BSA Processor Driver

This Script Generator helps you to create a ready-to-run script that you can use for running the BSA Processor Driver.
The BSA Processor Driver enables you to run the BSA Processor without BSA Flow Software.

You can choose between a Visual Basic or Java Script version to run with the build-in Windows Script Host (WSH).

You can also generate a XML file with the nessesery properties for setting up the BSA Processor with the SetXML method. Please refer to the BSA Processor Driver for more information.

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 ca quickly change between configurations and measurements.

The Script Generator automatically creates a Batch (.bat) file along with the script file, for executing the VBScript or JScript. This Batch file calls the correct 32-bit version of the Windows Script Host (WSH) on both a 32- and 64-bit operating system (OS).

1.6.4 Considerations Using 64-bit Windows

BSA Processor Driver is made as both a 32-bit (x86) and a 64-bit (x64) native Windows ActiveX. This means, that when installed on a 64-bit (x64) Windows operating system it will run as a 64-bit process, and on a 32-bit Windows operating system it will run as a 32-bit process. During normal use this will not cause any problems, but special considerations needs to be taken into account.

It is possible to install both the 32-bit and the 64-bit version of the BSA Processor Driver at the same time if required.

If the calling application, the calling script or the environment calling the BSA Processor Driver is 32-bit, it must use the 32-bit version of the driver, even the Windows operating system is 64-bit. In other words you can only call the 64-bit version of the BSA Processor Driver from a 64-bit application, script or environment running on 64-bit Windows operating system.

When calling the driver from e.g. LabVIEW or MATLAB it is necessary to check if they are running as 32-bit or 64-bit processes before knowing which version of the BSA Processor Driver to install.

Also be aware that some development environments (like Microsoft Visual Studio) runs as 32-bit even it compiles programs to 64-bit. In this situation it will be necessary to install both the 32-bit and the 64-bit version of the BSA Processor Driver.

Note
Currently we see warnings when uninstalling both versions of the driver at the same time, please ignore.

If used in a script it is important to know the correct version of the Windows Script Host (WSH) to use. A 32-bit version can be found in the Windows\SysWOW64 folder on a 64-bit Windows operating system.

1.6.5 Migrating Code to BSA3 Processor Driver

The interface for the BSA Processor Driver to support the BSA Processor and the BSA3 Processor is very similar. This means, that migrating from code supporting the BSA Processor to code supporting the BSA3 Processor or writing code that supports both processor models is very simple.

Since the support for the two processors is split into two different drivers, the initial call to create or instantiate the correct driver will be different. However the returned object can be treated the same and most methods and properties can be called using the same code.

Examples of difference in how to initially create or instantiate the driver.

[VBScript]

```
' creating interface to BSA Processor'
dim MyBSA: set MyBSA = createobject("BSA.Ctrl")

' --- or alternatively ---'

' creating interface to BSA3 Processor'
dim MyBSA: set MyBSA = createobject("BSA3.Ctrl")

' now MyBSA can be used to call any of the interface methods'
```

[Visual Basic]

```
' creating interface to BSA Processor'
Public MyBSA As New BSACtrl.BSA

' --- or alternatively ---'

' creating interface to BSA3 Processor'
Public MyBSA As New BSA3Ctrl.BSA

' now MyBSA can be used to call any of the interface methods'
```

[C++]

```

// importing the BSA.Ctrl
#import "C:\Program Files\Dantec Dynamics\BSA Processor Driver-
\BSA2ProcessorCtrl.dll" named_guids raw_interfaces_only

// --- or alternatively ---
// importing the BSA3.Ctrl
#import "C:\Program Files\Dantec Dynamics\BSA Processor Driver-
\BSA3ProcessorCtrl.dll" named_guids raw_interfaces_only

IBSA* m_pMyBSA;
```

There are a few differences between the BSA Processor and the BSA3 Processor driver interface when calling methods and properties. The difference in the interface is mainly related to Bragg cell outputs and synchronization outputs.

BSA Processor code	**Corresponding BSA3 Processor code**
The BSA Processor uses the BSA.Ctrl component. `MyBSA = CreateObject("BSA.Ctrl")`	The BSA3 Processor uses the BSA3.Ctrl component. `MyBSA = CreateObject("BSA3.Ctrl")`
The BSA Processor uses the BSA2ProcessorDriver.dll. `#import "C:\Program Files\Dantec Dynamics\BSA Processor Driver\BSA2ProcessorCtrl.dll"`	The BSA3 Processor uses the BSA3ProcessorDriver.dll. `#import "C:\Program Files\Dantec Dynamics\BSA Processor Driver\BSA3ProcessorCtrl.dll"`

The BSA Processor uses a specific 40 MHz Bragg cell output, which can be enabled or disabled.
`MyBSA.Set40MHzFreqShiftOutput(1)`
	The BSA3 Processor have 3 variable Bragg cell outputs. Setting the first to 40 MHz, where the first can be set to 40 MHz.
	`MyBSA.SetBraggCellOutput(1, 1)`
	`MyBSA.SetBraggCellOutputFreq(1, 40000000)`

`MyBSA.Set40MHzFreqShiftOutput(0)`	`MyBSA.SetBraggCellOutput(1, 0)`

The BSA Processor have a second variable Bragg cell output, which can be enabled or disabled.
`MyBSA.SetVarShiftOutput(1)`
`MyBSA.SetVarShiftFreq(80000000)`
	The BSA3 Processor have 3 variable Bragg cell outputs. Setting the first to 40 MHz, where the second can be set to 80 MHz.
	`MyBSA.SetBraggCellOutput(2, 1)`
	`MyBSA.SetBraggCellOutputFreq(2, 80000000)`

`MyBSA.SetVarShiftOutput(0)`	`MyBSA.SetBraggCellOutput(2, 0)`
The BSA Processor have one BNC output which can be assigned a signal. `MyBSA.SetBNC(signal)`	The BSA3 Processor have 3 BNC outputs, where the first can be used to assign the signal. `MyBSA.SetOutput(1, signal)`
The BSA Processor have one Shutter output which can be assigned a signal. `MyBSA.SetShutter(signal)`	The BSA3 Processor have 3 BNC outputs, where the second can be used to assign the signal. `MyBSA.SetOutput(2, signal)`
The BSA Processor have three DSub outputs, which can be assigned a signal. `MyBSA.SetDSUB(1-3, signal)`	The BSA3 Processor does not have any DSub output connectors. However there is a third BNC output which can be used. `MyBSA.SetOutput(3, signal)`

Other methods requires that the input is changed.

BSA Processor code	Corresponding BSA3 Processor code
The BSA Processor allows for a freely assigned bandwidth. The processor will automatically find the closest available value in combination with the selected center frequency. `MyBSA.SetBandwidth(channel, bandwidth)`	The interface is the same for the BSA3 Processor, but it only allows for valid values of the bandwidth. `MyBSA.SetBandwidth(channel, bandwidth)`
The BSA Processor uses a floating point value for the time, which means that you can specify acquisition between 0 and 1 sec. `MyBSA.SetMaxAcqTime(group, time)`	The interface is the same for the BSA3 Processor, but the time is treated as an integer, meaning that the acquisition is in steps of 1 sec. `MyBSA.SetMaxAcqTime(group, time)`

1.6.6 Quick Start

In this section we provide a quick step-by-step approach to use the BSA Processor Driver.

Initial Steps

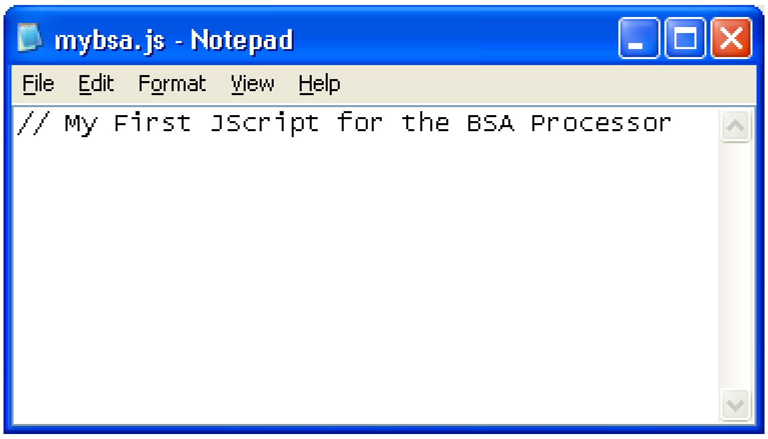
1. Ensure that you have a processor with a known IP-address connected to your network or in a peer-to-peer configuration with your controlling PC.
2. The flow signal to the processor should, for this test, be a stable flow with well-known characteristics, like frequency and variance.
3. For this quick 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 Processor Driver component on your PC, from the installation DVD or from the Dantec Dynamics Download section. On Windows x64 we recommend to install both the x64 and the x86 versions of the driver.
5. Identify your processor.

   BSA3 Processor uses the BSA3.Ctrl and the BSA3.SysMon.Ctrl controls.

Note
This quick example uses the BSA.Ctrl, but it can be replaced by the BSA3.Ctrl if a BSA3 Processor is used.

Writing a Script

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



```
// My First JScript for the BSA Processor
```

```
// My First JScript for the processor
```

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 processor. All the available methods can be seen in the "BSA Control Function Reference" (on page 35) section.

[JScript]

```
// My First JScript for the 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.

[JScript]

```
// My first JScript for the 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 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.

[JScript]

```
// My first JScript for the 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 processor for simplicity. Calling the `SetProcessor` and `SetConfiguration` methods both with the parameter 0 does this.

[JScript]

```
// My First JScript for the 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.

[JScript]

```
// My First JScript for the 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 Control Function Reference" (on page 35) 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.

[JScript]

```
// My First JScript for the 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.

[JScript]

```
// My First JScript for the 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 processor. You can extent the sample with all the functionality of the processor by adding methods from the "BSA Control Function Reference" (on page 35). 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 DVD, and for the latest samples and information on the Dantec Dynamics download section.

1.6.7 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.

Visual Basic Quick Example

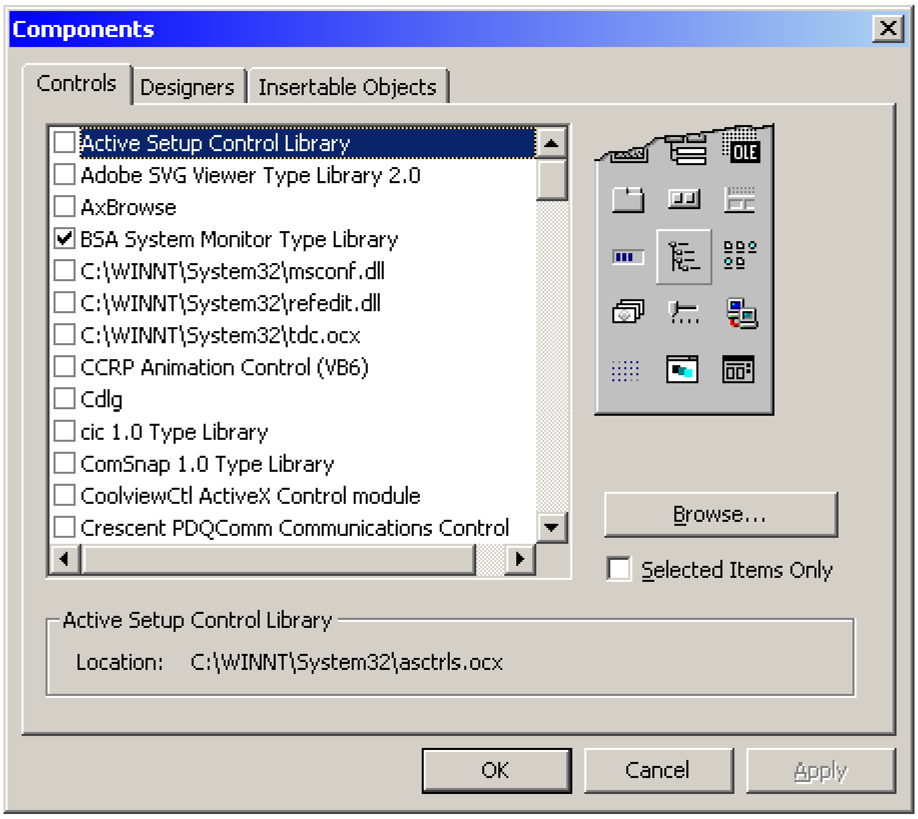
This simple Visual Basic 6 quick example shows how to select the BSA.SysMon.Ctrl in the Visual Basic environment.

Note

This quick example uses the BSA.Ctrl and the BSA.SysMon.Ctrl, but they can be replaced by the BSA3.Ctrl and the BSA3.SysMon.Ctrl if a BSA3 Processor is used.

Create a new blank project.

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.



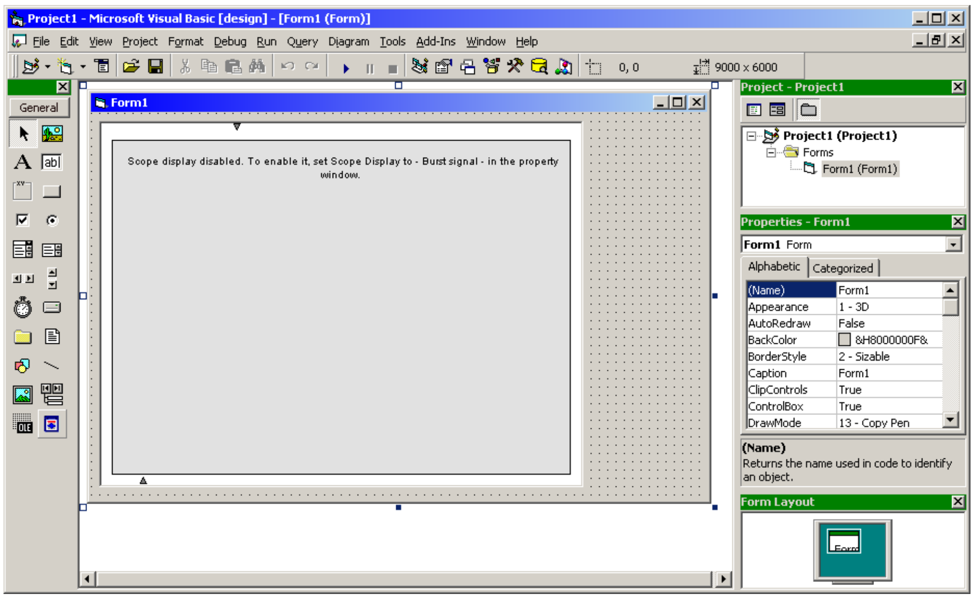
- [ ] Active Setup Control Library
- [ ] Adobe SVG Viewer Type Library 2.0
- [ ] AxBrowse
- [x] BSA System Monitor Type Library
- [ ] C:\WINNT\System32\msconf.dll
- [ ] C:\WINNT\System32\refedit.dll
- [ ] C:\WINNT\System32\tdc.ocx
- [ ] CCRP Animation Control (VB6)
- [ ] Cdlg
- [ ] cic 1.0 Type Library
- [ ] ComSnap 1.0 Type Library
- [ ] CoolviewCtl ActiveX Control module
- [ ] Crescent PDQComm Communications Control

- [ ] Selected Items Only

Active Setup Control Library

Location: C:\WINNT\System32\asctrls.ocx

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



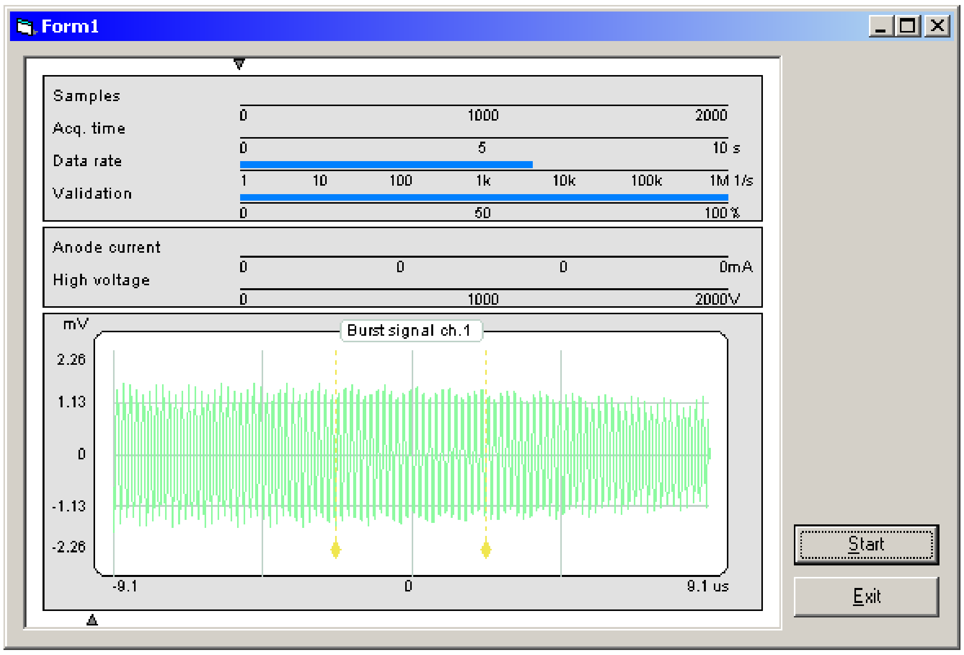
Scope display disabled. To enable it, set Scope Display to - Burst signal - in the property window.

(Name)	Form1
Appearance	1 - 3D
AutoRedraw	False
BackColor	▢ &H8000000F&
BorderStyle	2 - Sizable
Caption	Form1
ClipControls	True
ControlBox	True
DrawMode	13 - Copy Pen

(Name)
Returns the name used in code to identify an object.

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
```

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
```

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 processor.

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

[Visual Basic]

```
'------------------------------------------
' This code example assumes that the BSACtrl'
' component has been installed. '
' It will then initiate a data acquisition in'
' the 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.'
'---------------------------------------------'
```

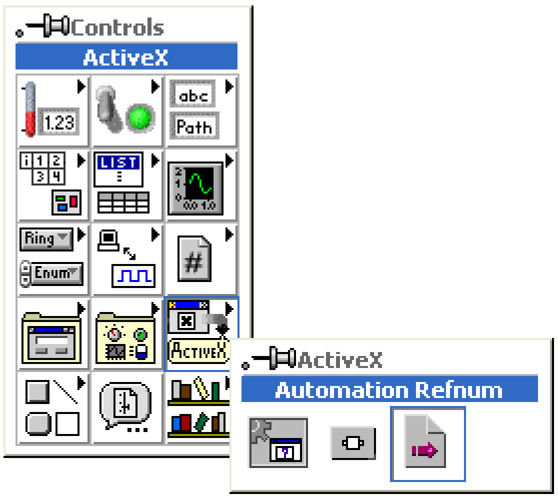
```
private sub cmdGetVelocityData_Click()
dim NumberOfSamples as long
dim SampleIdx as long
dim Velocity()

