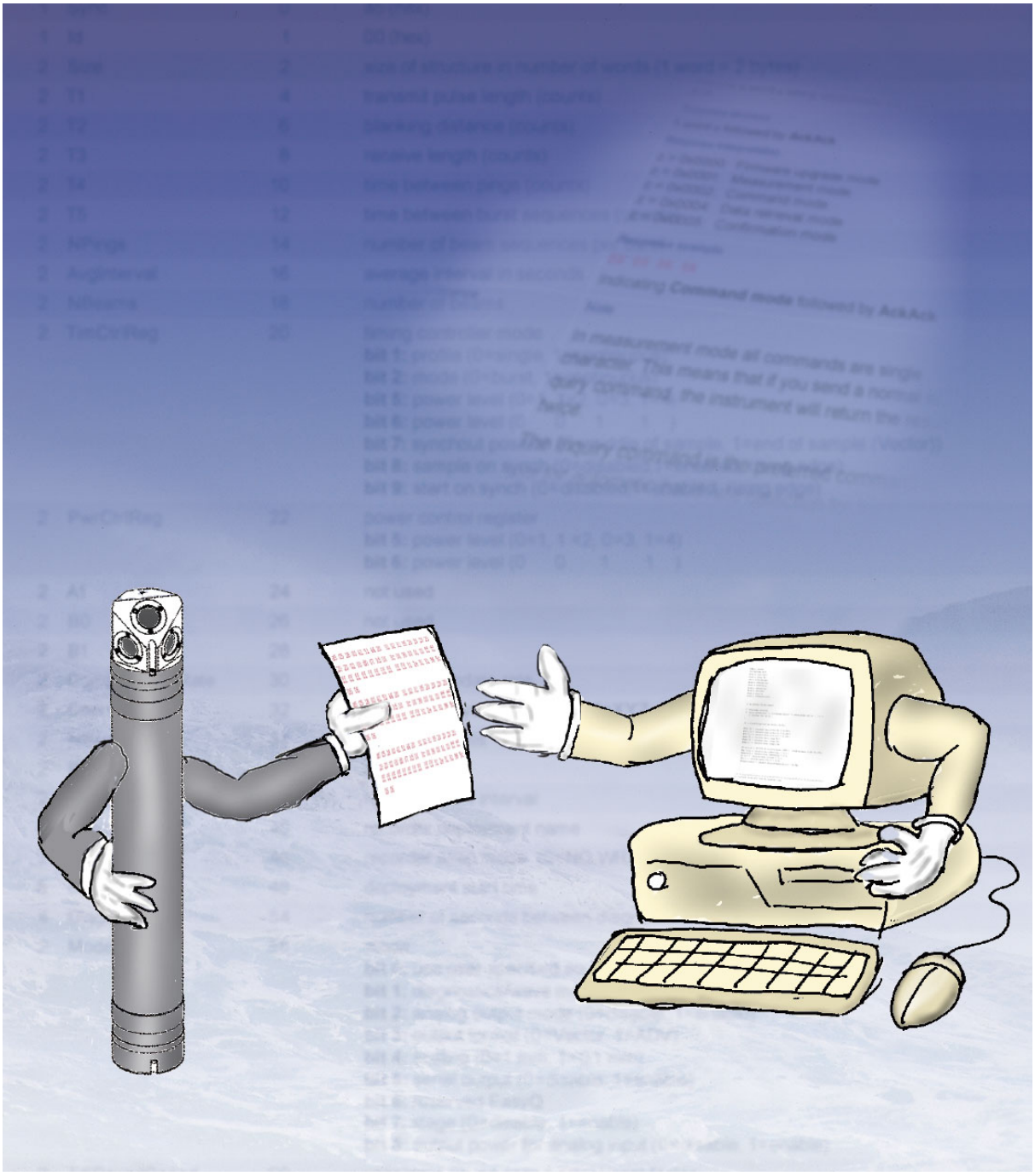
NORTEK AS

# SYSTEM INTEGRATOR GUIDE



## PARADOPP FAMILY OF PRODUCTS

OCTOBER 2004

N3001-101 • Rev. F

# SYSTEM INTEGRATOR MANUAL

# PARADOPP FAMILY OF PRODUCTS



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

Tel: +47 6717 4500 • Fax: +47 6713 6770 • e-mail: inquiry@nortek.no • www.nortek-as.com

NORTEK AS

## Software updates and technical support

Find us on the world wide web:

**www.nortek-as.com, www.nortek.no**

Here you will find software updates and technical support.

## Your Feedback is appreciated

If you find errors, misspelled words, omissions or sections poorly explained, please do not hesitate to contact us and tell us about it at:

**inquiry@nortek.no**

We appreciate your comments and your fellow users will as well.

## Nortek Forum Support

If you have comments, application tips, suggestions to improvements, etc. that you think will be of general interest you should register on Nortek's Forums at

**www.nortek-as.com/cgi-bin/ib/ikonboard.cgi**

and post your message there. The Forums also offer a great opportunity to share your experience using Nortek sensors with other users around the world, and to learn from their experience.

## Communicating with us

If you need more information, support or other assistance from us, 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**

# OVERVIEW

## What's in this Manual?

### Chapter 1 – Introduction

Here we introduce you to the Paradopp documentation and outlines the scope of this manual.

### Chapter 2 – Basic Interface Concepts

This chapter introduces you to just that, but also the fact that there are two types of break commands available. Here we tell which to use when.

### Chapter 3 – Use with a Controller

This chapter provides tips and tricks when setting up your Nortek instrument with a controller.

### Chapter 4 – Remote Control Terminal Commands

To control a Nortek instrument from remote, commands are needed. This chapter provides the remote control commands and their syntax.

### Chapter 5 – Data Structures

This chapter interprets the data received from your Nortek instrument.

## Chapter 6 – ASCII Output

The Aquadopp single current meter and the Continental are capable of outputting data directly in ASCII format. This chapter outlines the feature.

## Chapter 7 – SDI-12 Interface

The EasyQ and EasyV are available with SDI-12 interface as an option. This chapter presents SDI-12 basics and what you need to get data out of the EasyQ and EasyV.

## Chapter 8 – Program Examples

For your conveninence, we are pleased to provide a few example programs – all written in C.

# DETAILS

## The Table of Contents

# CHAPTER 1
# Introduction

This document aims at providing the information needed to control a Nortek Paradopp product (Aquadopp, Vector, etc.) from a non-PC controller. It is aimed at system integrators and engineers with interfacing experience. Examples are provided in C. The scope is limited to interfacing issues and the document does not address the general performance issues of the systems. For a more thorough understanding of the principle of operation, we recommend the User Guides for the individual instrument.

The document is not complete in the sense that it describes all available commands or modes of communication. For most users, it will make sense to let the Nortek Win32® software do most of the hardware configuration and then let the controller limit its task to starting/stopping data collection. For more in-depth information about specific commands, we urge you to contact Nortek to discuss how your particular problem is best solved. This is also the case for those who write Win32® applications to control one or more Paradopp products – an ActiveX® object is available to simplify the interfacing and to send/receive the complete data structures.

Before you start, please observe that the Paradopp products use binary format for communication. This makes it hard to "see" what is going on with a terminal emulator. On the other hand, however, the binary interface saves programming time because parsing won't be needed. Consequently, it may take you a little longer to get things up and running, but you will definitely move faster once the basics are in place. The natural use of check sums and CRC also contributes to make the binary interface a more robust choice.

As always, these types of documents are subject to changes. We recommend that you contact Nortek to make sure you have the latest version before embarking

on the job. We would also appreciate your comments regarding information you find missing or sections that should be expanded. Our general e-mail address is mailto:inquiry@nortek.no and we encourage you to use it.

## ActiveX®

The ActiveX®/DLL software interface provides functions to configure the instrument, control the data acquisition process and retrieve data from the recorder. In a DLL implementation C/C++ API calls are made to the Paradopp DLL. A Paradopp OCX implementation requires that the software development environment supports the OCX interface. Visual Basic, Visual C++ and Delphi, are a few environments that support the OCX interface.

The ActiveX® control interface is described in the *Paradopp ActiveX Automation Interface Manual,* which is available separately.

# CHAPTER 2
# Basic Interface Concepts

The Paradopp family of products communicates with a default protocol of 8 data bits, no parity and 1 stop bit. The baud rate is user selectable and can be configured either with the supplied Win32® programs or by using direct commands to the system after the direct communication has been initiated (see the appropriate commands in *Chapter 4*).

The only lines used are RxD, TxD, and GND. Status and handshaking lines are not used.

## The Break Command

A break command is used to change between the various operational modes of the instrument and to interrupt the instrument regardless of which mode it is in. It is used frequently when communicating with the instrument. Consequently, any system designed to control a Paradopp system must be able to send a break.

**The operational modes for any Paradopp system are:**

- **Command mode.** The system is waiting for instruction to come in over the serial line. After 5 minutes idle, the system will go in power down.
- **Power down mode.** This state is used to conserve power. A break must be sent to cause the instrument to leave power down mode.
- **Data collection mode.** The system cycles through a series of states when collecting data. To go out of data collection mode, a break and confirmation string must be sent.

After a power on/off, the system will generally remember what mode it is in.

# Determining Which Type of Break to Use

**Tip:** When you send a break to the instrument, it will respond as follows (using Aquadopp Profiler as an example):

```
AQUAPRO
NORTEK 2003
Version 1.23
Command mode
```

It may happen that a break is sent to the instrument at exactly the same time as the instrument receives an interrupt from its Real Time Clock (RTC). In such cases the RTC is always given priority.

This implies that if you receive no acknowledgement of the break you sent, you will need to send another.

Hence, your controller should be set up to verify that the break actually was received by the instrument before you proceed.

There are two types of break commands in Nortek instruments. These are denoted **soft break** and **hard break**, respectively.

Traditionally, sending a break used to be done by holding the transmit line high on the serial line for period of 500 ms. This is referred to as **hard break**. If you have direct connection between the controlling device and the instrument this will work fine. However, once modems or other devices are inserted in the communication path between the two, problems arise. Certain GSM modems, for example, handle breaks in their very own way. Typically, they have no problems in accepting a 500 ms break on the input, but they output a 100 ms break output only. This may not be accepted as a break by the instrument.

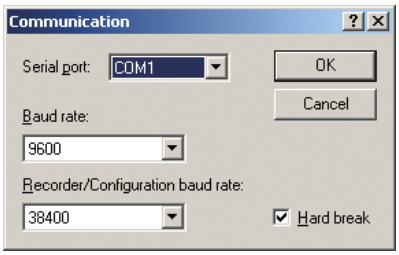
To circumvent this we have introduced the **soft break** principle, which consists entirely of characters. Hence it can be used without problems with any device capable of RS 232 or RS 422 communication.

Which break command to use for a particular instrument depends on the production date of the instrument, and most instruments manufactured after 2001 use soft break.

The easiest way to find out which type of break command to use is to run the Nortek software shipped with the instrument.

### To check which type of break command to use:

1. Run the deployment setup software shipped with the instrument and initiate an autodetect e.g. by clicking **Deployment** > **Planning** > **Load from instrument**.
2. Click **Communication** > **Serial Port** to produce a dialogue box like the one shown below.




The dialogue box used to determine whether to use soft or hard <break> with a particular instrument – see text for details.

3. When the **Hard break** box is checked, hard break is to be used, if unchecked, soft break is to be used.

## Checksum Control

Most data structures contain a 16-bit check sum control. An example program shown in *Chapter 5* shows how the checksum control can be implemented.

## Two-character ASCII Commands

The command interface is based upon two character commands where the two characters shall be treated as a single 16-bit word. The time delay between the two characters in a command must be less than 0.5 second, otherwise both characters will be discarded by the Paradopp. The data is transferred as words and the convention is *Intel-style*, which means that low byte is sent before high byte. The data types are shown explicitly in the section describing the various commands. More about this can be found in *Chapter 4*.

