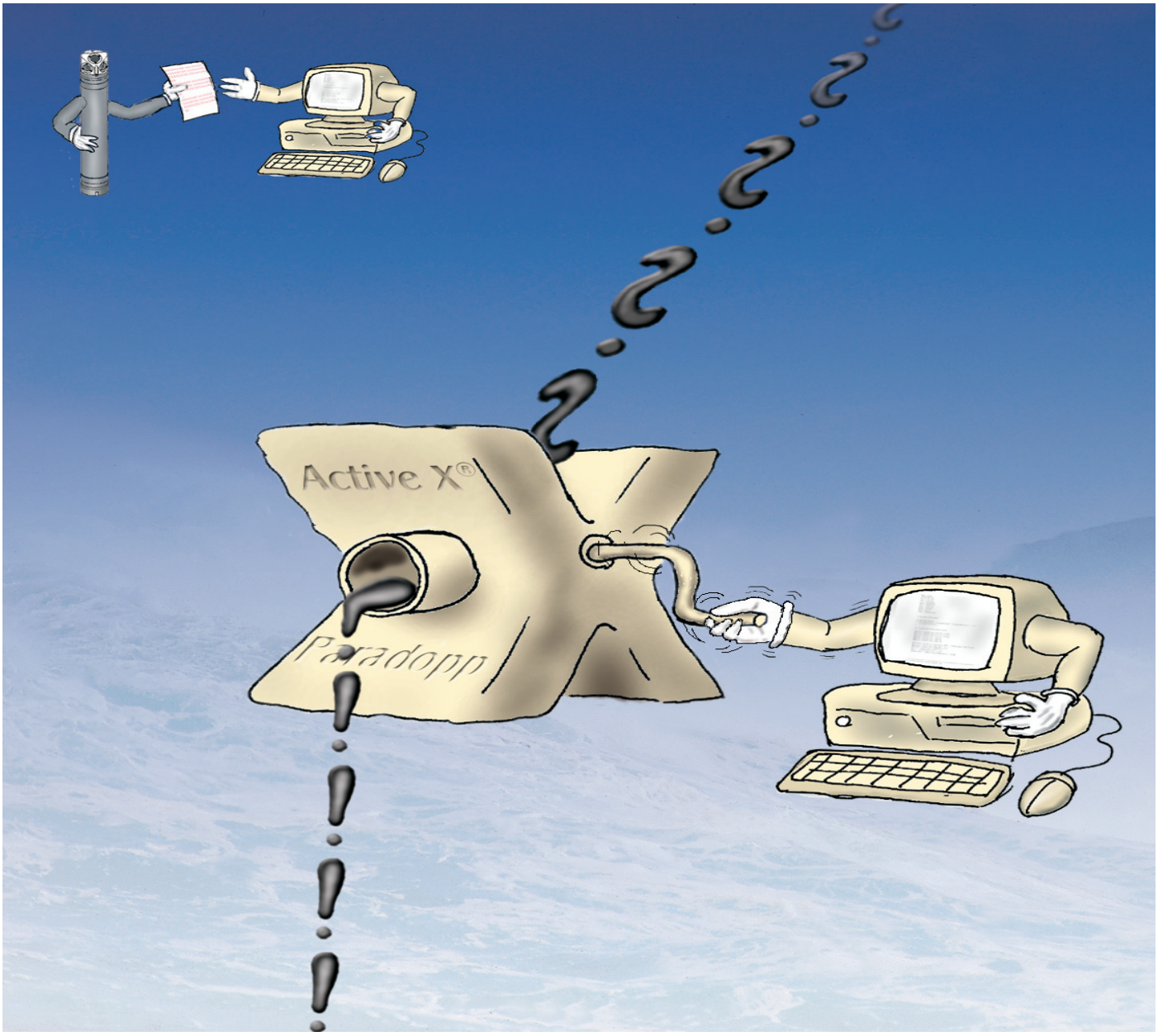
NORTEK AS

ACTIVE X MODULE FOR SYSTEM INTEGRATORS



PARADOPP FAMILY OF PRODUCTS

February 2008

ACTIVE X MODULE FOR SYSTEM INTEGRATORS

Nortek AS, Vangkroken 2, NO-1351 RUD, Norway.

Tel: +4767174500 • Fax: +4767136770 • e-mail: inquiry@nortek.no • www.nortek-as.com

NORTEK AS

Software updates and technical support

www.nortek-as.com - Our head office
www.nortekusa.com - Our US office
www.nortek.com.cn - Our Chinese office.

Nortek Support Forum

If you have any comments, tips, suggestions for improvement or any other information that you think may be of interest then you should join the Nortek on-line forum. Go to any of our web sites and click the “Forum” link.

The Forum offers a great opportunity for you to share your experiences of using Nortek instruments and to allow you to benefit from other’s.

Feedback

Your feedback is greatly appreciated. If you find errors, spelling mistakes, omissions, sections poorly explained or have other comments on this manual please let us know, e-mail us at:

inquiry@nortek.no

Communicating with us

If you need more information, support or other assistance, do not hesitate to contact us:

Nortek AS
Vangkroken 2
NO-1351 RUD, Norway
Phone: +47 6717 4500, Fax: +47 6713 6770
E-mail: inquiry@nortek.no

Contents

Code Examples

Introduction

This document provides the information necessary to control a Nortek Paradopp product (Aquadopp, Vector, etc.) using the PdCommX ActiveX® control function interface. The document is aimed at system integrators and engineers in need of writing their own Win32® or Windows® CE applications to control one or more Paradopp products. Examples are provided in Visual C++®, Visual Basic® and Delphi™. The document's scope is limited to interfacing issues and does not address general performance issues of the instruments. For a more thorough understanding of the principle of operation, we recommend the user guide that accompanies individual instruments.

Installation

Install the PdCommX component on your computer by using the program Setup.exe. The setup program copies files from the CD onto your hard disk and adds the necessary interface information to the Windows® registry.
For Windows® CE installations, a link between your mobile device and your desktop computer must be established using Microsoft® ActiveSync® before you run the setup.

Overview of the Technology

ActiveX® controls technology builds on a foundation of many lower-level objects and interfaces in OLE Automation. An external application accessing an Automation object does so by first connecting to the object and then requesting access to one or more of its published interfaces. An interface is an entry point that allows access to one or more related methods, properties and events. Once an application obtains an interface on the object, it proceeds to call any interface methods, as though they were part of the application itself.

Think of the PdCommX component as a programmable replacement for the standard Paradopp software user interface – the behaviour seen when using the standard software interface is to a large extent the same behaviour seen when using the ActiveX control. The PdCommX properties correspond to deployment planning and configuration dialog parameters and the PdCommX methods correspond to menu commands. A PdCommX event is generated whenever new data is available from the instrument.

Implementation

Implementing the Technology

When connected to the Automation object and its interfaces, identifier parameters, also known as Globally Unique Identifier (GUIDs), are passed to the automation library API functions. There is a single GUID representing the object itself and a GUID for each of the exposed interfaces. In order for the application to succeed in connecting to the object and its interfaces, these GUIDs must be present in the registry. The GUID entries are added to the registry by the PdCommX installation program.

The PdCommX control may be accessed by any language platform or software development environment (SDE) that supports OLE Automation (OCX interface). The most popular such languages platforms are Microsoft Visual C++®, Microsoft Visual Basic® and Borland Delphi™. The principles for coding for these platforms is similar, although in the Visual Basic® environment much of the low level work is performed behind the scenes by the Visual Basic® run-time system.

Note: The Connect method must be called prior to any other methods. Also, in order for some of the properties settings to take effect, the SetConfig method must be called prior to the Start command.

A general PdCommX OCX implementation is as follows:

1. Register PdCommX.ocx with the operating system (This is done by the PdCommX setup, however, most SDE's provide tools to register ActiveX® controls)
2. Include the PdCommX.ocx file into your application project.
3. Place the PdCommX control onto a destination form.
4. Use the property window to adjust default properties as desired.
5. Adjust properties in code as desired (Typically in the Visual Basic® Form_Load event or in the Delphi™ FormCreate procedure)
6. Implement the PdCommX methods and a NewData event handler as desired.

What follows is a brief overview of the programming techniques that allow easy access to the PdCommX interface from either of these languages.

Using Microsoft Visual C++®:

1. Create an MFC application using the MFC AppWizard (Ensure the ActiveX® Controls box is checked in step 3).
2. Open the Components and Controls gallery
3. Choose PdCommX in the Registered ActiveX® controls folder.
4. Click Insert to generate the CPdCommX class files.
5. Place the PdCommX control onto a dialogue (dialogue-based application) or a form view (form-based application)

Using Microsoft Visual Basic®:

1. Create a Standard EXE project.
2. Open the Components dialogue.
3. Check the PdCommX ActiveX® Control module in the Controls tab list.
4. Click OK to add the PdCommX control to the Toolbox.
5. Place the PdCommX control onto a form.