' 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
```

LabVIEW Quick Example

This quick example will provide you with the very basic steps to use the BSA3.Ctrl and BSA3.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.

From the front panel we will add a reference to the BSA3.Ctrl. Right click and select the *Controls - ActiveX - Automation Refnum* palette list.



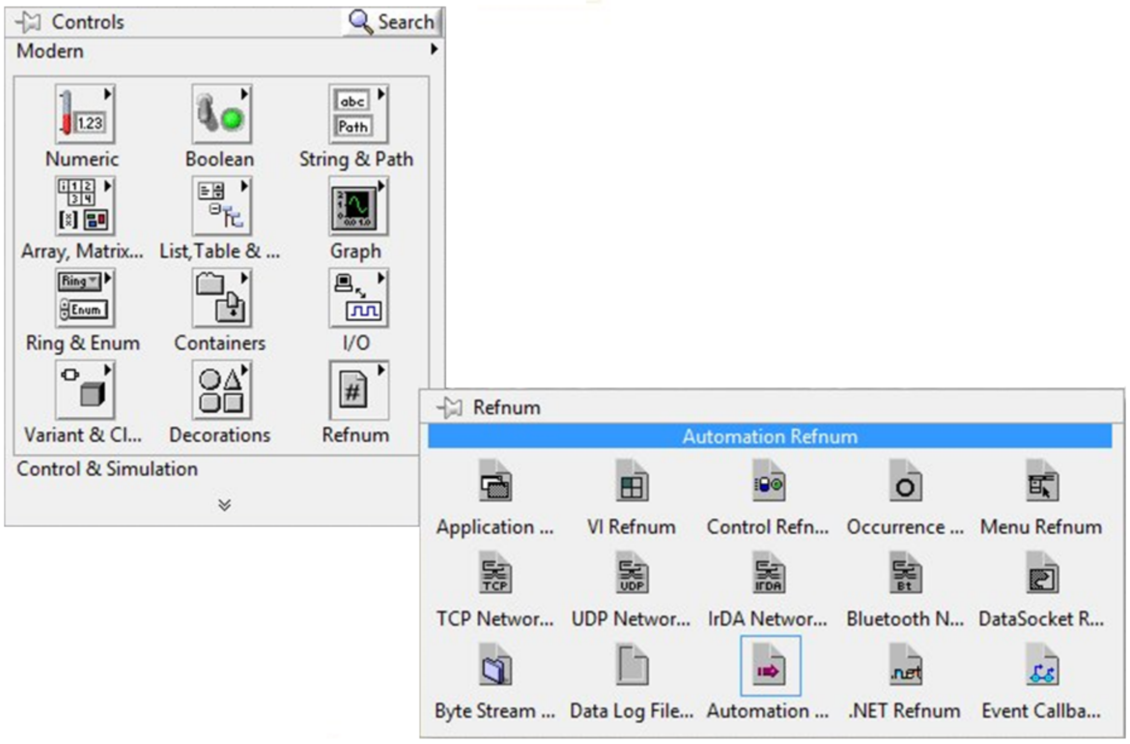
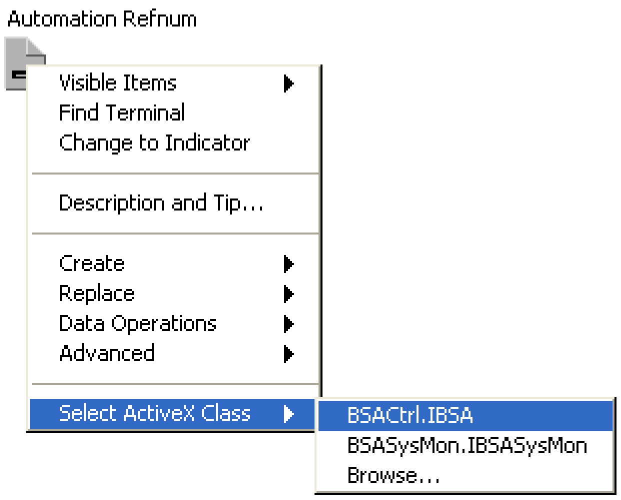


Figure 5-1: LabVIEW Controls Refnum Palette (older versions on top)

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.



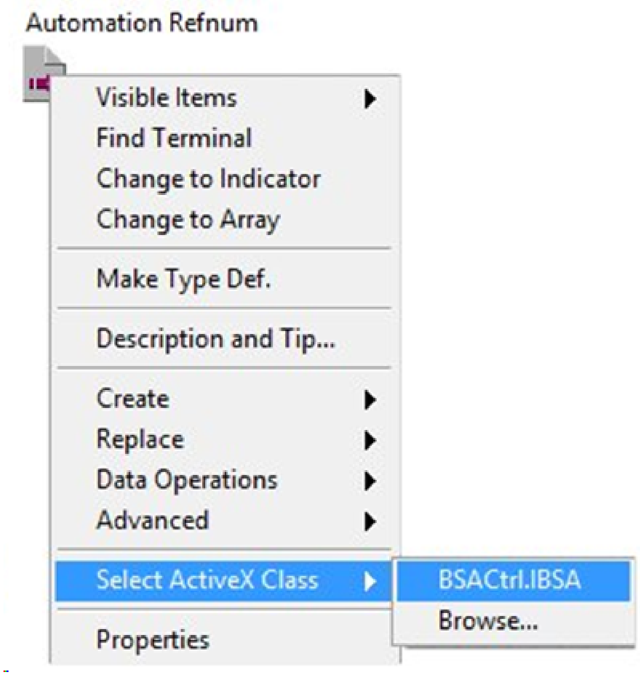
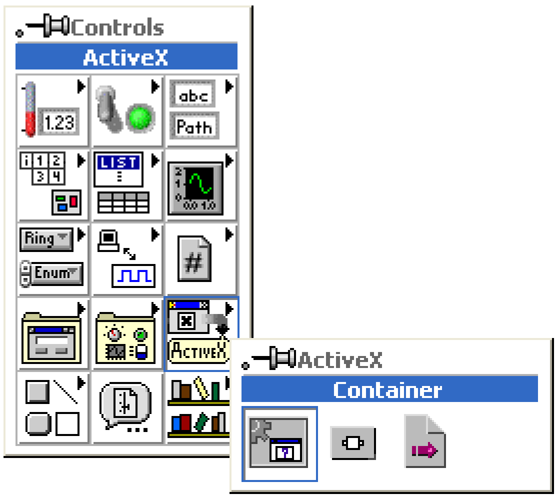


Figure 5-2: LabVIEW Automation Refnum Palette (older versions on top)

Now you have a reference to the BSA3.Ctrl.

From the front panel we will now add a reference to the BSA3.SysMon.Ctrl. Right click and select the Controls| ActiveX| Container palette list.



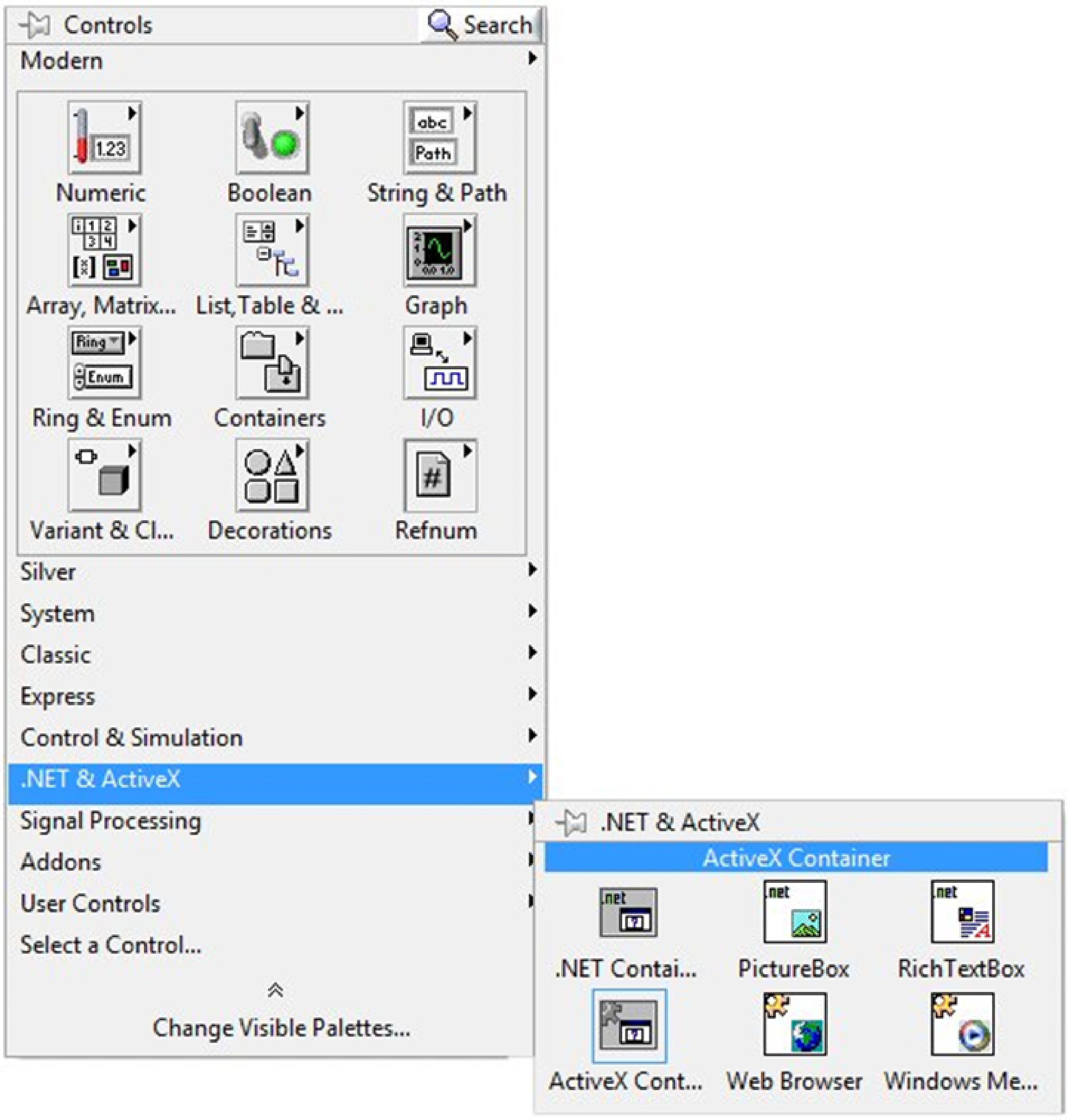
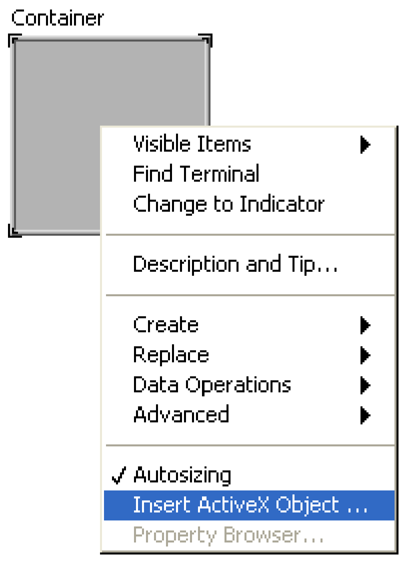


Figure 5-3: LabVIEW Controls Palette (older versions on top)

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



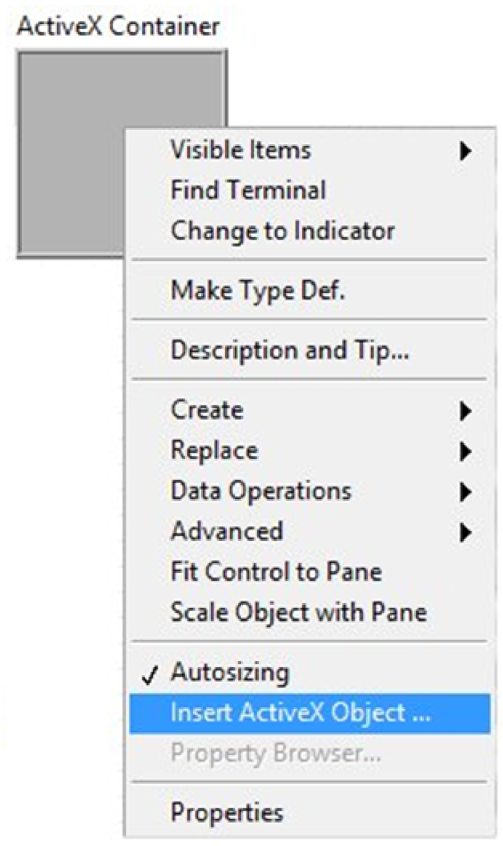


Figure 5-4: LabVIEW ActiveX Container Palette (older versions on top)

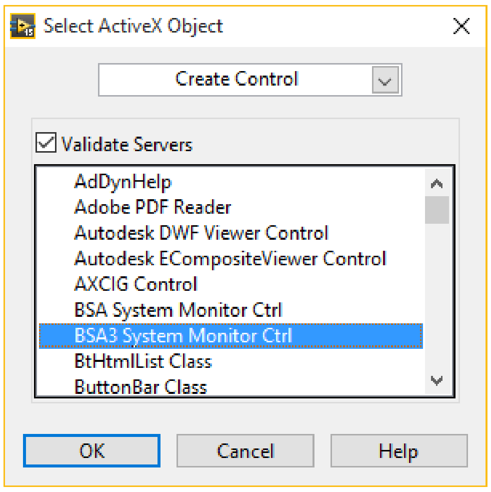


Figure 5-5: LabVIEW Select ActiveX Object Dialog

Now you have a reference to the BSA3.SysMon.Ctrl.

In the same way add a Stop button on the front panel that we can use later.

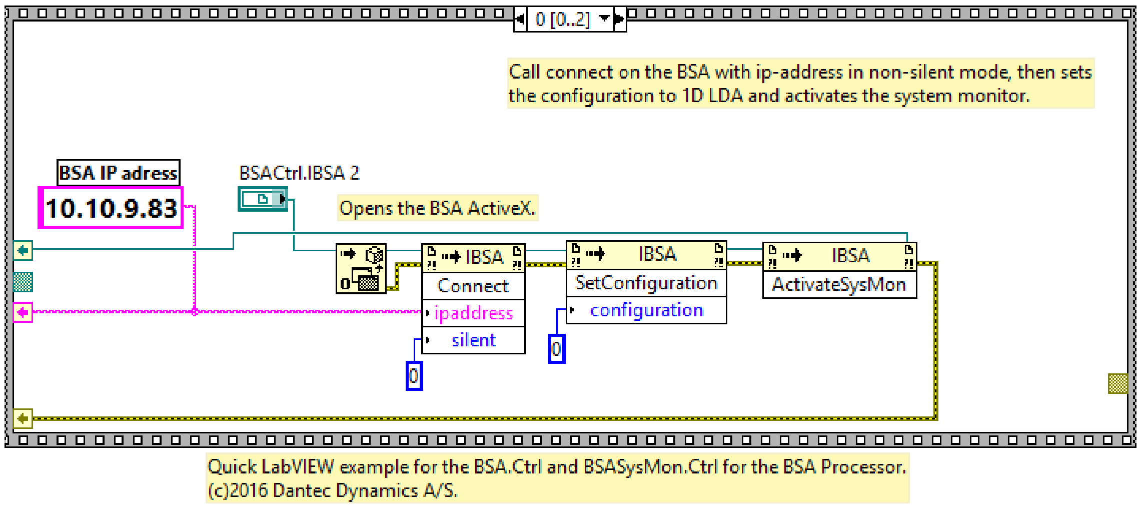
Now change to the diagram window.

Add three sequence-frames that we can use for scheduling the calls to the components.

In the first sequence add the reference to the BSA3.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 BSA3 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.

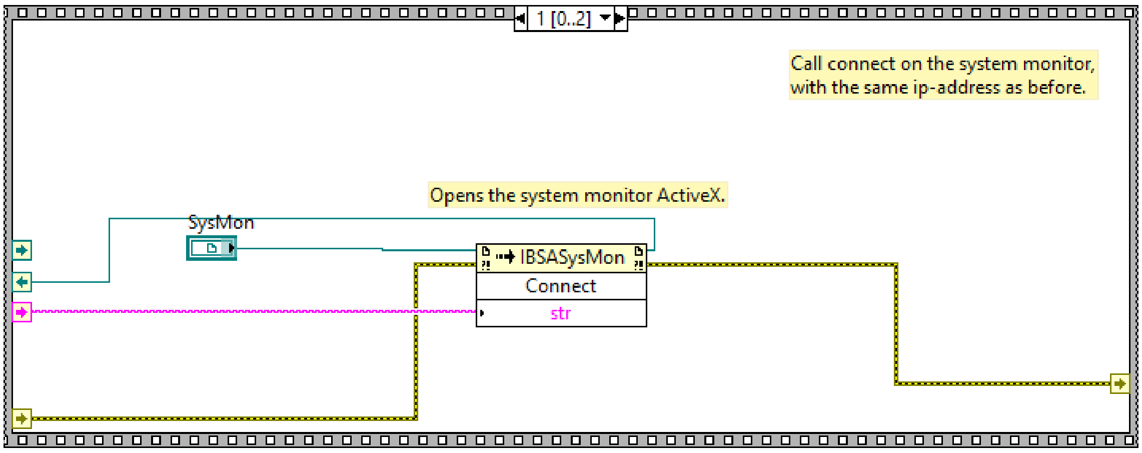
Wire up the different nodes like this:

[LabVIEW]


0 [0..2]
Call connect on the BSA with ip-address in non-silent mode, then sets the configuration to 1D LDA and activates the system monitor.
BSA IP adress
10.10.9.83
BSACtrl.IBSA 2
Opens the BSA ActiveX.
IBSA
Connect
ipaddress
silent
0
IBSA
SetConfiguration
configuration
0
IBSA
ActivateSysMon
Quick LabVIEW example for the BSA.Ctrl and BSASysMon.Ctrl for the BSA Processor.
(c)2016 Dantec Dynamics A/S.

In the second sequence add the reference to the BSA3.SysMon.Ctrl. Add one Invoke Node for the connect method, and specify the same IP-address as before.

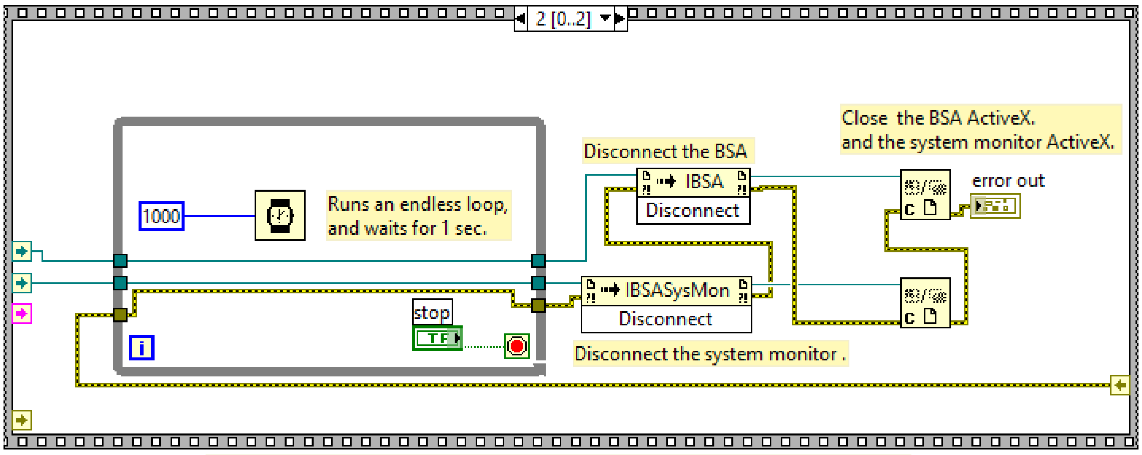
[LabVIEW]


1 [0..2]
Call connect on the system monitor, with the same ip-address as before.
Opens the system monitor ActiveX.
SysMon
IBSASysMon
Connect
str

Quick LabVIEW example for the BSA.Ctrl and BSASysMon.Ctrl for the BSA Processor.
(c)2016 Dantec Dynamics A/S.

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.

[LabVIEW]


2 [0..2]
1000
Runs an endless loop, and waits for 1 sec.
stop
Disconnect the BSA
IBSA
Disconnect
Close the BSA ActiveX. and the system monitor ActiveX.
error out
IBSASysMon
Disconnect
Disconnect the system monitor .

Quick LabVIEW example for the BSA.Ctrl and BSASysMon.Ctrl for the BSA Processor.
(c)2016 Dantec Dynamics A/S.

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.

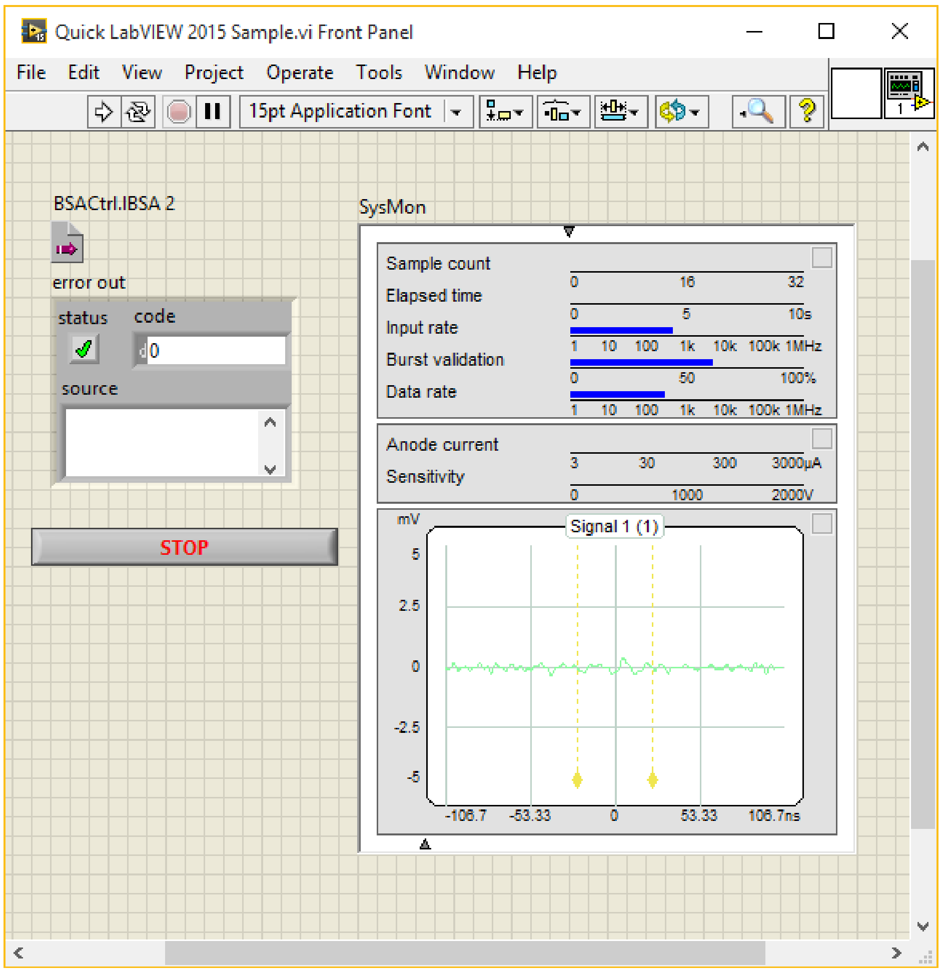


Figure 5-6: LabVIEW Quick Example Control Panel

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.

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>\BSA3ProcessorCtrl.dll" named_guids
using namespace BSA3Ctrl;
```

<path> is the full path to the component, the default location is C:\Program Files\Dantec Dynamics\BSA3 Processor Driver\BSA3ProcessorCtrl.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 BSA3 Processor you can add the following line to your code.

[c++]

```
pMyBSA->Connect("10.10.100.130", false);
```

For more examples see section "Examples" (on page 143) and on the installation DVD.

2 BSA Control Function Reference

The purpose of this chapter is to give an understanding of the functions in the BSA3 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 DVD 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 BSA3 Processor through a network IP-address. Connecting to the component can be done is 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 processor. Currently only LDA configurations are available.

All the 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 "IBSA::StartAcq" (on page 109) and "IBSA::StartAcq" (on page 109) 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 "IBSA::SetMaxAcqTime" (on page 76) and "IBSA::SetMaxSamples" (on page 75) methods.

2.1 Connecting

Connecting Methods	Description
Connect	Establishing a connection to the BSA3 Processor using a network IP-address.
ConnectEx	Establishing a connection to the BSA3 Processor using a network IP-address and a specific NIC address.
Disconnect	Closes the connection to the BSA3 Processor.
GetConnectState	Detects if the driver is connected to the BSA3 Processor or not.

2.2 Configuring

Configuring Methods	Description
SetProcessor	Specifies the BSA3 Processor model.
SetConfiguration	Specifies a LDA configuration in the BSA3 Processor.
SetProperties	Set multiple properties in the BSA3 Processor using one call to a configuration file.
ActivateSysMon	Starts the System Monitor.
DeactivateSysMon	Stops the System Monitor.
GetStatus	Retrieves the current BSA3 Processor status.
SetLog	Starts logging information to a file.

2.3 Setting-up

Acquisition Methods	Description
Set/GetHighVoltage Activation	Determines how the high voltage is handled during acquisition.
Set/GetAnodeCurrent WarningLevel	Specifies the warning level for the anode current during a measurement.
Set/GetDataCollectionMode	Specifies the method in which data is collected.
Set/GetDutyCycle	Sets the length of a continuous signal.
Set/GetDeadTime	Sets the minimum time-window between samples.

Frequency Shift Methods	Description
Set/GetBraggCellOutput	Enables the Bragg Cell frequency output, on the back of the BSA3 Processor.
Set/GetBraggCellOutputFreq	Sets the Bragg Cell frequency output.

Synchronization Input Methods	Description
Set/GetSync1	Assigns input for the sync. 1 signal.
Set/GetSync1Edge	Sets sync. 1 trigger edge direction.
Set/GetSync2	Assigns input for the sync. 2 signal.
Set/GetSync2Edge	Sets sync. 2 trigger edge direction.
Set/GetEncoderReset	Assigns input for the encoder reset signal.
Set/GetEncoderResetEdge	Sets encoder reset trigger edge direction.
Set/GetEncoder	Assigns input for the encoder clock signal.
Set/GetEncoderEdge	Sets encoder trigger edge direction.
Set/GetStartMeasurement	Assigns input for the start measurement signal.
Set/GetStartMeasurementEdge	Sets start measurement edge direction.
Set/GetStopMeasurement	Assigns input for the stop measurement signal.
Set/GetStopMeasurementEdge	Sets stop measurement edge direction.
Set/GetBurstDetectorEnable	Assigns trigger input for enabling the burst detector.
Set/GetBurstDetectorEnableLevel	Sets burst detector enable signal level.
Set/GetRefClk	Assigns input for external reference clock.

Synchronization Output Methods	Description
Set/GetOutput	Assigns a signal to one of the output connectors.

Group Methods	Description
Set/GetMaxSamples	Specifies the maximum number of samples to acquire for the group.
Set/GetMaxAcqTime	Specifies the maximum time for acquisition for the group.
Set/GetFilterMethod	Specifies the coincidence filtering method.
Set/GetBurstWindow	Sets the coincidence time window.
Set/GetScopeDisplay	Specifies the scope display type in the System Monitor.
Set/GetScopeTrigger	Sets the scope trigger method or channel in the System Monitor.
Set/GetScopeZoom	Sets the zoom factor of the x-axis scope display in the System Monitor.
Set/GetScopeYScale	Sets the scale factor of the y-axis scope display in the System Monitor.
Set/GetScopeAveraging	Sets the averaging factor of the spectrum display in the System Monitor.

LDA Methods	Description
Set/GetCenterFreq	Specifies the center frequency of the LDA chan-nel.
Set/GetBandwidth	Specifies the bandwidth of the LDA channel.
Set/GetRecordLengthMode	Sets the record length mode.
Set/GetRecordLength	Specifies the record length.
Set/GetMaxRecordLength	Specifies the maximum record length.
Set/GetHighVoltage	Sets the high voltage level of the LDA channel.
Set/GetSignalGain	Sets the gain for the LDA channel.
Set/GetBurstDetectorSNR	Defines the SNR threshold in the burst detector.
Set/GetLevelValidationRatio	Defines the level of peak validation for the chan-nel.

PDA Methods	Description
Set/GetPDACalibrationMode	Set the PDA phase calibration mode.
Set/GetPDAHighVoltage	Sets the high voltage level of the PDA channel.
Set/GetPDAAutoBalancing	Set automatic high voltage balancing for PDA channels.
CalibratePDA	Starts a PDA calibration.

Optics Methods	Description
Set/GetOpticsShift	Sets the frequency shift for a channel.
Set/GetOpticsConversionFactor	Sets the conversion factors for the channels.
Set/GetOpticsPhaseFactor	Sets the PDA phase factors.
Set/GetOpticsValidatioBand	Sets the PDA validation band.
Set/GetOpticsFringeDirection	Sets the fringe direction.

Advanced Methods	Description
Set/GetAnodeCurrentLimit	Sets the maximum allowed anode current.
Set/GetTransientData	Enables acquisition of transient data.

2.4 Acquiring

Acquiring Methods	Description
StartAcq	Starts an acquisition.
StopAcq	Aborts a running acquisition.
GetAcqState	Detects if an acquisition is running.
SetBlockSize	Sets the size of the data blocks received.
PreAllocateData	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.
ReadDiameterData	Reads PDA diameter data.
ReadEncoderData	Reads the encoder data.
ReadSyncDataLength	Gets the number of acquired sync. samples.
ReadSyncArrivalTimeData	Reads the sync. arrival time data.
ReadSyncData	Reads the sync. event data.
ReadStatValidation	Reads the statistical validation rate of the data.
ReadStatSphericalValidation	Reads the statistical spherical validation rate of the PDA diameter data.

2.5 Connecting

This section describes the methods that are involved in connecting to the BSA3 Processor.

The BSA3 Processor supports Ethernet network connection running the TCP/IPv4 (IPv4) protocol up to 1GBb/s, a wireless WiFi network connection up to IEEE 802.11n, and a USB connecting supporting the USBv2 standard.

During operation the BSA Ctrl. component must be connected to the BSA3 Processor, no other connections can be established at the same time, meaning that you cannot have simultaneous connection of BSA Flow Software and the BSA3 Processor Driver.

2.5.1 IBSA::Connect

Call this method for to connect to the BSA3 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 physical BSA3 Processor before trying to call any other function.

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

Sub Connect(
     IPAddress As String,
     Silent As Long)
```

Parameters

`IPAddress`

[in] The IP-address of the BSA3 Processor as it is seen on the network. The syntax consists of a common dotted octet IP-address, for example 10.10.100.130.

`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 Windows console application or from a script language set the silent mode to true.

Normally the BSA3 Processor will keep a fixed IP-address.

The default IP-address for the BSA3 Processor is 10.10.100.130.

The default IP-address 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 BSA3 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 BSA3 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 BSA3 Processor. If you are not comfortable with changing values in the Windows Registry Database please ask your local network administrator for advice.

2.5.2 IBSA::ConnectEx

An alternative way to connect to the BSA3 Processor is to use the ConnectEx, which allows for targeting a specific Network or Network Interface Card (NIC). This is especially useful if the PC have multiple NICs installed.

Please see the "IBSA::Connect" (on page 40) method for general information about connecting to the BSA3 Processor.

```
HRESULT ConnectEx(
     /*[in]*/ BSTR IPAddress,
     /*[in]*/ BSTR NICAddress);

Sub ConnectEx(
     IPAddress As String,
     NICAddress As String)
```

Parameters

`IPAddress`

[in] The IP-address of the BSA3 Processor as it is seen on the network. The syntax consists of a common dotted octet IP-address, for example 10.10.100.130.

`NICAddress`

[in] The IP-address of the NIC in the PC. The syntax consists of a common dotted octet IP-address, for example 10.10.1.1.

Remarks

If you experience problems connecting to the BSA3 Processor using the "IBSA::Connect" (on page 40) method, the ConnectEx can be used to target a specific network.

2.5.3 IBSA::Disconnect

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

```
HRESULT Disconnect();

Sub Disconnect()
```

Parameters

This method has no parameters.

Remarks

When disconnected to the processor, the "IBSA::GetConnectState" (on page 43) method will return 0.

2.5.4 IBSA::GetConnectState

In order to determine the connection state of the BSA3 Processor you can call the GetConnectionState method. It is recommended to call this function after a call to Connect to ensure that the connection is established.

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

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 "IBSA::Connect" (on page 40) 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.

2.6 Configuring

This section describes the methods necessary to call to configure the BSA3 Processor prior to setting up the processor.

2.6.1 IBSA::SetProcessor

The BSA3 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 5-1" (on page 44).