## Acknowledgement

Hexadecimal values are preceded by **0x** to indicate that they are in hexadecimal form.

**Example:** 0x4a2b

After a successful command is sent, the system returns an acknowledgement. The actual value for acknowledge (**AckAck**) is **0x0606**. Whenever the system firmware receives a command/word that is invalid, it immediately returns a negative acknowledge (**NackNack**). The value is **0x1515**.

# CHAPTER 3
# Use with a Controller

This chapter provides tips and tricks when setting up your Nortek instrument with a controller. However, we recommend that you read through the *User Guide* that came with your Nortek instrument before you embark on your controller project.

**Basically, a controller will act in one of the two following ways:**

- As a simple storage unit for the data acquired
- As a device controlling the Nortek instrument's behaviour, with or without transfer to the controller of the data acquired.

Always use the accompanying Nortek software to set up the instrument for deployments.

All Nortek instruments come with deployment software running on the Windows® platform. We strongly recommend that you use this software to set up the instrument properly.

The output signal to the controller is in binary format for all instruments. However, the Aquadopp Current Meter and the Continental can, in addition, output data in ASCII format – see *Chapter 6* for more.

All Nortek instruments are supplied with RS 232 interface unless specified otherwise. For long distance transmission (more than 50 –100 m) we recommend the use of RS 422, which is available as an option for all Paradopp instruments.

For the EasyQ and EasyV type of instruments the SDI-12 interface is also available – see *Chapter 7* for more.

## Using the Controller as a Simple Storage Device

If you decide to use your controller as a simple storage device you will have to make up your mind whether or not to use the internal recorder in addition to the controller. More about the internal recorder feature can be found in the User Guide for the Nortek instrument.

Data output from the Nortek instrument will be properly time stamped as long as the instrument remains powered, so you won't have to implement time stamping in the controller to keep track of the data acquisition.

How to interpret the received data is presented in *Chapter 5*.

## Using the Controller to Control the Instrument

**If you decide to use your controller to control the Nortek instrument, you have two options:**

If you choose to download deployment setups from the controller during deployment, you may want to send a **GA** command every time you download a new setup to store all configuration data together with the coresponding measurement data.

Deployment setups should always be generated by means of the software accompanying your Nortek instrument.

- The controller starts and stops the measurements by turning off the power to the Nortek instrument when not measuring. This opens up for longer deployments. However, this will require that the controller time-stamps the data acquired upon receipt, since Nortek instruments may loose their time information when the power is removed for more than 5 minutes.
- The controller starts and stops the measurements using a combination of a two-character ASCII command and a break command.

In addition you may want to store the data acquired in the controller by reading out the data. Some applications may also require that you download deployment setups from the controller at regular intervals.

For commands to be received and executed, the instrument must be in **Command** mode. If the instrument is in **Power down** mode a break must be sent to wake it up. If, on the other hand, the instrument is in **Data collection mode** (i.e. measuring) a break followed by a confirmation string must be sent. The confirmation string will be the **MC command**, which must be sent within 10 seconds after the break. Otherwise, the instrument will resume its measurement duties.

## Controlling Data Collection by Turning the Power On/Off

Always use the software accompanying your Nortek instrument to generate deployment files. The Nortek software must be set in a special mode to generate binary format deployment files – *see* **CC** command for details.

In this case, the system will automatically start measuring and outputting data when power is applied. To use this method effectively, you must:

- Make sure the appropriate system configuration has been downloaded to the instrument, either from a PC or from the controller.
- Start data collection from the PC or the controller before disconnecting. Once the power is shut down, the instrument will remember that it is in data collection mode and continue to collect data once the power is re-applied.

When using this technique of removing and applying power, the recorder will operate in *Append* mode, if started with the **SR** command (**Start with Recorder**), so that all the data will be recorded to the same file. This opens up for easy verification of the controller since what has been logged to the internal recorder is identical to what has been sent to the controller.

The internal clock may loose the correct time if the power is disconnected for more than 5 minutes.

## Controlling Data Collection Over the Serial Line

In this case, the data collection is controlled over the serial line. To start data collection, a two-character ASCII command is sent. To stop data collection – or to get system out from sleep mode – a break must be sent.

To start a measurement from Command mode, send the command **ST**. The system will send an acknowledge (**AckAck**) to show that the measurement is started. More about this can be found in *Chapter 4*.

### A typical sequence proceeds as follows:

- Send a break command to gain control of the system and put it in Command mode. If the system is busy collecting data (i.e. measuring), a verification is required, otherwise the instrument will not stop measuring. Send the characters **MC** within 10 seconds.
- To start a measurement from Command mode send the command **ST**. (**SR** if you want to also store data in the instrument's recorder – see the list of commands in *Chapter 4* for details)
- To stop data collection, send a break and the verification characters.
- To conserve power between measurement intervals, send the command **PD**.

### A typical session with an Aquadopp will look like this:

| Aquadopp sends | Controller sends | Comments |
|---|---|---|
| | <BREAK> | Aquadopp in power down |
| Aquadopp<br>Nortek AS 2003<br>Version 1.23<br>Command mode<br>AckAck | | In command mode |
| | ST | |
| AckAck | | In measurement mode |
| ....) (*&) (*&) (& | | Outputs binary data |
| &!%#&)*J ASH(#& | | |
| | <BREAK> | |
| Confirm:AckAck | | Waits 10 seconds for confirmation |
| | MC | |
| Aquadopp<br>Nortek AS 2003<br>Version 1.23<br>Command mode<br>AckAck | | Aquadopp is now in command mode |
| | PD | |
| AckAck | | Aquadopp is powered down |

# CHAPTER 4
# Remote Control Commands

**A few terms:**

**RTC:** Real Time Clock
**MSW:** Most Significant Word, bits 31–16 in a 32 bits data field
**LSW:** Least Significant Word, bits 15–0 in a 32 bits data field
**SW:** The software program in the computer or controller
**FW:** The software program in the instrument
**0x:** Indicates hex code

**Low byte before high byte.** When designing computers, there are two different architectures for handling memory storage. They are often called *Big Endian* and *Little Endian* and refer to the order in which the bytes are stored in memory. The Windows series of operating systems has been designed around Little Endian architecture and has not been designed to be compatible with Big Endian because most programs are written with some dependency on Little Endian.

Transmission is Intel style, i.e. low byte must be sent to (and will also be received from) the instrument before high byte.

Consequently, the Hex command code in the instruction tables is shown in reversed order to reflect a *16-bit word* point of view rather than a *per byte* point of view, see also text for details.

The time delay between the two characters in a command must be less than 0.5 second, otherwise both characters will be discarded by the Paradopp.

These two phrases are derived from “Big End In” and “Little End In.” They refer to the way in which memory is stored. On an Intel computer, the little end is stored first. This means a Hex word like 0x1234 is stored in memory as (0x34 0x12). The little end, or lower end, is stored first. The same is true for a four-byte value; for example, 0x12345678 would be stored as (0x78 0x56 0x34 0x12). For this reason we show the Hex values in reversed order in the below tables.

**Example:** The **R** corresponds to **0x52** and **C** to **0x43**. Shown in reversed order (to comply with the **Little Endian** principle) this will read **0x4352**, which is what is listed in the table.

## Remote Control Commands in Command Mode

| Read clock | |
|---|---|
| Execute command<br>**RC** | Reads the current time and date of the RTC in the instrument. |
| Hex<br>**4352** | Response<br>3 words clock data structure followed by **AckAck** |
| | Command parameter required<br>None |
| | Response example<br>28 18 13 11 04 02 06 06 |
| Reference<br>Chapter 5<br>Generic Structures | Note |

| Set clock | |
|---|---|
| Execute command<br>**SC** | Sets the current time and date of the RTC in the instrument. |
| Hex<br>**4353** | Response<br>**AckAck** |
| | Data received<br>3 word clock data structure |
| Reference<br>Chapter 5<br>Generic Structures | Note<br>The setting of the clock is synchronized to the transition of seconds i.e. the FW waits until a second transition has occurred and then sets the clock. |

| Inquiry | |
|---|---|
| Execute command<br>**II** | Returns a word **z** telling which mode the instrument is in. |
| Hex<br>**4949** | Response structure<br>1 word **z** followed by **AckAck** |
| | Response interpretation<br>z = 0x0000: Firmware upgrade mode<br>z = 0x0001: Measurement mode<br>z = 0x0002: Command mode<br>z = 0x0004: Data retrieval mode<br>z = 0x0005: Confirmation mode |
| | Response example<br>02 00 06 06<br>Indicating **Command mode** followed by **AckAck** |
| Reference | Note<br>In measurement mode all commands are single character. This means that if you send a normal inquiry command, the instrument will return the result twice.<br>The Inquiry command is the preferred command to use for automatic baud rate detection for the PC. |

## Set baud rate

Execute command

**BR**

Hex

**5242**

Sets the instrument baud rate.

Response

**AckAck**

Parameter to be sent

**Z**

Parameter structure

| | |
|---|---|
| z = 0x3030 | 300 baud |
| z = 0x3131 | 600 baud |
| z = 0x3232 | 1200 baud |
| z = 0x3333 | 2400 baud |
| z = 0x3434 | 4800 baud |
| z = 0x3535 | 9600 baud |
| z = 0x3636 | 19200 baud |
| z = 0x3737 | 38400 baud |
| z = 0x3838 | 57600 baud |
| z = 0x3939 | 115200 baud |

Example

```
5242 3232 06 06
```

Command for setting **baud rate to 1200 baud** followed by the response **AckAck**

Reference

Note

The baud rate is stored in the instrument only when the recorder is formatted, when a measurement is started, or an **SB** command is sent. This will ensure that the communication can be restored by waiting until the instrument powers down after 5 minutes of inactivity. The PC must make sure that the baud rate being used is sufficiently high to ensure that all data can be transferred over the serial line for the chosen data format and measurement interval. For example, at 300 baud it is only possible to transfer 30 bytes/s, so having a measurement interval of 1 second will not be possible. Using very low baud rates will inevitably have an impact on the power consumption because of the added time needed for data transfer on the serial line before the instrument can power down. For most applications, however, the difference will be negligible.

## Save baud rate

| | |
|---|---|
| Execute command<br>**SB**<br>Hex<br>**4253** | Saves the currently set baud rate.<br>Response<br>**AckAck**<br>Parameter<br>**0x4733 0x3241** |
| Reference | Note<br>ASCII counterpart: 3GA2. If you have set the baud with the **BR** command you must save it afterwards if you want the instrument to wake up with that baud rate after power down. The parameter is shown obeying the low-byte-before-high-byte principle, i.e. in the way it should be sent to the instrument. |

## Read complete configuration data

| | |
|---|---|
| Execute command<br>**GA**<br>Hex<br>**4147** | Reads out the currently used hardware configuration, the head configuration, and the deployment configuration from the instrument<br>Response<br>Complete setup information (48+224+512 bytes) followed by **AckAck**<br>Parameter<br>None<br>Response example |

```
a5 05 18 00 41 51 44 20 31 32 31 35 20 20 20 20
20 20 02 00 d0 07 0d 00 3c 00 90 00 01 00 ff ff
ff ff ff ff ff ff ff ff ff ff 31 2e 31 31 98 5c
a5 04 70 00 0d 00 d0 07 00 00 41 51 50 20 30 38
35 32 20 20 20 00 19 00 19 00 19 00 00 00 3d 19
                        .
                        .
                        .
cd ff 8b 00 e5 00 ee 00 0b 00 84 ff 3d ff 4c 52
06 06
```

| | |
|---|---|
| Reference<br>Chapter 5<br>Generic Structures | Note<br>Some lines in the above example have been removed for clarity. |