Using Borland Delphi™:

1. Create a new Application project.
2. Open the Import ActiveX® dialogue.
3. Select the PdCommX ActiveX® Control module in the controls list.
4. Click Install to create a PdCommX class unit and add the PdCommX control to your project.
5. Place the PdCommX control onto a form.

Your first application

It doesn't really matter which language you choose to write your application in or which developement environment you use. There are a few things that will be very similar or indeed the same for each. The example provided in this manual use the most frequently used. These notes follow these examples.

You'll note that for each of the examples there are three functions or seperate sections of code. One is a function called when the dialog or form you are using for your application is first displayed and initialised. It makes sense then to put code here that you need to do only once. The second common to all examples is the code that is executed when a button is clicked. These examples all assume that you have put a "Start" button somwhere on your dialog or form. This will make some basic checks of the instrument and if possible start a data measurement. The text for the button is changed so it functions as a "Stop" button too. The third function is the function the ActiveX object calls everytime there is new data from the instrument. If your measurement example is set to 10 seconds then this function will be called every ten seconds. These example do little apart from change the text on some label controls that it expects to find on the dialog or form. Not very exciting but it illustrates what is required.

Accessing Properties using Visual Basic

The Function Reference chapter lists all the properties and methods available within the PdCommX control. Included with the description for each is are **Visual C/C++** function definitions you need to use to access the properties. This is done differently in **Visual Basic.** All properties are referenced directly from the PdCommX control.

For example:

PdCommX1.MeasInterval = 1

Where PdCommX1 is an instance of our PdCommX control, MeasInterval is a property and an instrument parameter.

Here we are setting the MeasInterval property to a value of 1.

Properties are named as per their descriptions given below. The **C/C++** functions used to access these properties are typically prepended by "Get" or "Set". If you are unsure of the actual names to use within **Visual Basic** you can find them in the properties window of the control or when typing. Regular users of **Visual Basic** will be familiar with this.

Function Reference

Properties

All the properties listed here are properties if the PdCommX object. These properties are analagous to deployment parameters for the various instruments. It is important to note that these properties or instrument paramters are not written to or saved in the instrument without you explicitly do so by calling the SetConfig method. This means that whichever development environment you are using the PdcommX object's properties will not necessarily be that same as those on the instrument. Properties are set and returned in different ways depending upon the language you are using for your application. Examples are given below for C/C++ for each property. The previous chapter Implemention gives a little more information on this.

The second section of this chapter deals with methods available to you from within the PdCommX object. You will note there are some similarities between those functions and those here used to access properties. Note that those methods will return instrument parameters direct from the instrument, they can as the following paragraph show be different from what you might expect.

Since each instrument type has specific features, properties can apply to all, some or one of the instruments. The **Scope** heading in the property reference identifies the instrument type. It is important that you take notice of this, it is very clear about which instruments the properties apply to. If you are developing applications that are intended to be used across several instruments and you need to use properties that vary in their scope. You will need to verify the instrument type prior to calling these methods or functions. Error checking within the PdCommX object is robust but results from calling methods or functions out of scope can be unpredicable or misleading.

The instrument operation parameters may be specified at two levels. The **Standard** level provides default settings for various environments and mounting arrangements. The **Advanced** level allows for fine tuning the operation parameters. The **Category** heading in the reference specifies the level. Note that the **ConfigLevel** property must be set to **Advanced** for the property settings categorized as Advanced to be effective.

Property settings in the **Communication** category control the serial communication and must be set prior to the **Connect** method to be effective.

The description includes function definitions in the Microsoft Visual C++® language. For function definition details (return types and parameter types) in other languages use your development environments object browser.

AnalogInput1	
Scope All except Vectrino	Description The instrument can read two analogue inputs at the same time. The input range is 0-5 Volt, where 0 Volt equals 0 counts, 5 Volts equals 65535 counts and 2.5 Volts equals 32768 counts. The voltage output is fixed in production to either 5 Volts, 12 Volts or to the instrument voltage. The use of analogue inputs requires a special internal harness. Some systems are equipped with this at the time of purchase. It is also possible to purchase the harness separately and upgrade the instrument. Values: 0 = None 1 = Standard(Profile) 2 = Fast(Wave) 3 = StandardFast(Profile & Wave)
Category Advanced	

Default
0 (None)
	Visual C++®
	`short GetAnalogInput1()`
	`void SetAnalogInput1(short nNewValue)`

AnalogInput2	
Scope All except Vectrino	Description See AnalogInput1.
Category Advanced	

Default
0 (None)
	Visual C++®
	`short GetAnalogInput2()`
	`void SetAnalogInput2(short nNewValue)`

AnalogOutput	
Scope Vector, Vectrino	Description Enables or disables analogue output. When enabled, the 3D velocity is output as 0-5 Volt continuous signal over a separate set of three wires, one for each velocity component. Values are: 0 = OFF 1 = ON
Category Standard (Vectrino) Advanced (Vector)	

Default
0 (OFF)
	Visual C++®
	`short GetAnalogOutput()`
	`void SetAnalogOutput(short nNewValue)`

AnalogRange	
Scope Vector, Vectrino	Description AnalogRange specifies the velocity range that will correspond to the full analogue output range. See GetAnalogRangeCount and GetanalogRangeValue methods.
Category Standard (Vectrino) Advanced (Vector)	

Default	Visual C++®
	`short GetAnalogRange()`
	`void SetAnalogRange(short nNewValue)`

AutoConnect	
Scope All instruments	Description Enables or disables automatic baud rate scanning. Values are: 0 = OFF 1 = ON
Category Communication	

Default
0 (OFF)
	Visual C++®
	`short GetAutoConnect()`
	`void SetAutoConnect(short nNewValue)`

AvgInterval	
Scope All except Vector and Vectrino.	Description Averaging Interval specifies how long the instrument will be actively collecting data within each Measurement Interval. **Example:** Measurement Interval is set to 600 seconds, averaging interval set to 60 seconds means that the instrument will collect data for 1 minute every 10 minutes. Averaging Interval must be smaller than Measurement Interval.
Category Standard (EasyQ) Advanced (all others)	

Default	Visual C++®
	`short GetAvgInterval()`
	`void SetAvgInterval(short nNewValue)`

Baudrate	
Scope All instruments	Description The baud rate to use for the connection. Values are: 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200
Category Communication	

Default
9600
	Visual C++®
	`long GetBaudRate()`
	`void SetBaudRate(long nNewValue)`

BlankingDistance	
Scope All except Vector and Vectrino.	Description Blanking distance is the distance in metres from the Aquadopp sensor head to the start of the (first) measurement cell.
Category Standard (EasyQ) Advanced (all others)	

Default	Visual C++®
	`float GetBlankingDistance()`
	`void SetBlankingDistance(float newValue)`

BreakType	
Scope All instruments	Description Sets instrument break command type. Which break type to use depends on the production date of the Nortek instrument. Most instruments manufactured after 2001 use soft break. Run the software shipped with the instrument to find out which type of break to use. Values are: 0 = Hard Break 1 = Soft Break
Category Communication	

Default
1 (Soft Break)
	Visual C++®
	`short GetBreakType()`
	`void SetBreakType(short nNewValue)`

CellSize	
Scope Aquapro, AWAC, Continental	Description Cell size specifies the vertical length in metres of each depth cell in the profile. Applies to profiling instruments only. Standard deviation (accuracy) of the velocity measurements is inversely proportional to the depth cell size (larger depth cells give smaller standard deviations).
Category Standard	

Default	Visual C++®
	`float GetCellSize()`
	`void SetCellSize(float newValue)`

ConfigLevel	
Scope All instruments Category Application	Description Properties controlling the instrument may be specified at two levels. Use the Standard level to configure the system with default settings for various environments and mounting arrangements. Use the Advanced level to fine tune the operation parameters. Values are: 0 = Standard 1 = Advanced

Default
0 (Standard)
	Visual C++®
	`short GetConfigLevel()`
	`void SetConfigLevel(short nNewValue)`

CoordinateSystem	
Scope All Category Standard (Vectrino) Advanced (all others)	Description The coordinate system can be selected to Beam, XYZ, or ENU. Beam means that the recorded velocity will be in the coordinate system of the acoustic beams. XYZ means that the measurements are transformed to an orthogonal XYZ coordinate system fixed relative to the instrument and ENU means that the data are converted to geographic coordinates every second. Values are: 0 = ENU 1 = XYZ 2 = Beam The use of ENU requires that the instrument is equipped with a compass.