```
HRESULT SetProcessor(
     /*[in]*/ int Processor)

Sub SetProcessor(
     Processor As Long)
```

Parameters

`Processor`

[in] Specifies the BSA3 Processor type used with the configuration.

Remark

The BSA3 Processor can be set to one of the following configuration modes:

Processor Model	Value
bsaLDA	0
bsaPDA	1

Table 5-1: Processor models.

Good programming requires that the SetProcessor method must be called before calling "IBSA::SetConfiguration " (on page 44). However because of backward compatibility issues the default BSA3 Processor model is set to bsaLDA, and it is in that case not necessary to call this method before "IBSA::SetConfiguration " (on page 44).

Defaults

The default BSA3 Processor model is set to bsaLDA = 0.

2.6.2 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 tree 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 tree 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 BSA3 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 5-2" (on page 45).

```
HRESULT SetConfiguration(
     /*[in]*/ int Configuration)
```

```
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.

It is necessary to specify the BSA3 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 "IBSA::SetOpticsPhaseFactor" (on page 101) method must be called prior to running an acquisition.

In case of PDA measurements the "IBSA::SetProcessor" (on page 43) 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 5-2: 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
bsa2DLDANONCOINC	3	2 × 1	X	X
bsa3DLDANONCOINC	4	3 × 1	X	X
bsa3DLDASEMICOINC	5	1 × 2 and 1 × 1	X	X
bsa1DPDA	6	1 × 1		X
bsa2DPDACOINC	7	1 × 2		X
bsa3DPDACOINC	8	1 × 3		X
bsa2DPDANONCOINC	9	2 × 1		X
bsa3DPDANONCOINC	10	3 × 1		X
bsa3DPDASEMICOINC	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.SetOpticsPhaseFactor 1, 3.1
MyBSA.SetOpticsPhaseFactor 2, 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 BSA3 Processor, no default values are specified.

2.6.3 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 BSA3 Processor Driver, see section "BSA Flow Software Script Generator" (on page 15). Or it can be hand-coded following the rules of the XML file structure describe in this section.

```
HRESULT SetProperties(
     /*[in]*/ BSTR XMLFile);

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 BSA3 Processor must be specified using the "IBSA::SetProcessor" (on page 43) and "IBSA::SetConfiguration " (on page 44) 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 BSA3 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 BSA3 Processor to use for the current setup. The valid BSA3 Processor models can be found in "IBSA::SetProperties" on the previous page. 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 "IBSA::SetProperties" on the previous page.

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]

```
<?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:

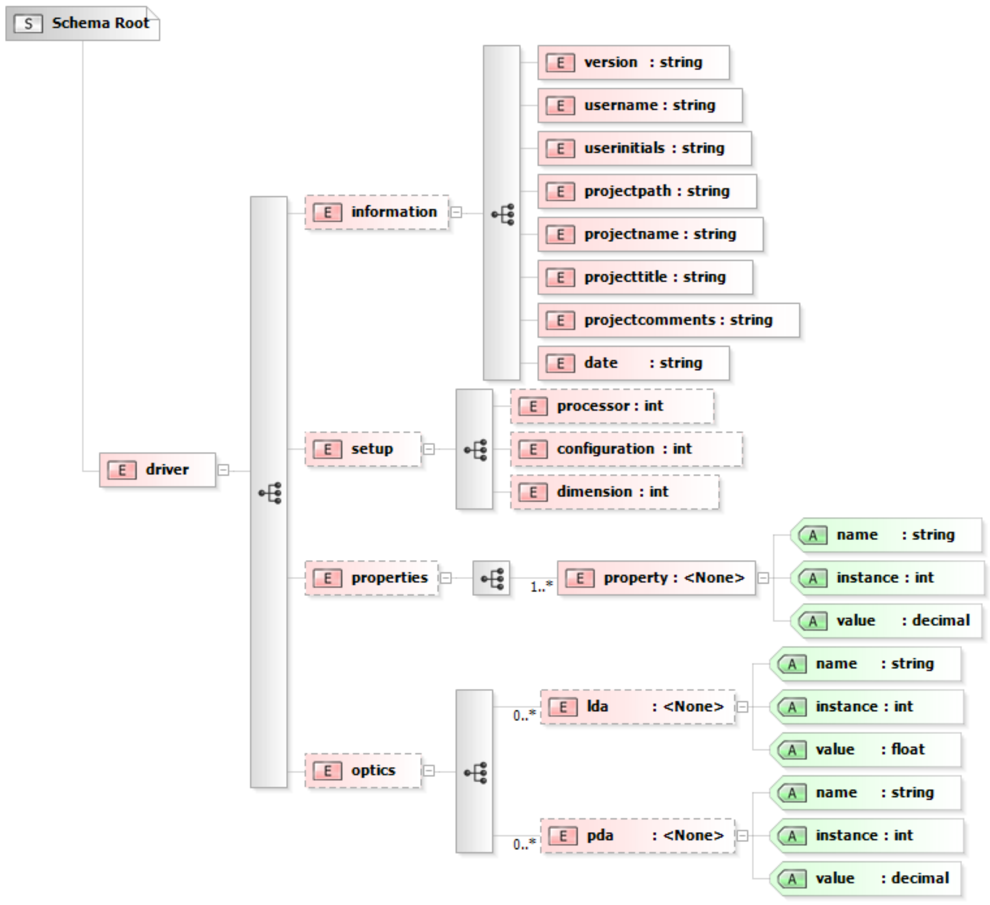
[XML]

```
<?xml version="1.0" ?>
<driver>
  <properties>
```

```
    <property name="MaxSamples" instance="0" value="10000" />
    <property name="MaxAcqTime" instance="0" value="10" />
  </properties>
</driver>
```

Schema

The following XSD schema can be used to interpreted and validate the XML file.



2.6.4 IBSA::ActivateSysMon

The System Monitor requires that the BSA3 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: "BSA System Monitor Control Function Reference" (on page 133).

```
HRESULT ActivateSysMon();

Sub ActivateSysMon()
```

Parameters

This method has no parameters.

Remarks

Please note, that the configuration of the BSA3 Processor must be set using the "IBSA::SetConfiguration " (on page 44) method, before the System Monitor can be activated.

This sample in a HTML VBScript block shows how to connect to the BSA3 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 in loaded.

[HTML VBScript]

```
<script language="vbscript">
set MyBSA = CreateObject("BSA.Ctrl")

' connects to the BSA3 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>
```

2.6.5 IBSA::DeactivateSysMon

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

```
HRESULT DeactivateSysMon();

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.

2.6.6 IBSA::SetLog

During test and debug of the script or program using the driver, it is often convenient to see debug log information of the execution of the methods in the driver.

Using the SetLog method this information is written to a file during execution of the methods in the driver.

```
HRESULT SetLog(
     /*[in]*/ BSTR Path);

Sub SetLog(
     Path As String)
```

Parameters

`Path`

[in] Path specifies a file path to the log file. The path must exist, and the access rights must be correct on the PC.

If the Path is empty a log file called “bsadriver.log” will be saved in the local temp folder.

[HTML VBScript]

```
<script language="vbscript">
set MyBSA = CreateObject("BSA.Ctrl")

' specify default logfile bsadriver.log'
' written to the temp folder'
MyBSA.SetLog ""

</script>
```

Remarks

If the SetLog method is not called, no log file will be written.

Please make sure that the filename and path is valid, and that both read and write access is allowed.

However, the BSA3 Processor Driver will always write debug information to Windows. The debug information can be read using e.g. DebugView from Windows Sysinternals.

2.7 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.

2.7.1 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.

```
HRESULT SetHighVoltageActivation(
     /*[in]*/ int Activation);

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 5-3: 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.

2.7.2 IBSA::GetHighVoltageActivation

This method gets the current state of the high voltage activation setting, set by the "IBSA::SetHighVoltageActivation" (on page 53).

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

Function GetHighVoltageActivation(
     ) As Long
```

Parameters

`pActivation`

[out, retval] A pointer to an int that is to hold voltage activation state. Refer to "Table 5-3" (on page 53) for possible return values.

Default

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

2.7.3 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 %.

```
HRESULT SetAnodeCurrentWarningLevel(
     /*[in]*/ int Level);

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 %.

2.7.4 IBSA::GetAnodeCurrentWarningLevel

This method gets the current anode current warning level.

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

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 %.

2.7.5 IBSA::SetDataCollectionMode

The BSA3 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 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, skrewness, 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.

```
HRESULT SetDataCollectionMode(
     /*[in]*/ int Mode);

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 5-4: 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 "IBSA::SetDutyCycle" (on page 56) method to set the duty-cycle when continuous mode is set, and use the "IBSA::SetDeadTime" (on page 57) method to set the values of the dead-time mode.

2.7.6 IBSA::GetDataCollectionMode

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

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

Function GetDataCollectionMode(
) As Long
```

Parameters

`pMode`

[out, retval] A pointer to an int that is to hold the current mode. Refer to "Table 5-4" (on page 55) for possible values.

Default

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

2.7.7 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.

```
HRESULT SetDutyCycle(
     /*[in]*/ float Cycle);

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 "IBSA::SetDataCollectionMode" (on page 55) method.

2.7.8 IBSA::GetDutyCycle

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

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

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 %.

2.7.9 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.

```
HRESULT SetDeadTime(
     /*[in]*/ float Time);

Sub SetDeadTime(
     Time As Single)
```

Parameters

`Time`

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

Remarks

This setting only has effect when the data collection mode is set to dead-time mode in the "IBSA::SetDataCollectionMode" (on page 55) method.

2.7.10 IBSA::GetDeadTime

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

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

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 µsec. to 10 sec.

Default

The default dead-time is 100 µsec.

2.8 Frequency Shift Settings

The functions in this section control the signals on the Bragg cell connectors at the rear panel of the BSA3 Processor. These can be used individually or in combination.

The three connectors can be configured to any frequency between 20 MHz and 120 MHz, supporting all combinations of old and new optical LDA systems like: FiberFlow, FlowLite and FlowExplorer.

2.8.1 IBSA::SetBraggCellOutput

Only applicable to the BSA3 Processor.

Bragg Cell Output directs a Bragg cell driver signal to on of the three BNC connectors named Bragg cell 1 - 3. The drive frequency is defined by the property "IBSA::SetBraggCellOutputFreq" (on page 59).

The BSA3 Processor supports up to 3 output Bragg Cell frequencies. Different variable Bragg Cell frequencies can be assigned in the range 30 - 120 MHz.

```
HRESULT SetBraggCellOutput(
     /*[in]*/ int Output,
     /*[in]*/ int State);

Sub SetBraggCellOutput(
     Output As Long,
     State As Long)
```

Parameters

`Output`

[in] Specifies the output. This value can be 1, 2 or 3 referring to the Bragg Cell 1-3.

`State`

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

Remarks

Output can be set to one of the following constants:

Table 5-5: Bragg Cell Output State Constants

Constant	Value
bsaOFF	0
bsaON	1

2.8.2 IBSA::GetBraggCellOutput

Only applicable to the BSA3 Processor.

This method gets the state of one of the Bragg cell outputs, set by the "IBSA::SetBraggCellOutput" (on page 58) method.

```
HRESULT GetBraggCellOutput(
     /*[in]*/ int Output,
     /*[out, retval]*/ int* pState);
```

```
Function Get40MHzFreqShiftOutput(
     Output As Long,
) As Long
```

Parameters

`Output`
: [in] Specifies the output. This value can be 1, 2 or 3 referring to the Bragg cell Outputs 1-3.

`pState`
: [out, retval] A pointer to an int that is to hold the Bragg cell output state for the specified output. Refer to "Table 5-5" (on page 58) for possible values.

Default

The default setting of all three Bragg cells outputs are on bsaON = 1.

2.8.3 IBSA::SetBraggCellOutputFreq

Only applicable to the BSA3 Processor.

Bragg cell output defines the frequency of the driver signal at each of the Bragg Cell 1-3 connectors on the back of the BSA3 Processor. The Bragg cell frequency signal must be enabled using the "IBSA::SetBraggCellOutput" (on page 58) method.

```
HRESULT SetBraggCellOutputFreq(
     /*[in]*/ int Output,
     /*[in]*/ int Shift);

Sub SetBraggCellOutputFreq(
     Output As Long,
     Shift As Long)
```

Parameters

`Output`
: [in] Specifies the output. This value can be 1, 2 or 3 referring to the Bragg Cell 1-3.

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

2.8.4 BraggCellOutputFreq

Only applicable to the BSA3 Processor.

This method gets the current Bragg cell output frequency.

```
HRESULT GetBraggCellOutputFreq(
     /*[in]*/ int Output,
     /*[out, retval]*/ int* pShift);

Function Get40MHzFreqShiftOutputFreq(
     Output As Long,
) As Long
```

Parameters

`Output`
: [in] Specifies the output. This value can be 1, 2 or 3 referring to the Bragg Cell 1-3.

`pShift`

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

Default

The default settings of the Bragg cell output frequencies are Bragg Cell 1 = 80 MHz, Bragg Cell 2 = 40 MHz, and Bragg Cell 3 = 80 MHz.

2.9 Synchronization Input Signals

The BSA3 Processor has a number of input connectors for synchronization and handshake with external equipment. The software controls the routing of these signals from the connectors on the rear panel through the synchronization input signals.

2.9.1 Dependencies

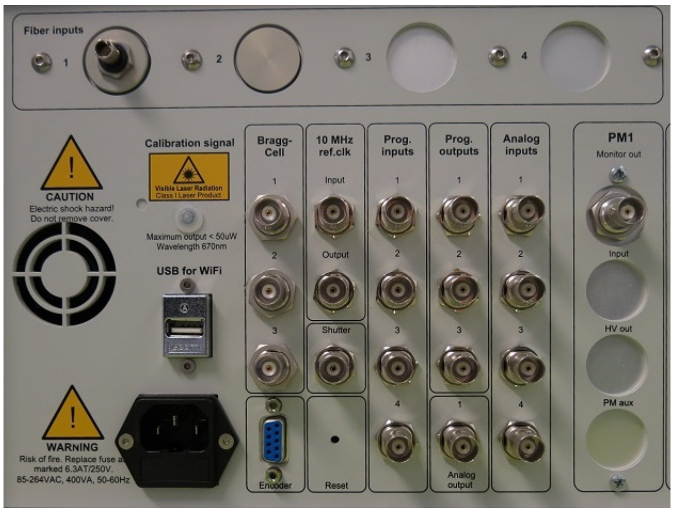
All processors do not support the input and output synchronization options. The following table shows the dependencies.

Figure 5-7: BSA3 Processor Models and Synchronization Capabilities

Processor Model	Remarks
F600, F800	Only supported through the optional 63S06 Synchronization Option.

2.9.2 Programmable Inputs

The synchronization input connectors can be found as four BNC connectors named Prog. inputs 1, 2, 3 and 4, and the 9-pin D-Sub connector named Encoder on the back of the BSA3 Processor, as shown on the image below.


Fiber inputs
1
2
3
4
CAUTION
Electric shock hazard! Do not remove cover.
Calibration signal
Visible Laser Radiation Class I Laser Product
Maximum output < 50uW Wavelength 670nm
USB for WiFi
WARNING
Risk of fire. Replace fuse as marked 6.3AT/250V. 85-264VAC, 400VA, 50-60Hz
Bragg-Cell
10 MHz ref.clk
Input
Output
Shutter
Prog. inputs
Prog. outputs
Analog inputs
Analog output
Encoder
Reset
PM1
Monitor out
Input
HV out
PM aux

Programmable Inputs (BNC Connectors)

Four general purpose programmable inputs are available on four BNC connectors. These connectors are refers as Prog. input 1, Prog. input 2, Prog. input 3, and Prog. input 4.

Encoder Inputs (D-Sub Connector)

The Encoder DSUB connector is designed to be used together with an digital quadrature encoder, for e.g. measure of the distance of the rotation of a wheel or shaft, a motion control systems and similar. The terminology often used for encoders are indicated by A±, B± and IDX±. The A+ and B+ are referred to as the signal lines and the IDX+ is known as the index pulse. The A-, B- and IDX- are the inverse of the + versions of those signals; these are useful when operating in a high-noise environment.

Note
Please consult the manual for the device generating the encoder pulses for correct connection.

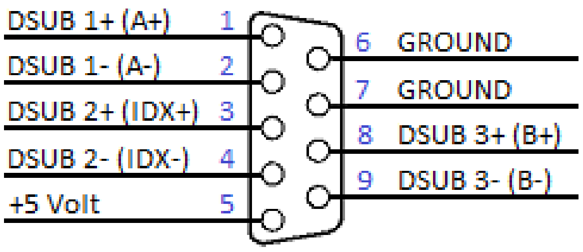


Table 5-6: The pin numbering of the Encoder connector.

Pin	Signal	Pin	Signal
1	DSUB 1+ (A+)	6	Ground
2	DSUB 1- (A-)	7	Ground
3	DSUB 2+ (IDX+)	8	DSUB 3+ (B+)
4	DSUB 2- (IDX-)	9	DSUB 3- (B-)
5	+ 5 Volt		

The DSUB connector can also be used as a general purpose differential input connectors. In this case the DSUB 1, DSUB 2, and DSUB 3 can be used individually.

2.9.3 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.

```
HRESULT SetSync1(
     /*[in]*/ int Connector);

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 "2.9" (on page 60).

Remarks

Type can be set to one of the following constants:

Table 5-7: Connector Constants

Constant	Value
bsaNONE	0
bsaBNC1	1
bsaBNC2	2
bsaBNC3	3
bsaBNC4	4
bsaDSUB1	5
bsaDSUB2	6
bsaDSUB3	7

2.9.4 IBSA::GetSync1

Gets the input connector to which the Sync1 in connected.

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

Function GetSync1(
) As Long
```

Parameters

`pConnector`
: [out, retval] A pointer to an int that is to hold the connector to witch the Sync1 signal is connected. Refer to "Table 5-7" (on page 62) for possible return values.

Default

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

2.9.5 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.

```
HRESULT SetSync1Edge(
     /*[in]*/ int Edge);

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 5-8: Trigger Edge Constants

Constant	Value
bsaNEGATIVE	0
bsaPOSITIVE	1

2.9.6 IBSA::GetSync1Edge

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

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

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 5-8" (on page 63) for possible return values.

Default

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

2.9.7 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.

```
HRESULT SetSync2(
     /*[in]*/ int Connector);

Sub SetSync2(
     Connector As Long)
```

Parameters

`Connector`

[in] Specifies the connector that the Sync2 signal is connected. Refer to "Table 5-7" (on page 62) for possible input values.

2.9.8 IBSA::GetSync2

Gets the input connector to which the Sync2 in connected.

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

Function GetSync2(
) As Long
```

Parameters

`pConnector`

[out, retval] A pointer to an int that is to hold the connector to witch the Sync2 signal is connected. Refer to "Table 5-7" (on page 62) for possible return values.

Default

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

2.9.9 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.

```
HRESULT SetSync2Edge(
     /*[in]*/ int Edge);

Sub SetSync2Edge(
     Edge As Long)
```

Parameters

`Edge`

[in] Specifies the edge on which to trigger Sync2. Refer to "Table 5-8" (on page 63) for possible input values.

2.9.10 IBSA::GetSync2Edge

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

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

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 5-8" (on page 63) for possible return values.

Default

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

2.9.11 IBSA::SetEncoderReset

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

```
HRESULT SetEncoderReset(
     /*[in]*/ int Connector);

Sub SetEncoderReset(
     Connector As Long)
```

Parameters

`Connector`
[in] Specifies the connector that the encoder reset signal is to be connected. Refer to "Table 5-7" (on page 62) 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.

2.9.12 IBSA::GetEncoderReset

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

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

Function GetResetEncSignal(
) As Long
```

Parameters

`pConnector`
[out, retval] A pointer to an int that is to hold the connector to witch the encoder reset signal is connected. Refer to "Table 5-7" (on page 62) for possible return values.

Default

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

2.9.13 IBSA::SetEncoderResetEdge

Only applicable to the BSA3 Processor.

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

```
HRESULT SetEncoderResetEdge(
     /*[in]*/ int Edge);
```

```
Sub SetEncoderResetEdge(
     Edge As Long)
```

Parameters

`Edge`

[in] Specifies the edge on which to trigger the Encoder Reset. Refer to "Table 5-8" (on page 63) for possible values.

2.9.14 IBSA::GetEncoderResetEdge

Only applicable to the BSA3 Processor.

This method returns the edge on which the Encoder Reset is being triggered.

```
HRESULT GetEncoderResetEdge(
     /*[out, retval]*/ int* pEdge);

Function GetEncoderResetEdge(
) As Long
```

Parameters

`pEdge`

[out, retval] A pointer to an int that is to hold the edge on which the Encoder Reset is triggered. Refer to "Table 5-8" (on page 63) for possible return values.

Default

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

2.9.15 IBSA::SetEncoder

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

```
HRESULT SetEncoder(
     /*[in]*/ int Connector);

Sub SetEncoder(
     Connector As Long)
```

Parameters

`Connector`

[in] Specifies the connector that the encoder clock signal is to be connected. Refer to "Table 5-7" (on page 62) 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.

2.9.16 IBSA::GetEncoder

Gets the input connector to which the Encoder is connected.

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

Function GetEncoder(
) As Long
```

Parameters

`pConnector`

[out, retval] A pointer to an int that holds the connector to witch the encoder clock signal is connected. Refer to "Table 5-7" (on page 62) for possible return values.

Default

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

2.9.17 IBSA::SetEncoderEdge

Only applicable to the BSA3 Processor.

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

```
HRESULT SetEncoderEdge(
     /*[in]*/ int Edge);

Sub SetEncoderEdge(
     Edge As Long)
```

Parameters

`Edge`

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

2.9.18 IBSA::GetEncoderEdge

Only applicable to the BSA3 Processor.

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

```
HRESULT GetEncoderEdge(
     /*[out, retval]*/ int* pEdge);

Function GetEncoderEdge(
) As Long
```

Parameters

`pEdge`

[out, retval] A pointer to an int that is to hold the edge on which the Encoder is triggered. Refer to "Table 5-8" (on page 63) for possible return values.

Default

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

2.9.19 IBSA::SetStartMeasurement

Start measurement is a trigger input, that will start data acquisition and hence the BSA3 Processors arrival time counter on the rising edge of a TTL pulse. Start measurement signal is mapped to one of the selectable connectors.

```
HRESULT SetStartMeasurement(
     /*[in]*/ int Connector);

Sub SetStartMeasurement(
     Connector As Long)
```

Parameters

`Connector`

[in] Specifies the connector that the start measurement signal is to be connected. Refer to "Table 5-7" (on page 62) for possible input values.

2.9.20 IBSA::GetStartMeasurement

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

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

Function GetStartMeasurement(
) As Long
```

Parameters

`pConnector`

[out, retval] A pointer to an int that is to hold the connector to witch the start measurement signal is connected. Refer to "Table 5-7" (on page 62) for possible return values.

Default

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

2.9.21 IBSA::SetStartMeasurementEdge

Only applicable to the BSA3 Processor.

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

```
HRESULT SetStartMeasurementEdge(
     /*[in]*/ int Edge);

Sub SetStartMeasurementEdge(
     Edge As Long)
```

Parameters

`Edge`

[in] Specifies the edge on which to trigger Start Measurement.

2.9.22 IBSA::GetStartMeasurementEdge

Only applicable to the BSA3 Processor.

This method returns the edge on which Start Measurement is being triggered.

```
HRESULT GetStartMeasurementEdge(
     /*[out, retval]*/ int* pEdge);

Function GetStartMeasurementEdge(
) As Long
```

Parameters

`pEdge`

[out, retval] A pointer to an int that is to hold the edge on which Start Measurement is triggered. Refer to "Table 5-8" (on page 63) for possible return values.

Default

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

2.9.23 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.

```
HRESULT SetStopMeasurement(
     /*[in]*/ int Connector);

Sub SetStopMeasurement(
     Connector As Long)
```

Parameters

`Connector`

[in] Specifies the input, which stops measurement, is to be connected to. Refer to "Table 5-7" (on page 62) for possible input values.

2.9.24 IBSA::GetStopMeasurement

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

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

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 5-7" (on page 62) for possible return values.

Default

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

2.9.25 IBSA::SetStopMeasurementEdge

Only applicable to the BSA3 Processor.

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

```
HRESULT SetStopMeasurementEdge(
     /*[in]*/ int Edge);

Sub SetStopMeasurementEdge(
     Edge As Long)
```

Parameters

`Edge`

[in] Specifies the edge on which to trigger Stop Measurement.

2.9.26 IBSA::GetStopMeasurementEdge

Only applicable to the BSA3 Processor.

This method returns the edge on which Stop Measurement is being triggered.

```
HRESULT GetStopMeasurementEdge(
     /*[out, retval]*/ int* pEdge);

Function GetStopMeasurementEdge(
) As Long
```

Parameters

`pEdge`

[out, retval] A pointer to an int that is to hold the edge on which Stop Measurement is triggered. Refer to "Table 5-8" (on page 63) for possible return values.

Default

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

2.9.27 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.

```
HRESULT SetBurstDetectorEnable(
     /*[in]*/ int Connector);

Sub SetBurstDetectorEnable(
     Connector As Long)
```

Parameters

`Connector`

[in] Specifies the connector that the burst detector enable signal is to be connected. Refer to "Table 5-7" (on page 62) for possible input values.

2.9.28 IBSA::GetBurstDetectorEnable

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

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

Function GetBurstDetectorEnable(
) As Long
```

Parameters

`pConnector`

[out, retval] A pointer to an int that is to hold the input connector to witch the burst detector enable signal is connected. Refer to "Table 5-7" (on page 62) for possible return values.

Default

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

2.9.29 IBSA::SetBurstDetectorEnableLevel

Only applicable to the BSA3 Processor.

Burst detector enable level defines when the burst detector is enabled. The burst detector can be enabled on either a low or a high input signal.