## Read deployment configuration data

| | |
|---|---|
| Execute command<br>**GC** | Reads out the currently used deployment configuration from the instrument. |
| Hex<br>**4347** | Response<br>Deployment setup (512 bytes) followed by **AckAck** |
| | Parameter<br>None |
| | Response example<br>a5 00 00 01 5c 00 22 00 18 00 b6 02 00 02 0f 00<br>01 00 03 00 02 00 60 00 00 00 00 00 00 00 01 00<br>01 00 14 00 14 17 01 00 00 00 00 00 00 00 00 00<br>50 01 06 10 04 02 06 00 00 00 20 00 11 41 01 00<br>01 00 0f 00 00 00 00 00 a8 2f 5e 01 ca 3c e6 3c<br>.<br>.<br>.<br>cd ff 8b 00 e5 00 ee 00 0b 00 84 ff 3d ff 4c 52<br>06 06 |
| Reference<br>Chapter 5<br>Generic Structures | Note<br>Some lines in the above example have been removed for clarity. |

## Read hardware configuration data

| | |
|---|---|
| Execute command<br>**GP** | Reads out the currently used hardware configuration from the instrument. |
| Hex<br>**5047** | Response<br>HW setup information (48bytes) followed by **AckAck** |
| | Parameter<br>None |
| | Response example<br>a5 05 18 00 41 51 44 20 31 32 31 35 20 20 20 20<br>20 20 02 00 d0 07 0d 00 3c 00 90 00 01 00 ff ff<br>ff ff ff ff ff ff ff ff ff ff 31 2e 31 31 98 5c<br>06 06 |
| Reference<br>Chapter 5<br>Generic Structures | Note |

| Read head configuration data | |
|---|---|
| Execute command<br>**GH**<br>Hex<br>**4847** | Reads out the currently used head configuration from the instrument<br>Response<br>Head configuration (224 bytes) followed by **AckAck**<br>Parameter<br>None<br>Response example<br>15 15 a5 04 70 00 0d 00 d0 07 00 00 41 51 50 20<br>30 38 35 32 20 20 20 00 19 00 19 00 19 00 00 00<br>.<br>.<br>.<br>38 0e 10 0e 10 0e 10 27 64 00 00 00 00 00 00 00<br>00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00<br>f0 60 06 06 |
| Reference<br>Chapter 5<br>Generic Structures | Note<br>Some lines in the above example have been removed for clarity. |

| Format recorder | |
|---|---|
| Execute command<br>**FO**<br>Hex<br>**4f46** | The format recorder command formats the recorder's memory. This affects the data acquired only. Configuration files – including the deployment setup – are not affected.<br>Response<br>**AckAck**<br>Parameter to be sent<br>**0xd412 0xef1e** |
| Reference | Note<br>This parameter has no ASCII counterpart. The parameter is shown obeying the low-byte-before-high-byte principle, i.e. in the way it should be sent to the instrument. |

## Configure instrument

| | |
|---|---|
| Execute command<br>**CC** | Use this command to download a new deployment file to the instrument. |
| Hex<br>**4343** | Response<br>**AckAck** |
| | Parameter sent out<br>The setup file to be downloaded |
| Reference<br>User guide for your Nortek instrument | Note<br>The command must be followed by a deployment setup file – how to generate this, see below. |

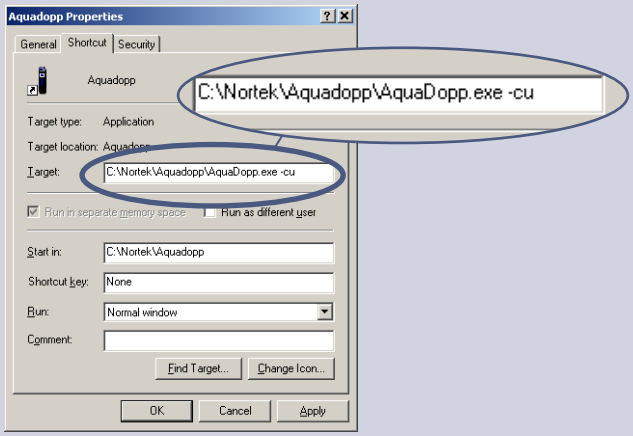
### Generating a Deployment File in Binary Format

Always use the Nortek software accompanying your Nortek instrument when making deployment files. This will save you from a lot of unneeded efforts! When you have generated the file, you may save it. However, this will not generate a file in binary format suitable for direct download to your controller.

**To generate deployment files in binary format:**

1 Generate a new shortcut to the Nortek software.

2 Append the characters -cu in the target line as shown below (using Aquadopp as example).




3 Start the Nortek software using the new shortcut.

4 When you now save the deployment file, this will generate two files – the regular file and a file in binary format with the file extension .pcf. This file is the one to download with your controller.

| Read File Allocation Table (FAT) | |
|---|---|
| Execute command<br>**RF**<br>Hex<br>**4652** | Reads the FAT in the instrument.<br>Response<br>FAT followed by **AckAck**<br>Parameter sent out<br>None |
| Reference<br>Chapter 5<br>Generic Structures | Note<br>The FAT is a 16 byte × 32 lines matrix containing information about names and start & stop addresses of the measurement files. Up to 31 different measurement files can be stored in the instrument's recorder (the last line in the matrix is spare and not used). |

| Read file configuration | |
|---|---|
| Execute command<br>**FC**<br>Hex<br>**4346** | Returns the hardware, head and deployment setup for a particular measurement file.<br>Response<br>Complete setup information (48+224+512 bytes) for the selected file followed by **AckAck**<br>Parameter sent out<br>**0x0000 0x001e** using the file's location in the FAT as index |
| Reference<br>GA command<br>RF command | Note<br>This command corresponds to the **GA** command, but for a specific file. |

| Read recorder data file | |
|---|---|
| Execute command<br>**RD** | Returns the contents of a part of the recorder memory as defined by a start and a stop address. |
| Hex<br>**4452** | Response<br>As many bytes of data as requested followed by **AckAck** |
| | Parameter sent out<br>**<Start address><Stop address>** |
| | Parameter example<br>Assume you want to read in the contents from **0x00002a00** to **0x00003a00**<br>What will be sent is (hex values):<br>4452002a0000003a0000 |
| Reference<br>RF command | Note<br>You will normally pick the Start address and the Stop address from the FAT when programming your controller. However, you may also use this command to transfer only a part of the measurement file. |

| Read recorder data block with CRC | |
|---|---|
| Execute command<br>**DC** | As the **RD** command, but with CRC appended to the data transferred. |
| Hex<br>**4344** | Response<br>As many bytes of data as requested followed by the CRC and **AckAck** |
| | Parameter sent out<br>As for **RD** |
| Reference<br>RD command | Note<br>As for **RD** |

| Battery voltage | |
|---|---|
| Execute command<br>**BV**<br>Hex<br>**5642** | Reads out the battery voltage from the instrument.<br>Response<br>4 bytes followed by **AckAck**<br>Parameter sent out<br>None<br>Response example<br>`a7 1f 06 06` |
| Reference | Note<br>Bearing in mind the low-byte-before-high-byte principle, the response should be interpreted as **0x1fa7** which corresponds to decimal **8103**, which in turn is the voltage directly in mV. |

| Power down | |
|---|---|
| Execute command<br>**PD**<br>Hex<br>**4450** | The Power down command puts the instrument in sleep mode (switches off the power)<br>Response<br>**AckAck**<br>Parameter<br>None |
| Reference | Note |

## Transparent compass

| | |
|---|---|
| Execute command<br>**TC**<br>Hex<br>**4354** | The transparent compass command powers up the compass and makes a transparent channel from the compass to the PC. |
| | Response<br>**AckAck** |
| | Parameter<br>None |
| Reference | Note<br>This command enables the PC to read the data strings output from the compass and to send commands to the compass. Observe that this command sets the baud rate of the instrument to 38400. However, the baud rate is set back to the current instrument baud rate once a break is sent to the instrument from the controller. You can use this command to let the controller to verify that the compass is outputting the correct data. It can also be used to set up the compass to match the required setup for the instrument.<br>**Caution!** The FW will never attempt to change the setup of the compass, so if the controller sends commands to the compass, these must set up the compass correctly before you send a break to end the transparent compass session! |

## Get identification string

| | |
|---|---|
| Execute command<br>**ID**<br>Hex<br>**4449** | Reads out the identification string from the instrument. |
| | Response<br>14 bytes ASCII string followed by **AckAck** |
| | Parameter<br>None |
| | Response example<br>41 51 44 20 31 32 31 35 20 20 20 20 20 20 06 06<br>corresponding to AQD1215 |
| Reference | Note |

| Start measurement without recorder | |
|---|---|
| Execute command<br>**ST**<br>Hex<br>**5453** | Immediately starts a measurement based on the current configuration of the instrument without storing data to the recorder. Data is output on the serial port. |
| | Response<br>**AckAck** or **NackNack** |
| | Parameter<br>None |
| Reference | Note<br>If the measurement was successfully started, **AckAck** is returned. If the measurement could not be started **NackNack** is returned. The reason for failing to start is usually that the instrument configuration is invalid. |

| Acquire data | |
|---|---|
| Execute command<br>**AD**<br>Hex<br>**4441** | Starts a single measurement based on the current configuration of the instrument without storing data to the recorder. Instrument enters Command mode when measurement has been made. |
| | Response<br>**AckAck** or **NackNack** |
| | Parameter<br>None |
| Reference | Note<br>If the measurement was successfully started, **AckAck** is returned. If the measurement could not be started **NackNack** is returned. The reason for failing to start is usually that the instrument configuration is invalid. |

| Start measurement with recorder | |
|---|---|
| Execute command<br>**SR**<br>Hex<br>**5253** | Immediately starts a measurement based on the current configuration of the instrument. Data is stored to a new file in the recorder and also output on the serial port |
| | Response<br>**AckAck** or **NackNack** |
| | Parameter<br>None |
| Reference | Note<br>If the measurement was successfully started, **AckAck** is returned. If the measurement could not be started **NackNack** is returned. The reason for failing to start is usually that the instrument configuration is invalid or that the recorder is full. |

| Start measurement with recorder, but with no output on serial port | |
|---|---|
| Execute command<br>**SD**<br>Hex<br>**4453** | Starts a measurement at a specified time based on the current configuration of the instrument. Data is stored to a new file in the recorder. Data is not output on the serial port. |
| | Response<br>**AckAck** or **NackNack** |
| | Parameter<br>None |
| Reference | Note<br>If the measurement was successfully started, **AckAck** is returned. If the measurement could not be started **NackNack** is returned. The reason for failing to start is usually that the instrument configuration is invalid or that the recorder is full. |

# Remote Control Commands in Data Collection Mode

The commands available in data collection mode are all single character commands. Before sending these commands, the controller must transmit a character with binary value 0x00 or the character **@** and then wait for 100 ms before sending the command.

The idea behind the commands in data collection mode is to allow the controller to find out where the instrument is in its measurement cycle. It is thus possible to interrogate the system without disturbing the data collection.

The inquiry (*see* **II** command) is present in all modes. In data collection mode only one character 'I' is used (*see* overleaf). If the standard inquiry command is sent ('II'), the system will send the response twice.

## Time remaining of average interval

Execute command

**A**

Hex

**41**

This command returns a word that indicates the number of seconds left of the average or burst interval.

Response

A 16-bit word followed by **AckAck**

Parameter

None

Response example

`0x1e00 0606`

Reference

Note

The commands available in data collection mode are all single character commands. Before sending these commands, the controller must transmit a character with binary value 0x00.

## Time remaining of measurement interval

Execute command

**M**

Hex

**4D**

This command returns a word that indicates the number of seconds left of the measurement interval.