Default
0 (ENU)
	Visual C++®
	`short GetCoordinateSystem()`
	`void SetCoordinateSystem(short nNewValue)`

DefaultInstrument	
Scope All instruments	Description Sets the instrument type. Values are: 0 = AQUADOPP 1 = VECTOR 2 = AQUAPRO 3 = AWAC. 4 = EASYQ. 5 = CONTINENTAL 6 = VECTRINO
Category Communication	

Default
0 (AQUADOPP)
	Visual C++®
	`short GetDefaultInstrument()`
	`void SetDefaultInstrument(short nNewValue)`

DeploymentComment	
Scope All, except Vectrino	Description Max 180 characters of text which will be included in the data file for the purpose of documenting the data set.
Category Deployment	

Default
“N/A”
	Visual C++®
	`BSTR GetDeploymentComment()`
	`void SetDeploymentComment(LPCTSTR lpszNewValue)`

DeploymentName	
Scope All, except Vectrino	Description Recorder data file name. Max 6 characters of text.
Category Deployment	

Default
“N/A”
	Visual C++®
	`BSTR GetDeploymentName()`
	`void SetDeploymentName(LPCTSTR lpszNewValue)`

DeploymentTime	
Scope All, except Vectrino	Description Sets date and time to start data recording.
Category Deployment	

Default
Current time
	Visual C++®
	`DATE GetDeploymentTime()`
	`void SetDeploymentTime(DATE newValue)`

DiagnosticsInterval	
Scope Aquadopp, EasyQ	Description Sets the Interval in seconds for diagnostics data.
Category Advanced	

Default	Visual C++®
	`short GetDiagnosticsInterval()`
	`void SetDiagnosticsInterval(short nNewValue)`

DiagnosticsMode	
Scope Aquadopp, EasyQ	Description Enables or disables diagnostics. If enabled, the diagnostic mode will set the instrument to collect a number of diagnostics samples at regular intervals. For example, if the interval is set to 720 and number of samples is set to 30, the instrument will record 30 single second data points ever 12 hours. The number of samples cannot be larger than the Measurement interval minus the Averaging interval. Values are: 0 = OFF 1 = ON
Category Advanced	

Default
0 (OFF)
	Visual C++®
	`short GetDiagnosticsMode()`
	`void SetDiagnosticsMode(short nNewValue)`

DiagnosticsSamples	
Scope Aquadopp, EasyQ	Description Sets the number of diagnostics samples.
Category Advanced	

Default
20
	Visual C++®
	`short GetDiagnosticsSamples()`
	`void SetDiagnosticsSamples(short nNewValue)`

FileWrapping	
Scope All, except Vectrino	Description Enables or disables recorder file wrapping. If enabled, data is logged to the internal instrument recorder in ring-buffer mode. This ensures that the recorder always holds the latest data. If disabled data logging will stop when the recorder is full. Values are: 0 = OFF 1 = ON
Category Advanced	

Default
0 (OFF)
	Visual C++®
	`short GetFileWrapping()`
	`void SetFileWrapping(short nNewValue)`

FixedTemperature	
Scope All instruments	Description Temperature to use for calculation of Speed of Sound if TemperatureMode is Fixed.
Category Standard (Vector and Vectrino) Advanced (All others)	

Default
0.0
	Visual C++®
	`short GetFixedTemperature()`
	`void SetFixedTemperature(float nNewValue)`

HighRate	
Scope Aquadopp	Description Enables or disables 4 Hz sampling rate. Values are: 0 = OFF 1 = ON
Category Advanced	

Default
0 (OFF)
	Visual C++®
	`short GetHighRate()`
	`void SetHighRate(short nNewValue)`

MaxDepth	
Scope EasyQ Category Standard	Description The estimated maximum depth (meters) is used to determine the transmit level for stage measurements. For direct control, set ConfigLevel = Advanced and use the advanced properties.

Default
10
	Visual C++®
	`float GetMaxDepth()`
	`void SetMaxDepth(float newValue)`

MeasInterval	
Scope All, except Vectrino Category Standard	Description Measurement Interval describes how often the instrument will collect data and the value can be set from 1 to 3600 seconds. See AvgInterval to see how it affects MeasInterval **Example:** Measurement Interval = 600 seconds means that data will be written to the recorder and/or to disk file every 10 minutes.

Default
600
	Visual C++®
	`short GetMeasInterval()`
	`void SetMeasInterval(short nNewValue)`

MeasLoad	
Scope All, except Vectrino Category Advanced	Description Within each second, the instrument can either be in active mode (collecting data) or in idle mode (not collecting data). The Measurement load is the relative time spent in active mode within each second. Values are: From 0 (no data collection) to 100 (always in active mode).

Default	Visual C++®
	`short GetMeasLoad()`
	`void SetMeasLoad(short nNewValue)`

MeasLoadMode	
Scope All, except Vectrino	Description Enables or disables automatic calculation of an appropriate measurement load based on the averaging interval. Values are: 0 = Manual 1 = Automatic See **MeasLoad** command.
Category Advanced	

Default
1 (Automatic)
	Visual C++®
	`short GetMeasLoadMode()`
	`void SetMeasLoadMode(short nNewValue)`

Messages	
Scope All instruments	Description Enables or disables pop-up message boxes. Values are: 0 = OFF 1 = ON
Category Application	

Default
1 (ON)
	Visual C++®
	`short GetMessages()`
	`void SetMessages(short nNewValue)`

ModemConfig	
Scope All instruments	Description Modem AT configuration string.
Category Communication	

Default
ATM0&D2
	Visual C++®
	`BSTR GetModemConfig()`
	`void SetModemConfig(LPCTSTR lpszNewValue)`

ModemTimeout	
Scope All instruments	Description Modem response timeout in seconds.
Category Communication	

Default
60
	Visual C++®
	`short GetModemTimeout()`
	`void SetModemTimeout(short nNewValue)`

NumCells	
Scope Aquapro, AWAC, Continental	Description Number of depth cells to collect per profile. Applies to profiling instruments only. The maximum number of depth cells with accurate velocity data will vary with system frequency, depth cell size and deployment conditions.
Category Standard	

Default	Visual C++®
	`short GetNumCells()`
	`void SetNumCells(short nNewValue)`

PhoneNumber	
Scope All instruments	Description Modem phone number to dial.
Category Communication	

Default	Visual C++®
	`BSTR GetPhoneNumber()`
	`void SetPhoneNumber(LPCTSTR lpszNewValue)`

PowerLevel	
Scope All	Description The power level sets how much acoustic energy the instrument transmits into the water. The difference between the highest level and the lowest level is about 20dB. Values are: 0 = High 1 = HighLow 2 = LowHigh 3 = Low
Category Advanced	

Default	Visual C++®
	`short GetPowerLevel()`
	`void SetPowerLevel(short nNewValue)`

PowerOutMode	
Scope All, except Vectrino	Description Enable to supply power from the instrument to an external sensor. Values are: 0 = OFF 1 = ON
Category Advanced	

Default
0 (Off)
	Visual C++®
	`short GetPowerOutMode()`
	`void SetPowerOutMode(short nNewValue)`

PowerSource	
Scope EasyQ	Description Sets the instrument power source. The power source determines the default EasyQ transmit level (flow measurements). For direct control, set ConfigLevel = Advanced and use the advanced properties. Values are: 0 = Battery 1 = External
Category Standard	

Default
0 (Battery)
	Visual C++®
	`short GetPowerSource()`
	`void SetPowerSource(short nNewValue)`

Priority	
Scope All Instruments	Description Sets the desired priority of the data collection thread. Values are: 0 = Above Normal 1 = Below Normal 2 = Highest 3 = Idle 4 = Lowest 5 = Normal 6 = Time Critical
Category Communication	

Default
5 (Normal)
	Visual C++®
	`short GetPriority()`
	`void SetPriority(short nNewValue)`

QualityThreshold	
Scope EasyQ Category Standard	Description The stage algorithm looks for the first echo peak for which its quality crosses above this threshold.

Default
150
	Visual C++®
	`short GetQualityThreshold()`
	`void SetQualityThreshold(short nNewValue)`

Salinity	
Scope All Category Standard (Vector and Vectrino) Advanced (All others)	Description Salinity to use for sound speed calculation. The salinity is 0 for fresh water and typically 35 for the ocean.