```
HRESULT SetBurstDetectorEnableLevel(
     /*[in]*/ int Level);

Sub SetBurstDetectorEnableLevel(
     Level As Long)
```

Parameters

`Level`

[in] Specifies the level on which the burst detector is enabled.

Remarks

Level can be set to one of the following constants:

Table 5-9: Enable Level Constants

Constant	Value
bsaLOW	0
bsaHIGH	1

2.9.30 IBSA::GetBurstDetectorEnableLevel

Only applicable to the BSA3 Processor.

This method returns the level on which the burst detector is enabled.

```
HRESULT GetBurstDetectorEnableLevel(
     /*[out, retval]*/ int* pLevel);
```

```
Function GetBurstDetectorEnableLevel(
) As Long
```

Parameters

pLevel
[out, retval] A pointer to an int that is to hold the level on which the burst detector is enabled. Refer to "Table 5-9" (on page 71) for possible return values.

Default

The default setting of the edge direction is low bsaLOW = 0.

2.9.31 IBSA::SetRefClk

Reference clock mode can be set to internal or external. Default is internal. If two or more BSA3 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.

```
HRESULT SetRefClk(
     /*[in]*/ int Clk);

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 5-10: Reference Clock Constants

Constant	Value
bsaINTERNCLK	0
bsaEXTERNCLK	1

2.9.32 IBSA::GetRefClk

This method gets the setting of reference clock.

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

Function GetRefClk(
) As Long
```

Parameters

`pClk`

[out, retval] A pointer to an int that holds the reference clock setting. Refer to "Table 5-10" (on page 72) for possible return values.

Default

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

2.10 Synchronization Output Signals

The BSA3 Processor has a number of output connectors for synchronization and handshake with external equipment. The software controls the routing of these signals to the connectors on the rear panel through the synchronization output signals.

2.10.1 Dependencies

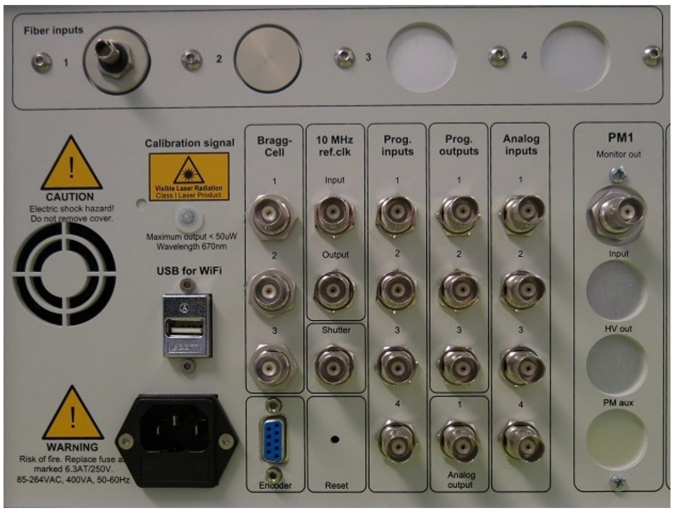
All processors do not support the input and output synchronization options. The following table shows the dependencies.

Figure 5-8: BSA3 Processor Models and Synchronization Capabilities

Processor Model	Remarks
F600, F800	Only supported through the optional 63S06 Synchronization Option.

2.10.2 Programmable Outputs

The synchronization output connectors can be found as three BNC connectors named Prog. outputs 1, 2, and 3 on the back of the BSA3 Processor, as shown on the image below.



Programmable Outputs (BNC Connectors)

Three general purpose programmable outputs are available on three BNC connectors. These connectors are refers as Prog. output 1, Prog. output 2, and Prog. output 3

2.10.3 IBSA::SetOutput

Only applicable to the BSA3 Processor.

This setting is used to specify the signal that is to be output on each of the three Programmable Output BNC connectors.

```
HRESULT SetOutput(
     /*[in]*/ int Output,
     /*[in]*/ int Signal);

Sub SetOutput(
     Output As Long,
     Signal As Long)
```

Parameters

`Output`
: [in] Specifies the output. This value can be 1, 2 or 3 referring to the Prog. Output 1-3.

`Signal`
: [in] Specifies the signal.

Remarks

Signal can be set to one of the following constants:

Figure 5-9: Synchronization Signal Constants

Constant	Value
bsaUNUSED	0
bsaBURSTDETECT1	1
bsaBURSTDETECT2	2
bsaBURSTDETECT3	3
bsaBURSTDETECT4	4
bsaBURSTDETECT5	5
bsaMEASUREMENTRUNNING	7

2.10.4 IBSA::GetOutput

Only applicable to the BSA3 Processor.

This method gets the current signal mapped to one of the three Programmable Output BNC connectors.

```
HRESULT GetBNC(
     /*[in]*/ int Output,
     /*[out, retval]*/ int* pSignal);

Function GetBNC(
     Output As Long,
) As Long
```

Parameters

`Output`

[in] Specifies the output. This value can be 1, 2 or 3 referring to the Prog. Output 1-3.

`pSignal`

[out, retval] A pointer to an int that is to hold the selected signal for the specified output. Refer to "Figure 5-9" (on page 74) for possible return values.

Default

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

2.11 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 "IBSA::SetConfiguration " (on page 44) method.

2.11.1 IBSA::SetMaxSamples

Maximum number of samples; stop criterion for data acquisition for the selected group. Together with the "IBSA::SetMaxAcqTime" (on page 76) method this constitutes the limits of the acquisition for the group.

```
HRESULT SetMaxSamples(
     /*[in]*/ int Group,
     /*[in]*/ int Samples)

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 "IBSA::SetMaxSamples" (on page 75); the actual number of acquired samples can be found by calling the "IBSA::ReadDataLength" (on page 112) method for the group.

2.11.2 IBSA::GetMaxSamples

This method gets the number of samples set by the method "IBSA::SetMaxSamples" (on page 75).

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

```
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.

2.11.3 IBSA::SetMaxAcqTime

The maximum acquisition time is one of the stop criteria for data acquisition for the selected coincidence group. Together with the "IBSA::SetMaxSamples" (on page 75) method this constitutes the limits of the acquisition for the group.

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

Sub SetMaxAcqTime(
     Group As Long,
     Time As Single)
```

Parameters

`Group`
: [in] Specifies the group.

`Time`
: [in] Specifies the maximum acquisition time in seconds.

Limitations

The `Time` value in the BSA3 Processor will be rounded to the nearest integer value. This means, that you can only specify the maximum acquisition time in steps of 1 second.

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.

2.11.4 IBSA::GetMaxAcqTime

This method gets the maximum acquisition time set by the method "IBSA::SetMaxAcqTime" (on page 76).

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

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.

2.11.5 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 "IBSA::SetBurstWindow" (on page 78) method. This setting may be useful when measuring through curved window surfaces, or other situations where burst overlap may not be possible.

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

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 5-11: Filter Method Constants

Constant	Value
bsaWINDOW	0
bsaOVERLAP	1

2.11.6 IBSA::GetFilterMethod

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

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

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 5-11" (on page 77) for possible return values.

Default

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

2.11.7 IBSA::SetBurstWindow

Burst window defines the time window for coincidence filtering by the windowing mode set by the "IBSA::SetFilterMethod" (on page 77) method.

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

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 µsec to 1 sec.

2.11.8 IBSA::GetBurstWindow

This method gets the time set by the method "IBSA::SetBurstWindow" (on page 78).

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

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 μsec to 1 sec set by the method "IBSA::SetBurstWindow" (on page 78).

Default

The default setting of the burst windowing width is 10 μsec.

2.11.9 IBSA::SetScopeDisplay

Determines what to show in the scope display in the System Monitor for a specific group.

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

Sub SetScopeDisplay(
    Group As Long,
     Display As Long)
```

Parameter

`Group`
: [in] Specifies the group.

`Display`
: [in] Specifies what to show in the scope display.

Remarks

Display can be set to one of the following constants:

Table 5-12: Scope Monitor Constants

Constant	Value
bsaOFF	0
bsaBURSTSIGNAL	1
bsaBURSTSPECTRUM	2

2.11.10 IBSA::GetScopeDisplay

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

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

Function GetScopeDisplay(
     Group As Long
) As Long
```

Parameter

`Group`
: [in] Specifies the group.

`pDisplay`

[out, retval] A pointer to a int that describes what is shown in the scope display for the specified group. Refer to "Table 5-12" (on page 79) for possible return values.

Default

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

2.11.11 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 bust 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.

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

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 5-13: Trigger Channel Constants

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

2.11.12 IBSA::GetScopeTrigger

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

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

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 5-13" (on page 80) 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.

2.11.13 IBSA::SetScopeZoom

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

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

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 %.

2.11.14 IBSA::GetScopeZoom

This method gets the zoom value set by the "IBSA::SetScopeZoom" (on page 81) method.

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

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 %.

2.11.15 IBSA::SetScopeYScale

Only applicable to the BSA3 Processor.

Y scale determines the scaling of the scope display y axis in the System Monitor.

```
HRESULT SetScopeYScale(
     /*[in]*/ int Group,
     /*[in]*/ float Scale);

Sub SetScopeYScale(
     Group As Long,
     Scale As Single)
```

Parameters

`Group`
: [in] Specifies the group.

`Scale`
: [in] Specifies the y scale. `Scale` can be set in the range from 0.01 to 100.

2.11.16 IBSA::GetScopeYScale

This method gets the y scale value set by the "IBSA::SetScopeYScale" (on page 82) method.

```
HRESULT GetScopeYScale(
     /*[in]*/ int Group,
     /*[out, retval]*/ float* pScale);

Function GetScopeYScale(
     Group As Long
) As Single
```

Parameters

`Group`
: [in] Specifies the group.

`pScale`
: [out, retval] A pointer to a floating point holding the value of the scope y scale for the specified group. On return `pScale` is set in the range from 0.01 to 100.

Default

The default setting of the scope y scale is 1.

2.11.17 IBSA::SetScopeAveraging

Only applicable to the BSA3 Processor.

The Scope averaging is implemented as an first order IIR filter. The filter length is given by the averaging parameter.

```
HRESULT SetScopeAveraging(
     /*[in]*/ int Group,
     /*[in]*/ int Avg);

Sub SetScopeAveraging(
     Group As Long,
     Avg As Long)
```

Parameters

`Group`
: [in] Specifies the group.

`Avg`
: [in] Specifies the averaging parameter. `Avg` can be set in the range from 1 to 100.

Remarks

The scope averaging only works for spectra in the System Monitor. The method "IBSA::SetScopeDisplay" (on page 79) can be used to specify what to show in the scope display of the System Monitor. The averaging ha only effect if this value is set to bsaBURSTSPECTRUM, see "Table 5-12" (on page 79).

2.11.18 IBSA::GetScopeAveraging

This method gets the zoom value set by the "IBSA::SetScopeAveraging" (on page 82) method.

```
HRESULT GetScopeAveraging(
     /*[in]*/ int Group,
     /*[out, retval]*/ int* pAvg);

Function GetScopeAveraging(
     Group As Long
) As Long
```

Parameters

`Group`
: [in] Specifies the group.

`pAvg`
: [out, retval] A pointer that holds the scope averaging for the specified group. On return `pAvg` is set in the range from 1 to 100.

Default

The default setting of the scope averaging is 5.

2.12 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 processor model.

Table 5-14: BSA3 Processor models and Frequencies and Bandwidths

Parameter	BSA F/P600	BSA F/P800
Max. input frequency (MHz)	120	200
Min. input frequency (MHz)	15	15
Max. bandwidth (MHz)	80	160
Min. bandwidth (MHz)	0.080	0.080
Max. center frequency (MHz)	?	?
Min. center frequency (MHz)	?	?
No. of bandwidths	31	34

2.12.1 IBSA::SetCenterFreq

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

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

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 is it has changed using the "IBSA::GetBandwidth" (on page 86) method.

2.12.2 IBSA::GetCenterFreq

This method gets the center frequency from a specified channel. The center frequency can be set using the "IBSA::SetCenterFreq" (on page 84) method or it can have been changed because the bandwidth has been changed using the "IBSA::SetBandwidth" (on page 85) method.

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

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.

2.12.3 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 width, make an acquisition, and zoom in to proper center frequency and bandwidth afterward.

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

Sub SetBandwidth(
     Channel As Long,
     Bandwidth As Single)
```

Parameters

`Channel`
: [in] Specifies the input channel.

`Bandwidth`
: [in] Bandwidth in Hz.

Limitations

The bandwidth in the BSA3 Processor needs to be set to a valid value. The valid values for the bandwidth are dependent on the specified center frequency set by the "IBSA::SetCenterFreq" (on page 84) method.

Table 5-15: BSA3 ProcessorValid Bandwidths

BSA3 Processor F/P600 and F/P800 Models			
80000 MHz	100000 MHz	120000 MHz	160000 MHz
200000 MHz	240000 MHz	310000 MHz	390000 MHz
490000 MHz	630000 MHz	780000 MHz	980000 MHz
1250000 MHz	1560000 MHz	1950000 MHz	2500000 MHz
3130000 MHz	3910000 MHz	5000000 MHz	6250000 MHz
7810000 MHz	10000000 MHz	12500000 MHz	15630000 MHz
20000000 MHz	25000000 MHz	31250000 MHz	40000000 MHz
50000000 MHz	62500000 MHz	80000000 MHz	
BSA3 Processor F/P800 Models Only			
100000000 MHz	125000000 MHz	160000000 MHz	

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 is it has changed using the "IBSA::GetCenterFreq" (on page 84) method.

The resulting frequency is also dependent of the optical shift (Bragg cell) frequency set by the "IBSA::SetOpticsShift" (on page 98) method.

2.12.4 IBSA::GetBandwidth

Gets the center frequency from a specified channel.

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

Function GetBandwidth(
     Channel As Long
) 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.

2.12.5 IBSA::SetRecordLengthMode

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

In auto-adaptive record length mode, the BSA3 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 burst signal in the System Monitor display.

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

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 5-16: Record Length Mode Constants

Constant	Value
bsaFIXED	0
bsaAUTO	1

2.12.6 IBSA::GetRecordLengthMode

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

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

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 5-16" (on page 87) for possible return values.

Default

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

2.12.7 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.

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

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 5-17: Record Length Constants

Constant	Value
bsaREC16	16
bsaREC32	32
bsaREC64	64
bsaREC128	128
bsaREC256	256
bsaREC512	512
bsaREC1024	1024

2.12.8 IBSA::GetRecordLength

This method gets the record length from a specified channel.

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

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 5-17" (on page 88) 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.

2.12.9 IBSA::SetMaxRecordLength

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

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

Sub SetMaxRecordLength(
     Channel As Long,
     Length As Long)
```

Parameters

`Channel`

[in] Specifies the input channel.

`Length`

[in] The maximum record length. Refer to "Table 5-17" (on page 88) 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.

2.12.10 IBSA::GetMaxRecordLength

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

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

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 5-17" (on page 88) 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.

2.12.11 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 "IBSA::SetSignalGain" (on page 91) 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.

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

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.

Remarks

For setting the high voltage on PDA channels, please refer to "IBSA::SetPDAHighVoltage" (on page 95).

2.12.12 IBSA::GetHighVoltage

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

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

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.

2.12.13 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.

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

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

2.12.14 IBSA::GetSignalGain

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

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

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.

2.12.15 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.

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

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.

2.12.16 IBSA::GetBurstDetectorSNR

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

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

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.

2.12.17 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.

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

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 5-18: Level Validation Constants

Constant	Value
bsaLEVELOFF	0
bsaLEVEL3	3
bsaLEVEL4	4
bsaLEVEL5	5
bsaLEVEL6	6
bsaLEVEL7	7
bsaLEVEL8	8

Limitations

2.12.18 IBSA::GetLevelValidationRatio

This method gets the validation ratio from the BSA3 Processor set by the "IBSA::SetLevelValidationRatio" (on page 92) method.

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

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 5-18" (on page 93) for possible return values.

Default

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

2.13 PDA Settings

PDA settings are specific to P-series processors. The methods involve the automatic phase calibration in the processor, and phase and diameter validation.

The methods are only valid for PDA measurements.

2.13.1 IBSA::SetPDACalibrationMode

Only applicable to the BSA3 Processor.

Defines the optical PDA phase calibration mode. PDA phase calibration is necessary for keeping relation between BSA3 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).

```
HRESULT SetPDACalibrationMode(
     /*[in]*/ int Mode);

Sub SetPDACalibrationMode(
     Mode As Long)
```

Parameter

`Mode`

[in] Specifies if PDA calibration is performed.

Remarks

The phase calibration modes determine when the PDA BSA3 Processor performs a calibration. The calibration will be performed before an acquisition is started.

The default setting is automatic calibration bsaCALIBRATEYES. This will calibrate whenever one of the involved properties are changed.

The second calibration mode bsaCALIBRATENO 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.

Figure 5-10: PDA Calibration Modes

Constant	Value
bsaCALIBRATEYES	1
bsaCALIBRATENO	0

2.13.2 IBSA::GetPDACalibrationMode

Only applicable to the BSA3 Processor.

This method gets the calibration mode from the BSA3 Processor set by the "IBSA::SetPDACalibrationMode" (on page 94) method.

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

Function GetPDACalibrationMode(
) As Long
```

Parameter

`pMode`

[out, retval] A pointer to the PDA calibration mode. See "Figure 5-10" (on page 94) for possible return values.

Default

The default value is bsaCALIBRATEYES.

2.13.3 IBSA::SetPDAHighVoltage

This method sets the high voltage to the photo-multiplier for a specified PDA channel. Please refer to the description of the "IBSA::SetHighVoltage" (on page 90) method.

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

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.

2.13.4 IBSA::GetPDAHighVoltage

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

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

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.

2.13.5 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.

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

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 BSA3 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 5-19: Auto Balancing Modes

Constant	Value
bsaON	1
bsaOFF	0

2.13.6 IBSA::GetPDAAutoBalancing

This method gets the automatic balancing mode for a specific phase channel set by the "IBSA::SetPDAAutoBalancing" (on page 95) method.

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

Function GetPDAAutoBalancing(
     Channel As Long
) As Long
```

Parameter

`pBalancing`

[out, retval] A pointer to the automatic balancing mode. See "Table 5-19" (on page 96) for possible return values.

Default

The default value is bsaOFF.

2.13.7 IBSA::CalibratePDA

When CalibratePDA is called, a command is send to the BSA3 Processor to perform a PDA calibration. This method only works when the BSA3 Processor is configured to be a PDA processor using the "IBSA::SetProcessor" (on page 43) method.

```
HRESULT CalibratePDA();

Sub CalibratePDA();
```

Parameters

This method has no parameters.

Remarks

A PDA calibration typically takes between 2 – 5 seconds to complete, depending on the BSA3 Processor settings etc. The CalibratePDA method will however return immediately. The "IBSAEvents::onCalDone" (on page 123) event can be used to determine when the PDA calibration has finished.

2.14 Optics Settings

The BSA3 Processor supports three different optical systems: LDA Optics, PDA Optics and DualPDA Optics.

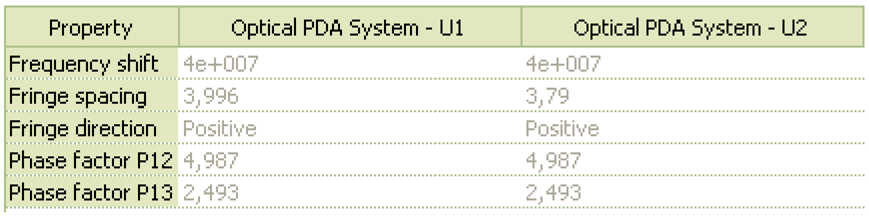
To be able to measure zero velocities and to remove the directional ambiguity the BSA3 Processor supports frequency shifted beam pairs. The optical shift between the beams defines the measurement range and is typically 40 MHz or 80 MHz for Dantec Dynamics optical systems.

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 Flow Software application, and then entered into your appropriate driver methods.

Figure 5-11: 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

The relation between the methods in the BSA Processor Driver and the optical properties are:

IBSA::SetOpticsShift(<Frequency shift property>)

IBSA::SetOpticsConversionFactor(<Fringe spacing property>)

IBSA::SetOpticsValidationBand(<Phase ratio validation property>

IBSA::SetOpticsFringeDirection(<Fringe direction property>)

IBSA::SetOpticsPhaseFactor(1, <Phase factor P12 property>)

IBSA::SetOpticsPhaseFactor(2, <Phase factor P13 property>)

2.14.1 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.

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

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

We recommend using BSA Flow Software to calculate and find the optical settings based on a specific optical PDA configuration. Alternatively the BSA Flow Software Script Generator can be used, please refer to section "BSA Flow Software Script Generator" (on page 15). The optical frequency shift value corresponds to the output property *Frequency shift*, which can be read for each channel in the output properties in BSA Flow Software. Please refer to the BSA Flow Software Installation and User's Guide for more information.

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 "IBSA::SetBandwidth" (on page 85) and "IBSA::SetCenterFreq" (on page 84) 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 BSA3 Processor provides three frequency outputs than can be used as inputs to your Bragg cell in your optical system. Please refer to the "IBSA::SetBraggCellOutput" (on page 58) method.

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.

2.14.2 IBSA::GetOpticsShift

This method gets the optical shift frequency for a given channel earlier set by the method "IBSA::SetOpticsShift" (on page 98).

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

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.

2.14.3 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 "IBSA::ReadVelocityData" (on page 114) method, will be returned using the conversion factor.

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

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

We recommend using BSA Flow Software to calculate and find the optical settings based on a specific optical PDA configuration. Alternatively the BSA Flow Software Script Generator can be used, please refer to section "BSA Flow Software Script Generator" (on page 15). The optical conversion factor value corresponds to the output property *Fringe spacing*, which can be read in the output properties in BSA Flow Software. For calibrated probes the conversion factor is directly indicated on the probe. Please refer to the BSA Flow Software Installation and User's Guide for more information.

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 chancing the conversion factor in ft/s.

[VBScript]

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

2.14.4 IBSA::GetOpticsConversionFactor

This method gets the optical conversion factor for a given channel earlier set by the method "IBSA::SetOpticsConversionFactor" (on page 99).

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

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.

2.14.5 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 "IBSA::SetConfiguration " (on page 44) method, this must be called prior to running an acquisition.

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

Sub SetOpticsPhaseFactor(
     Channel As Long,
     Phase As Single)
```

Parameters

`Channel`
: [in] Specifies the phase channel.

`Phase`
: [in] Specifies the phase factor.

Remarks

We recommend using BSA Flow Software to calculate and find the optical settings based on a specific optical PDA configuration. Alternatively the BSA Flow Software Script Generator can be used, please refer to section "BSA Flow Software Script Generator" (on page 15). The optical phase factor values corresponds to the output property *Phase factor P12 and P13*, which can be read in the output properties in BSA Flow Software. Please refer to the BSA Flow Software Installation and User's Guide for more information.

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 5-20: 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.

2.14.6 IBSA::GetOpticsPhaseFactor

This method gets the optical phase factor for a given phase channel earlier set by the method "IBSA::SetOpticsPhaseFactor" (on page 101).

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

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. micro-meter (deg / µm = 10^6 deg / m).

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.

2.14.7 IBSA::SetOpticsValidationBand

Specifies the optical validation band. The optical validation band is used for PDA configurations in the determination of valid particles or diameters.

```
HRESULT SetOpticsValidationBand(
     /*[in]*/ float Band);

Sub SetOpticsValidationBand(
     Band As Single)
```

Parameter

`Band`

[in] Specifies the validation band in percent.

Default

The default value is 5 %.

Remarks

We recommend using BSA Flow Software to calculate and find the optical settings based on a specific optical PDA configuration. Alternatively the BSA Flow Software Script Generator can be used, please refer to section "BSA Flow Software Script Generator" (on page 15). The optical validation band value corresponds to the output property *Phase ratio validation*, which can be read in the output properties in BSA Flow Software. Please refer to the BSA Flow Software Installation and User's Guide for more information.

2.14.8 IBSA::GetOpticsValidationBand

This method gets the optical validation band earlier set by the method "IBSA::SetOpticsValidationBand" (on page 102).

```
HRESULT GetOpticsValidationBand(
     /*[out, retval]*/ float* pBand);

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.

2.14.9 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.

```
HRESULT SetOpticsFringeDirection(
     /*[in]*/ int Direction);

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.

Remarks

We recommend using BSA Flow Software to calculate and find the optical settings based on a specific optical PDA configuration. Alternatively the BSA Flow Software Script Generator can be used, please refer to section "BSA Flow Software Script Generator" (on page 15). The optical fringe shift direction value corresponds to the output property *Fringe direction*, which can be read in the output properties in BSA Flow Software. Please refer to the BSA Flow Software Installation and User's Guide for more information.

2.14.10 IBSA::GetOpticsFringeDirection

This method gets the optical fringe direction earlier set by the method "IBSA::SetOpticsFringeDirection" (on page 103).

```
HRESULT GetOpticsFringeDirection(
     /*[out, retval]*/ float* pDirection);

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.

2.15 Advanced Settings

2.15.1 IBSA::SetAnodeCurrentLimit

To protect the photo-multiplier tube e.g. when approaching walls, the BSA3 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.

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

Sub SetAnodeCurrentLimit(
     Channel As Long,
     Limit As Long)
```

Parameters

`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.

2.15.2 IBSA::GetAnodeCurrentLimit

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

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

Function GetAnodeCurrentLimit(
     Channel As Long
) As Long
```

Parameters

`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 1.500 µA.

2.15.3 IBSA::SetTransientData

Only applicable to the BSA3 Processor.

Enables acquisition of transient data.

```
HRESULT SetTransientData(
     /*[in]*/ int Channel,
     /*[in]*/ int Trans);

Sub SetTransientData(
     Channel As Long,
     Trans As Single)
```

Parameters

`Channel`
: [in] Specifies the channel.

`Trans`
: [in] Enables or disables transient data acquisition. `Trans` can be set to 1 to enable transient data acquisition and 0 to disable it..

2.15.4 IBSA::GetTransientData

This method gets the value set by the "IBSA::SetTransientData" (on page 105) method.

```
HRESULT GetTransientData(
     /*[in]*/ int Channel,
     /*[out, retval]*/ int* pTrans);

Function GetTransientData(
     Channel As Long
) As Long
```

Parameters

`Channel`
: [in] Specifies the channel.