Response

A 16-bit word followed by **AckAck**

Parameter

None

Response example

`0x1c01 0606`

Reference

Note

The commands available in data collection mode are all single character commands. Before sending these commands, the controller must transmit a character with binary value 0x00.

| Inquiry | |
|---|---|
| Execute command<br>**I**<br>Hex<br>**49** | Returns a word **z** telling which mode the instrument is in.<br>Response<br>See **II** command<br>Parameter<br>See **II** command |
| Reference<br>See **II** command | Note<br>The commands available in data collection mode are all single character commands. Before sending these commands, the controller must transmit a character with binary value 0x00. |

## Confirmation Commands

| Enter Command mode | |
|---|---|
| Execute command<br>**MC**<br>Hex<br>**434d** | Preceded by a break command, this command is sent to force the instrument to exit Data Collection mode and enter Command mode.<br>Response<br>**AckAck**<br>Parameter<br>None |
| Reference | Note<br>The MC command must be sent within 10 seconds after the break was sent. Otherwise the measurement will go on uninterrupted.<br>Within 2 seconds after **AckAck** is sent, the instrument will enter Command mode |

# CHAPTER 5
# Firmware Data Structures

**Tip:** See Chapter 8 for data structures in a form easy to copy and paste into your own programs when reading this manual on screen as a PDF document.

This section describes the data structures that are used for the Paradopp products. They are grouped in generic data structures that are common to all instruments and instrument specific structures.

**The following firmware data structures are described:**

- Generic Structures
- Aquadopp-specifc Structures
- Vector-specific Structures
- Aquadopp Profiler-specific Structures
- AWAC-specific Structures
- Continental
- EasyQ-specific Structures

## Generic structures

### CLOCK DATA (6 bytes)

**BCD format.** The BCD format is a a way of representing decimal figures in a binary manner. Four bits are used per digit.

**Example** (using clock data):
10 00 05 11 04 02
(all values shown in hex)
corresponds to:
**5 February 2004 11:10:00**.

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Minute | 0 | minute (BCD format) |
| 1 | Second | 1 | second (BCD format) |
| 1 | Day | 2 | day (BCD format) |
| 1 | Hour | 3 | hour (BCD format) |
| 1 | Year | 4 | year (BCD format) |
| 1 | Month | 5 | month (BCD format) |

## FILE ALLOCATION TABLE (16×32 bytes)

| Byte 0 | Byte 1 | Byte 2 | Byte 3 | Byte 4 | Byte 5 | Byte 6 | Byte 7 | Byte 8 | Byte 9 | Byte 10 | Byte 11 | Byte 12 | Byte 13 | Byte 14 | Byte 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| File name (ASCII) of file #0 | | | | | | Seq | Status | Start address | | | | Stop address | | | |
| File name (ASCII) of file #1 | | | | | | Seq | Status | Start address | | | | Stop address | | | |
| File name (ASCII) of file #2 | | | | | | Seq | Status | Start address | | | | Stop address | | | |
| ⋮ | | | | | | | | | | | | | | | |
| File name (ASCII) of file #29 | | | | | | Seq | Status | Start address | | | | Stop address | | | |
| File name (ASCII) of file #30 | | | | | | Seq | Status | Start address | | | | Stop address | | | |
| Not used | | | | | | | | | | | | | | | |

**Seq:** If several files share the same file name, the sequence byte will be used to append a number to the file name to discriminate between them. **Example:** Multiple use of the file name ANTHON will produce ANTHON1, ANTHON2 etc., in which 1 and 2 are the contents of the Sequence byte (added automatically).

**Status:** If bit 0 (the LSB) has been set to 1, file wrapping has been enabled. If bit 1 has been set to 1, a complete wrap-around has occurred, i.e. all the data initially stored has been overwritten at least once.

**Start address:** The start address of the measured data

**Stop address:** The stop address of the measured data

Altogether a maximum of 31 different measurement files may be stored in the instrument's internal recorder. The length of these files depends on the amount of memory installed.

# Aquadopp specific structures

## AQUADOPP VELOCITY DATA (42 bytes)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 01 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 1 | Minute | 4 | minute (BCD) |
| 1 | Second | 5 | second (BCD) |
| 1 | Day | 6 | day (BCD) |
| 1 | Hour | 7 | hour (BCD) |
| 1 | Year | 8 | year (BCD) |
| 1 | Month | 9 | month (BCD) |
| 2 | Error | 10 | error code |
| 2 | AnaIn1 | 12 | analog input 1 |
| 2 | Battery | 14 | battery voltage (0.1 V) |
| 2 | SoundSpeed/AnaIn2 | 16 | speed of sound (0.1 m/s) or analog input 2 |
| 2 | Heading | 18 | compass heading (0.1°) |
| 2 | Pitch | 20 | compass pitch (0.1°) |
| 2 | Roll | 22 | compass roll (0.1°) |
| 1 | PressureMSB | 24 | pressure MSB (mm) (Pressure = 65536×PressureMSB + PressureLSW) |
| 1 | Status | 25 | status code |
| 2 | PressureLSW | 26 | pressure LSW (mm) (Pressure = 65536×PressureMSB + PressureLSW) |
| 2 | Temperature | 28 | temperature (0.01 °C) |
| 2 | Vel B1/X/E | 30 | velocity beam1 or X or East coordinates (mm/s) |
| 2 | Vel B2/Y/N | 32 | velocity beam2 or Y or North coordinates (mm/s) |
| 2 | Vel B3/Z/U | 34 | velocity beam3 or Z or Up coordinates (mm/s) |
| 1 | Amp B1 | 36 | amplitude beam1 (counts) |
| 1 | Amp B2 | 37 | amplitude beam2 (counts) |
| 1 | Amp B3 | 38 | amplitude beam3 (counts) |
| 1 | Fill | 39 | fill byte |
| 2 | Checksum | 40 | = b58c(hex) + sum of all bytes in structure |

# Aquadopp specific structures *cont…*

## AQUADOPP DIAGNOSTICS DATA HEADER (36 bytes)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 06 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 2 | Records | 4 | number of diagnostics samples to follow |
| 2 | Cell | 6 | cell number of stored diagnostics data |
| 1 | Noise1 | 8 | noise amplitude beam 1 (counts) |
| 1 | Noise2 | 9 | noise amplitude beam 2 (counts) |
| 1 | Noise3 | 10 | noise amplitude beam 3 (counts) |
| 1 | Noise4 | 11 | noise amplitude beam 4 (counts) |
| 2 | ProcMagn1 | 12 | processing magnitude beam 1 |
| 2 | ProcMagn2 | 14 | processing magnitude beam 2 |
| 2 | ProcMagn3 | 16 | processing magnitude beam 3 |
| 2 | ProcMagn4 | 18 | processing magnitude beam 4 |
| 2 | Distance1 | 20 | distance beam 1 |
| 2 | Distance2 | 22 | distance beam 2 |
| 2 | Distance3 | 24 | distance beam 3 |
| 2 | Distance | 26 | distance beam 4 |
| 6 | Spare | 28 | spare |
| 2 | Checksum | 34 | = b58c(hex) + sum of all bytes in structure |

## AQUADOPP DIAGNOSTICS DATA (42 bytes)

Same as Aquadopp Velocity Data, except Id = 0x80.

# Vector specific structures

## VECTOR VELOCITY DATA HEADER (42 bytes)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 12 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 1 | Minute | 4 | minute (BCD) |
| 1 | Second | 5 | second (BCD) |
| 1 | Day | 6 | day (BCD) |
| 1 | Hour | 7 | hour (BCD) |
| 1 | Year | 8 | year (BCD) |
| 1 | Month | 9 | month (BCD) |
| 2 | NRecords | 10 | number of velocity samples to follow |
| 1 | Noise1 | 12 | noise amplitude beam 1 (counts) |
| 1 | Noise2 | 13 | noise amplitude beam 2 (counts) |
| 1 | Noise3 | 14 | noise amplitude beam 3 (counts) |
| 1 | Noise4 | 15 | noise amplitude beam 4 (counts) |
| 1 | Correlation | 16 | noise correlation beam 1 |
| 1 | Correlation | 17 | noise correlation beam 2 |
| 1 | Correlation | 18 | noise correlation beam 3 |
| 1 | Correlation | 19 | noise correlation beam 4 |
| 20 | Spare | 20 | spare |
| 2 | Checksum | 40 | = b58c(hex) + sum of all bytes in structure |

# Vector specific structures *cont…*

## VECTOR VELOCITY DATA (24 bytes)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 10 (hex) |
| 1 | AnaIn2LSB | 2 | analog input 2 LSB |
| 1 | Count | 3 | ensemble counter |
| 1 | PressureMSB | 4 | pressure MSB (mm) (Pressure = 65536×PressureMSB + PressureLSW) |
| 1 | AnaIn2MSB | 5 | analog input 2 MSB |
| 2 | PressureLSW | 6 | pressure LSW (mm) (Pressure = 65536×PressureMSB + PressureLSW) |
| 2 | AnaIn1 | 8 | analog input 1 |
| 2 | Vel B1/X/E | 10 | velocity beam1 or X or East (mm/s) |
| 2 | Vel B2/Y/N | 12 | velocity beam2 or Y or North (mm/s) |
| 2 | Vel B3/Z/U | 14 | velocity beam3 or Z or Up (mm/s) |
| 1 | Amp B1 | 16 | amplitude beam1 (counts) |
| 1 | Amp B2 | 17 | amplitude beam2 (counts) |
| 1 | Amp B3 | 18 | amplitude beam3 (counts) |
| 1 | Corr B1 | 19 | correlation beam1 (%) |
| 1 | Corr B2 | 20 | correlation beam2 (%) |
| 1 | Corr B3 | 21 | correlation beam3 (%) |
| 2 | Checksum | 22 | = b58c(hex) + sum of all bytes in structure |

# Vector specific structures *cont…*

## VECTOR SYSTEM DATA (28 bytes)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 11 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 1 | Minute | 4 | minute (BCD) |
| 1 | Second | 5 | second (BCD) |
| 1 | Day | 6 | day (BCD) |
| 1 | Hour | 7 | hour (BCD) |
| 1 | Year | 8 | year (BCD) |
| 1 | Month | 9 | month (BCD) |
| 2 | Battery | 10 | battery voltage (0.1 V) |
| 2 | SoundSpeed | 12 | speed of sound (0.1 m/s) |
| 2 | Heading | 14 | compass heading (0.1 deg) |
| 2 | Pitch | 16 | compass pitch (0.1 deg) |
| 2 | Roll | 18 | compass roll (0.1 deg) |
| 2 | Temperature | 20 | temperature (0.01 deg C) |
| 1 | Error | 22 | error code |
| 1 | Status | 23 | status code |
| 2 | AnaIn | 24 | analog input |
| 2 | Checksum | 26 | = b58c(hex) + sum of all bytes in structure |

# Aquadopp Profiler specific structures

## AQUADOPP PROFILER VELOCITY DATA (variable length)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 21 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 1 | Minute | 4 | minute (BCD) |
| 1 | Second | 5 | second (BCD) |
| 1 | Day | 6 | day (BCD) |
| 1 | Hour | 7 | hour (BCD) |
| 1 | Year | 8 | year (BCD) |
| 1 | Month | 9 | month (BCD) |
| 2 | Error | 10 | error code |
| 2 | AnaIn1 | 12 | analog input 1 |
| 2 | Battery | 14 | battery voltage (0.1 V) |
| 2 | SoundSpeed/AnaIn2 | 16 | speed of sound (0.1 m/s) or analog input 2 |
| 2 | Heading | 18 | compass heading (0.1°) |
| 2 | Pitch | 20 | compass pitch (0.1°) |
| 2 | Roll | 22 | compass roll (0.1°) |
| 1 | PressureMSB | 24 | pressure MSB (mm) (Pressure = 65536×PressureMSB + PressureLSW) |
| 1 | Status | 25 | status code |
| 2 | PressureLSW | 26 | pressure LSW (mm) (Pressure = 65536×PressureMSB + PressureLSW) |
| 2 | Temperature | 28 | temperature (0.01 °C) |
| 2 | Vel 1 B1/X/E | 30 | velocity cell 1, beam1 or X or East (mm/s) |
| 2.. | Vel 2⋯n | 32⋯ | ...repeated for cells 2 through n |
| 2 | Vel 1 B2/Y/N | | velocity cell 1, beam2 or Y or North (mm/s) |
| 2.. | Vel 2⋯n | | ...repeated for cells 2 through n |
| 2 | Vel 1 B3/Z/U | | velocity cell 1, beam3 or Z or Up (mm/s) |
| 2.. | Vel 2⋯n | | ...repeated for cells 2 through n |
| 1 | Amp 1 B1 | | amplitude cell 1, beam1 (counts) |
| 1.. | Amp 2⋯n | | ...repeated for cells 2 through n |
| 1 | Amp 1 B2 | | amplitude cell 1, beam2 (counts) |
| 1.. | Amp 2⋯n | | ...repeated for cells 2 through n |
| 1 | Amp 1 B3 | | amplitude cell 1, beam3 (counts) |
| 1.. | Amp 2⋯n | | ...repeated for cells 2 through n |
| 1 | Fill | | fill byte if number of cells mod 2 is not equal to 0 |
| 2 | Checksum | | = b58c(hex) + sum of all bytes in structure |