Default
35.0
	Visual C++®
	`float GetSalinity()`
	`void SetSalinity(float newValue)`

SampleOnSynch	
Scope Vector/Vectrino Category Standard (Vectrino) Advanced (Vector)	Description Enables or disables start data collection by external trigger signal. Values are: 0 = OFF 1 = ON

Default
0 (OFF)
	Visual C++®
	`short GetSampleOnSynch()`
	`void SetSampleOnSynch(short nNewValue)`

SamplesPerMI	
Scope Vector Category Standard	Description Sets the number of samples per burst interval.

Default
10
	Visual C++®
	`short GetSamplesPerMI()`
	`void SetSamplesPerMI(short nNewValue)`

SamplingMode	
Scope Vector	Description Select continuous sampling to collect data continuously, without pause. Select burst interval to collect data for a number of samples/sampling rate seconds. The burst interval is the time between each measurement burst. After each burst, the instrument will go to sleep to conserve power and recorder capacity. Values are: 0 = Burst 1 = Continuous
Category Standard	

Default
1 (Continuous)
	Visual C++®
	`short GetSamplingMode()`
	`void SetSamplingMode(short nNewValue)`

SamplingRate	
Scope Vector/Vectrino	Description Sets the output rate in Hz for the velocity, amplitude, correlation, and pressure data. Values are: Vector 1,2,4,8,16,32,64 Hz Vectrino 1-25 Hz Vectrino Plus 1-200 Hz
Category Standard	

Default
8Hz Vector
25Hz Vectrino
200Hz Vectrino Plus
	Visual C++®
	`short GetSamplingRate()`
	`void SetSamplingRate(short nNewValue)`

SamplingVolume	
Scope Vector/Vectrino	Description When reducing the sampling volume size, the total number of samples used for the velocity calculation is reduced. The effect of this reduction is that the precision of the measured velocity is reduced. See GetSamplingVolumeCount and GetSamplingVolumeValue methods.
Category Standard	

Default	Visual C++®
	`short GetSamplingVolume()`
	`void SetSamplingVolume(short newValue)`

SerialOutput	
Scope All instruments	Description Enables or disables serial data output. Values are: 0 = Disabled 1 = Enabled
Category Communication	

Default
1 (Enabled)
	Visual C++®
	`BSTR GetSerialOutput()`
	`void SetSerialOutput(short nNewValue)`

SerialPort	
Scope All instruments	Description The name of the serial port connected to the instrument. Valid names are “COM1”, “COM2” etc.
Category Communication	

Default
“COM1”
	Visual C++®
	`BSTR GetSerialPort()`
	`void SetSerialPort(LPCTSTR lpszNewValue)`

SoundSpeed	
Scope All	Description Speed of sound to use if SondSpeedMode is Fixed.
Category Standard (Vector and Vectrino) Advanced (All others)	

Default
1525.0 m/s
	Visual C++®
	`float GetSoundSpeed()`
	`void SetSoundSpeed(float newValue)`

SoundSpeedMode	
Scope All	Description Speed of sound can be set by the user (Fixed) or calculated by the instrument based on the measured temperature and a user-input value for salinity (Measured). Values are: 0 = Measured 1 = Fixed
Category Standard (Vector and Vectrino) Advanced (All others)	

Default
0 (Measured)
	Visual C++®
	`short GetSoundSpeedMode()`
	`void SetSoundSpeedMode(short nNewValue)`

StageAvgInterval	
Scope EasyQ	Description The stage averaging Interval specifies how long the instrument will be actively collecting stage data within each Measurement Interval. **Example:** Measurement Interval is set to 600 seconds, averaging interval set to 60 seconds means that the instrument will collect data for 1 minute every 10 minutes. The averaging Intervals (stage + flow) must be smaller than Measurement Interval.
Category Advanced	

Default
60
	Visual C++®
	`short GetStageAvgInterval()`
	`void SetStageAvgInterval(short nNewValue)`

StageMode	
Scope EasyQ	Description Enables or disables stage measurements. Values are: 0 = OFF 1 = ON
Category Standard	

Default
1 (ON)
	Visual C++®
	`short GetStageMode()`
	`void SetStageMode(short nNewValue)`

StageOffset	
Scope EasyQ	Description The StageOffset is added to the stage measurement after stage is computed. The stage algorithm first compares measured stage to pressure, and then adds the offset.
Category Standard	

Default
0
	Visual C++®
	`float GetStageOffset()`
	`void SetStageOffset(float newValue)`

StagePowerLevel	
Scope EasyQ Category Advanced	Description The stage power level sets how much acoustic energy the instrument transmits into the water for stage measurements. The difference between the highest level and the lowest level is about 20dB. Values are: 0 = High 1 = HighLow 2 = LowHigh 3 = Low

Default
1 (HighLow)
	Visual C++®
	`short GetStagePowerLevel()`
	`void SetStagePowerLevel(short nNewValue)`

StartOnSynch	
Scope Vector/Vectrino Category Advanced	Description Enables or disables start data collection by external trigger signal. Values are: 0 = OFF 1 = ON

Default
0 (OFF)
	Visual C++®
	`short GetStartOnSynch()`
	`void SetStartOnSynch(short nNewValue)`

SynchMaster	
Scope Vector Category Standard	Description Enable for instruments acting as a master in a synchronization setup. Values are: 0 = OFF 1 = ON

Default
1 (ON)
	Visual C++®
	`short GetSynchMaster()`
	`void SetSynchMaster(short nNewValue)`

TemperatureMode	
Scope All Instruments	Description Speed of sound can be set by the user (Fixed) or calculated by the instrument based on the measured or fixed temperature and a user-input value for salinity (Measured) Values are: 0 = Measured 1 = Fixed
Category Vector/Vectrino Standard All others Advanced	

Default
0 (Measured)
	Visual C++®
	`short GetTemperatureMode()`
	`void SetTemperatureMode(short nNewValue)`

Timeout	
Scope All instruments	Description Instrument command response timeout in milliseconds.
Category Communication	

Default
1000 m/s
	Visual C++®
	`short GetTimeout()`
	`void SetTimeout(short nNewValue)`

TransmitLength	
Scope Vector/Vectrino	Description The effect of increasing the transmit length is that the signal-to-noise ratio is increased. When changing the transmit length the available sampling volumes sizes is also changed. Values are: 0 = 2 mm 1 = 4 mm
Category Advanced (Vector) Standard (Vectrino)	

Default
1 (4 mm)
	Visual C++®
	`short GetTransmitLength()`
	`void SetTransmitLength(short nNewValue)`

TriggerOut	
Scope Vector	Description Sets the output sync signal type for external devices. Values are: 0 = Vector 1 = Other
Category Advanced	

Default
0 (Vector)
	Visual C++®
	`short GetTriggerOut()`
	`void SetTriggerOut(short nNewValue)`

VelRange	
Scope Vector/Vectrino	Description Nominal velocity range should be set to cover the range of the velocities anticipated during the deployment. A higher velocity range give more noise in the data and vice versa. See the User Guide for more information. Values are: Vector 0 = 7.0 m/s, 1 = 4.0 m/s, 2 = 2.0 m/s, 3 = 1.0 m/s, 4 = 0.3 m/s, 5 = 0.1 m/s, 6 = 0.01 m/s Vectrino 0 = 4.0 m/s, 1 = 2.5 m/s, 2 = 1.0 m/s, 3 = 0.3 m/s, 4 = 0.1 m/s, 5 = 0.03 m/s
Category Standard	

Default
Vector 4 (0.3 m/s)
Vectrino 3 (0.3 m/s)
	Visual C++®
	`short GetVelRange()`
	`void SetVelRange(short nNewValue)`

WaveASTThreshold	
Scope AWAC AST	Description This parameter is used in conjunction with Find First Peak, and specifies the minimum acoustic return to identify the free surface. This is a parameter has no units of measurement, but 100 represents the default value for most cases. Applies to AST systems only.
Category Advanced	

Default
100
	Visual C++®
	`short GetWaveASTThreshold()`
	`void SetWaveASTThreshold(short nNewValue)`

WaveASTSUV	
Scope AWAC AST	Description Enables a wave data collection mode to be used for deployment on a subsurface buoy. It allows for the instrument to rotate and move laterally. Values are: 0 = OFF 1 = ON
Category Standard	

Default
0 (OFF)
	Visual C++®
	`short GetWaveASTSUV()`
	`void SetWaveASTSUV(short nNewValue)`

WaveASTWindowSize	
Scope AWAC AST	Description This is the size of the AST window and should be designed such that the free surface is contained within it. Applies to AST systems only.
Category Advanced	

Default
10.0 m
	Visual C++®
	`short GetWaveASTWindowSize()`
	`void SetWaveASTWindowSize(short nNewValue)`

WaveBursts	
Scope Aquapro, AWAC	Description Enables or disables wave measurements. Values are: 0 = OFF 1 = ON
Category Standard	