`pTrans`

[out, retval] A pointer to an integer indicating if transient data acquisition is enabled for the specified channel. On return `pTrans` is either 1 indicating that transient data acquisition is enabled or 0.

Default

The default setting is that transient data acquisition is disabled for all channels.

2.16 Acquiring

This section describes methods for acquiring data from the BSA3 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 Java natively don't understand variant and safearray types, a build-in object VBArray extents 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 array
ReadTransitTimeData	32-bit floating point, VT_R4, float array
ReadVelocityData	32-bit floating point, VT_R4, float array
ReadDiameterData	32-bit floating point, VT_R4, float array
ReadEncoderData	32-bit integer, VT_I4, int array
ReadSyncArrivalTimeData	64-bit double precision floating point, VT_R8, double array
ReadSyncData	32-bit integer, VT_I4, int array
ReadStatValidation	32-bit floating point, VT_R4, float value
ReadStatSphericalValidation	32-bit floating point, VT_R4, float value

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]

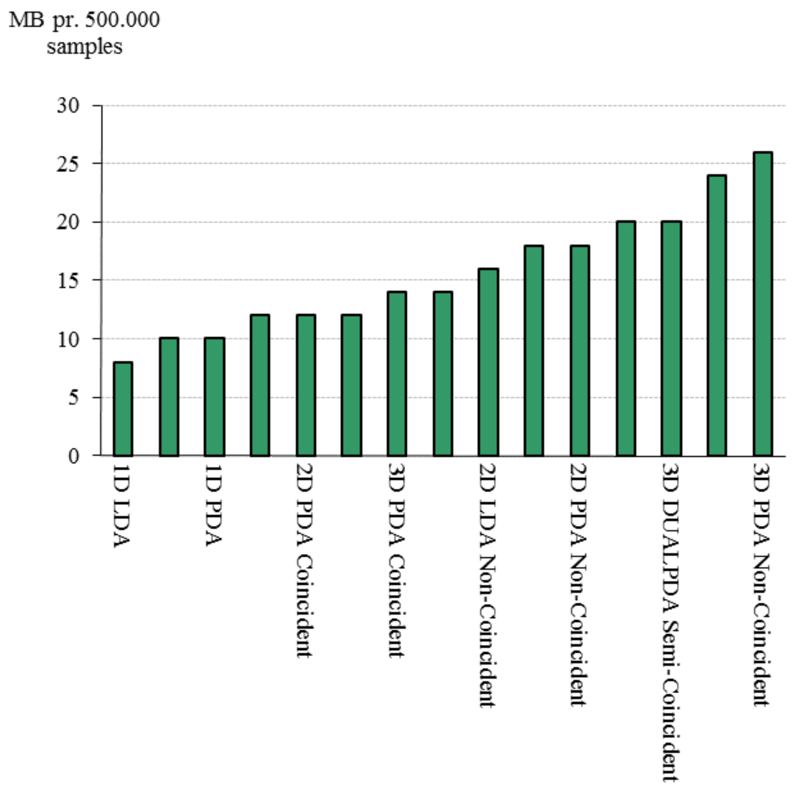
Variant To Data

2.16.1 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 BSA3 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 "IBSA::SetMaxSamples" (on page 75) and "IBSA::SetMaxAcqTime" (on page 76) methods.

The following graph shows the memory consumption by the driver for different BSA3 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 "IBSA::PreAllocateData" (on page 111) 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 "IBSA::PreAllocateData" (on page 111) method.

2.16.2 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 "IBSA::SetMaxSamples" (on page 75) method or the duration of the acquisition is equal to the maximum acquisition time set by calling "IBSA::SetMaxAcqTime" (on page 76). Data can after the acquisition is ended be read using the data access functions "IBSA::ReadArrivalTimeData" (on page 113), "IBSA::ReadVelocityData" (on page 114) etc.

```
RESULT StartAcq(
     /*[in]*/ int Wait);

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 "IBSA::ReadDataLength" (on page 112) 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 "IBSA::StopAcq" (on page 109) method.

2.16.3 IBSA::StopAcq

Forces an acquisition to abort before it ends. Normally the acquisition ends when the maximum number of samples set by the "IBSA::SetMaxSamples" (on page 75) or the maximum acquisition time set by "IBSA::SetMaxAcqTime" (on page 76) is reached

```
HRESULT StopAcq();

Sub StopAcq()
```

Parameters

This method has no parameters.

Remarks

This method only applies to an acquisition that is started with Wait parameter in the "IBSA::StartAcq" (on page 109) method set to false.

When the acquisition is ended the "IBSAEvents::onAcqDone" (on page 122) event is fired.

2.16.4 IBSA::GetAcqState

Call GetAcqState in order to determine if an acquisition is currently running or stopped.

```
HRESULT GetAcqState(
     /*[out, retval]*/ int* pAcq);

Function GetAcqState(
) As Long
```

Parameters

`pAcq`

[out, retval] A pointer that holds the acquisition state. `pAcq` is set to the value 1 if an acquisition is currently running, and 0 if the acquisition is stopped.

[HTML VBScript]

```
<script language="vbscript">Set MyBSA = CreateObject("BSA.Ctrl")

' start acquisition'
MyBSA.StartAcq False

' this loop will run until the acquisition is done
do
  ' query the acquisition state'
  dim statusAcq
  statusAcq = MyBSA.GetAcqState()
  ' if no acquisition running exit the loop'
  if statusAcq=0 Then
    exit do
  end if
loop until False

</script>
```

Remarks

The GetAcqState method is useful to test whether an acquisition is running or not, see the "IBSA::StartAcq" (on page 109) method.

Alternatively the "IBSAEvents::onAcqDone" (on page 122) event can be used to get the same information, in an asynchronous way.

2.16.5 IBSA::SetBlockSize

The BSA3 Processor sends data in blocks to the driver. The SetBlockSize method can be used to adjust the maximum size of the data blocks send to the driver during an acquisition. If the number is low, more blocks will be send to the driver, and if the size is large, less blocks will be send.

```
HRESULT SetBlockSize(
     /*[in]*/ int Size);

Sub SetBlockSize(
     Size As Long)
```

Parameters

`Size`

[in] Specifies the block size. The value can be specified in the range 1 to 1024.

Remarks

In most situations the default value of the block size is adequate, and it is only recommended to call SetBlockSize if the data update is either too slow or fast.

Note
Block size is not the same as number of data samples, since a block from the BSA3 Processor contains more information than data samples.

Defaults

The default block size is 512.

2.16.6 IBSA::PreAllocateData

During normal use all possible data is received from the 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.

```
HRESULT PreAllocateData(
     /*[in]*/ int Flags);

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 5-21" (on page 111)

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 5-21: Pre-allocation Data Flags

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

Default

By default all possible data is received from the BSA3 Processor bsaDATAALL = 0xFFF.

2.16.7 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 "IBSA::SetMaxSamples" (on page 75) and "IBSA::SetMaxAcqTime" (on page 76) methods control the data-collection for each group, the ReadDataLength will return the actual number of acquired samples limited by these methods. The data length cannot be larger than the value specified in "IBSA::SetMaxSamples" (on page 75) for the group.

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

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
```

2.16.8 IBSA::ReadArrivalTimeData

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

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

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.

2.16.9 IBSA::ReadTransitTimeData

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

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

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 cannel is calculated as the mean transit time for all the LDA channels in the group.

2.16.10 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.

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

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.

2.16.11 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 BSA3 Processor is set to run a PDA measurement, this function will fail when called after a LDA measurement.

```
HRESULT ReadDiameterData(
     /*[out, retval]*/ VARIANT* pData);

Function ReadDiameterData(
) As Variant
```

Parameters

`pData`
: [out, retval] A pointer to a VARIANT data that is to hold the diameters.

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. Diameter values will be returned in micrometers (µm = 10^{-6} m).

2.16.12 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 "IBSA::SetEncoder " (on page 66) and "IBSA::SetEncoderReset " (on page 65); the encoder counts can be found calling this method.

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

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.

2.16.13 IBSA::ReadSyncDataLength

When synchronization is enabled with the "IBSA::SetSync1" (on page 61) and/or "IBSA::SetSync2" (on page 63) methods, synchronization data will be acquired along with LDA data. This function returns the number of acquired synchronization samples.

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

Function ReadSyncDataLength(
) As Long
```

Parameters

`pLength`

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

Remarks

This number is **not** the same as the number of acquired data found with the "IBSA::ReadDataLength" (on page 112) method. Synchronization data is acquires 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 "IBSA::ReadSyncArrivalTimeData" (on page 116) and "IBSA::ReadSyncData" (on page 116) subsequent to a call to this method will return the synchronization data events.

2.16.14 IBSA::ReadSyncArrivalTimeData

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

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

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.

2.16.15 IBSA::ReadSyncData

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

```
HRESULT ReadSyncData(
     /*[out, retval]*/ VARIANT* pData);

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 5-22: Synchronization Event Values

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

2.16.16 IBSA::ReadStatValidation

When an acquisition has ended, calling this method will return the data validation rate for each group during the measurement.

```
HRESULT ReadStatValidation(
     /*[in]*/ int Group,
     /*[in]*/ int Average,
     /*[out, retval]*/ float* pVal);

Function ReadStatValidation(
     Group As Long,
     Average As Long
) As Single
```

Parameters

`Group`
: [in] Specifies the group.

`Average`
: [in] Specifies whether to read the average value or the instant value of the statistical validation.

`pVal`
: [out, retval] A pointer to a float that contains the validation rate in percent for the group.

Remarks

The validation rate describes how many bursts that are validated during a measurement. It is calculated as a statistical average over the full measurement period. If the validation fluctuates, or changes from the beginning to the end of the measurement, this will affect the validation rate. E.g. a validation rate of 10 % means, that 90 % of the burst signals received is invalidated and therefore discarded from the data.

The validation can be read in two different ways. During asynchronous acquisition (or at the end of a synchronous acquisition) the instant value of the validation can be read. Please note that this is an instant value that can change during the measurement. Alternatively the average value of the validation of the whole measurement can be read. For steady flows this would give a more true value of the validation of the measurement. See "Table 5-23" (on page 117).

The reason for a low validation rate can be many. To get the highest possible validation rate it is often necessary to optimize the BSA3 Processor settings to the nature of the incoming signal. Here the System Monitor is a valuable tool for observing the signals and see how the settings affect the validation.

Table 5-23: Validation Flags

Constant	Value
bsaAVERAGE	1
bsaINSTANT	0

2.16.17 IBSA::ReadStatSphericalValidation

When an acquisition has ended, calling this method will return the spherical validation rate for the PDA measurement. This is only available if the BSA3 Processor is set to run a PDA measurement, and will fail when called after a LDA measurement.

```
HRESULT ReadStatSphericalValidation(
     /*[in]*/ int Average,
     /*[out, retval]*/ float* pVal);

Function ReadStatSphericalValidation(
     Average As Long
) As Single
```

Parameters

`Average`

[in] Specifies whether to read the average value or the instant value of the statistical spherical validation.

`pVal`

[out, retval] A pointer to a float that contains the spherical validation rate in percent for the PDA measurement.

Remarks

The spherical validation rate describes how many particles that are validated during a PDA measurement. It is calculated as a statistical average over the full measurement period. If the validation fluctuates, or changes from the beginning to the end of the measurement, this will affect the spherical validation rate. E.g. a spherical validation rate of 10 % means, that 90 % of the particle signals received is invalidated and therefore discarded from the diameter data.

The validation can be read in two different ways, please refer to "IBSA::ReadStatValidation" (on page 117) for explanation. See "Table 5-23" (on page 117).

The spherical validation is dependent on a correct alignment of the PDA transmitter and receiver optics. During the alignment process the Phase plot available in BSA Flow Software can be used.

The "IBSA::SetOpticsValidationBand" (on page 102) method can be used to adjust the validation band in the processor.

2.17 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 component supports the following events.

2.17.1 BSA.Ctrl Component Events

Connecting Methods	Description
onStatus	Notifies that the status has changed.
onState	Notifies that the state has changed.
onAcqDone	Notifies when an acquisition is done.
onCalDone	Notifies when a calibration is done.
onData	Notifies when acquisition data is available.

2.17.2 Receiving Events

The programming language or environment used must be able to receive asynchronous events from ActiveX/COM objects. This is normally part of supporting ActiveX/COM bojects, but the way it is set up is very different between programming language and environments.

In the following we will use the `onAcqDone` event as example, but handling the other events will be done is a similar way. The onAcqDone event is called when an acquisition is started and ended.

In VBScript there is a special method called `WScript.ConnectObject`. This method attached an event to a prefix argument to an event method. It is set up like this:

[VBScript]

```
dim MyBSA
' create the BSA3.Ctrl object (BSA3 Processor)'
set MyBSA = createobject("BSA3.Ctrl")
' attach a prefix argument to the events'
WScript.ConnectObject MyBSA,"MyEvent_"
' here we do something to trigger some events
' connect, configure and start acquisition
' each event function can be defines as a sub routine'
sub MyEvent_onAcqDone(nDone)
    WScript.Echo "My AcqDone event. " & nDone
end sub
' clean up'
set MyBSA = nothing
```

In HTML this prefix argument to the event function is based on the id of the object, like this:

HTML

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
    <title>BSA Processor Events</title>
    <object classid="clsid:83E16B05-CDAD-11D0-89E3-0000C0C27240" id="MyBSA" >
    </object>
  </head>
  <body>
```

```
    <!--
      here we do something to trigger some events
      connect, configure and start acquisition
      :
    -->

    <script type="text/vbscript" language="vbscript">
      Sub MyBSA_onAcqDone(done)
        MsgBox "Fire onAcqDone event."
      End Sub
    </script>
  </body>
</html>
```

In MATLAB it is important to register the ActiveX/COM events to use. Based on this registration the event is mapped to a MATLAB function.

MATLAB

```
global MyBSA;
% create the BSA3.Ctrl object (BSA3 Processor)
MyBSA = actxserver('BSA3.Ctrl');
% register the onAcqDone event
registerevent(MyBSA, {'onAcqDone', 'onacqdone'});


% here we do something to trigger some events
% connect, configure and start acquisition
% :

function onacqdone(varargin)  %varargin holds a variable number of arguments
  disp(varargin)   %display the arguments
  disp('BSA measurement is done!')   %display the message
```

For more information about handling events, please refer to the samples.

2.17.3 IBSAEvents::onStatus

The onStatus event is triggered when the status of the BSA3 Processor has changed.

```
HRESULT onStatus(
     /*[in]*/ int Status);

Event onStatus(
     Status As Long);
```

Parameters

`Status`

[in] A flag indicating the new status of the BSA3 Processor. Refer to "Table 5-24" (on page 120) for definitions.

Table 5-24: BSA3 Processor Status Flags

Value	Status Flag
0	Disabled.
1	Monitor.
2	Acquisition.
3	Level.
4	Calibration.
5	Test2.
6	Test1.
7	Error.
8	Emission.
9	Manual.
-1	An unknown status received.

Remarks

The status flag must be seen in combination with the state flag received in the "IBSAEvent-s::onState" (on page 121) event. E.g.. the Calibration status can be combined with the "BSA3 Processor State Flags" (on page 121) to find out if the calibration has failed.

2.17.4 IBSAEvents::onState

Only applicable to the BSA3 Processor.

The onState event is triggered when the state of the BSA3 Processor has changed.

```
HRESULT onState(
     /*[in]*/ int State);

Event onState(
     State As Long);
```

Parameters

`State`

[in] A flag indicating the new state of the BSA3 Processor. Refer to "Table 5-25" (on page 121) for definitions.

Table 5-25: BSA3 Processor State Flags

Value	State Flag
0	Idle.
1	Applying profile.
2	Adjust settings.
3	Startup delay.
4	Enabling HDL.
5	Enabling start signal.
6	Running.
7	Stopping.
8	Disabling HDL.
-1	An unknown state received.

Remarks

The state flag must be seen in combination with the status flag received in the "IBSAEvent-s::onStatus" (on page 120) event. E.g.. the Running state can be combined with the "BSA3 Pro-cessor Status Flags" (on page 120) to find out if it is an acquisition is running or monitor running.

2.17.5 IBSAEvents::onAcqDone

The onAcqDone event is triggered when a measurement is started and ended, typically triggered by a previous call to the "IBSA::StartAcq" (on page 109) and "IBSA::StopAcq" (on page 109) methods.

```
HRESULT onAcqDone(
     /*[in]*/ int Done);

Event onAcqDone(
     Done As Long);
```

Parameters

`Done`

[in] A flag indicating if the measurement is started or ended. Refer to "Table 5-26" (on page 122) for definitions.

Table 5-26: Acquisition Done Flags

Value	Synchronization Event
0	The acquisition is started after calling "IBSA::StartAcq" (on page 109).
1	The acquisition stopped normally, on maximum number of samples or maximum acquisition time, specified by the functions "IBSA::SetMaxSamples" (on page 75) and "IBSA::SetMaxAcqTime" (on page 76).
2	The acquisition was aborted by calling "IBSA::StopAcq" (on page 109).
-1	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.
-2	An unknown error occurred during acquisition.
-3	An unknown error occurred during acquisition.

Remarks

This event is a helper event to look at the measurement progress specifically. Alternatively the more general events "IBSAEvents::onStatus" (on page 120) and "IBSAEvents::onState" (on page 121) can be used in combination to get the same information. Use the "IBSAEvent-s::onStatus" (on page 120) to see if the measurement succeeded or failed.

2.17.6 IBSAEvents::onCalDone

The onCalDone event is triggered when a PDA calibration is started and ended, typically triggered by a previous call to the "IBSA::CalibratePDA" (on page 97) method, or when dependent properties are changed.

```
HRESULT onCalDone(
     /*[in]*/ int Done);

Event onCalDone(
     Done As Long);
```

Parameters

`Done`

[in] A flag indicating if the PDA calibration is started or ended. Refer to "Table 5-27" (on page 123) for definitions.

Table 5-27: Calibration Done Flags

Value	Done Flag
0	The PDA calibration is started.
1	The PDA calibration is ended.

Remarks

This event is a helper event to look at the PDA calibration specifically. Alternatively the more general events "IBSAEvents::onStatus" (on page 120) and "IBSAEvents::onState" (on page

121) can be used in combination to get the same information. Use the "IBSAEvents::onStatus" (on page 120) to see if the calibration succeeded or failed.

There can be many reasons for the PDA calibration to fail. The most likely reason is an invalid combination of properties or defect or misaligned optics. Please use BSA Flow Software to investigate the problem.

2.17.7 IBSAEvents::onData

The onData event is triggered every time data is received by the driver. This means, that it is only be called during an acquisition, as the result of a previous call to the "IBSA::StartAcq" (on page 109) method

```
HRESULT onData(
     /*[in]*/ int Flag,
     /*[in]*/ int Group);

Event onCalDone(
     Flag As Long,
     Group As Long);
```

Parameters

`Flag`

[in] A flag indicating what kind of data is received. Refer to "Table 5-28" (on page 124) for definitions.

`Group`

[in] Specifies which data groups the data belongs to.

Table 5-28: Data Type Flags

Value	Synchronization Event
0	Indicates that the data is velocity or diameter data. The Group parameter indicates which group the data belongs to.
1	Indicates that the data is Sync data. The Group parameter is not used for Sync data.

[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 onData'
sub MyBSA_onData(flag,group)
  WScript.Echo "Data event received."

  dim numSamples
  if flag=0 then
```

```
    ' read number of samples for group'
    numSamples = MyBSA.ReadDataLength(group)
  elseif flag=1 then
    ' read number of sync. samples'
numSamples = MyBSA.ReadSyncDataLength()
  end if

  WScript.Echo numSamples & " samples."
end sub
```

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.

[HTML JScript]

```
<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>
```

2.18 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.

2.18.1 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(err) {
     WScript.Echo("MyError: " + err.description);
}
```

In HTML and JavaScript exception handling is also carried out using try-catch statements. Here we assume a `<div id="myDIV"></div>` tag is already made in the body of the HTML to include the status message.

[HTML]

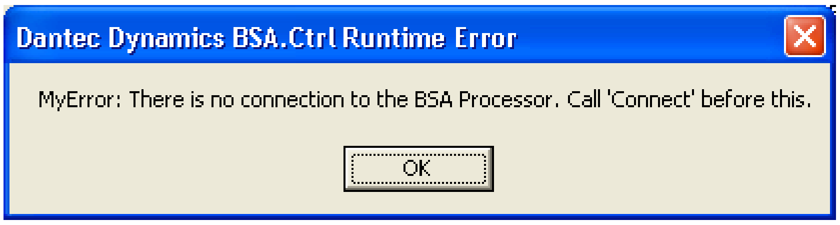
```
var MyBSA = new ActiveXObject("BSA.Ctrl");
// error handling in jscript is done through
// try-catch exception handling
try {
     document.getElementById("MyDIV").innerHTML = "Connecting";
     MyBSA.ConnectEx("10.10.100.100","0.0.0.0",true);
}
catch(err) {
     document.getElementById("MyDIV").innerHTML = err.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.


Dantec Dynamics BSA.Ctrl Runtime Error

MyError: There is no connection to the BSA Processor. Call 'Connect' before this.

Figure 5-12: 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.

2.18.2 Processor Errors

BSA3 Processor status, state and errors are retrieved through events, please see "Event Handling" (on page 118).

2.19 BSA Control 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
bsaBNC4	4
bsaDSUB1	5
bsaDSUB2	6
bsaDSUB3	7
bsa1DLDA	0
bsa2DLDACOINC	1
bsa3DLDACOINC	2
bsa2DLDANONCOINC	3
bsa3DLDANONCOINC	4
bsa3DLDASEMICOINC	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
bsaLOW	0
bsaHIGH	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
bsaMEASUREMENTRUNNING	7
bsaUNUSED	0
bsaREC16	16
bsaREC32	32
bsaREC64	64
bsaREC128	128
bsaREC256	256
bsaREC512	512

bsaREC1024	1024
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
bsaLEVELOFF	0
bsaLEVEL3	3
bsaLEVEL4	4
bsaLEVEL5	5
bsaLEVEL6	6
bsaLEVEL7	7
bsaLEVEL8	8

bsaINSTANT	0
bsaAVERAGE	1
bsaLDA	0
bsaPDA	1
bsaAUTOCALIBRATE	0
bsaALWAYSCALIBRATE	1
bsaNEVERCALIBRATE	2
bsaDATAAT	0x001
bsaDATATT	0x002
bsaDATAVEL	0x004
bsaDATAENC	0x008
bsaDATADIA	0x020
bsaDATASYNC	0x040
bsaDATAALL	0xFFF

3 BSA System Monitor Control Function Reference

The purpose of this chapter is to give an understanding of the functions in the IBSASysMon interface for the BSA3 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 BSA3 Processor. However the SysMon.Ctrl requires that the System Monitor is activated before displaying any information. To activate the System Monitor call the BSA.Ctrl ActivateSysMon method or start the System Monitor in the BSA Flow Software. Please refer to the "HTML Example" (on page 147) section.

3.1 Connecting

Connecting Methods	Description
Connect	Connects the System Monitor to the BSA3 Processor.
ConnectEx	Connects the System Monitor to the BSA3 Processor using a network IP-address and a specific NIC address.
Disconnect	Disconnects the System Monitor from the BSA3 Processor.
GetConnectState	Detects if the System Monitor is connected to the BSA3 Processor or not.

3.2 Advanced

Advanced Methods	Description
ReadData	Read data value(s) from the System Monitor.
SetStyles	Specify the look and feel of the System Monitor .

3.3 Using the BSA System Monitor Control

The BSA System Monitor Control is a valuable tool for monitoring the current online condition of the signal. You will instantaneously have an indication of the how your BSA3 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.

3.3.1 Resizing the BSA System Monitor Control

The BSA System Monitor Control must be embedded into a OCX/ActiveX containers or form; in other words the BSA System Monitor Control, needs a host window to live in. The System Monitor window can automatically be resized when the size of the host window is changed.

In some programing languages like C++/C#/Visual Basic etc. the host window will be a Windows view, form or dialog, and the resizing will be controlled directly from within the code.

In Windows generally the `WM_ONSIZE` message handles resizing and the WIN32 call `::SetWindowPos` is used to specify a new size of a window.

In other environments like HTML, MATLAB, LabVIEW etc. the resizing is handled by the standard OCX/ActiveX hosting interface rules, which means that is basically is handled automatically if the BSA System Monitor Control is embedded correctly.

Often some tricks needs to be applied to the embedding of the System Monitor window.

Below is described how to ensure proper resizing in MATLAB by setting the property `units` to `normalized` for the figure window. This is MATLAB specific and maybe changed in other versions of MATLAB why this is up to the implementer to find out.

```
% create the BSA3.SysMon.Ctrl object in a figure window
f = figure('pos',[300 300 500 500]);
global MySysMon;
[MySysMon, WinSysMon] = actxcontrol('BSA3.SysMon.Ctrl', [0 0 500 500], f);
% ensure resizing of the BSA3.SysMon.Ctrl in the figure window
set(WinSysMon, 'units', 'normalized');
```

In a web browser the System Monitor window can be placed inside a table, or some other HTML container, to control the resizing. In modern web browsers CSS or HTML5 should be used to ensure the correct resizing.

```
<table width="100%" height="100%">
<tr>
<td>
<object classid="clsid:83E16B06-CDAD-11D0-89E3-0000C0C27240" id="MySysMon" width-
h="100%" height="100%" border="0">
</object>
</td>
</tr>
</table>
```

3.4 Connecting

This section describes the methods that are involved in connecting the System Monitor component to the processor. While connected; the System Monitor component will display online information from the processor.

3.4.1 IBSASysMon::Connect

This method connects the System Monitor component to the BSA3 Processor through the IP-address of the processor. The System Monitor must be configured by calling "IBSA::SetConfiguration " (on page 44) and activated by calling "IBSA::ActivateSysMon" (on page 51) before showing any data.

