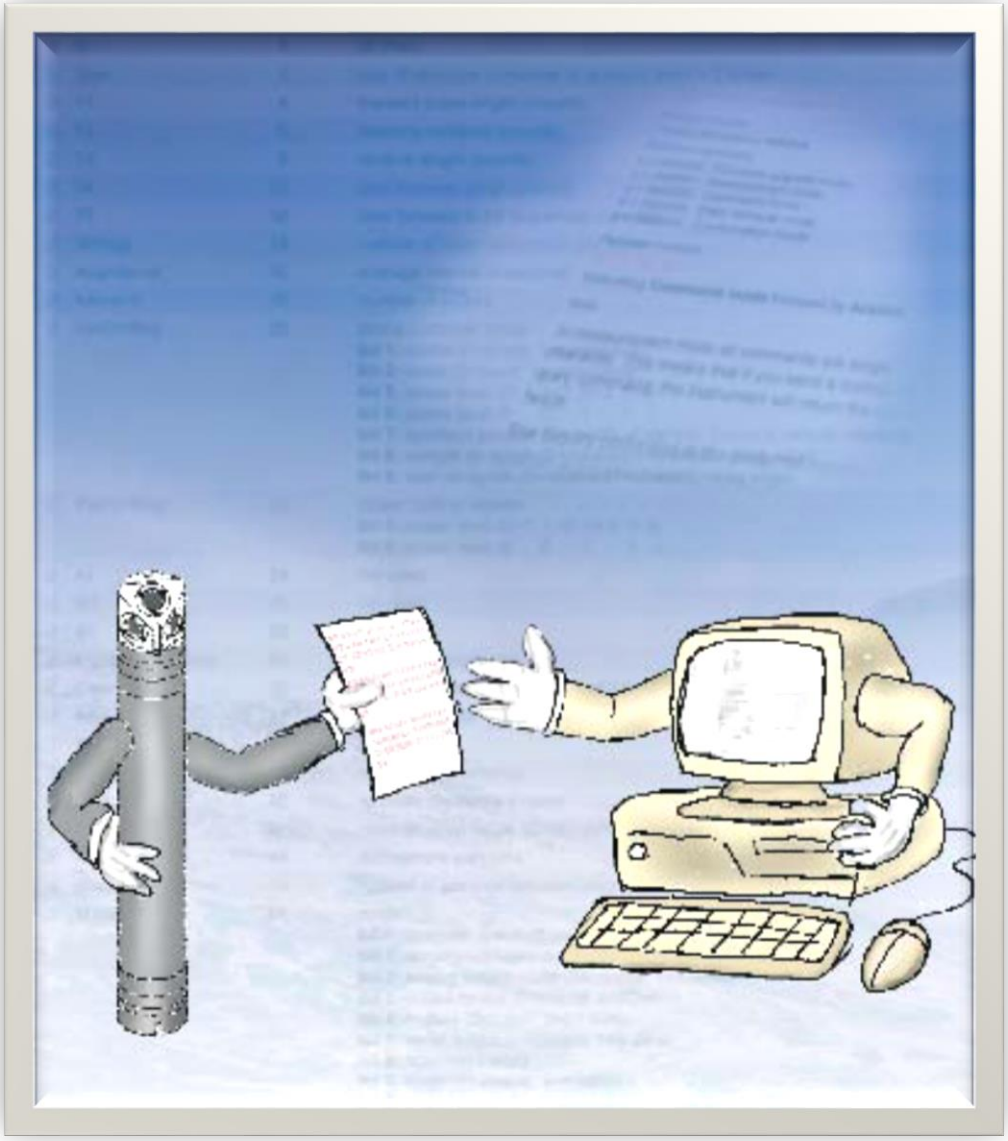
NORTEK AS

VECTRINO PROFILER INTEGRATOR MANUAL



January 2015

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

Date	**Change Notes**	**Authour**
August 2012	Initial Release	Robert Craig
October 2012	Added « velocityExponent » to User Configuration structure (0.1 mm/s or 1mm/s resolution).	Robert Craig
February 2014	-Added units information to ping interval parameters -Vectrino II changed to Vectrino Profiler. -Status information is now passed in the command header during measurement mode.	Robert Craig
July 2014	Updated sample code. The velocity data stucture may contain a pad byte to make the overall structure size an even number. Added section on how to use the binary configuration file created by MIDAS.	Robert Craig
Sept 2014	Section on how to use the vProToNTK utility added.	Robert Craig
January 2015	Probe calibration description added	Robert Craig

CONTENTS

1 INTRODUCTION

This document provides the information needed to control the Vectrino Profiler with user supplied software. It is aimed at system integrators and engineers with interfacing experience. Code examples are provided in C. The document's scope is limited to interfacing and does not address general performance issues of the instruments. For a more thorough understanding of the principle of operation, we recommend the user guide that accompanies the individual instruments.

The document is complete in the sense that it describes all available commands and modes of communication. For most users, it will make sense to let the supplied Nortek 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.

Note that the Nortek products use a binary data format for communication. This makes it hard to "see" what is going on with a terminal emulator. However, the binary interface saves programming time because parsing of text data isn't needed. It may take more time initially to put the basic communication in place, but once done the remainder of the work should be straightforward. The use of checksums and CRC helps to make the binary data interface more robust.

As always, these types of documents are subject to change. We recommend that you check http://www.nortek-as.com/en/support or contact Nortek to ensure you have the all the latest information and versions of any software you plan to use.