Default
1 (ON) if AWAC
0 (OFF) if Aquapro
	Visual C++®
	`short GetWaveBursts()`
	`void SetWaveBursts(short nNewValue)`

WaveCellPosition	
Scope AWAC (non AST)	Description This is the vertical position (distance from the instrument head) of the depth cell used for wave velocity measurements. Applies to AST systems only.
Category Advanced	

Default
15.0 m
	Visual C++®
	`float GetWaveCellPosition()`
	`void SetWaveCellPosition(float newValue)`

WaveCellSize	
Scope Aquapro, AWAC	Description The cell size specifies the vertical length in metres of the depth cell used for wave spectra velocity measurements.
Category Advanced	

Default	Visual C++®
	`float GetWaveCellSize()`
	`void SetWaveCellSize(float newValue)`

WaveFindPeak	
Scope AWAC AST	Description The standard method for finding the free surface is to use the maximum return from AST time series. Using this method chooses a first peak in the AST time series over a specified threshold (AST threshold). Applies to AST systems only.
Category Advanced	

Default
0 (Off)
	Visual C++®
	`short GetWaveFindPeak()`
	`void SetWaveFindPeak(short nNewValue)`

WaveInterval	
Scope Aquapro, AWAC	Description Sets how often the instrument will collect wave data (seconds).
Category Standard	

Default
3600
	Visual C++®
	`long GetWaveInterval()`
	`void SetWaveInterval(long nNewValue)`

WaveSamples	
Scope Aquapro, AWAC	Description Sets the number of data samples to collect for wave spectra computations.
Category Standard	

Default
1024
	Visual C++®
	`short GetWaveSamples()`
	`void SetWaveSamples(short nNewValue)`

WaveSamplingRate	
Scope Aquapro, AWAC	Description Sets the sampling rate to use for wave data collection. Values are: 0 = 1Hz 1 = 2Hz.
Category Standard	

Default
0 (1 Hz)
	Visual C++®
	`short GetWaveSamplingRate()`
	`void SetWaveSamplingRate(short nNewValue)`

WaveStaticMode	
Scope AWAC	Description This mode is chosen over the automatic mode when the user wants to have a fixed positioned velocity cell position or AST window. One example deployment would be at a location with no tidal variations and wave heights that are small. Generally speaking this mode is rarely used. Values are: 0 = Off 1 = On
Category Advanced	

Default
0 (Off)
	Visual C++®
	`short GetWaveStaticMode()`
	`void SetWaveStaticMode(short nNewValue)`

WaveASTWindowStart	
Scope AWAC AST	Description This is where the AST window begins to measure, referenced from the centre transducer in the direction of the centre beam; (beam 4). Applies to AST systems only.
Category Advanced	

Default
15.0 m
	Visual C++®
	`short GetWaveASTWindowStart()`
	`void SetWaveASTWindowStart(short nNewValue)`

Methods

The following methods are used to control the instrument data collection.

AquireData	
Scope	Description
AWAC	Starts a single ware burst measurement
	Visual C++® `BOOL AquireData()`

Connect	
Scope	Description
All instruments	Connects to the instrument and checks its status. Returns TRUE if successful. Call this method prior to any other method!
	Visual C++® `BOOL Connect()`

DialModem	
Scope	Description
All instruments	Dials the phone number. Returns TRUE if successful.
	Visual C++® `BOOL DialModem()`

Disconnect	
Scope	Description
All instruments	Disconnects from the instrument. Returns TRUE if successful.
	Visual C++® `BOOL Disconnect()`

DownloadData	
Scope	Description
All instruments, except Vectrino	Download (dumps) data from the recorder. Returns recorder file stop address (end of file) strFileName - name of disk file nStartAddress - recorder address to start from (most recent data) 0 = read all data
	Visual C++® `long DownloadData(LPCTSTR strFileName, long nStartAddress, BOOL bAppend)`

EmptyInBuffer	
Scope	Description
All instruments	Clears the serial port input buffer.
	Visual C++® `BOOL EmptyInBuffer(long nTimeout)`

EraseRecorder	
Scope	Description
All instruments, except Vector/Vectrino	Erases all recorder files
	Visual C++® `BOOL EraseRecorder()`

GetAmp	
Scope	Description
All instruments	Gets the most recent amplitude data for the specified cell and beam (counts). Cell and beam numbering start at 1.
	Visual C++® `short GetAmp(short nCell, short nBeam)`

GetAmplitude	
Scope	Description
All instruments	Gets the most recent amplitude data for the specified cell number. Returns TRUE if successful.
	Visual C++® `BOOL GetAmplitude(short nCell, short FAR* nAmp1, short FAR* nAmp2, short FAR* nAmp3)`

GetAnalogInput	
Scope	Description
All instruments, except Vectrino	Gets the most recent analogue input data for the specified channel (counts). Channel is 1 or 2.
	Visual C++® `short GetAnalogInput(short nChannel)`

GetAnalogInputs	
Scope	Description
All instruments, except Vectrino	Gets the most recent analog input data. Returns TRUE if successful.
	Visual C++® `BOOL GetAnalogInputs(short FAR* nVal1, short FAR* nVal2)`

GetAnalogRangeCount	
Scope	Description
Vector, Vectrino	Returns the number of available choices for the Analogue Output Range. See TransmitLength property and GetTransmitLengthValue method.
	Visual C++® `short GetAnalogRangeCount()`

GetAnalogRangeValue	
Scope	Description
Vector, Vectrino	Returns the Analogue Output Range value corresponding to the index. See AnalogRange property and GetAnalogRangeCount method.
	Visual C++® `float GetAnalogRangeValue(short nIndex)`

GetBattery	
Scope	Description
All instruments	Gets the most recent battery voltage (V).
	Visual C++® `float GetBattery()`

GetClock	
Scope	Description
All instruments	Returns the current date and time of the RTC in the instrument. Returns 0 if not successful.
	Visual C++® `DATE GetClock()`

GetCompassCalib	
Scope	Description
All instruments except Vectrino	Gets the compass calibration results.
	Visual C++® `BOOL GetCompassCalib(float FAR* fOffX, float FAR* fOffY, float FAR* fK11, float FAR* fK22, float FAR* fMaxError)`

GetConfig	
Scope	Description
All instruments	Reads the configuration (properties) from the instrument. Returns TRUE if successful.
	Visual C++® `BOOL GetConfig()`

GetCorr	
Scope	Description
Vector, Vectrino	Returns the most recent correlation data for the specified cell and beam number.
	Visual C++® `short GetCorr(short nCell, short nBeam)`

GetDataBlock	
Scope	Description
All instruments	Returns the most recent measurement data structure as a VARIANT array. See the Paradopp system integrators manual for a description of the binary data structures.
	Visual C++® `VARIANT GetDataBlock(short hType)`

GetDateTime	
Scope	Description
All instruments	Get date and time for the most recent data.
	Visual C++® `DATE GetDateTime()`

GetError	
Scope	Description
All instruments	Gets the most recent error word (V).
	Visual C++® `short GetErrorWord()`

GetErrorCode	
Scope	Description
All instruments	Gets the most recent error code.
	Visual C++® `short GetErrorCode()`

GetErrorMessage	
Scope	Description
All instruments	Gets the most recent error message.
	Visual C++® `BSTR GetErrorMessage()`

GetFileInfo	
Scope	Description
All instruments	Gets recorder file information
	Visual C++® `BOOL GetFileInfo(short nIndex, BSTR FAR* strFileName, long FAR* nSize, BSTR FAR* strComment, DATE FAR* dateTime)`

GetFileName	
Scope	Description
All instruments, except Vectrino	Returns the recorder filename
	Visual C++® `BSTR GetFileName(short nIndex)`

GetFirmwareVersion	
Scope	Description
All instruments	Returns the firmware version. Must be called after Connect() which reads the hardware and software configuration from the instrument.
	Visual C++® `BSTR GetFirmwareVersion()`

GetFirstFile	
Scope	Description
All instruments, except Vectrino	Finds the first data file in the recorder and returns the file position number to be used in consecutive calls to GetNextFile. Returns –1 if there are no files on the recorder.
	Visual C++® `short GetFirstFile(BSTR FAR* strFileName)`

GetCommHandle	
Scope	Description
All instruments	Returns the Windows API serial port handle for the instrument connection.
	Visual C++® `long GetCommHandle()`

GetHeading	
Scope	Description
All instruments, except Vectrino	Gets the most recent compass heading (°).
	Visual C++® `float GetHeading()`

GetHeadSerialNo	
Scope	Description
All instruments	Returns the head serial number.
	Visual C++® `BSTR GetHeadSerialNo()`