```
HRESULT Connect(
     /*[in]*/ BSTR IPAddress);

Sub Connect(
     IPAddress As String)
```

Parameters

`IPAddress`

[in] The IP-address of the BSA3 Processor as it is seen on the network. The syntax consists of a common dotted octet IP-address, for example 10.10.100.130.

Remarks

It is possible to have multiple instances of the System Monitor component connected at the same time.

Normally the BSA3 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 BSA3 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.

3.4.2 IBSASysMon::ConnectEx

An alternative way to connect the System Monitor component to the BSA3 Processor is to use the ConnectEx, which allows for targeting a specific Network or Network Interface Card (NIC). This is especially useful if the PC have multiple NICs installed.

Please see the "IBSASysMon::Connect" (on page 135) method for general information about connecting the System Monitor component.

```
HRESULT ConnectEx(
     /*[in]*/ BSTR IPAddress,
     /*[in]*/ BSTR NICAddress);

Sub ConnectEx(
     IPAddress As String,
     NICAddress As String)
```

Parameters

`IPAddress`

[in] The IP-address of the BSA3 Processor as it is seen on the network. The syntax consists of a common dotted octet IP-address, for example 10.10.100.130.

`NICAddress`

[in] The IP-address of the NIC in the PC. The syntax consists of a common dotted octet IP-address, for example 10.10.1.1.

Remarks

If you experience problems connecting to the BSA3 Processor using the "IBSASysMon::Connect" (on page 135) method, the ConnectEx can be used to target a specific network.

3.4.3 IBSASysMon::Disconnect

Calling this method disconnects the System Monitor component from the processor. Calling the "IBSASysMon::Connect" (on page 135) method will connect to the System Monitor.

```
HRESULT Disconnect();

Sub Disconnect()
```

Parameters

This method has no parameters.

Remarks

Disconnecting then System Monitor component will stop the System Monitor for receiving data from the processor. The System Monitor component will be removed from the processors list of listening System Monitor clients.

3.4.4 IBSASysMon::GetConnectState

In order to determine the connection state of the System Monitor control, you can call the GetConnectionState method. It is recommended to call this function after a call to Connect to ensure that the connection is established.

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

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 System Monitor control, and 0 if there is no connection.

Remarks

The GetConnectState method is useful to test a connection, see the "IBSASysMon::Connect" (on page 135) method.

3.5 Advanced Settings

3.5.1 IBSASysMon::ReadData

This method is used to read data from the System Monitor control. The purpose is to get the data displayed in the System Monitor as numbers instead of bars and graphs.

The data in the System Monitor comes directly from the processor, together with the setup and naming of the data bars and graphs. Therefore it is not known up front what data is available in the System Monitor and the name of the data must be identified and read from the System Monitor.

```
HRESULT ReadData(
     /*[in]*/ BSTR Name,
     /*[in]*/ int Instance,
     /*[out, retval]*/ VARIANT *pData);

Sub ReadData(
     Name As String,
     Instance As Long
) as Variant
```

Parameters

`Name`

[in] Specifies the name of the data value(s) to return. The name must match the exact name displayed in the System Monitor.

`Instance`

[in] Specifies the instance of the data value(s). This often represents either the channel or group index of the data.

`pData`

[out, retval] A pointer to a VARIANT data that is to hold the data value(s).

Remarks

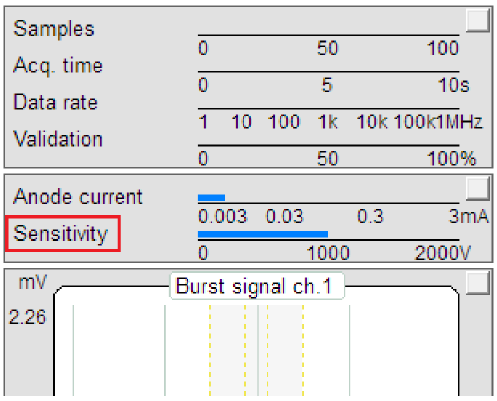
The System Monitor control must be running for the ReadData method to return values. This also means, that it is only possible to call the ReadData method if the BSA.SysMon.Ctrl is embedded into a COM container or another window.

To use this method it is necessary to know the name and instance of the data value to read from the System Monitor. Both can be found looking at the System Monitor.

The `pData` buffer returned is a VARIANT. Depending on the type of data value it can be a single value represented by either a VT_I1 byte, a VT_I4 integer, a VT_R4 32-bit single-precision floating point or a VT_R8 64-bit double-precision floating point.

If the type of the data value is an array the pData buffer will return a VARIANT of the type VT_ARRAY | VT_VARIANT, where each element is a VARIANT of the type of data values described above.

This example shows how to locate a data value name, instance and read the data value.



Locate the name, in this case Sensitivity and note the instance of the data value. In this case Sensitivity is part of the channel properties, and since there is only one bar it corresponds to channel 1, meaning instance number 0. We know that it will return a single value as a floating point value.

[c++]

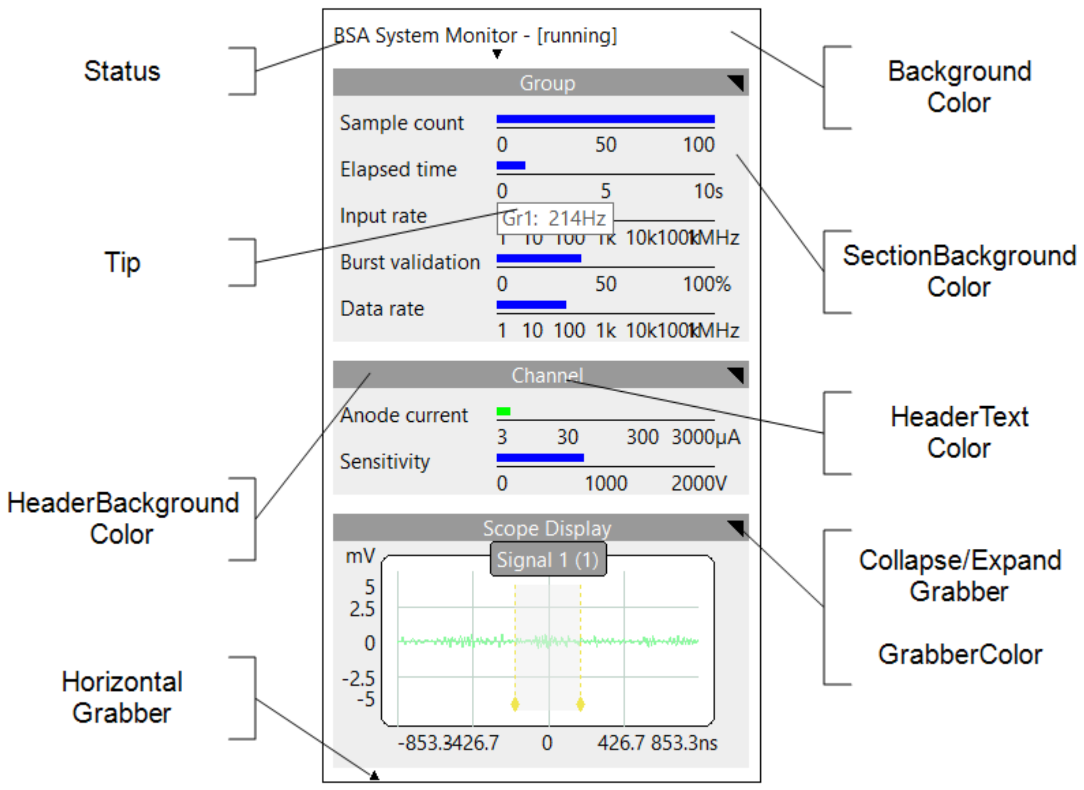
```
_variant_t vReadData;
vReadData = m_MySysMon.ReadData("Sensitivity",0);
if (vReadData.vt!=VT_EMPTY)
{
  if (vReadData.vt&VT_R4)
  {
     float fDataValue = vReadData.fltVal;
  }
}
```

[c#]

```
object obj1 = axSysMon1.ReadData("Acq. time", 0);
if (obj1 != null)
{
   float floatVal = (float)obj1;
}

object obj2 = axSysMon1.ReadData("Burst signal ch.1", 0);
if (obj2 != null)
{
   object[] objArray = obj2 as object[];
   if (objArray != null)
   {
      int nLen = objArray.Length;
      if (nLen > 0)
      {
         sbyte byteVal = (sbyte)objArray[0];
      }
   }
}
```

3.5.2 IBSASysMon::SetStyles

You can change the look and functionality of the System Monitor by specifying the styles. This way it is possible to design the System Monitor to fit better into custom applications.



```
HRESULT SetStyles(
     /*[in]*/ int Flags,
     /*[in]*/ BSTR BackgroundColor,
     /*[in]*/ BSTR SectionBackgroundColor,
     /*[in]*/ BSTR HeaderBackgroundColor,
     /*[in]*/ BSTR TextColor,
     /*[in]*/ BSTR HeaderTextColor,
     /*[in]*/ BSTR GrabberColor);

Sub SetStyles(
     Flags As Long
     BackgroundColor As String,
     SectionBackgroundColor As String,
     HeaderBackgroundColor As String,
     TextColor As String,
     HeaderTextColor As String,
     GrabberColor As String
) as Variant
```

Parameters

`Flags`

[in] Specifies the style flags. The style flag defines if element in the System Monitor is displayed or not. Refer to "Table 5-29" (on page 140) for possible combinations.

`BackgroundColor`

[in] The background color of the System Monitor area. Default color white #FFFFFF.

`SectionBackgroundColor`

[in] The background color of each of the sections; Group, Channels etc. Default color white #FFFFFF.

`HeaderBackgroundColor`

[in] The background color of the section headers. Default color white #FFFFFF.

`TextColor`
: [in] The color of status and plot text. Default color black #000000.

`HeaderTextColor`
: [in] The color of the section header text. Default color black #000000.

`GrabberColor`
: [in] The color of the horizontal grabber and the section collapse/expand grabber. Default color black #000000.

Remarks

Different element of the System Monitor can be display or hidden, using the following constants:

Table 5-29: System Monitor Style Flag Values

Value	Synchronization Event
0x00	Default. Status text and tip window are displayed, and horizontal and collapse/expand grabbers are hidden.
0x01	Hide Status. This will hide the status text.
0x02	Show Grabbers. This will show the horizontal grabbers.
0x04	Allow Collapse. This will show the collapse/expand grabbers for sections.
0x08	Hide Tip Window. This will prevent the tip window from appearing.

By default the System Monitor have a flat modern look resembling the look of the system Monitor in the BSA Flow Software, which can be expressed like this:

[c++]

```
m_MySysMon.SetStyles(0x00, "#FFFFFF", "#FFFFFF", "#FFFFFF", "#000000", "#000000",
"#000000");
```

The string representing the colors is in hexadecimal (HEX) format. This hexadecimal value must be in the form: #RRGGBB, where RR (red), GG (green) and BB (blue) are hexadecimal values between 00 and FF (same as decimal 0-255). In the above default example #FFFFFF represents while, and #000000 represents black.

The example shown in the beginning of this section can be defined as:

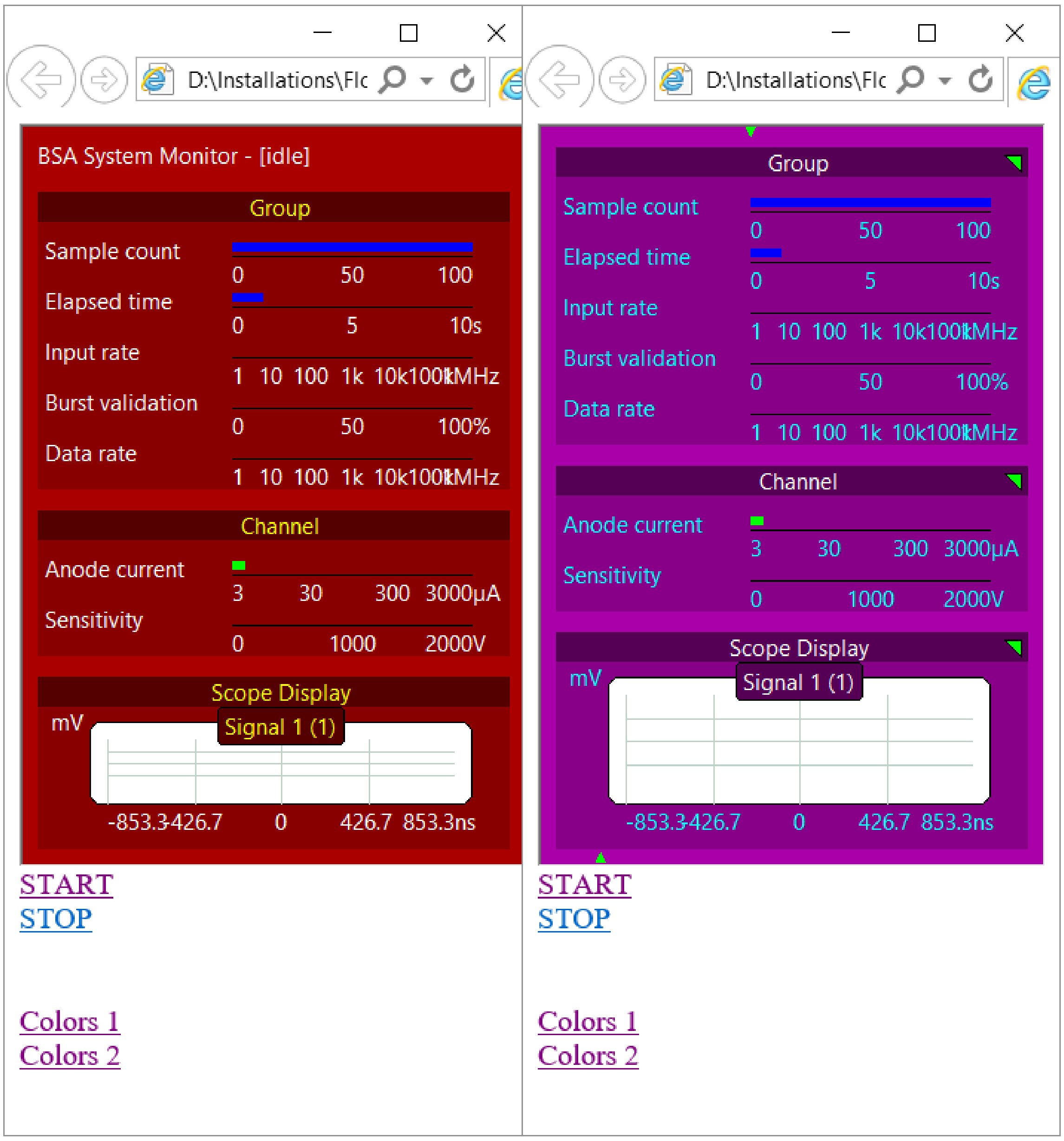
[c++]

```

m_MySysMon.SetStyles(0x06, "#FFFFFF", "#EEEEEE", "#999999", "#000000", "#FFFFFF",
"#000000");

```

More "special" looks can be defined as in this example:

[HTML]

```
<object classid="clsid:83E16B06-CDAD-11D0-89E3-0000C0C27240" id="MySysMon" width-
h="100%" height="400" border="0">
</object>

<br /><a href="javascript:MySysMon.Connect('10.10.100.130')">START</a>
<br /><a href="javascript:MySysMon.Disconnect()">STOP</a>

<br /><br />
<br /><a href="javascript:MySysMon.SetStyles
(0,'#AA0000','#880000','#550000','#FFFFFF','#FFFF00','#006600')">Colors 1</a>
<br /><a href="javascript:MySysMon.SetStyles
(7,'#AA00AA','#880088','#550055','#00FFFF','#FFFFFF','#00FF00')">Colors 2</a>
```



3.6 BSA System Monitor Control Constants

For convenience a number of constants are defined. These constants are located in the BSA.SysMon.Ctrl module and can be accessed by languages that import the definitions.

Constant	Value
sysmonNone	0x00
sysmonHideStatus	0x01
sysmonShowGrabbers	0x02
sysmonAllowCollaps	0x04
sysmonHideTip	0x08

4 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 DVD in its full contents. Examples can also be downloaded from the Dantec Dynamics web site.

4.1 How to Get Started with the Examples

When the driver is installed the examples can be found in the sub-folder called `Samples`. The typical path is:

```
<root>\Program Files (x86)\Dantec Dynamics\BSA Processor Driver\Samples
```

and for the 64-bit version:

```
<root>\Program Files\Dantec Dynamics\BSA Processor Driver\Samples
```

Before running the samples it is a good idea to know the settings of the BSA3 Processor, and to ensure that the processor is in a known configuration with a stable data rate. The way to do this is to first use BSA Flow Software to setup the connection, and to adjust the configuration and settings using the System Monitor. We recommend to set the processor in a 1D LDA Configuration and to setup the properties of the processor to measure a stable signal with a data rate around 100-1000 Hz.

4.2 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 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
```

4.3 JScript Example

[JScript]

```
//==========================================================
// 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 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 nessesery 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;
```

4.4 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 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
```

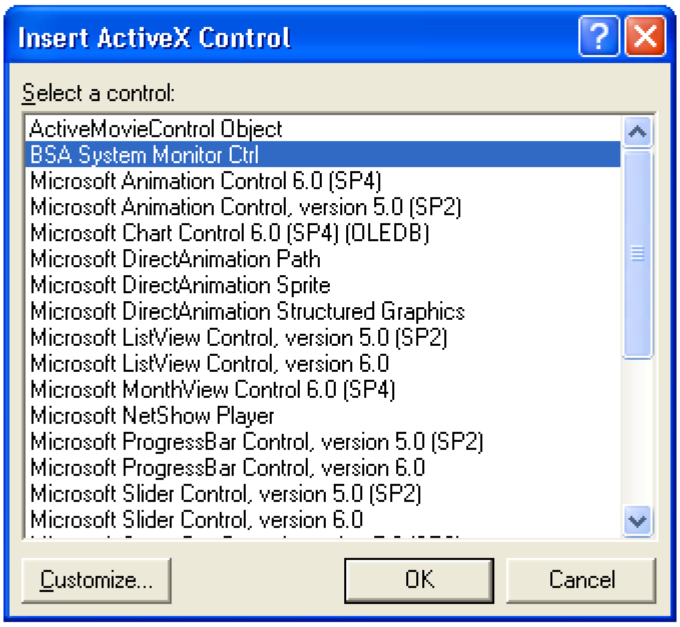
4.5 HTML Example

This example shows how to activate the System Monitor from the BSA3.Ctrl and how to display the System Monitor embedded on a web page using the BSA3.SysMon.Ctrl.

Note
This will only work in Internet Explorer, since ActiveX/COM is not allowed in other browsers. Also notice that the default security settings in the Internet Explorer needs to be changed to allow for ActiveX/COM objects to run.

The example uses a script block in VBScript to create an instance of the BSA3.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 BSA3.SysMon.Ctrl is placed in an object block, referencing the GUID of the component. The GUID of the BSA3.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:


Insert ActiveX Control
Select a control:
ActiveMovieControl Object
BSA System Monitor Ctrl
Microsoft Animation Control 6.0 (SP4)
Microsoft Animation Control, version 5.0 (SP2)
Microsoft Chart Control 6.0 (SP4) (OLEDB)
Microsoft DirectAnimation Path
Microsoft DirectAnimation Sprite
Microsoft DirectAnimation Structured Graphics
Microsoft ListView Control, version 5.0 (SP2)
Microsoft ListView Control, version 6.0
Microsoft MonthView Control 6.0 (SP4)
Microsoft NetShow Player
Microsoft ProgressBar Control, version 5.0 (SP2)
Microsoft ProgressBar Control, version 6.0
Microsoft Slider Control, version 5.0 (SP2)
Microsoft Slider Control, version 6.0
Customize...
OK
Cancel

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-
t="14">
</object>
```

The BSA3.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 VBScript]

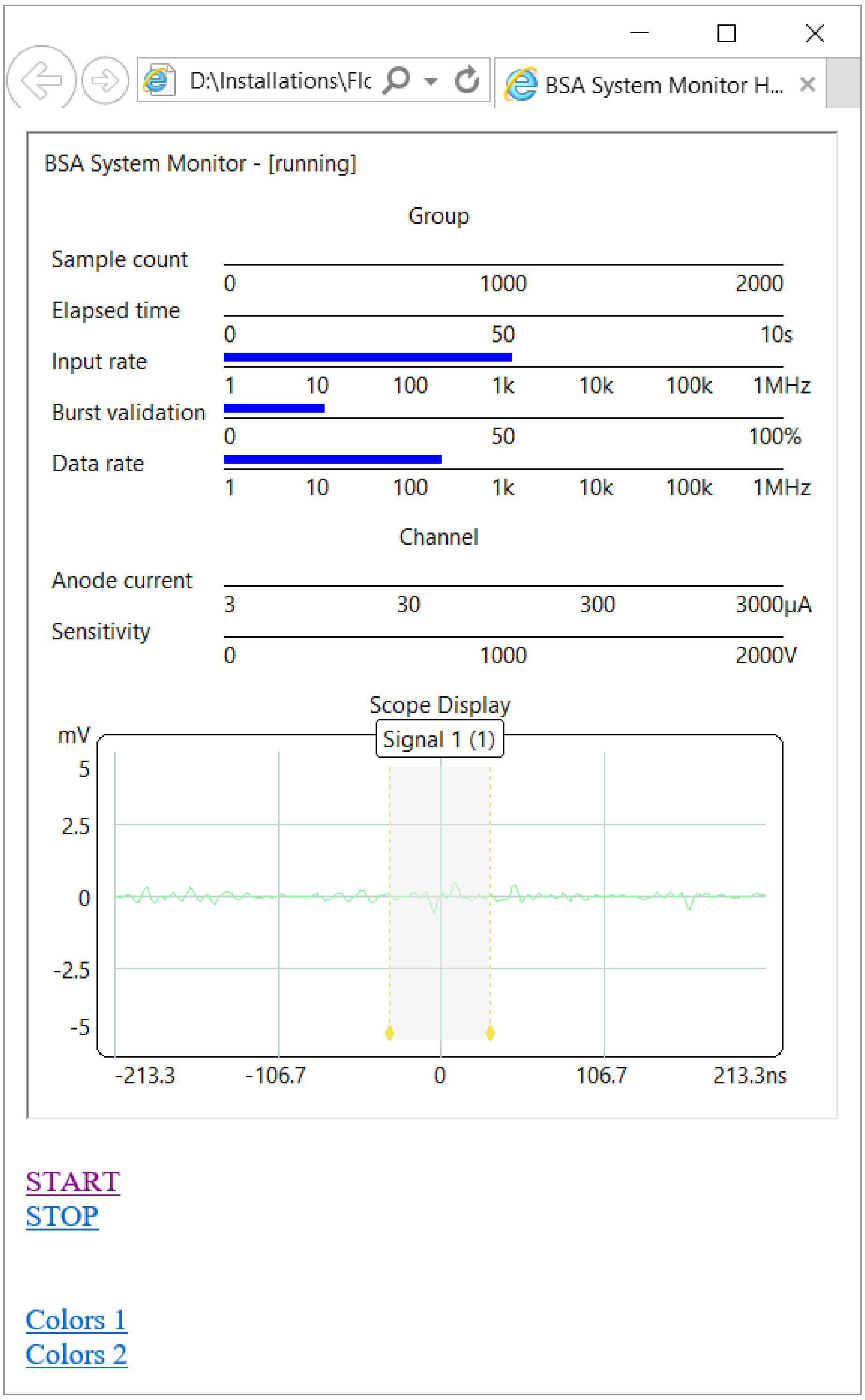
```
<!--
HTML Example
This example is made for the Dantec Dynamics Processor Driver
Copyright Dantec Dynamics A/S, 2003 All Rights Reserved.
This example will activate the BSA System Monitor using
the BSA.Ctrl and display the System Monitor using the BSA.SysMon.Ctrl
-->

<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 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>
```

[HTML JScript]

```
<--
HTML Example
This example is made for the Dantec Dynamics Processor Driver
Copyright Dantec Dynamics A/S, 2017 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
```

```
NOTE: This will only work in Internet Explorer, since ActiveX/COM is not allowed
in other browsers. Also notice that the default security settings in the Internet
Explorer needs to be changed to allow for ActiveX/COM objects to run.
-->

<html>

<head>
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252" />
<title>BSA System Monitor HTML Example</title>

<script type="text/javascript" language="jscript">

var MyBSA = new ActiveXObject("BSA3.Ctrl");
// connects to the BSA3 Processor
MyBSA.Connect("10.10.100.130", false);

// set configuration to 1D LDA
MyBSA.SetProcessor(0);
MyBSA.SetConfiguration(0);

// TODO specify special settings

// activates the system monitor
MyBSA.ActivateSysMon();

</script>

</head>

<body>

<table width="100%" height="100%">
<tr>
<td>
<object classid="clsid:83E16B06-CDAD-11D0-89E3-0000C0C27240" id="MySysMon" width-
h="100%" height="100%" border="0">
</object>

</td>
</tr>
<tr>
<td>

<br /><a href="javascript:MySysMon.Connect('10.10.100.130')">START</a>
<br /><a href="javascript:MySysMon.Disconnect()">STOP</a>

<br /><br />
<br /><a href="javascript:MySysMon.SetStyles
(0,'#AA0000','#880000','#550000','#FFFFFF','#FFFF00','#006600')">Colors 1</a>
<br /><a href="javascript:MySysMon.SetStyles
(7,'#AA00AA','#880088','#550055','#00FFFF','#FFFFFF','#00FF00')">Colors 2</a>
```

```
</td>
</tr>
</table>


</body>
</html>
```



HTML example of embedding the System Monitor.