# AWAC specific structures

## AWAC VELOCITY PROFILE DATA (variable length)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 20 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 1 | Minute | 4 | minute (BCD) |
| 1 | Second | 5 | second (BCD) |
| 1 | Day | 6 | day (BCD) |
| 1 | Hour | 7 | hour (BCD) |
| 1 | Year | 8 | year (BCD) |
| 1 | Month | 9 | month (BCD) |
| 2 | Error | 10 | error code |
| 2 | AnaIn1 | 12 | analog input 1 |
| 2 | Battery | 14 | battery voltage (0.1 V) |
| 2 | SoundSpeed/AnaIn2 | 16 | speed of sound (0.1 m/s) or analog input 2 |
| 2 | Heading | 18 | compass heading (0.1°) |
| 2 | Pitch | 20 | compass pitch (0.1°) |
| 2 | Roll | 22 | compass roll (0.1°) |
| 1 | PressureMSB | 24 | pressure MSB (mm)<br>(Pressure = 65536×PressureMSB + PressureLSW) |
| 1 | Status | 25 | status code |
| 2 | PressureLSW | 26 | pressure LSW (mm)<br>(Pressure = 65536×PressureMSB + PressureLSW) |
| 2 | Temperature | 28 | temperature (0.01 °C) |
| 88 | Spare | 30 | spare |
| 2 | Vel 1 B1/X/E | 118 | velocity cell 1, beam1 or X or East (mm/s) |
| 2.. | Vel 2⋯n | 120 | ...repeated for cells 2 through n |
| 2 | Vel 1 B2/Y/N | | velocity cell 1, beam2 or Y or North (mm/s) |
| 2.. | Vel 2⋯n | | ...repeated for cells 2 through n |
| 2 | Vel 1 B3/Z/U | | velocity cell 1, beam3 or Z or Up (mm/s) |
| 2.. | Vel 2⋯n | | ...repeated for cells 2 through n |
| 1 | Amp 1 B1 | | amplitude cell 1, beam1 (counts) |
| 1.. | Amp 2⋯n | | ...repeated for cells 2 through n |
| 1 | Amp 1 B2 | | amplitude cell 1, beam2 (counts) |
| 1.. | Amp 2⋯n | | ...repeated for cells 2 through n |
| 1 | Amp 1 B3 | | amplitude cell 1, beam3 (counts) |
| 1.. | Amp 2⋯n | | ...repeated for cells 2 through n |
| 1 | Fill | | fill byte if number of cells mod 2 is not equal to 0 |
| 2 | Checksum | | = b58c(hex) + sum of all bytes in structure |

# AWAC specific structures *cont…*

## AWAC WAVE DATA HEADER (60 bytes)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 31 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 1 | Minute | 4 | minute (BCD) |
| 1 | Second | 5 | second (BCD) |
| 1 | Day | 6 | day (BCD) |
| 1 | Hour | 7 | hour (BCD) |
| 1 | Year | 8 | year (BCD) |
| 1 | Month | 9 | month (BCD) |
| 2 | NRecords | 10 | number of wave data records to follow |
| 2 | Blanking | 12 | blanking distance (counts) |
| 2 | Battery | 14 | battery voltage (0.1 V) |
| 2 | SoundSpeed | 16 | speed of sound (0.1 m/s) |
| 2 | Heading | 18 | compass heading (0.1°) |
| 2 | Pitch | 20 | compass pitch (0.1°) |
| 2 | Roll | 22 | compass roll (0.1°) |
| 2 | MinPress | 24 | min pressure value of previous profile (mm) |
| 2 | hMaxPress | 26 | max pressure value of previous profile (mm) |
| 2 | Temperature | 28 | temperature (0.01 °C) |
| 2 | CellSize | 30 | cell size in counts of T3 |
| 1 | Noise1 | 32 | noise amplitude beam 1 (counts) |
| 1 | Noise2 | 33 | noise amplitude beam 2 (counts) |
| 1 | Noise3 | 34 | noise amplitude beam 3 (counts) |
| 1 | Noise4 | 35 | noise amplitude beam 4 (counts) |
| 2 | ProcMagn1 | 36 | processing magnitude beam 1 |
| 2 | ProcMagn2 | 38 | processing magnitude beam 2 |
| 2 | ProcMagn3 | 40 | processing magnitude beam 3 |
| 2 | ProcMagn4 | 42 | processing magnitude beam 4 |
| 14 | Spare | 44 | spare |
| 2 | Checksum | 58 | = b58c(hex) + sum of all bytes in structure |

# AWAC specific structures *cont…*

## AWAC WAVE DATA (24 bytes)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 30 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 2 | Pressure | 4 | pressure (mm) |
| 2 | Distance | 6 | distance to vertical beam |
| 2 | AnaIn | 8 | analog input |
| 2 | Vel1 | 10 | velocity beam 1 (mm/s) |
| 2 | Vel2 | 12 | velocity beam 2 (mm/s) |
| 2 | Vel3 | 14 | velocity beam 3 (mm/s) |
| 2 | Vel4 | 16 | velocity beam 4 (mm/s) |
| 1 | Amp1 | 18 | amplitude beam 1 (mm/s) |
| 1 | Amp2 | 19 | amplitude beam 2 (mm/s) |
| 1 | Amp3 | 20 | amplitude beam 3 (mm/s) |
| 1 | Amp4 | 21 | amplitude beam 4 (mm/s) |
| 2 | Checksum | 22 | = b58c(hex) + sum of all bytes in structure |

## CONTINENTAL DATA

Same as AWAC Profiler Data, except ID = 0x24

# EasyQ specific structures

## EASYQ VELOCITY DATA (58 bytes)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 40 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 1 | Minute | 4 | minute (BCD) |
| 1 | Second | 5 | second (BCD) |
| 1 | Day | 6 | day (BCD) |
| 1 | Hour | 7 | hour (BCD) |
| 1 | Year | 8 | year (BCD) |
| 1 | Month | 9 | month (BCD) |
| 2 | Error | 10 | error code |
| 2 | Temperature | 12 | temperature (0.01 °C) |
| 2 | Battery | 14 | battery voltage (0.1 V) |
| 2 | SoundSpeed | 16 | speed of sound (0.1 m/s) |
| 1 | Spare | 18 | spare |
| 1 | Status | 19 | status code |
| 8 | Spare | 20 | spare |
| 2 | Vel 1 B1/X/E | 28 | velocity cell 1, beam1 or X or East (mm/s) |
| 2 | Vel 2 B1/X/E | 30 | velocity cell 2, beam1 or X or East (mm/s) |
| 2 | Vel 3 B1/X/E | 32 | velocity cell 3, beam1 or X or East (mm/s) |
| 2 | Vel 1 B2/Y/N | 34 | velocity cell 1, beam2 or Y or North (mm/s) |
| 2 | Vel 2 B2/Y/N | 36 | velocity cell 2, beam2 or Y or North (mm/s) |
| 2 | Vel 3 B2/Y/N | 38 | velocity cell 3, beam2 or Y or North (mm/s) |
| 2 | Vel 1 B3/Z/U | 40 | velocity cell 1, beam3 or Z or Up (mm/s) |
| 2 | Vel 2 B3/Z/U | 42 | velocity cell 2, beam3 or Z or Up (mm/s) |
| 2 | Vel 3 B3/Z/U | 44 | velocity cell 3, beam3 or Z or Up (mm/s) |
| 1 | Amp 1 B1 | 46 | amplitude cell no 1, beam1 (counts) |
| 1 | Amp 2 B1 | 47 | amplitude cell no 2, beam1 (counts) |
| 1 | Amp 3 B1 | 48 | amplitude cell no 3, beam1 (counts) |
| 1 | Amp 1 B2 | 49 | amplitude cell no 1, beam2 (counts) |
| 1 | Amp 2 B2 | 50 | amplitude cell no 2, beam2 (counts) |
| 1 | Amp 3 B2 | 51 | amplitude cell no 3, beam2 (counts) |
| 1 | Amp 1 B3 | 52 | amplitude cell no 1, beam3 (counts) |
| 1 | Amp 2 B3 | 53 | amplitude cell no 2, beam3 (counts) |
| 1 | Amp 3 B3 | 54 | amplitude cell no 3, beam3 (counts) |
| 1 | Fill | 55 | fill |
| 2 | Checksum | 56 | = b58c(hex) + sum of all bytes in structure |

# EasyQ specific structures *cont...*

## EASYQ RANGE CHECK DATA (46 bytes)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 41 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 1 | Minute | 4 | minute (BCD) |
| 1 | Second | 5 | second (BCD) |
| 1 | Day | 6 | day (BCD) |
| 1 | Hour | 7 | hour (BCD) |
| 1 | Year | 8 | year (BCD) |
| 1 | Month | 9 | month (BCD) |
| 2 | Beams | 10 | number of beams |
| 2 | Blanking | 12 | blanking distance (counts) |
| 1 | Noise B1 | 14 | noise amplitude beam 1 (counts) |
| 1 | Noise B2 | 15 | noise amplitude beam 2 (counts) |
| 1 | Noise B3 | 16 | noise amplitude beam 3 (counts) |
| 1 | Noise B4 | 17 | noise amplitude beam 4 (counts) |
| 14 | Spare | 18 | spare |
| 1 | Amp 1 B1 | 32 | amplitude cell 1, beam1 (counts) |
| 1 | Amp 2 B1 | 33 | amplitude cell 2, beam1 (counts) |
| 1 | Amp 3 B1 | 34 | amplitude cell 3, beam1 (counts) |
| 1 | Amp 1 B2 | 35 | amplitude cell 1, beam2 (counts) |
| 1 | Amp 2 B2 | 36 | amplitude cell 2, beam2 (counts) |
| 1 | Amp 3 B2 | 37 | amplitude cell 3, beam2 (counts) |
| 1 | Amp 1 B3 | 38 | amplitude cell 1, beam3 (counts) |
| 1 | Amp 2 B3 | 39 | amplitude cell 2, beam3 (counts) |
| 1 | Amp 3 B3 | 40 | amplitude cell 3, beam3 (counts) |
| 1 | Amp 1 B4 | 41 | amplitude cell 1, beam4 (counts) |
| 1 | Amp 2 B4 | 42 | amplitude cell 2, beam4 (counts) |
| 1 | Amp 3 B4 | 43 | amplitude cell 3, beam4 (counts) |
| 2 | Checksum | 44 | = b58c(hex) + sum of all bytes in structure |

# EasyQ specific structures *cont…*

## EASYQ STAGE DATA (variable length)

| Size | Name | Offset | Description |
|---|---|---|---|
| 1 | Sync | 0 | a5 (hex) |
| 1 | Id | 1 | 42 (hex) |
| 2 | Size | 2 | size of structure in number of words (1 word = 2 bytes) |
| 2 | Blanking | 4 | blanking distance (mm) |
| 2 | Pitch | 6 | compass pitch (0.1 deg) |
| 2 | Roll | 8 | compass roll (0.1 deg) |
| 2 | Pressure | 10 | pressure (mm) |
| 2 | Stage | 12 | stage (mm) |
| 2 | Quality | 14 | quality (counts) |
| 2 | Soundspeed | 16 | speed of sound (0.1 m/s) |
| 2 | StageP | 18 | stage using pressure window (mm) |
| 2 | QualityP | 20 | quality (counts) |
| 10 | Spare | 22 | spare |
| 1 | Amp 1 | 32 | amplitude cell 1 (counts) |
| 1··· | Amp 2···n | 33··· | repeated for cells 2 through n |
| 2 | Checksum | | = b58c(hex) + sum of all bytes in structure |

# CHAPTER 6
# ASCII Output

ASCII output is available for the Aquadopp single point current meter and for the Continental current profiler.