GetHeadConf	
Scope	Description
All instruments	Returns the head configuration structure as a VARIANT array. See the Paradopp system integrators manual for a description of the binary data structures
	Visual C++® `VARIANT GetHeadConf()`

GetHorizontalVelPrec	
Scope	Description
All instruments	Returns the horizontal velocity precision.
	Visual C++® `float GetHorizontalVelPrec()`

GetId	
Scope	Description
All instruments	Returns the instrument head ID.
	Visual C++® `BSTR GetID()`

GetInstrument	
Scope	Description
All instruments	Returns the instrument type. 0 = AQUADOPP 1 = VECTOR 2 = AQUAPRO 3 = AWAC 4 = EASYQ 5 = CONTINENTAL 6 = VECTRINO
	Visual C++® `SHORT GetInstrument()`

GetLastError	
Scope	Description
All instruments	Gets the most recent error message string.
	Visual C++® `BSTR GetLastError()`

GetLastMeasStatus	
Scope	Description
All instruments	Returns the most recent status word.
	Visual C++® `short GetLastMeasStatus()`

GetLastMessage	
Scope	Description
All instruments	Returns the last error message as a string. If the Messages property is ON all errors are reported by the PdCommX control. If Messages is OFF, the application can use GetLastMessage to control the message pop-up itself.
	Visual C++® `BSTR GetLastMessage()`

GetMemoryRequired	
Scope	Description
All instruments	Returns the memory required for the deployment configuration.
	Visual C++® `float GetMemoryRequired()`

GetModemReply	
Scope	Description
All instruments	Returns the modem response string.
	Visual C++® `BSTR GetModemReply()`

GetNextFile	
Scope	Description
All instruments, except Vectrino	Finds the next data file in the recorder and returns the file position number to be used in consecutive calls to GetNextFile.

	Visual C++®
	`short GetNextFile(short nPos,`
	`BSTR FAR* strFileName)`

GetNoiseAmp	
Scope	Description
All instruments	Gets the noise amplitude for the specified beam (counts). Beam numbering starts at 1.
	Visual C++® `short GetNoiseAmp(short nBeam)`

GetNoiseLevel	
Scope	Description
All instruments	Gets the instrument noise level measurement (counts).
	Visual C++® `boolean GetNoiseLevel(short* nAmp1, short* nAmp2, short* nAmp3, short* nAmp4)`

GetNumBeams	
Scope	Description
All instruments	Returns the number of acoustic beams
	Visual C++® `short GetNumBeams()`

GetNumFiles	
Scope	Description
All instruments; Except Vectrino	Returns the number of files in the instrument recorder.
	Visual C++® `short GetNumFiles()`

GetPitch	
Scope	Description
All instruments, except Vectrino	Gets the most recent pitch measurement (°).
	Visual C++® `float GetPitch()`

GetPowerConsumption	
Scope	Description
All instruments	Returns the battery utilization for the deployment configuration.
	Visual C++® `float GetPowerConsumption()`

GetPressure	
Scope	Description
All instruments, except Vectrino	Gets the most recent pressure data.
	Visual C++® `float GetPressure()`

GetProdConf	
Scope	Description
All instruments	Returns the hardware configuration structure as a VARIANT array. See the Paradopp system integrators manual for a description of the binary data structures
	Visual C++® `VARIANT GetProdConf()`

GetRangeNBeams	
Scope	Description
EasyQ	Gets the number of beams used for range measurement.
	Visual C++® `short GetRangeNBeams()`

GetRoll	
Scope	Description
All instruments, except Vectrino	Gets the most recent roll measurement (°).
	Visual C++® `float GetRoll()`

GetSamplingVolumeCount	
Scope	Description
Vector, Vectrino	Returns the number of available choices for the Sampling Volume. See SamplingVolume property and GetSamplingVolumeValue method.
	Visual C++® `short GetSamplingVolumeCount()`

GetSamplingVolumeValue	
Scope	Description
Vector, Vectrino	Returns the Sampling Volume corresponding to the Sampling Volume and Transmit Length indices See SamplingVolume property and GetSamplingVolumeCount method.

	Visual C++®
	`float GetSamplingVolumeValue(short nSVIndex,`
	`short nTLIndex)`

GetSensors	
Scope	Description
All instruments, except Vector	Gets the most recent sensors data. Returns TRUE if successful.
	Visual C++® `BOOL GetSensors(float FAR* fTemperature, float FAR* fPitch, float FAR* fRoll, float FAR* fHeading)`

GetSerialNo	
Scope	Description
All instruments	Returns the instrument ID string.
	Visual C++® `BSTR GetSerialNo()`

GetSNR	
Scope	Description
All instruments	Gets the most recent SNR data for the specified cell number. Returns TRUE if successful.
	Visual C++® `BOOL GetSNR(short nCell, float FAR* fB1, float FAR* fB2, float FAR* fB3)`

GetSSpeed	
Scope	Description
All instruments	Gets the most recent sound speed used (m/s).
	Visual C++® `float GetSSpeed()`

GetStage	
Scope	Description
EasyQ	Gets the most recent stage measurement (m).
	Visual C++® `float GetStage()`

GetStageP	
Scope	Description
EasyQ	Gets the most recent stage measurement (m).
	Visual C++® `float GetStageP()`

GetStageAndQuality	
Scope	Description
EasyQ	Gets the most recent stage (m) and quality measurements.
	Visual C++®
	`boolean GetStageAndQuality(float* fStage, short* nQuality, float* fStageP, short* nQualityP)`

GetStageBlanking	
Scope	Description
EasyQ	Gets the most recent stage blanking distance (m) (EasyQ).
	Visual C++®
	`float GetStageBlanking()`

GetStageQuality	
Scope	Description
EasyQ	Gets the most recent quality number.
	Visual C++®
	`short GetStageQuality()`

GetStageQualityP	
Scope	Description
EasyQ	Gets the most recent quality number.
	Visual C++®
	`short GetStageQualityP()`

GetStatus	
Scope	Description
All instruments	Gets the most recent status data. Returns TRUE if successful.
	Visual C++®
	`BOOL GetStatus(float FAR* fSoundSpeed, float FAR* fBattery, short FAR* nError, short FAR* nStatus)`

GetStatusWord	
Scope	Description
All instruments	Gets the most recent status word.
	Visual C++®
	`short GetStatusWord()`

GetTemperature	
Scope	Description
All instruments	Gets the most recent temperature measurement (°C).
	Visual C++® `float GetTemperature()`

GetTransmitLengthCount	
Scope	Description
Vector, Vectrino	Returns the number of available choices for the Transmit Length.
	Visual C++® `short GetTransmitLengthCount()`

GetTransmitLengthValue	
Scope	Description
All Vector, Vectrino	Returns the Sampling Volume corresponding to the index.
	Visual C++® `float GetTransmitLengthValue(short nIndex)`

GetUserConf	
Scope	Description
All instruments	Returns the user configuration structure as a VARIANT array. See the Paradopp system integrators manual for a description of the binary data structures
	Visual C++® `VARIANT GetUserConf()`

GetVarAmplitude	
Scope	Description
All instruments	Gets the most recent amplitude data as a variant type array.
	Visual C++® `VARIANT GetVarAmplitude()`

GetVarAnalogInputs	
Scope	Description
All instruments, except Vectrino	Gets the most recent analogue input data as a variant type array.
	Visual C++® `VARIANT GetVarAnalogInputs()`

GetVarCorrelation	
Scope	Description
Vector, Vectrino	Returns the most recent correlation data as a variant type array.
	Visual C++® `VARIANT GetVarCorrelation()`

GetVarNoiseLevel	
Scope	Description
All instruments	Gets the instrument noise level measurement as a variant type array (counts).
	Visual C++® `VARIANT GetVarNoiseLevel()`

GetVarSensors	
Scope	Description
All instruments, except Vectrino	Gets the most recent sensors data as a variant type array.
	Visual C++® `VARIANT GetVarSensors()`

GetVarSNR	
Scope	Description
All instruments	Gets the most recent SNR data as a variant type array.
	Visual C++® `VARIANT GetVarSNR()`

GetVarStage	
Scope	Description
EasyQ	Gets the most recent stage measurement (m) as a variant type array.
	Visual C++® `VARIANT GetVarStage()`

GetVarStatus	
Scope	Description
All instruments, except Vectrino	Gets the most recent status data as a variant type array.
	Visual C++® `VARIANT GetVarStatus()`

GetVarVelocity	
Scope	Description
All instruments	Gets the most recent velocity data as a variant type array.
	Visual C++® `VARIANT GetVarVelocity()`

GetVarWaveAmplitude	
Scope	Description
AWAC, Aquapro	Gets the most recent wave burst amplitude data as a variant type array (counts).
	Visual C++® `VARIANT GetVarWaveAmplitude()`

GetVarWaveVelocity	
Scope	Description
AWAC, Aquapro	Gets the most recent wave burst velocity data (m/s) as a variant type array.
	Visual C++® `VARIANT GetVarWaveVelocity()`

GetVel	
Scope	Description
All instruments	Gets the most recent velocity data for the specified cell and beam (m/s). Cell and beam numbering start at 1
	Visual C++® `float GetVel(short nCell, short nBeam)`

GetVelocity	
Scope	Description
All instruments	Gets the most recent velocity data for the specified cell number. Returns TRUE if successful.