4.6 LabVIEW Example

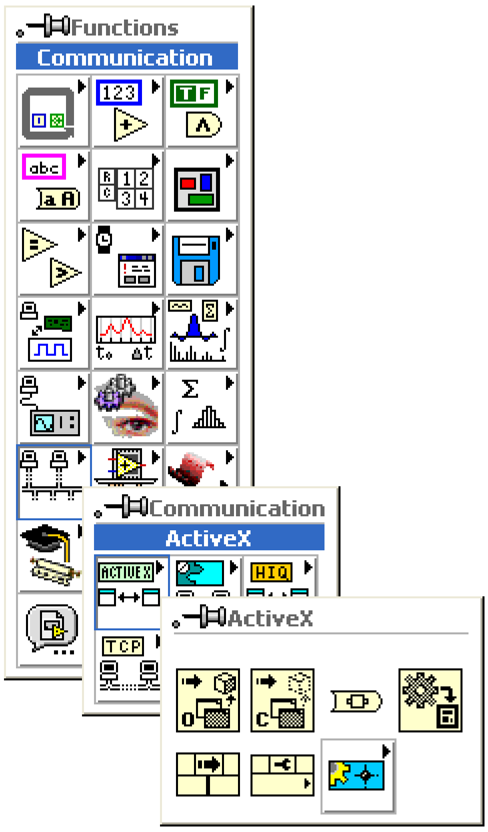
To start ActiveX programing in LabVIEW, add an Automation Refnum to the front panel and select the proper ActiveX class. See "LabVIEW Quick Example" (on page 26) how to do this.

"LabVIEW Example" (on page 151) 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 5-30: Relevant LabVIEW VIs

Automation Function	Description
Automation Open	Used to open an Automation refnum, which points to a specific ActiveX object.
Automation Close (old LabVIEW)	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.


Functions
Communication
Communication
ActiveX
ActiveX
HIQ
TCP

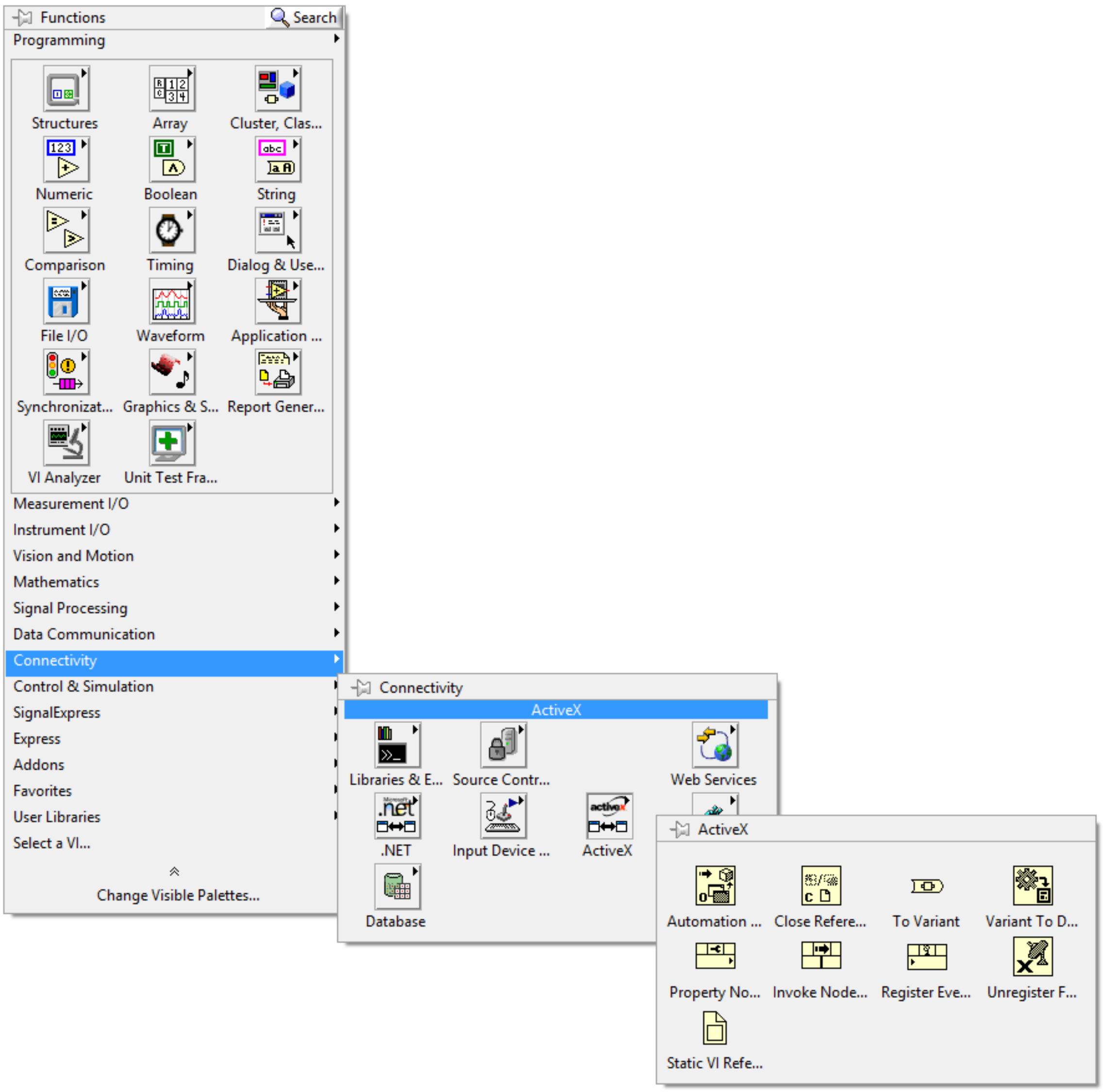


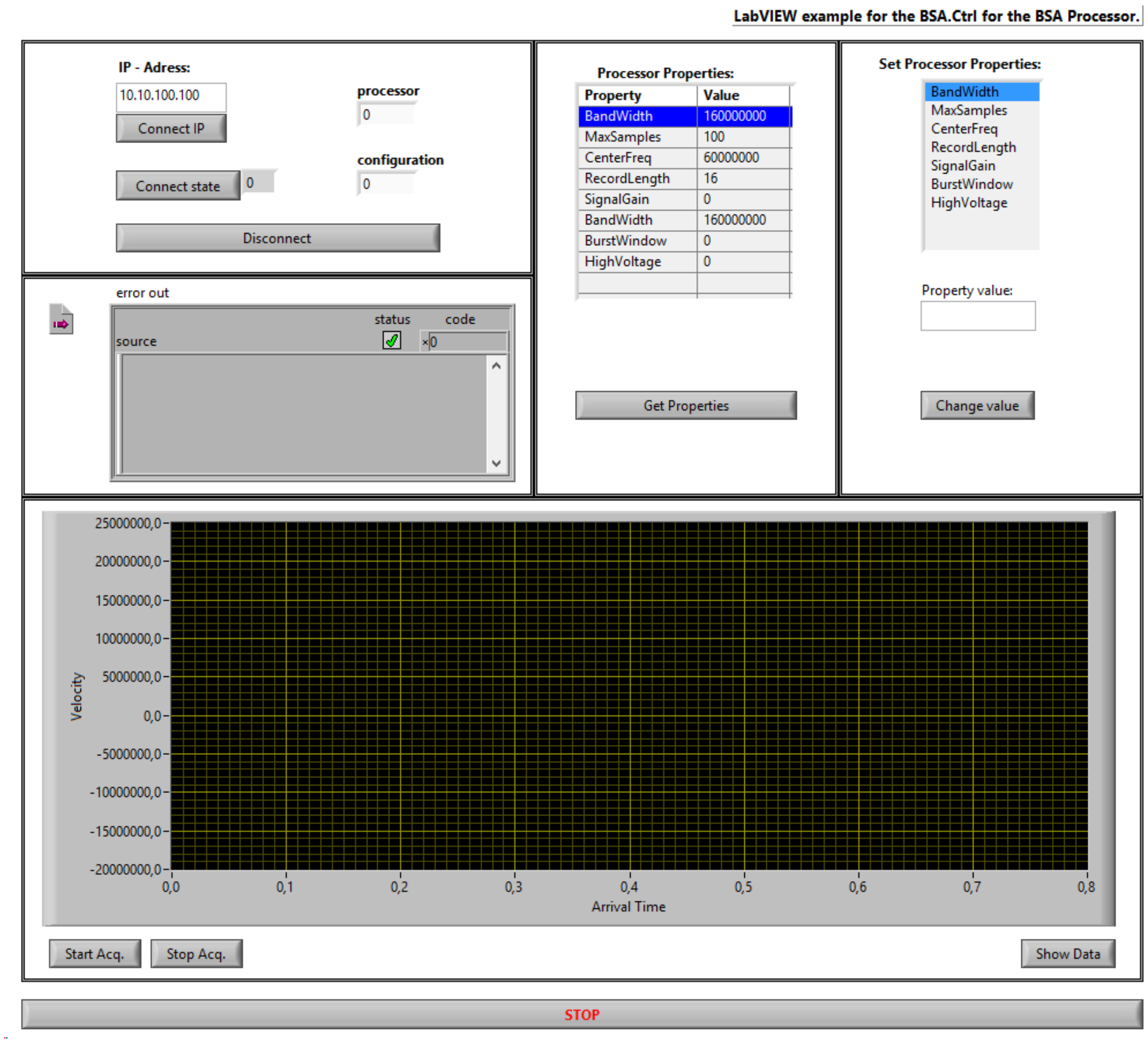
Figure 5-13: LabVIEW ActiveX Palette (older versions on top)

Now we will use these LabVIEW functions to develop a sample VI, which will connect to the BSA3.Ctrl, and read arrival time data and velocity data and display it as a time series.

The following will provide a description of each part of the diagram to make this.

The diagram is divided into the following parts:

- Connect to the BSA3 Processor
- Disconnect from it
- Get the connection state
- Set one property at a time
- Get a list of current properties
- Start an acquisition
- Stop it
- Show data in a graph



IP - Adress: 10.10.100.100

Connect state: 0

processor: 0

configuration: 0

error out

status ✓ code 0

source

Processor Properties:

Property	Value
BandWidth	160000000
MaxSamples	100
CenterFreq	60000000
RecordLength	16
SignalGain	0
BandWidth	160000000
BurstWindow	0
HighVoltage	0

Set Processor Properties:

- BandWidth
- MaxSamples
- CenterFreq
- RecordLength
- SignalGain
- BurstWindow
- HighVoltage

Property value:

Figure 5-14: LabVIEW Example Control Panel

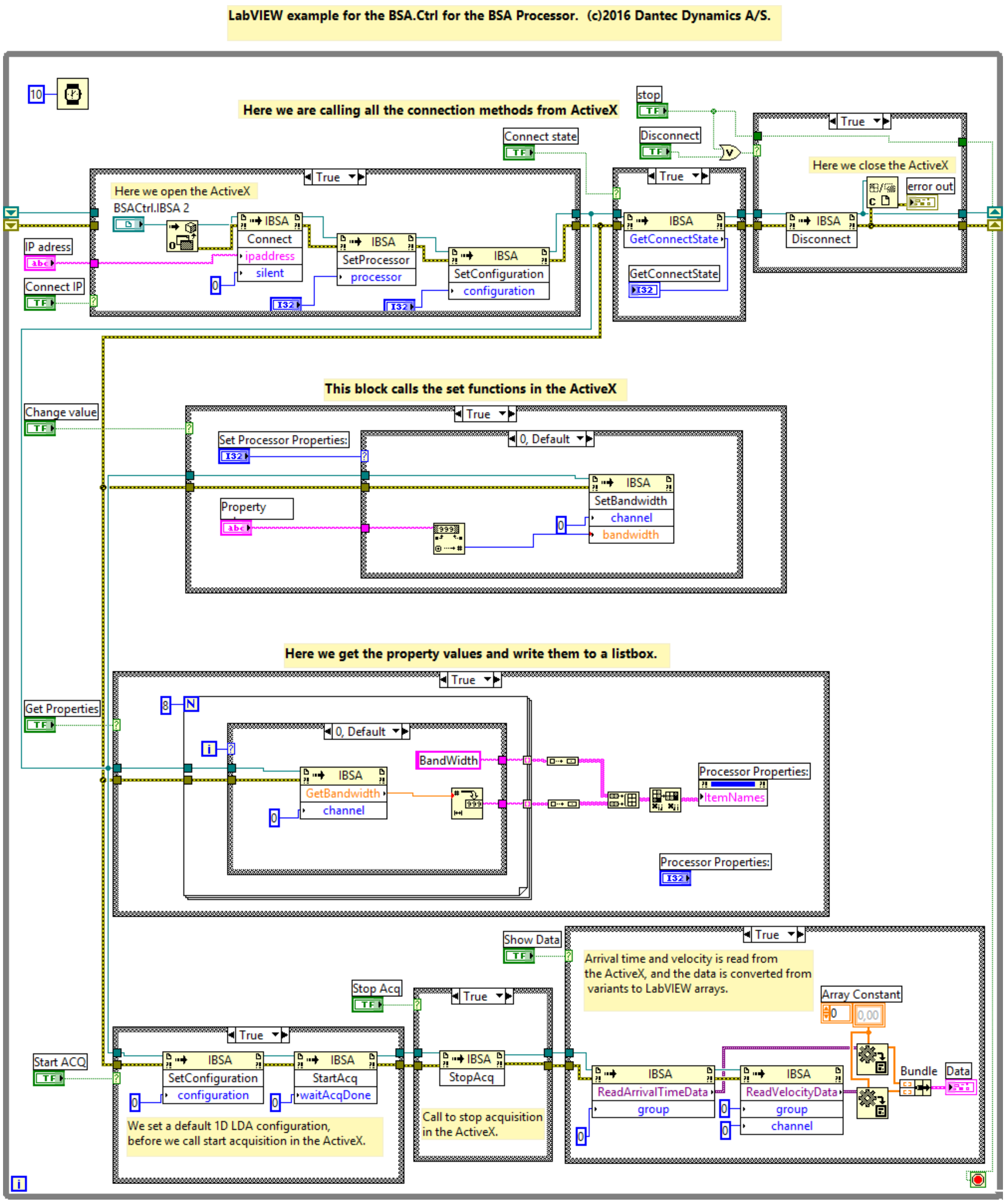
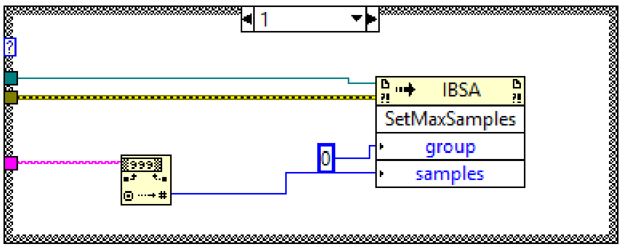
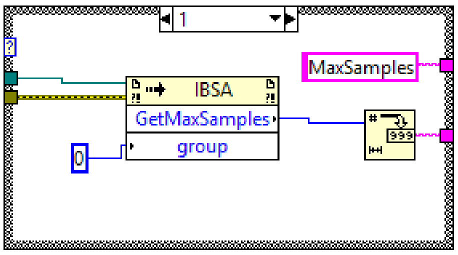

LabVIEW example for the BSA.Ctrl for the BSA Processor. (c)2016 Dantec Dynamics A/S.
Here we are calling all the connection methods from ActiveX
Here we open the ActiveX
BSACtrl.IBSA 2
IP adress
Connect IP
IBSA Connect ipaddress silent
IBSA SetProcessor processor
IBSA SetConfiguration configuration
Connect state
Disconnect
stop
IBSA GetConnectState
GetConnectState
Here we close the ActiveX
error out
IBSA Disconnect
This block calls the set functions in the ActiveX
Change value
Set Processor Properties:
Property
0, Default
IBSA SetBandwidth channel bandwidth
Here we get the property values and write them to a listbox.
Get Properties
BandWidth
IBSA GetBandwidth channel
Processor Properties:
ItemNames
Show Data
Arrival time and velocity is read from the ActiveX, and the data is converted from variants to LabVIEW arrays.
Array Constant
Stop Acq
Start ACQ
IBSA SetConfiguration configuration
IBSA StartAcq waitAcqDone
We set a default 1D LDA configuration, before we call start acquisition in the ActiveX.
IBSA StopAcq
Call to stop acquisition in the ActiveX.
IBSA ReadArrivalTimeData group
IBSA ReadVelocityData group channel
Bundle
Data

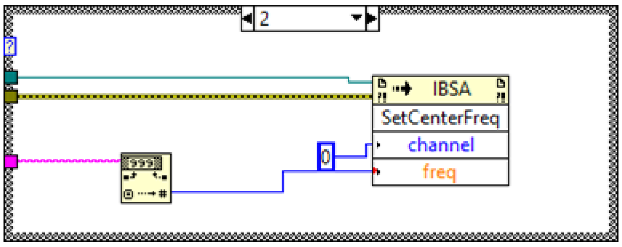
Figure 5-15: LabVIEW Example Diagram

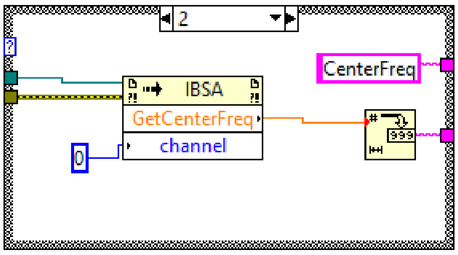
This sub-block calls the set functions in the ActiveX

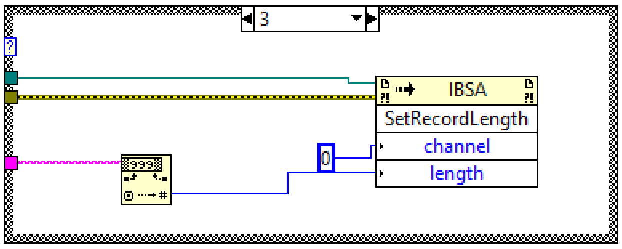


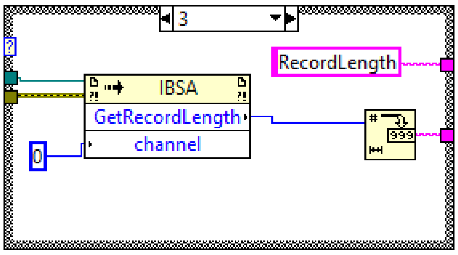
Here we get the property values and write them to a listbox.

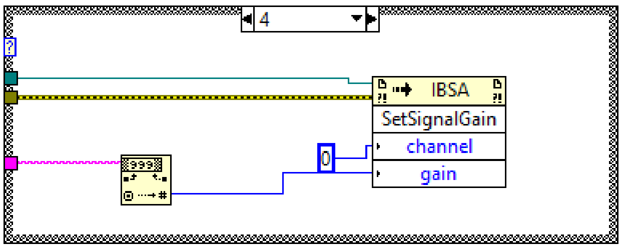


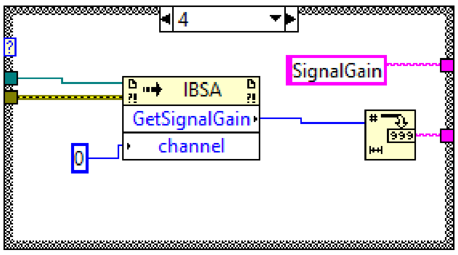


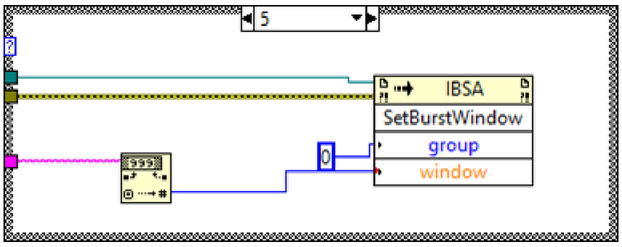


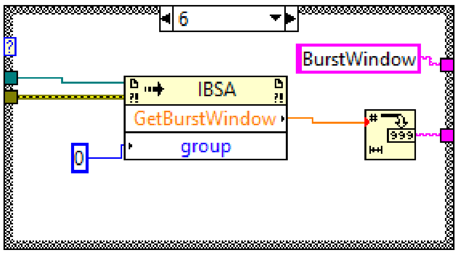


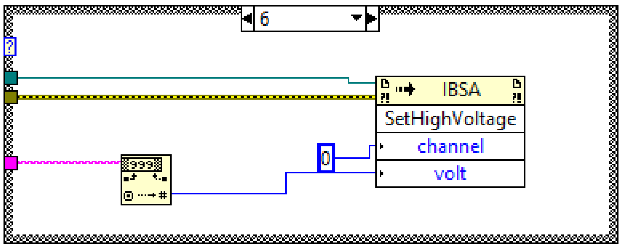


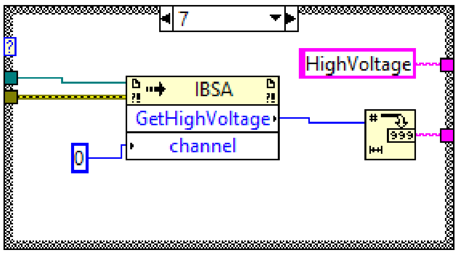










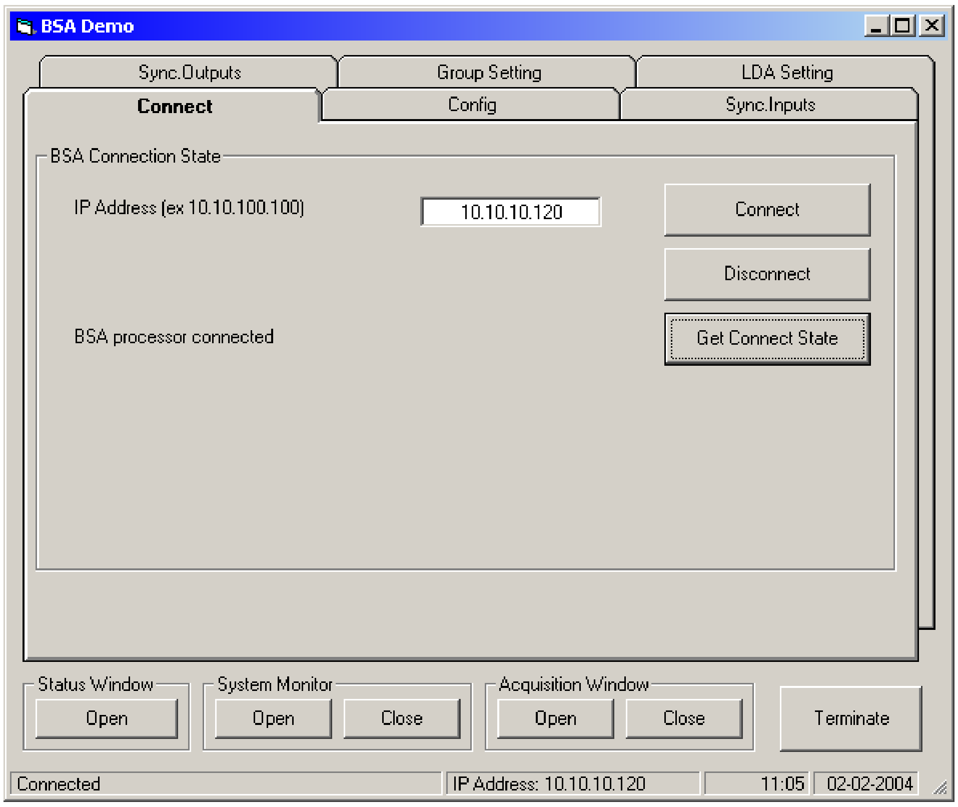


4.7 Visual Basic Example

On the DVD 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 screen shots from the program.

From the Connect page you can specify the IP-address of the BSA3 Processor. By pressing the buttons you can connect and disconnect from the BSA3 Processor and get the connection status.



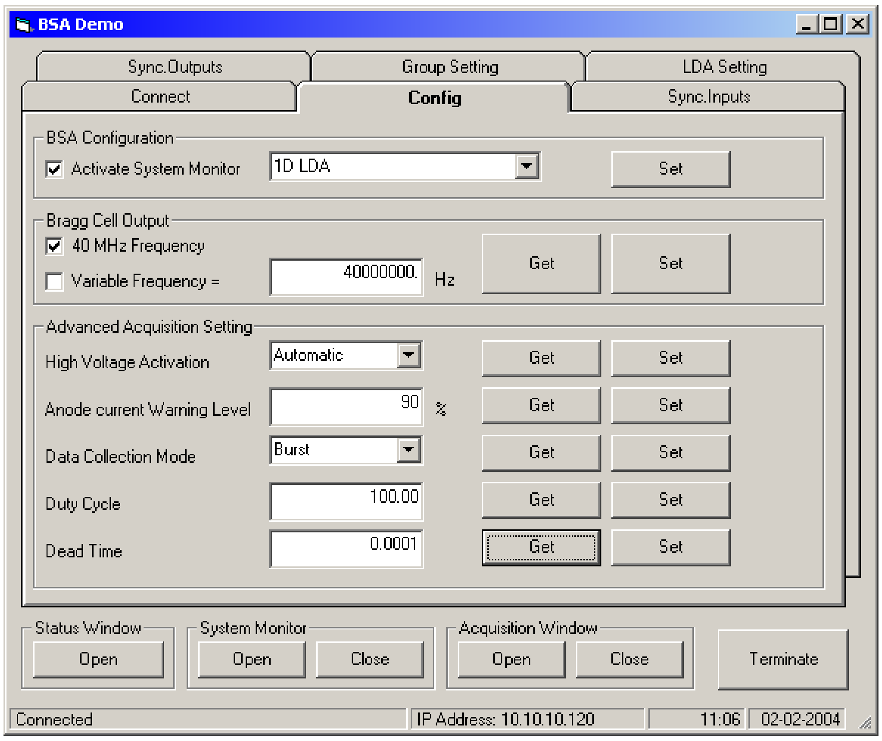
BSA Connection State

IP Address (ex 10.10.100.100) 10.10.10.120

BSA processor connected

Connected | IP Address: 10.10.10.120 | 11:05 | 02-02-2004

The Config page sets the configuration along with global BSA3 Processor properties.