## Aquadopp ASCII Output

The Aquadopp single point current meter can also output data in ASCII format. To use the ASCII output feature you must upgrade your Aquadopp firmware to version 1.13 or higher. You can download new firmware from the *Support Pages of www.nortek-as.com*.

**Note:** The ASCII output ability applies to the Aquadopp single point current meter only and not to the Aquadopp profiler.

The output format is based on the current output of the ASCII conversion with our software. All positions are the same, the only differences are that the error and status codes are output as decimal numbers (e.g. $177_{10}$ instead of $10110001_2$), and the field delimiter is always just a single space. The description of the format is found in the .hdr file that is generated when you convert an .aqd data file to ASCII – shown overleaf.

### There are two ways to enable the ASCII output:

1. The command **AS** (**A**scii**S**tart) is the ASCII equivalent to the regular **ST** command. It starts a measurement with the current configuration and outputs the data in ASCII format. To get back into command mode you must send the confirmation characters **MC** after sending a break.
2. The command **MA** (**M**easure**A**scii) makes one measurement with the current configuration (unless when configured for continuous measurement – see below)

and outputs the data in ASCII format. There is a new binary equivalent to this command, **AD** (**A**quire**D**ata). If you want to control the data timing from a data logger you should use one of these commands. By using either the **MA** or the **AD** command, the instrument will automatically power down after the measurement is finished. Sending a break will cause the instrument to enter command mode directly, for example, if you want to stop a continuous measurement.

**The following should be observed:**

- ASCII commands consist of a pair of ASCII characters, sent when the instrument is in **Command** mode. No other characters are required (i.e. checksum or end of line characters).
- Make sure you have configured the instrument correctly before using the ASCII output commands. To use the ASCII commands, you will first configure the instrument from the Aquadopp software by entering the required setup parameters and starting a measurement.
- Note that for the **MA** command to work properly, the current meter cannot be in **Continuous** mode. This means that it must have a measuring interval that is at least 4s longer than its averaging interval.
- When you stop the measurement to enter **Command** mode, the instrument will remember the last configuration, even when power is removed.

## The ASCII Format

The output format is the same as the standard output of the ASCII conversion using the Aquadopp software, with the exception that speed and direction are not included. The sequence is the same, but the error and status codes are decimal numbers instead of binary (i.e. $177_{10}$ instead of $10110001_2$) Also, the field delimiter is always just a single space. The description of the format is found in the .hdr file that is generated when you convert an .aqd data file to ASCII. Here is an example with three lines of data (sampled at 10 minutes intervals):

| Month | Day | Year | Hour | Minute | Second | Error code | Status code | Velocity (Beam1/X/East) | Velocity (Beam2/Y/North) | Velocity (Beam3/Z/Up) | Amplitude (Beam1) | Amplitude (Beam2) | Amplitude (Beam3) | Battery voltage | Soundspeed | Heading | Pitch | Roll | Pressure | Temperature | Analogue input 1 | Analogue input 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 1 | 2004 | 11 | 3 | 40 | 0 | 172 | -0.104 | -0.226 | 0.061 | 21 | 20 | 20 | 13.3 | 1525.5 | 350.5 | 52.9 | -53.9 | 1.552 | 22.48 | 6304 | 6287 |
| 4 | 1 | 2004 | 11 | 3 | 44 | 0 | 172 | -0.375 | -0.366 | -0.082 | 21 | 20 | 20 | 13.3 | 1525.5 | 350.5 | 52.9 | -53.9 | 1.560 | 22.48 | 6280 | 6270 |
| 4 | 1 | 2004 | 11 | 3 | 48 | 0 | 172 | -0.705 | -0.526 | -0.359 | 21 | 20 | 20 | 13.3 | 1525.5 | 350.3 | 52.9 | -53.9 | 1.550 | 22.48 | 6204 | 6205 |

## Calculating the Speed

The speed is not included in the output file, but can be calculated from the following:

$$\text{Speed} = \sqrt{V_{EAST}^2 + V_{NORTH}^2}$$

In Excel® or MATLAB® this yields:

```
speed = SQRT(Veast*Veast+Vnorth*Vnorth)
```

## Calculating the Direction

The direction is not included in the output file either, but can be calculated from the following:

Excel®: `direction=MOD(90-ATAN2(Veast;Vnorth)*180/PI,360)`

MATLAB®: `direction=mod(90-atan2(Vnorth, Veast)*180/pi,360)`

# Continental ASCII Output

The Continental is also capable of sending out ASCII formatted data.

**To start a measurement with output in ASCII format the following steps must be used:**

1. Set the relevant deployment parameters using the Continental software that is shipped with the instrument.
2. Download the deployment configuration to the instrument by starting a measurement.
3. Stop the measurement, the instrument has now stored the configuration internally.
4. Start an ASCII measurement from the terminal emulator using the two character command **AS** (**A**scii **S**tart)
5. Stop the measurement using **Stop Data Collection** in the Continental SW. Alternatively, the measurement can be stopped by sending a soft break followed by the characters **MC** (**M**ode **C**ommand).

Observe that there is no storage of data to the internal recorder when data are output in ASCII format.

The format is as follows (overleaf):

## Header Line

| Size (characters) | Name |
|---|---|
| 12 | Serial No |
| 1 | Separator |
| 4 | Year |
| 1 | Separator |
| 2 | Month |
| 1 | Separator |
| 2 | Day |
| 1 | Year |
| 1 | Separator |
| 2 | Hour |
| 1 | Separator |
| 2 | Min |
| 1 | Separator |
| 2 | Second |
| 1 | Separator |
| 6 | Temperature |
| 1 | Separator |
| 7 | Extra field |

## Data Line

| Size (characters) | Name |
|---|---|
| 3 | CellNo |
| 1 | Separator |
| 6 | Speed |
| 1 | Separator |
| 4 | Direction |

# CHAPTER 7
# The SDI12 Interface

In the mid 1980's a group of water monitoring instrumentation users met to discuss a specific problem – the unreliability of low power analogue water sensors and the complexity of interfacing analogue sensors and dataloggers. The solution they came up with was the introduction of a digital interface – the SDI-12, which is short for *Serial Digital Interface-1200 baud.*

SDI-12 is an option for the EasyQ and EasyV type of instruments.

Among the features offered by SDI-12 are the ability to supply power to the sensor through the interface cable itself, the ability to have several sensors transmitting along one cable, instead of on cable per sensor, and the ability to use any SDI-12 compatible sensor with any SDI-12 compatible logger (with compatible connectors)

The SDI-12 supports the use of up to 200 feet (60 m) of cable between a sensor and a data recorder.

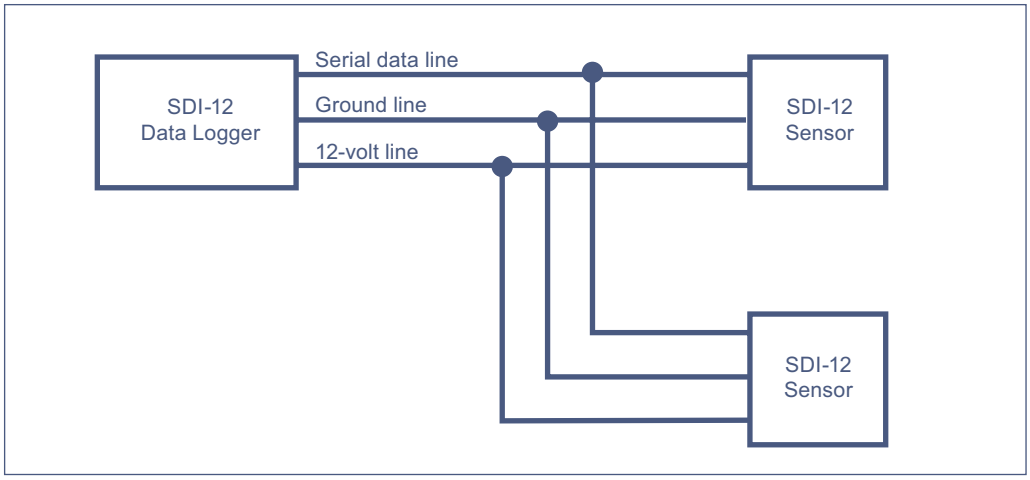
## A Few Technical Details

The SDI-12 electrical interface uses the SDI-12 bus to transmit serial data between SDI-12 data recorders and sensors. The SDI-12 bus is the cable that connects multiple SDI-12 devices. This is a cable with three conductors:

1. a serial data line
2. a ground line
3. a 12-volt line

The Fig. overleaf shows the SDI-12 bus connecting one data recorder with two sensors. The SDI-12 bus is capable of having at least 10 sensors connected to it.

The SDI-12 bus schematics




## The Serial Data Line

The data line is a bidirectional, three-state, data transfer line. The logic and voltage levels for the transmission of serial data is as shown below. Observe that the data line uses negative logic.

| Condition | Binary state | Voltage range |
|---|---|---|
| marking | 1 | –0.5 V to 1.0 V |
| spacing | 0 | 3.5 V to 5.5 V |
| transition | undefined | 1.0 V to 3.5 V |

When an SDI-12 device has its transmitter on, its direct current (DC) source resistance must be greater than 1000 Ω and less than 2000 Ω. When any SDI-12 device's transmitter is off, including a low-power standby mode, the DC resistance to ground must be within 160 kΩ to 360 kΩ . If an SDI-12 sensor does not use the 12-volt line for power, its data line resistance to ground while powered down must be within 160 kΩ to 360 kΩ.

## The Ground Line

The ground line must be connected to the circuit ground and the earth ground at the data logger. The sensor circuit ground must also be connected to the ground line, but not normally to its own earth ground. The ground conductor should be large enough to keep the voltage drop between the data logger and all sensors less than 0.5 V during the maximum combined sensor current drain.

## The 12-volt Line

The data logger (or the external power supply) provides between 9.6 V and 16 V to the 12-volt line, with respect to ground, as measured under a maximum sensor load of 0.5 A. SDI-12 does not require the data recorder to be the source of power to the 12-volt line.

# SDI-12 Communications Protocol Basics

SDI-12 data loggers and sensors communicate by an exchange of ASCII characters on the data line. The data logger sends a break to wake up the the sensors on the data line. A break is continuous spacing on the data line for at least 12 milliseconds. The data logger then sends a command. The sensor, in turn, returns the appropriate response. Each command is for a specific sensor. The first character of each command is a unique sensor address that specifies with which sensor the data logger wants to communicate. Other sensors on the SDI-12 bus ignore the command and return to low-power standby mode. When a data logger tells a sensor to start its data output procedure, the recorder does not communicate with any other sensor until the data collection from the first sensor is complete.

However, during a concurrent measurement command, a data logger can communicate with other sensors while one or more sensors are making measurements.

### A typical recorder/sensor measurement sequence proceeds as follows:

1. The data logger wakes all sensors on the SDI-12 bus with a break.
2. The data logger transmits a command to a specific, addressed sensor, instructing it to output data.
3. The addressed sensor responds within 15.0 milliseconds, returning the maximum time until measurement data will be come available for transfer to the data logger and the number of values to be transferred.
4. If the data is available immediately, the data logger transmits a command to the sensor instructing it to output the data. If the sensor is not ready to output data, the data logger waits for the sensor to send a request to the data logger. This will indicate that data are available for transfer to the data logger. The data logger will then transmit a command to get the data.