	Visual C++®
	`BOOL GetVelocity(short nCell,`
	`float FAR* fVel1XE, float FAR* fVel2YN,`
	`float FAR* fVel3ZU)`

GetVerticalVelPrec	
Scope	Description
All instruments	Returns the vertical velocity precision.
	Visual C++® `float GetVerticalVelPrec()`

GetWaveAmp	
Scope	Description
AWAC, Aquapro	Gets the most recent wave burst amplitude data (counts) for the specified beam. Beam numbering starts at 1.
	Visual C++® `short GetWaveAmp(short nBeam)`

GetWaveAmplitude	
Scope	Description
AWAC, Aquapro	Gets the most recent wave burst amplitude data (counts). Returns TRUE if successful.

	Visual C++®
	`BOOL GetWaveAmplitude(short FAR* nAmp1, short FAR* nAmp2, short FAR* nAmp3,`
	`short FAR* nAmp4)`

GetWaveAnalogInput	
Scope	Description
AWAC, Aquapro	Returns the most recent analogue input reading collected with the wave burst data.
	Visual C++® `short GetWaveAnalogInput();`

GetWaveDateTime	
Scope	Description
AWAC, Aquapro	Gets Date and Time for the most recent wave burst measurement.
	Visual C++® `DATE GetWaveDateTime()`

GetWaveDistance	
Scope	Description
AWAC with AST	Gets the most recent wave AST distance measurement (m).
	Visual C++® `float GetWaveDistance()`

GetWaveDistance2	
Scope	Description
AWAC with AST	Gets the most recent wave AST distance2 measurement (m).
	Visual C++® `float GetWaveDistance2()`

GetWavePressure	
Scope	Description
AWAC, Aquapro	Gets the most recent wave burst pressure data (m).
	Visual C++® `float GetWavePressure()`

GetWaveQuality	
Scope	Description
AWAC with AST	Gets the most recent wave AST quality number (counts).
	Visual C++® `short GetWaveQuality()`

GetWaveVel	
Scope	Description
AWAC, Aquapro	Gets the most recent wave burst velocity data (m/s) for the specified beam. Beam numbering starts at 1.
	Visual C++® `float GetWaveVel(short nBeam)`

GetWaveVelocity	
Scope	Description
AWAC, Aquapro	Gets the most recent wave burst velocity data (m/s). Returns TRUE if successful.
	Visual C++® `BOOL GetWaveVelocity(float FAR* fVel1, float FAR* fVel2, float FAR* fVel3, float FAR* fVel4)`

HangUpModem	
Scope	Description
All instruments	Hangs up the modem. Returns TRUE if successful.
	Visual C++® `BOOL HangUpModem()`

InitializeModem	
Scope	Description
All instruments	Initializes the modem. Returns TRUE if successful.
	Visual C++® `BOOL InitializeModem()`

InquireState	
Scope	Description
All instruments	Returns the instrument state. Values are 0 = Firmware upgrade mode, 1 = Measurement mode, 2 = Command mode, 4 = Data retrieval mode, 5 = Confirmation mode. Returns –1 if unknown or not successful.
	Visual C++® `short InquireState()`

IsASTFirmware	
Scope	Description
AWAC	Returns TRUE if the instrument is running the AST firmware version.
	Visual C++® `BOOL IsASTFirmware()`

IsConnected	
Scope	Description
All instruments	Returns TRUE if connected.
	Visual C++® `BOOL IsConnected()`

IsModemOnline	
Scope	Description
All instruments	Returns TRUE if the modem is online.
	Visual C++® `BOOL IsModemOnline()`

IsVMFirmware	
Scope	Description
AWAC.Continental	Returns TRUE if the instrument is running the Vessel Mounted firmware version.
	Visual C++® `BOOL IsVMFirmware()`

LoadDeployment	
Scope	Description
All instruments, except Vectrino	Loads the specified deployment (.dep) file and configures the instrument. Deployment files are instrument configuration files generated and saved during the standard instrument software.
	Visual C++® `boolean LoadDeployment(BSTR strFileName)`

ReadFile	
Scope	Description
All instruments, except Vectrino	Downloads a data file to the specified disk filename. Performs a CRC check on the data if bCRC = TRUE. Returns TRUE if successful.
	Visual C++® `BOOL ReadFile(short nPos, LPCTSTR strDiskFileName, BOOL bCRC)`

SaveDeployment	
Scope	Description
All instruments	Saves the deployment configuration to file.
	Visual C++® `BOOL SaveDeployment(LPCTSTR strFileName)`

SendModemCommand	
Scope	Description
All instruments	Sends a modem command string. Returns TRUE if successful. Use **GetModemReply** to get the modem response.
	Visual C++® `BOOL SendModemCommand(LPCTSTR strCommand)`

SetClock	
Scope	Description
All instruments	Sets the date and time of the RTC in the instrument. Returns TRUE if successful.
	Visual C++® `BOOL SetClock(DATE dateTime)`

SetConfig	
Scope	Description
All instruments	Writes the configuration (properties) to the instrument. Returns TRUE if successful. Note that for most of the property settings to take effect, the SETCONFIG method must be called prior to Start command. See also CONNECT.
	Visual C++® `BOOL SetConfig()`

SetInstrumentBaudRate	
Scope	Description
All instruments	Sets the instrument baudrate. Set bSave = TRUE to make permanent.
	Visual C++® `BOOL SetInstrumentBaudRate(long nBaudRate, BOOL bSave)`

SetInstrumentDelay	
Scope	Description
All instruments	Sets the instrument serial comms delay (msec).
	Visual C++® `BOOL SetInstrumentDelay(short nDelay, BOOL bSave)`

SetPressureOffset	
Scope	Description
All instruments	Sets the pressure sensor offset (depth in meters).
	Visual C++® `float SetPressureOffset(float fDepth)`

ShowProgress	
Scope	Description
All instruments	Shows or hides the progress status box during lengthly operations.
	Visual C++® `void ShowProgress(BOOL bShow)`

Start	
Scope	Description
All instruments	Starts online measurements based on the current configuration of the instrument. If bRecorder = TRUE data is stored to a new file in the recorder. Returns TRUE if successful. See also SetConfig (above).
	Visual C++® `BOOL Start(BOOL bRecorder)`

StartCompassCalib	
Scope	Description
All instruments except Vectrino	Starts compass calibration.
	Visual C++® `BOOL StartCompassCalib(BOOL bPitchRollTransformation, BOOL bResetScaling)`

StartDeployment	
Scope	Description
All instruments, except Vectrino	Starts a deployment based on the current configuration of the instrument. Data is stored to a new file in the recorder. Returns TRUE if successful.
	Visual C++® `BOOL StartDeployment()`

StartDistanceCheck	
Scope	Description
Vectrino	Starts distance measurements.
	Visual C++® `BOOL StartDistanceCheck()`

StartDiskRecording	
Scope	Description
All instruments	Starts data recording to disk. Specify the filename without extension. If AutoName = TRUE a new file will be opened for data recording each time the specified time interval has elapsed. The current date and time is then automatically added to the filename. Returns TRUE if successful.
	Visual C++® `BOOL StartDiskRecording(CString strFileName, BOOL bAutoName)`

StartRangeCheck	
Scope	Description
EasyQ	Starts the range check mode to verify that the signal strength is sufficient for the specified blanking distance and cell size.
	Visual C++® `BOOL StartRangeCheck()`

StartStageCheck	
Scope	Description
EasyQ	Starts the stage check mode.
	Visual C++® `BOOL StartStageCheck()`

Stop	
Scope	Description
All instruments	Stops online measurements and deployment data recording. Returns TRUE if successful.
	Visual C++® `BOOL Stop()`

StopCompassCalib	
Scope	Description
All instruments except Vectrino	Stops compass calibration.
	Visual C++® `BOOL StopCompassCalib()`

StopDiskRecording	
Scope	Description
All instruments	Stops data recording to disk.
	Visual C++®
	`BOOL StopDiskRecording()`

UpdateCompassCalib	
Scope	Description
All instruments except Vectrino	Saves the compass calibration results to instrument.
	Visual C++®
	`BOOL UpdateCompassCalib()`

ValidateConfig	
Scope	Description
All instruments	Returns TRUE if the instrument configuration is valid.
	Visual C++®
	`BOOL ValidateConfig()`

Code Examples

Implementation

Implementation examples are provided in Visual C++®, Visual Basic® and Delphi™.