BSA Configuration

☑ Activate System Monitor: 1D LDA

Bragg Cell Output

☑ 40 MHz Frequency

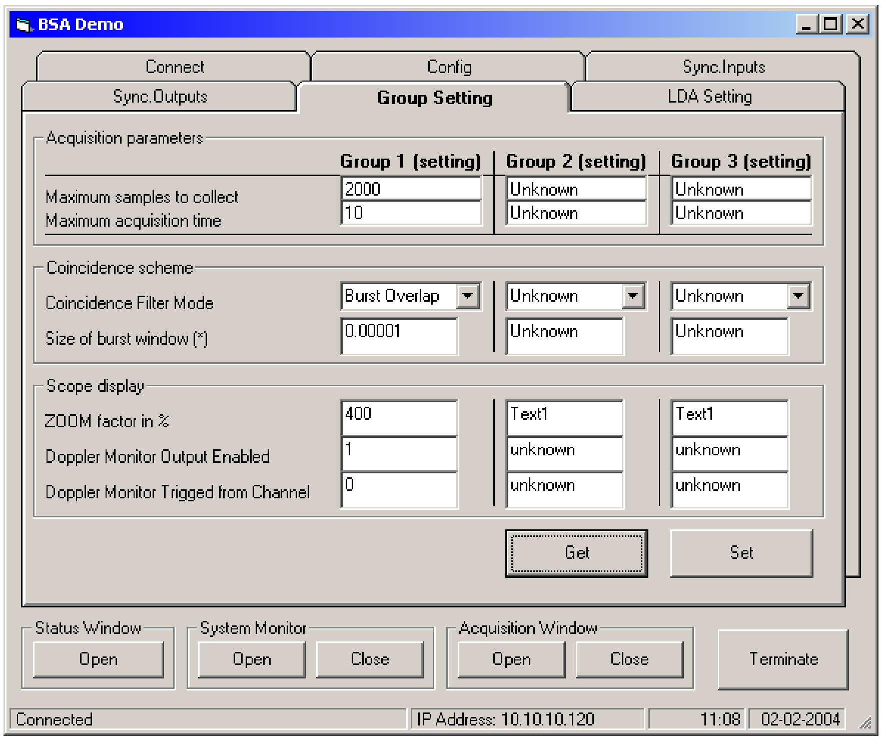
☐ Variable Frequency = 40000000. Hz

Advanced Acquisition Setting

High Voltage Activation	Automatic
Anode current Warning Level	90 %
Data Collection Mode	Burst
Duty Cycle	100.00
Dead Time	0.0001

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

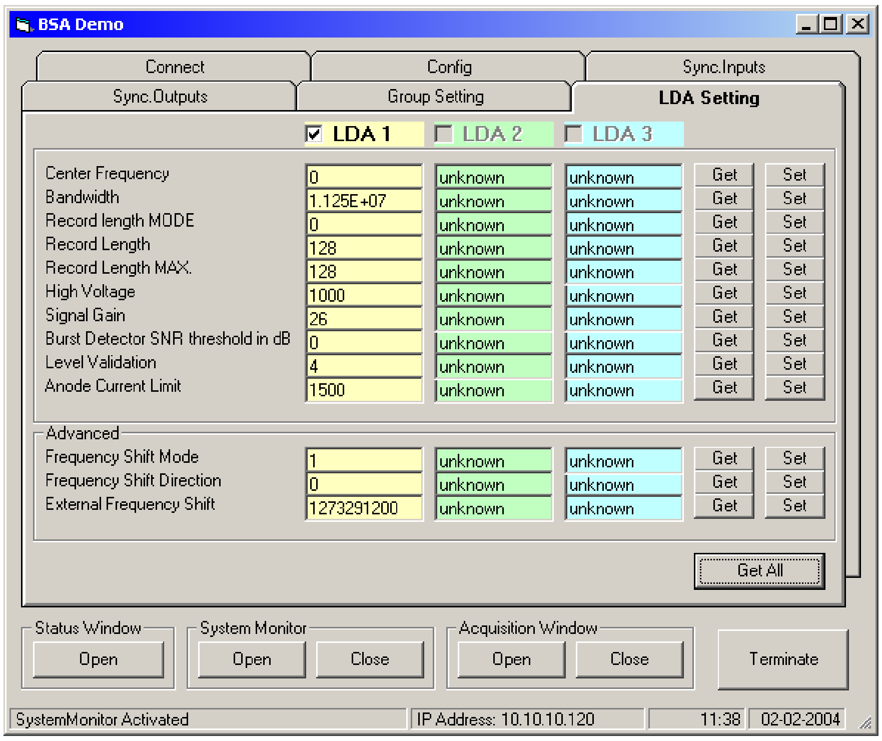
ZOOM factor in %	400	Text1	Text1
Doppler Monitor Output Enabled	1	unknown	unknown
Doppler Monitor Trigged from Channel	0	unknown	unknown

Get | Set

Status Window: Open | System Monitor: Open, Close | Acquisition Window: 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.



BSA Demo

Connect | Config | Sync.Inputs | Sync.Outputs | Group Setting | **LDA Setting**

	☑ LDA 1	☐ LDA 2	☐ LDA 3		
Center Frequency	0	unknown	unknown	Get	Set
Bandwidth	1.125E+07	unknown	unknown	Get	Set
Record length MODE	0	unknown	unknown	Get	Set
Record Length	128	unknown	unknown	Get	Set
Record Length MAX.	128	unknown	unknown	Get	Set
High Voltage	1000	unknown	unknown	Get	Set
Signal Gain	26	unknown	unknown	Get	Set
Burst Detector SNR threshold in dB	0	unknown	unknown	Get	Set
Level Validation	4	unknown	unknown	Get	Set
Anode Current Limit	1500	unknown	unknown	Get	Set

Advanced

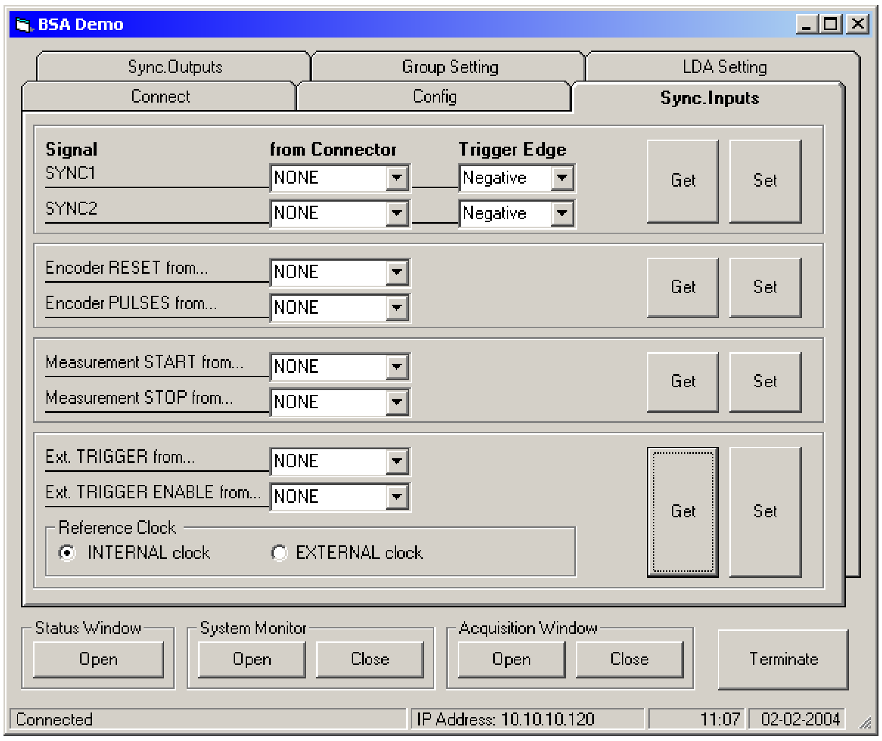
Frequency Shift Mode	1	unknown	unknown	Get	Set
Frequency Shift Direction	0	unknown	unknown	Get	Set
External Frequency Shift	1273291200	unknown	unknown	Get	Set

Get All

Status Window: Open | System Monitor: Open, Close | Acquisition Window: Open, Close | Terminate

SystemMonitor Activated | IP Address: 10.10.10.120 | 11:38 | 02-02-2004

On the Sync. Inputs page all synchronization settings can be set.


BSA Demo

Sync.Outputs | Group Setting | LDA Setting
Connect | Config | **Sync.Inputs**

Signal	from Connector	Trigger Edge
SYNC1	NONE	Negative
SYNC2	NONE	Negative

Get Set

Encoder RESET from... NONE
Encoder PULSES from... NONE

Get Set

Measurement START from... NONE
Measurement STOP from... NONE

Get Set

Ext. TRIGGER from... NONE
Ext. TRIGGER ENABLE from... NONE

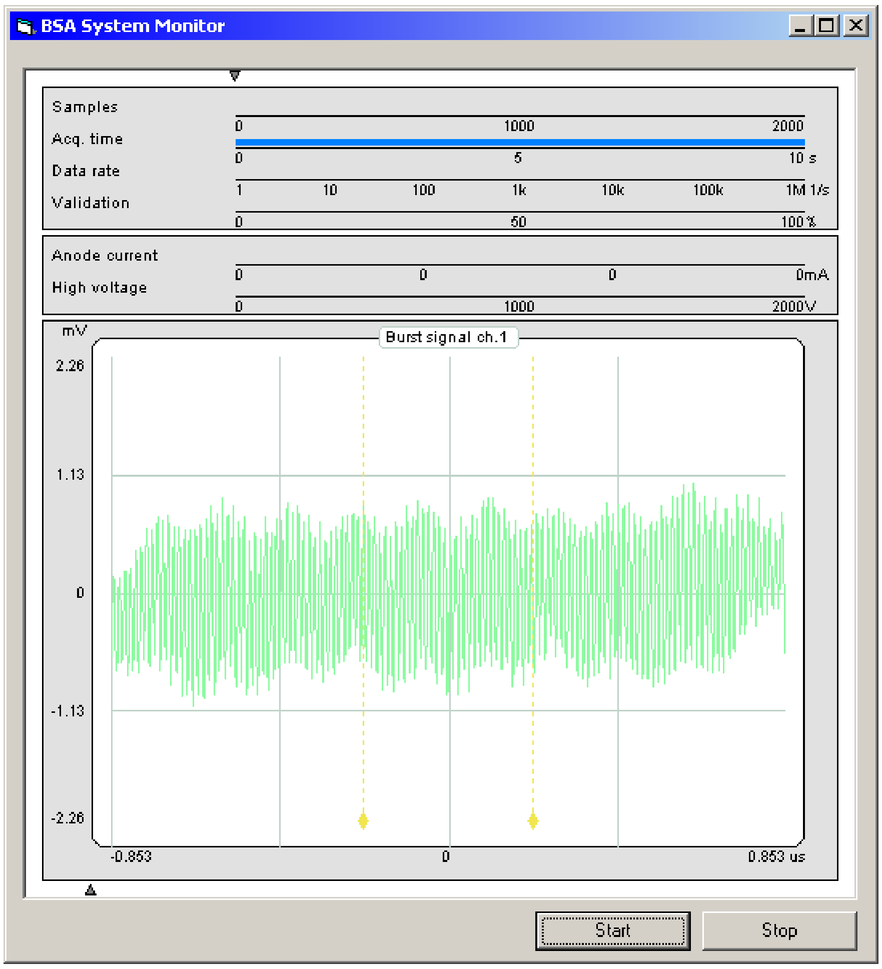
Reference Clock
[✓ INTERNAL clock] EXTERNAL clock

Get Set

Status Window: Open
System Monitor: Open Close
Acquisition Window: Open Close
Terminate

Connected | IP Address: 10.10.10.120 | 11:07 | 02-02-2004

When selecting System Monitor Open, the online system monitor window appears.



Before starting the system monitor, make sure the Activate System Monitor check box 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.

4.8 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.

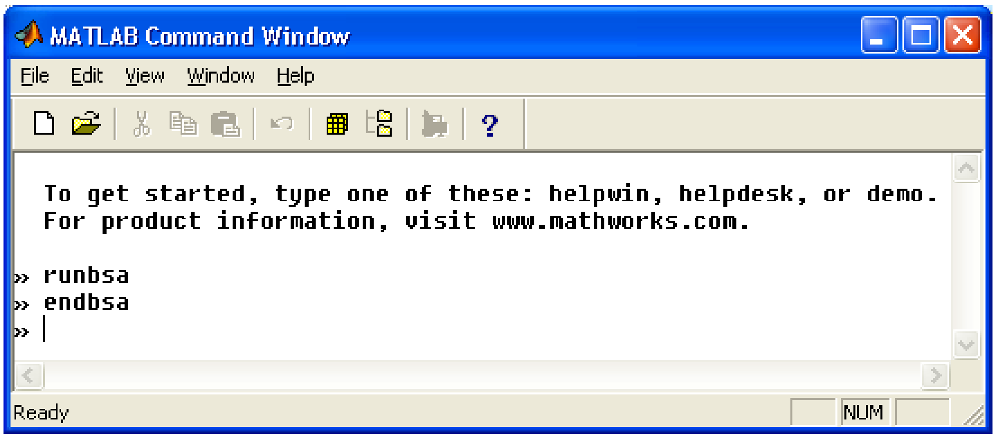
The API can be called from the command window or from an m-file. The following example consists of two m-files, one for creating the controls, connecting, measuring and plotting, and one for ending and cleaning up. Be sure to place the m-files in a path known by MATLAB; global paths can be found using 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','10.10.100.100',0);
while invoke(MyBSA,'GetConnectState')==0
end

% call connect on the system monitor
invoke(MySysMon,'Connect','10.10.100.100');

% 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
```

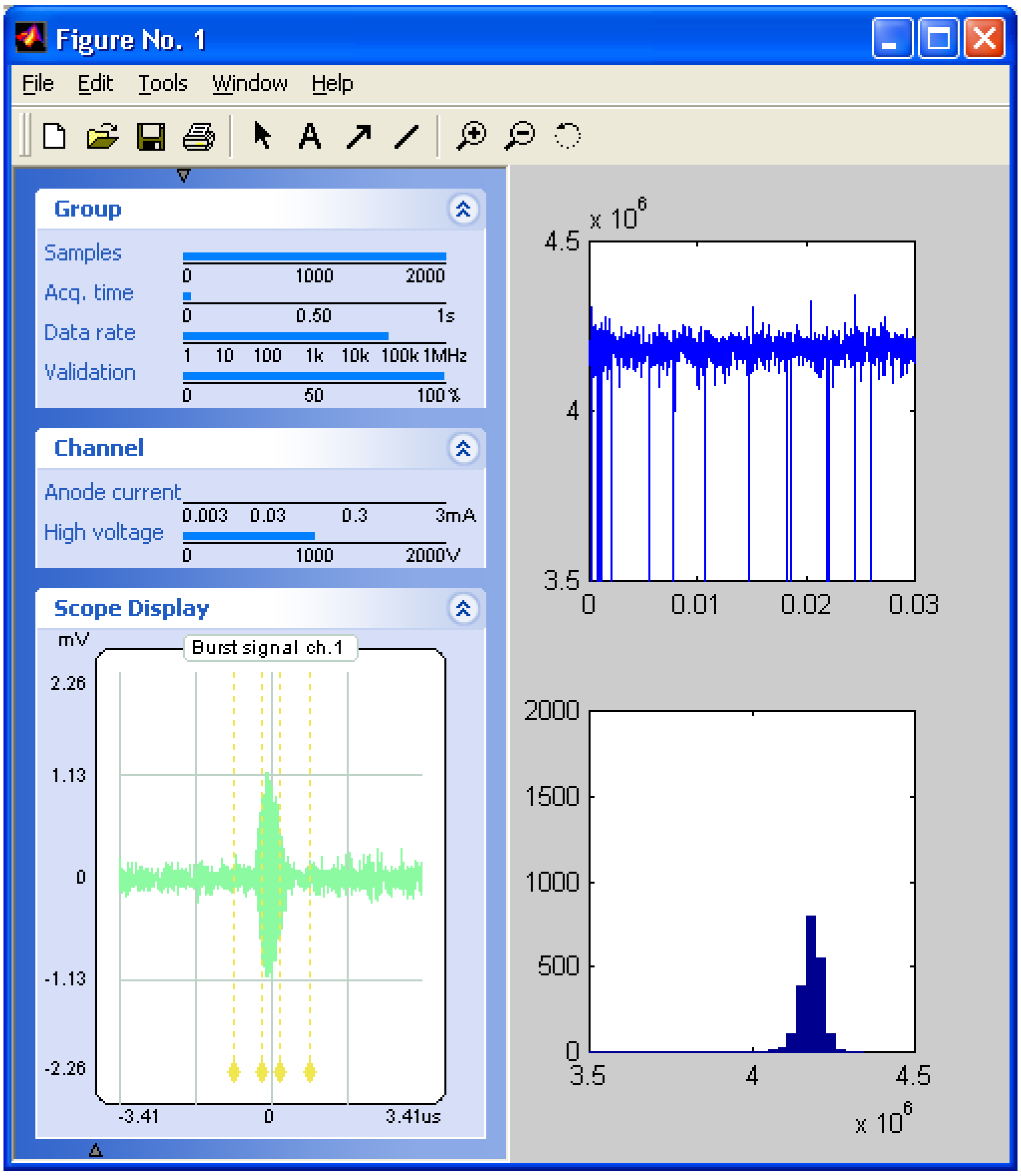
Call the function script directly from the MATLAB Command Windows using the m-file name runbsa, and when finished call endbsa.

Figure 10 MATLAB Command Windows calling script running the BSA.Ctrl and BSA.SysMon.Ctrl.



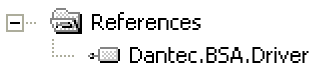
```
  To get started, type one of these: helpwin, helpdesk, or demo.
  For product information, visit www.mathworks.com.

>> runbsa
>> endbsa
>> |
```

Running the sample produces the following figure, displaying a running System Monitor and a small acquisition.



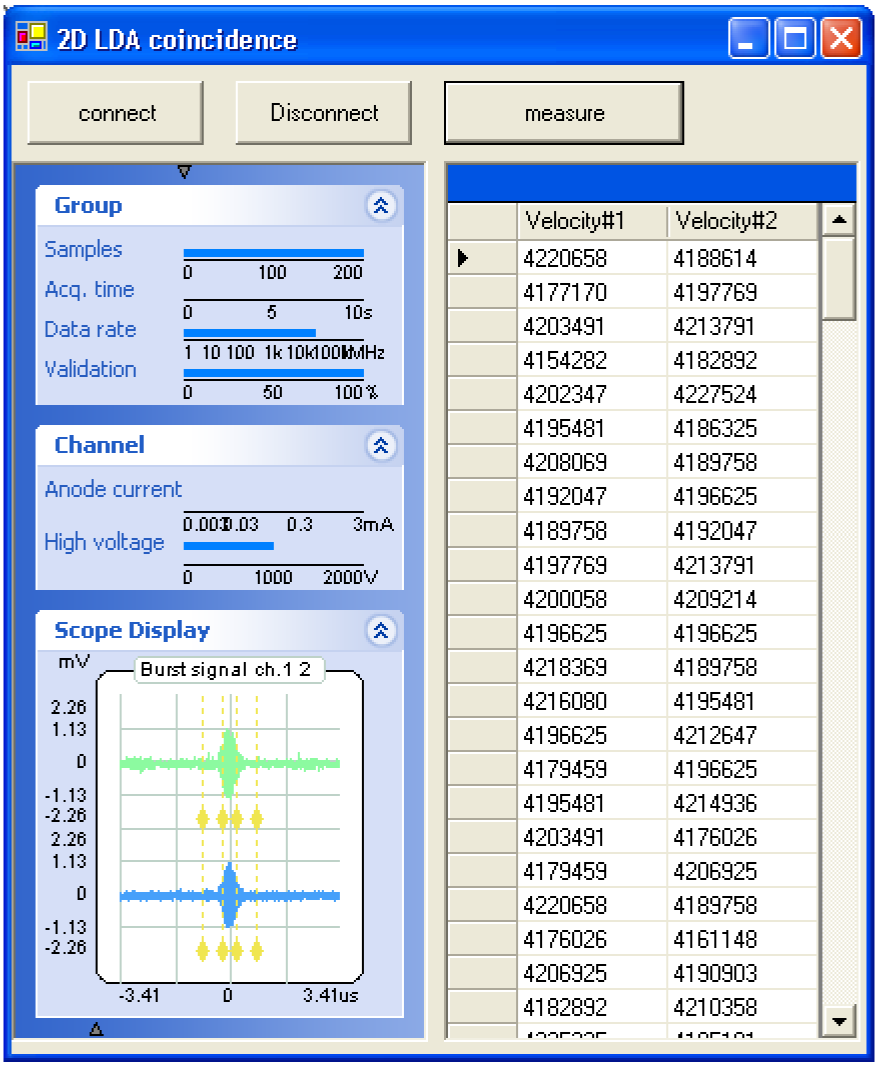
4.9 C# .NET Example

When using the .NET version of the BSA3 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.

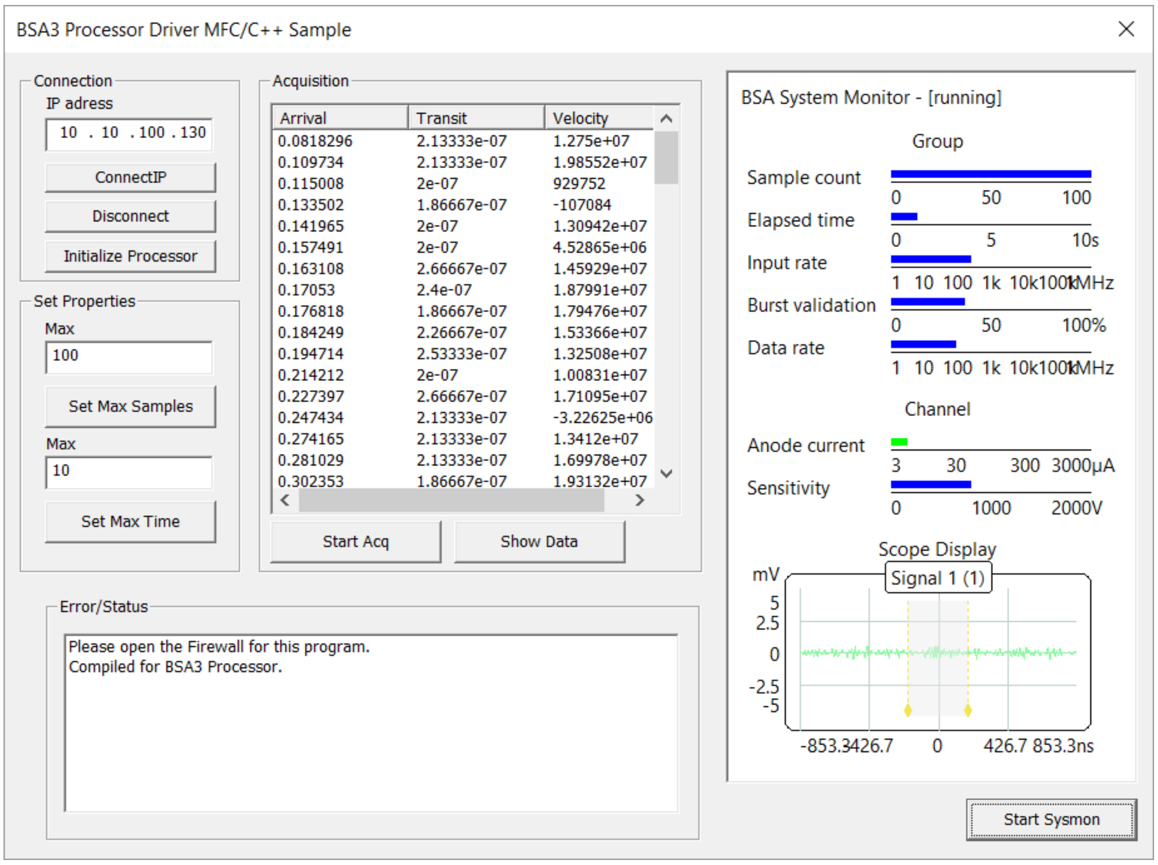


	Velocity#1	Velocity#2
▶	4220658	4188614
	4177170	4197769
	4203491	4213791
	4154282	4182892
	4202347	4227524
	4195481	4186325
	4208069	4189758
	4192047	4196625
	4189758	4192047
	4197769	4213791
	4200058	4209214
	4196625	4196625
	4218369	4189758
	4216080	4195481
	4196625	4212647
	4179459	4196625
	4195481	4214936
	4203491	4176026
	4179459	4206925
	4220658	4189758
	4176026	4161148
	4206925	4190903
	4182892	4210358

4.10 C++ MFC Example

The C++ example used the MFC framework for connecting to the components. Alternatively ATL can be use, or native C++ interfaces.

Here is an example of a small application using C++ and MFC.



IP adress: 10 . 10 . 100 . 130

Max: 100

Max: 10

Arrival	Transit	Velocity
0.0818296	2.13333e-07	1.275e+07
0.109734	2.13333e-07	1.98552e+07
0.115008	2e-07	929752
0.133502	1.86667e-07	-107084
0.141965	2e-07	1.30942e+07
0.157491	2e-07	4.52865e+06
0.163108	2.66667e-07	1.45929e+07
0.17053	2.4e-07	1.87991e+07
0.176818	1.86667e-07	1.79476e+07
0.184249	2.26667e-07	1.53366e+07
0.194714	2.53333e-07	1.32508e+07
0.214212	2e-07	1.00831e+07
0.227397	2.66667e-07	1.71095e+07
0.247434	2.13333e-07	-3.22625e+06
0.274165	2.13333e-07	1.3412e+07
0.281029	2.13333e-07	1.69978e+07
0.302353	1.86667e-07	1.93132e+07

```
Please open the Firewall for this program.
Compiled for BSA3 Processor.
```