### Baud rate and byte frame format:

- Baud rate is 1200 baud.
- Byte frame format is 1 start bit; 7 databits (least significant bit transmitted first); 1 parity bit (even parity); 1 stop bit.

### Allowable characters:

- All characters transmitted on the SDI-12 bus must be printable ASCII characters, i.e. from the character **<space>** (**$32_{10}$**) through the character **~** (**$126_{10}$**).

  However, there are two exceptions:

  - All responses from from an SDI-12 sensor end with a **carriage return** (**0x0d**, **$13_{10}$**) and a line feed (**0x0a**, **$10_{10}$**).
  - In some cases the second and the third character of a CRC code may not be printable ASCII characters.

**Device addresses:**

- The first character of every command must be a sensor address. Likewise, the first character of a response is also the address character. This serves to verify that the response has come from the right sensor. An address is a single character used to indicate the sensor which is to respond to the command.
- The ASCII character **0** (zero), corresponding to $\mathbf{48}_{10}$ and **0x30** is the default address. All devices are initially set to **0** (zero) by the manufacturer for use in a single sensor system.
- The ASCII characters from **1** to **9**, corresponding to from $\mathbf{49}_{10}$ to $\mathbf{57}_{10}$ and from **0x31** to **0x39** are then to be used as addresses for additional sensors in a multi-sensor setup.
- Should there be a need for more than 10 sensors in a setup, addresses in the range ASCII **A** through ASCII **Z** ($\mathbf{65}_{10}$ through $\mathbf{90}_{10}$) and ASCII **a** through **z** ($\mathbf{97}_{10}$ through $\mathbf{122}_{10}$) can be used.

# Use with EasyQ and EasyV

As an option, the EasyQ and EasyV may include both RS232 and SDI-12 interfaces. You set up the EasyQ through the RS232 port using our standard Windows EasyQ software. You can set the SDI-12 address when you tell the instrument to switch to SDI-12 mode.

Once in SDI-12 mode, you will communicate through the SDI-12 interface. The following two SDI-12 commands give you access to water velocity and stage separately:

**0M!** Velocity measurement
**0M1!** Stage measurement

The above assumes the EasyQ is set for address 0.

# Velocity Data

**Nine values are returned in both cases for an EasyQ, for velocity these are:**

X Velocity cell 1 (ft/s)
Y Velocity cell 1 (ft/s)
Amplitude cell 1 counts×100+counts
X Velocity cell 2 (ft/s)
Y Velocity cell 2 (ft/s)
Amplitude cell 2 counts×100+counts

X Velocity cell 3 (ft/s)
Y Velocity cell 3 (ft/s)
Amplitude cell 3 counts×100+counts

The amplitude data is in the form **AABB** where **AA** is the amplitude for beam 1 and **BB** is the amplitude for beam 2. The range for **AA** and **BB** is 0–99, each count corresponding to about 1.1 dB.

## Stage Data

**Nine values are returned for stage:**

Stage (ft)
Quality counts
Pressure (ft)
Pitch degrees
Roll degrees
Stage(p) (ft), based on pressure
Quality(p) counts, based on pressure
%Good1
%Good2

Stage and Stage(p) both use acoustic stage measurement.

Stage(p) is the preferred stage measurement because it uses the pressure to reject echoes from debris and out-of-range echoes.

Stage (without using the pressure sensor) is included to provide a fallback in case the pressure sensor is defective or given the wrong offset. The stage quality is a measure of how well the echo conforms to the shape that is expected for the echo from a flat surface.

%Good tells you the percentage of stage measurements that produced good stage values. Our observation thus far is that %Good as little as 1% is sufficient for a good reading, so you should reject stage values only when %Good = 0.

# CHAPTER 8
# Example Programs (coded in C)

For your conveninence, we are pleased to provide a few example programs.

**The following examples are provided:**

- Generating a break
- Decoding the data structures – using Aquadopp as an example
- Structure definitions

## Generating a Break

```
//////////////////////////////////////////////////////////////////////////
// Sample code using the Microsoft Win32 API to open a handle to COM1,
// configure the serial port and send a break signal to wake up the instrument.

   .
   .
   .

   DCB dcb;
   HANDLE hComm;
   DWORD dwError;
   DWORD nBytesWritten;
   char cCommand[10];

   // Open a handle to COM1
   hComm = CreateFile("COM1",GENERIC_READ|GENERIC_WRITE,0,NULL,
                      OPEN_EXISTING,0,NULL);
   if (hComm == INVALID_HANDLE_VALUE) {
      dwError = GetLastError();
      // Handle the error.
   }
   // Omit the call to SetupComm to use the default queue sizes.
   // Get the current configuration.
   if (!GetCommState(hComm,&dcb)) {
      dwError = GetLastError();
```

```
   // Handle the error.
}

// Fill in the DCB: baud=9600, 8 data bits, no parity, 1 stop bit.
dcb.BaudRate = 9600;
dcb.ByteSize = 8;
dcb.Parity = NOPARITY;
dcb.StopBits = ONESTOPBIT;

if (!SetCommState(hComm, &dcb)) {
   dwError = GetLastError();
   // Handle the error.
}

// Send a soft break signal
memset(cCommand,64,6);     // @@@@@@
if (!WriteFile(hComm,cCommand,6,&nBytesWritten,NULL))
   dwError = GetLastError();
   // Handle the error.
}
Sleep(100);
strcpy(cCommand,"K1W%!Q");
if (!WriteFile(hComm,cCommand,6,&nBytesWritten,NULL))
   dwError = GetLastError();
   // Handle the error.
}

// Send a hard break signal
// Place the transmission line in a break state for 500 milliseconds
SetCommBreak(hComm);
Sleep(500);
ClearCommBreak(hComm);

.
.
.
```

# Decoding the Data Structures – Aquadopp Example

```
////////////////////////////////////////////////////////////////////////
// Sample code for decoding the Aquadopp data structure

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification (0x01=normal, 0x80=diag)
   unsigned short hSize;         // size of structure (words)
   PdClock        clock;         // date and time
   short          hError;        // error code
   short          hSpare;
   unsigned short hBattery;      // battery voltage (0.1 V)
   unsigned short hSoundSpeed;   // speed of sound  (0.1 m/s)
   short          hHeading;      // compass heading (0.1 deg)
   short          hPitch;        // compass pitch   (0.1 deg)
   short          hRoll;         // compass roll    (0.1 deg)
   unsigned char  cMSB;          // pressure MSB
   char           cStatus;       // status code
   unsigned short hLSW;          // pressure LSW
   short          hTemperature;  // temperature (0.01 deg C)
   short          hVel[3];       // velocity  (mm/s)
   unsigned       char cAmp[3];  // amplitude (counts)
```

```
   char           cFill;
   short          hChecksum;     // checksum
} PdMeas;


{
   .
   .
   .

   PdMeas meas;
   SYSTEMTIME st;
   double dVel[3];
   double dAmp[3];
   short hChecksum;
   double dPressure;
   double dBattery;
   double dHeading;
   double dPitch;
   double dRoll;
   double dTemperature;


   // Assuming three beams

   // Checksum control
   if (meas.hChecksum != Checksum((short *)&meas,meas.hSize - 1)) {
      // Handle the error.
   }

   st = ClockToSystemTime(meas.clock);

   dVel[0] = (double)meas.hVel[0] * 0.001;
   dVel[1] = (double)meas.hVel[1] * 0.001;
   dVel[2] = (double)meas.hVel[2] * 0.001;
   dAmp[0] = (double)meas.cAmp[0];
   dAmp[1] = (double)meas.cAmp[1];
   dAmp[2] = (double)meas.cAmp[2];

   dPressure = (65536.0*(double)meas.cMSB + (double)meas.hLSW)*0.001;
   dBattery = (double)meas.hBattery * 0.1;
   dHeading = (double)meas.hHeading * 0.1;
   dPitch = (double)meas.hPitch * 0.1;
   dRoll = (double)meas.hRoll * 0.1;
   dTemperature = (double)meas.hTemperature * 0.01;
   .
   .
   .

}//////////////////////////////////////////////////////////////////////
// Convert from BCD time to system time

SYSTEMTIME ClockToSystemTime(PdClock clock)
{
   SYSTEMTIME systime;
   WORD wYear;

   wYear = (WORD)BCDToChar(clock.cYear);
   if (wYear >= 90) {
      wYear += 1900;
   }
   else {
      wYear += 2000;
   }

   systime.wYear = wYear;
```

```
   systime.wMonth = (WORD)BCDToChar(clock.cMonth);
   systime.wDay = (WORD)BCDToChar(clock.cDay);
   systime.wHour = (WORD)BCDToChar(clock.cHour);
   systime.wMinute = (WORD)BCDToChar(clock.cMinute);
   systime.wSecond = (WORD)BCDToChar(clock.cSecond);
   systime.wMilliseconds = 0;

   return systime;
}


////////////////////////////////////////////////////////////////////////
// Convert from BCD to char

unsigned char BCDToChar(unsigned char cBCD)
{
   unsigned char c;

   cBCD = min(cBCD,0x99);
   c  = (cBCD & 0x0f);
   c += 10 * (cBCD >> 4);

   return c;
}



////////////////////////////////////////////////////////////////////////
// Compute checksum

short Checksum(short *phBuff,int n)
{
   int i;
   short hChecksum = 0xb58c;

   for (i=0; i<n; i++)
      hChecksum += phBuff[i];

   return hChecksum;
}
```