Visual C++®

```
BOOL COCXTestDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
   .
   .
   .

   m_pdcommx.SetMeasInterval(1);
   m_pdcommx.SetAvgInterval(1);
   m_pdcommx.SetDiagnosticsMode(0) ;

   return TRUE;
}

void COCXTestDlg::OnBtnStartstop()
{
   CString str;

   m_btnStartStop.GetWindowText(str);
   if (str == "&Start") {
      m_pdcommx.Connect();
      if (m_pdcommx.InquireState() != 2) {    // not in command mode
         m_pdcommx.Stop();
         m_pdcommx.Disconnect();
         return;
      }
      if (!m_pdcommx.SetConfig()) {
         m_pdcommx.Stop();
         m_pdcommx.Disconnect();
         return;
      }
      if (!m_pdcommx.Start(FALSE)) {
         m_pdcommx.Stop();
         m_pdcommx.Disconnect();
         return;
```

```
        }
        m _ btnStartStop.SetWindowText("&Stop");
    }
    else {
        m _ pdcommx.Stop();
        m _ pdcommx.Disconnect();
        m _ btnStartStop.SetWindowText("&Start");
    }
}

BEGIN _ EVENTSINK _ MAP(COCXTestDlg, CDialog)
    //{{AFX _ EVENTSINK _ MAP(COCXTestDlg)
      ON _ EVENT(COCXTestDlg,  IDC _ PDCOMMXCTRL,  1  /*  OnNewData  */,
OnOnNewDataPdcommxctrl, VTS _ I2)
    //}}AFX _ EVENTSINK _ MAP
END _ EVENTSINK _ MAP()

void COCXTestDlg::OnOnNewDataPdcommxctrl(short hType)
{
    float dVel[3];
    float dTemperature, dPitch, dRoll, dHeading, dPressure;
    COleDateTime odtTime;
    short nAmp[3];

    odtTime = m _ pdcommx.GetDateTime();
    m _ pdcommx.GetVelocity(0, &dVel[0], &dVel[1], &dVel[2]);
    m _ pdcommx.GetAmplitude(0, &nAmp[0], &nAmp[1], &nAmp[2]);
    dPressure = m _ pdcommx.GetPressure();
    m _ pdcommx.GetSensors(&dTemperature, &dPitch, &dRoll, &dHeading);

    m _ strVel1.Format("%.2f",dVel[0]);
    m _ strVel2.Format("%.2f",dVel[1]);
    m _ strVel3.Format("%.2f",dVel[2]);
    m _ strAmp1.Format("%d",nAmp[0]);
    m _ strAmp2.Format("%d",nAmp[1]);
    m _ strAmp3.Format("%d",nAmp[2]);

    m _ strTemp.Format("%.1f",dTemperature);
    m _ strPitch.Format("%.1f",dPitch);
    m _ strRoll.Format("%.1f",dRoll);
    m _ strHeading.Format("%.1f",dHeading);

    m _ strPressure.Format("%.1f",dPressure);
    m _ strTime = odtTime.Format();

    UpdateData(FALSE);
}
```

Visual Basic®

```
Dim DateTime As Date
Dim Vel(3) As Single
Dim Amp(3) As Integer
```

```
Dim Temperature As Single
Dim Pitch As Single
Dim Roll As Single
Dim Heading As Single
Dim Pressure As Single

Private Sub cmdStartStop _ Click()
If InStr(cmdStartStop.Caption, "Start") > 0 Then
    pdcommx.Connect
    If pdcommx.InquireState <> 2 Then  'not in command mode
        pdcommx.Stop
        pdcommx.Disconnect
        Exit Sub
    End If
    'initialize instrument
    If Not pdcommx.SetConfig Then
        pdcommx.Stop
        pdcommx.Disconnect
        Exit Sub
    End If
    If Not pdcommx.Start(False) Then
        pdcommx.Stop
        pdcommx.Disconnect
        Exit Sub
    End If
    cmdStartStop.Caption = "&Stop"
Else
    pdcommx.Stop
    pdcommx.Disconnect
    cmdStartStop.Caption = "Start"
End If
End Sub

Private Sub Form _ Load()
    pdcommx.MeasInterval = 1
    pdcommx.AvgInterval = 1
    pdcommx.DiagnosticsMode = DiagOff
    ' more properties .... or use property window
End Sub

Private Sub pdcommx _ OnNewData(ByVal hType As Integer)
    DateTime = pdcommx.GetDateTime
    Call pdcommx.GetVelocity(0, Vel(1), Vel(2), Vel(3))
    Call pdcommx.GetAmplitude(0, Amp(1), Amp(2), Amp(3))
    Pressure = pdcommx.GetPressure
    Call pdcommx.GetSensors(Temperature, Pitch, Roll, Heading)

    lblVel1.Caption = Str(Vel(1))
    lblVel2.Caption = Str(Vel(2))
    lblVel3.Caption = Str(Vel(3))
    lblAmp1.Caption = Str(Amp(1))
    lblAmp2.Caption = Str(Amp(2))
    lblAmp3.Caption = Str(Amp(3))

    lblTemp.Caption = Str(Temperature)
```

```
    lblPitch.Caption = Str(Pitch)
    lblRoll.Caption = Str(Roll)
    lblHead.Caption = Str(Heading)

    lblPress.Caption = Str(Pressure)
    lblTime.Caption = Str(DateTime)
End Sub
```

Delphi™

```
var
  OCXTestForm: TOCXTestForm;
  DateTime: TDateTime;
  Vel: array[1..3] of Single;
  Amp: array[1..3] of Smallint;
  Temperature, Pitch, Roll, Heading, Pressure: Single;
  I, J, K: Smallint;

procedure TOCXTestForm.FormCreate(Sender: TObject);
begin
  PdCommX.MeasInterval := 1;
  PdCommX.AvgInterval := 1;
  PdCommX.DiagnosticsMode := 0;
  // more properties .... or use property window
end;

procedure TOCXTestForm.btnStartStopClick(Sender: TObject);
begin
  if btnStartStop.Caption = '&Start' then
    begin
      PdCommX.Connect;
      if PdCommX.InquireState <> 2 then  // not in command mode
        begin
          PdCommX.Stop;
          PdCommX.Disconnect;
          Exit;
        end;

      // initialize instrument
      if PdCommX.SetConfig <> True then
        begin
          PdCommX.Stop;
          PdCommX.Disconnect;
          Exit;
        end;

      if PdCommX.Start(0) = 0 then
        begin
          PdCommX.Stop;
          PdCOmmX.Disconnect;
          Exit;
        end;
```

```
      btnStartStop.Caption := '&Stop'
    end
  else
    begin
      PdCommX.Stop;
      PdCommX.Disconnect;
      btnStartStop.Caption := '&Start';
    end;
end;

procedure TOCXTestForm.NewData(Sender: TObject; hType: Smallint);
begin
  DateTime := PdCommX.GetDateTime;
  PdCommX.GetVelocity(0, Vel[1], Vel[2], Vel[3]);
  PdCommX.GetAmplitude(0, Amp[1], Amp[2], Amp[3]);
  Pressure := PdCommX.GetPressure;
  PdCommX.GetSensors(Temperature, Pitch, Roll, Heading);

  edtVel1.Text := FloatToStrF(Vel[1],ffFixed,7,2);
  edtVel2.Text := FloatToStrF(Vel[2],ffFixed,7,2);
  edtVel3.Text := FloatToStrF(Vel[3],ffFixed,7,2);
  edtAmp1.Text := IntToStr(Amp[1]);
  edtAmp2.Text := IntToStr(Amp[2]);
  edtAmp3.Text := IntToStr(Amp[3]);

  edtTemp.Text := FloatToStrF(Temperature,ffFixed,7,1);
  edtPitch.Text := FloatToStrF(Pitch,ffFixed,7,1);
  edtRoll.Text := FloatToStrF(Roll,ffFixed,7,1);
  edtHead.Text := FloatToStrF(Heading,ffFixed,7,1);

  edtPress.Text := FloatToStrF(Pressure,ffFixed,7,1);
  edtTime.Text := DateTimeToStr(DateTime);
end;

end.
```