# Structure Definitions

```
#define PD_MAX_BEAMS                3
#define PD_MAX_BINS                 128
#define PD_MAX_STAGECELLS           1024


#pragma pack(push)
#pragma pack(1)      // 1 byte struct member alignment used in firmware


////////////////////////////////////////////////////////////////////////////
// Clock data (6 bytes)  NOTE! BCD format

typedef struct {
   unsigned char  cMinute;       // minute
   unsigned char  cSecond;       // second
   unsigned char  cDay;          // day
   unsigned char  cHour;         // hour
   unsigned char  cYear;         // year
   unsigned char  cMonth;        // month
} PdClock;


////////////////////////////////////////////////////////////////////////////
// Aquadopp diagnostics header data

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification = 0x06
   unsigned short hSize;         // total size of structure (words)
   unsigned short nRecords;      // number of diagnostics samples to follow
   unsigned short nCell;         // cell number of stored diagnostics data
   unsigned char  cNoise[4];     // noise amplitude (counts)
   PdClock        clock;         // date and time
   unsigned short hSpare1;
   unsigned short hDistance[4];  // distance
   unsigned short hSpare[3];
   short          hChecksum;     // checksum
} PdDiagHead;


////////////////////////////////////////////////////////////////////////////
// Aquadopp velocity data 3 beams

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification (0x01=normal, 0x80=diag)
   unsigned short hSize;         // size of structure (words)
   PdClock        clock;         // date and time
   short          hError;        // error code:
                                 //    bit 0: compass (0=ok, 1=error)
                                 //    bit 1: measurement data (0=ok, 1=error)
                                 //    bit 2: sensor data (0=ok, 1=error)
                                 //    bit 3: tag bit (0=ok, 1=error)
                                 //    bit 4: flash (0=ok, 1=error)
                                 //    bit 5:
                                 //    bit 6: serial CT sensor read (0=ok,
                                 //                                  1=error)
   unsigned short hAnaIn1;       // analog input 1
   unsigned short hBattery;      // battery voltage (0.1 V)
   union {
      unsigned short hSoundSpeed;   // speed of sound (0.1 m/s)
      unsigned short hAnaIn2;       // analog input 2
   } u;
```

```
   short          hHeading;      // compass heading (0.1 deg)
   short          hPitch;        // compass pitch (0.1 deg)
   short          hRoll;         // compass roll (0.1 deg)
   unsigned char  cPressureMSB;  // pressure MSB
   char           cStatus;       // status:
                                 //    bit 0: orientation (0=up, 1=down)
                                 //    bit 1: scaling (0=mm/s, 1=0.1mm/s)
                                 //    bit 2: pitch (0=ok, 1=out of range)
                                 //    bit 3: roll (0=ok, 1=out of range)
                                 //    bit 4: wakeup state:
                                 //    bit 5: (00=bad power, 01=break, 10=power
                                 //            applied, 11=RTC alarm)
                                 //    bit 6: power level:
                                 //    bit 7: (00=0(high), 01=1, 10=2,
                                 //            11=3(low))
   unsigned short hPressureLSW;  // pressure LSW
   short          hTemperature;  // temperature (0.01 deg C)
   short          hVel[3];       // velocity
   unsigned char  cAmp[3];       // amplitude
   char           cFill;
   short          hChecksum;     // checksum
} PdMeas;

////////////////////////////////////////////////////////////////////////////
// Vector velocity data header (18 bytes)

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification = 0x12
   unsigned short hSize;         // total size of structure (words)
   PdClock        clock;         // date and time
   unsigned short nRecords;      // number of velocity samples to follow
   unsigned char  cNoise[4];     // noise amplitude (counts)
   unsigned char  cCorr[4];      // noise correlation
   unsigned short hSpare[10];    // spare values
   short          hChecksum;     // checksum
} PdVecHead;


////////////////////////////////////////////////////////////////////////////
// Vector velocity data 3 beams

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification = 0x10
   unsigned char  cAnaIn2LSB;    // analog input 2 LSB
   unsigned char  cCount;        // ensemble counter
   unsigned char  cPressureMSB;  // pressure MSB
   unsigned char  cAnaIn2MSB;    // analog input 2 MSB
   unsigned short hPressureLSW;  // pressure LSW
   unsigned short hAnaIn1;       // analog input 1 (fast)
   short          hVel[3];       // velocity
   unsigned char  cAmp[3];       // amplitude
   unsigned char  cCorr[3];      // correlation  (0-100)
   short          hChecksum;     // checksum
} PdVecVel;


////////////////////////////////////////////////////////////////////////////
// Vector system data (28 bytes)

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification = 0x11
   unsigned short hSize;         // size of structure (words)
   PdClock        clock;         // date and time
```

```
   unsigned short hBattery;      // battery voltage (0.1 V)
   unsigned short hSoundSpeed;   // speed of sound (0.1 m/s)
   short          hHeading;      // compass heading (0.1 deg)
   short          hPitch;        // compass pitch (0.1 deg)
   short          hRoll;         // compass roll (0.1 deg)
   short          hTemperature;  // temperature (0.01 deg C)
   char           cError;        // error code
   char           cStatus;       // status
   unsigned short hAnaIn;        // analog input (slow)
   short          hChecksum;
} PdVecSys;


////////////////////////////////////////////////////////////////////////////
// Aquadopp velocity profile data

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification (0x21 = 3 beams, 0x22 = 2
                                 // beams, 0x21 = 1 beam)
   unsigned short hSize;         // size of structure (words)
   PdClock        clock;         // date and time
   short          hError;        // error code
   unsigned short hAnaIn1;       // analog input 1
   unsigned short hBattery;      // battery voltage (0.1 V)
   union {
      unsigned short hSoundSpeed;// speed of sound (0.1 m/s)
      unsigned short hAnaIn2;    // analog input 2
   } u;
   short          hHeading;      // compass heading (0.1 deg)
   short          hPitch;        // compass pitch (0.1 deg)
   short          hRoll;         // compass roll (0.1 deg)
   union {
      struct {
         unsigned char  cMSB;    // pressure MSB
         char           cStatus; // status
         unsigned short hLSW;    // pressure LSW
      } Pressure;                // (mm)
      struct {
         unsigned char  cQuality;// distance quality
         char           cStatus; // status
         unsigned short hDist;   // distance (mm)
      } Distance;
   } u1;
   short          hTemperature;  // temperature (0.01 deg C)
   // actual size of the following = nBeams*nBins*3 + 2
   short          hVel[PD_MAX_BEAMS][PD_MAX_BINS];
// short          hVel[nBeams][nCells];   // velocity
   unsigned char  cAmp[PD_MAX_BEAMS][PD_MAX_BINS];
// char  cAmp[nBeams][nCells];   // amplitude
// char  cFill                   // if nCells % 2 != 0
   short          hChecksum;     // checksum
} PdAqdProf;


////////////////////////////////////////////////////////////////////////////
// Continental velocity profile data (variable length)

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification (0x24 = 3 beams, 0x25 = 2
                                                    beams, 0x26 = 1 beam)
   unsigned short hSize;         // size of structure (words)
   PdClock        clock;         // date and time
   short          hError;        // error code
   unsigned short hAnaIn1;       // analog input 1
```

```
   unsigned short hBattery;      // battery voltage (0.1 V)
   union {
      unsigned short hSoundSpeed;// speed of sound (0.1 m/s)
      unsigned short hAnaIn2;    // analog input 2
   } u;
   short          hHeading;      // compass heading (0.1 deg)
   short          hPitch;        // compass pitch (0.1 deg)
   short          hRoll;         // compass roll (0.1 deg)
   unsigned char  cPressureMSB;  // pressure MSB
   char           cStatus;       // status
   unsigned short hPressureLSW;  // pressure LSW
   short          hTemperature;  // temperature (0.01 deg C)
   short          hSpare[44];
   // actual size of the following = nBeams*nBins*3 + 2
   short          hVel[PD_MAX_BEAMS][PD_MAX_BINS];
// short hVel[nBeams][nCells];   // velocity
   unsigned char  cAmp[PD_MAX_BEAMS][PD_MAX_BINS];
// char  cAmp[nBeams][nCells];   // amplitude
// char  cFill                   // if nCells % 2 != 0
   short          hChecksum;     // checksum
} PdFarProf;


////////////////////////////////////////////////////////////////////////////
// AWAC velocity profile data (variable length)

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification (0x20)
   unsigned short hSize;         // size of structure (words)
   PdClock        clock;         // date and time
   short          hError;        // error code
   unsigned short hAnaIn1;       // analog input 1
   unsigned short hBattery;      // battery voltage (0.1 V)
   union {
      unsigned short hSoundSpeed;   // speed of sound (0.1 m/s)
      unsigned short hAnaIn2;       // analog input 2
   } u;
   short          hHeading;      // compass heading (0.1 deg)
   short          hPitch;        // compass pitch (0.1 deg)
   short          hRoll;         // compass roll (0.1 deg)
   unsigned char  cPressureMSB;  // pressure MSB
   char           cStatus;       // status
   unsigned short hPressureLSW;  // pressure LSW
   short          hTemperature;  // temperature (0.01 deg C)
   short          hSpare[44];
   // actual size of the following = nBeams*nBins*3 + 2
   short          hVel[PD_MAX_BEAMS][PD_MAX_BINS];
// short hVel[nBeams][nCells];   // velocity
   unsigned char  cAmp[PD_MAX_BEAMS][PD_MAX_BINS];
// char  cAmp[nBeams][nCells];   // amplitude
// char  cFill                   // if nCells % 2 != 0
   short          hChecksum;     // checksum
} PdProf;


////////////////////////////////////////////////////////////////////////////
// Wave header data (60 bytes)

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification = 0x31
   unsigned short hSize;         // total size of structure (words)
   PdClock        clock;         // date and time
   unsigned short nRecords;      // number of wave data records to follow
   unsigned short hBlanking;     // T2 used for wave data measurements (counts)
```

```
   unsigned short hBattery;      // battery voltage (0.1 V)
   unsigned short hSoundSpeed;   // speed of sound (0.1 m/s)
   short          hHeading;      // compass heading (0.1 deg)
   short          hPitch;        // compass pitch (0.1 deg)
   short          hRoll;         // compass roll (0.1 deg)
   unsigned short hMinPress;     // minimum pressure value of previous profile
                                 //(mm)
   unsigned short hMaxPress;     // maximum pressure value of previous profile
                                 // (mm)
   short          hTemperature;  // temperature (0.01 deg C)
   unsigned short hCellSize;     // cell size in counts of T3
   unsigned char  cNoise[4];     // noise amplitude (counts)
   unsigned short hProcMagn[4];  // processing magnitude
   unsigned short hWindRed;      // number of samples of AST window past
                                 // boundary
   unsigned short hASTWindow;    // AST window size (# samples)
   short          hSpare[5];     // spare values
   short          hChecksum;     // checksum
} PdWaveHead;


////////////////////////////////////////////////////////////////////////////
// Wave data (24 bytes)

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification (0x30)
   unsigned short hSize;         // size of structure (words)
   unsigned short hPressure;     // pressure (mm)
   unsigned short hDistance;     // AST distance1 on vertical beam (mm)
   unsigned short hAnaIn;        // analog input
   short          hVel[4];       // velocity, hVel[3] = AST distance2 on
                                 // vertical beam (mm)
   unsigned char  cAmp[4];       // amplitude, cAmp[3] = AST quality (counts)
   short          hChecksum;     // checksum
} PdWave;


////////////////////////////////////////////////////////////////////////////
// EasyQ range check data (variable length)

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification = 0x41
   unsigned short hSize;         // size of structure (words)
   PdClock        clock;         // date and time
   unsigned short nBeams;        // number of beams
   unsigned short hBlanking;     // blanking distance (counts)
   unsigned char  cNoise[4];     // noise amplitude (counts)
   short          hSpare[7];
   // actual size of the following = nBeams*nCells + 2
   unsigned char  cAmp[4*PD_MAX_BINS];
   // char  cAmp[nBeams][nCells]; // amplitude
   // char  cFill                  // if nCells % 2 != 0
   short          hChecksum;     // checksum
} PdRange;


////////////////////////////////////////////////////////////////////////////
// EasyQ stage check data (variable length)

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification = 0x42
   unsigned short hSize;         // size of structure (words)
   unsigned short hBlanking;     // blanking distance (mm)
```

```
   short          hPitch;        // compass pitch (0.1 deg)
   short          hRoll;         // compass roll (0.1 deg)
   unsigned short hPressure;     // pressure (mm)
   short          hStage;        // stage (mm)
   unsigned short hQuality;      // quality (counts)
   unsigned short hSoundSpeed;   // speed of sound (0.1 m/s)
   short          hStageP;       // stage using pressure window (mm)
   unsigned short hQualityP;     // quality (counts)
   short          hSpare[5];
   // actual size of the following = nCells + 2
   unsigned char  cAmp[PD_MAX_STAGECELLS];
   // char cAmp[nStageCells]; // amplitude
   // char  cFill             // if nCells % 2 != 0
   short          hChecksum;     // checksum
} PdStage;


////////////////////////////////////////////////////////////////////////////
// EasyQ velocity data (variable length)

typedef struct {
   unsigned char  cSync;         // sync = 0xa5
   unsigned char  cId;           // identification (0x40)
   unsigned short hSize;         // size of structure (words)
   PdClock        clock;         // date and time
   short          hError;        // error code
   short          hTemperature;  // temperature (0.01 deg C)
   unsigned short hBattery;      // battery voltage (0.1 V)
   unsigned short hSoundSpeed;   // speed of sound (0.1 m/s)
   unsigned char  cSpare;
   unsigned char  cStatus;       // status
   unsigned short hAnaIn1;       // analog input 1
   unsigned short hAnaIn2;       // analog input 2
   short          hSpare[2];
   // actual size of the following = nBeams*nBins*3 + 2
   short          hVel[PD_MAX_BEAMS][PD_MAX_BINS];
// short hVel[nBeams][nCells];   // velocity
   unsigned char  cAmp[PD_MAX_BEAMS][PD_MAX_BINS];
// char  cAmp[nBeams][nCells];   // amplitude
// char  cFill                   // if nCells % 2 != 0
   short          hChecksum;     // checksum
} PdEasyQ;

#pragma pack(pop)
```


Nortek AS, Vangkroken 2, NO-1351 Rud, Norway. Tel +47 67174500. Fax +47 67136770. inquiry@nortek.no. www.nortek-as.com