We recommend you do this as part of your project planning before you start any development work. If you have any comments or suggestions on the information given here, please let us know. Your comments are always appreciated; our general e-mail address is inquiry@nortek.no.

You can always join our forum and post your comments, suggestions or questions there, visit our website www.nortek-as.com and click the link to the forum.

2 BASIC INTERFACE CONCEPTS

The Nortek products communicate using serial ports 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 Windows programs or by using direct commands to the system after the direct communication has been initiated (see the chapter on Remote Control Terminal Commands). The only lines used are RxD, TxD, and GND. Status and handshaking lines are not used.

2.1 OPERATIONAL MODES

The operational modes for the Vectrino Profiler are:

- Command mode. The system is waiting for commands to be sent over the serial line.
- Measurement mode. The system cycles through a series of states when collecting data. To exit measurement mode, a break and confirmation string must be sent.

2.2 THE BREAK

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.

To send a break you first send “@@@@@@” followed by a delay of 10 ms and then send “K1W%!Q”. The “MC” command (embedded in a full command header) must follow the break on the Vectrino Profiler to receive a response (see section 4.2).

2.3 Checksum Control

Most data structures contain a 16-bit checksum. An example program is given in the chapter on Data Structures to help explain how the checksum is calculated.

2.4 Protocol Header with Two-character ASCII Commands

The command interface uses two character commands where the two characters are treated as a single 16-bit word. These commands are embedded in a protocol header described below. The protocol header must be sent as one unit within 1 second; otherwise all characters will be discarded.

Data transfer is carried out using the "little Endian" convention, which means that the low byte is sent before the high byte. The data types are given in the section describing the various commands. More about this can be found in the Commands chapter.

2.5 Acknowledgement

After a successful command is sent, the system returns an acknowledgement embedded within a protocol header. The value for acknowledge (AckAck) is 0x0606. Whenever the firmware receives a command/word that is invalid, it immediately returns a negative acknowledge (NackNack). The value is 0x1515. With the Vectrino Profiler, this is usually followed with an error string indicating the reason for the failure.

3 USE WITH A CONTROLLER

This chapter provides useful information when setting up your Nortek instrument with a controller. 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 behavior, with or without data transfer to the controller.

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

The data output to the controller is in binary format. The Vectrino Profiler uses a high speed RS-422 interface for data transfer. This interface allows transfer rates of up to 1.25MBaud between the controller and instrument.

3.1 SIMPLE STORAGE DEVICE

Data output from the Vectrino Profiler instrument is time stamped with relative to the start of data collection. In order to synchronize instrument time with the real time of day, the controller must store the time of day that the first data record is received. Comparing the time of day with the instrument time gives the offset that must be applied to the instrument time to convert it into time of day. Note that there is currently no software available from Nortek that allows raw data files from the Vectrino Profiler to be read. It is up to the user to interpret the file contents.

See the chapter on Data Structures for more information on how to interpret the data received from the instrument.

3.2 CONTROL THE INSTRUMENT DIRECTLY

Direct control involves having the controller start and stop the measurements using a combination of a two character ASCII command (embedded in a protocol header) and a break command.

For commands to be received and executed, the instrument must be in Command mode. If the instrument is in Measurement Mode a break followed by the MC command must be sent. The MC command must be sent within 1 second of the break. The Vectrino Profiler continues measuring until the break / MC command is received.

3.3 Control via the Serial Line

To start a measurement from Command mode, send the command ST in a protocol header. The system will send an acknowledge header to show that the measurement is started. More about this can be found in the Terminal Commands chapter.

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 command MC (in a protocol header) within 60 seconds.
- To start a measurement from command mode, send the command ST in the protocol header.
- To stop data collection, send a break string followed by the verification command MC in a protocol header.

4 REMOTE CONTROL COMMANDS

Note that it is *not* possible to control the Vectrino Profiler through a terminal emulator. All commands sent to and received from the Vectrino Profiler are binary in nature and, as such, contain non-printable characters. All communications must be carried out using a controller running appropriate interpretation software.

All commands in the Vectrino Profiler are incorporated into a format that use a data header followed by a data section protocol. The data header contains information about the command and the amount of parameter data following the header. The same header is used when reading data from the instrument to identify the type and amount of data in the data section.

To send a command, create a header with the sync byte set to 0xA5, the refer byte set to 0, the ID set to the command ID (as described below) and fill in the dataSize parameter with the size of any command parameters to follow (often 0). Then calculate the checksum for these 6 bytes and store it in the checksum word. Send the header followed by any associated data.

The instrument will respond with a header containing ID 0x0606 (ACKACK) on success and a value indicating the amount of response data (if applicable) in dataSize. On failure, a header containing an ID 0x1515 (NACKNACK) will be sent and the data following will usually be an error string describing the reason for the failure (with the length of the error string in dataSize).

A few terms:

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 running on the computer or controller

FW: The software program running on the instrument

0x: Indicates hexadecimal representation

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 is not compatible with Big Endian.

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 tables below.

Example: For the RC command the character ‘R’ corresponds to 0x52 and the character ‘C’ to 0x43. Shown in reversed order (to comply with the Little Endian principle) this will read 0x4352, which is what you will find listed in the table: Remote Control Commands in Command Mode.

4.1 Command Mode

Protocol Header

Size	Name	Offset	Description
1	Sync	0	0xA5 (hex)
1	Status / Refer	1	Bits 0 – 3: The lower 4 bits are reserved for internal use. In measurement mode, the upper four bits contain status information. Bit 4 – The instrument is buffering and the internal memory is almost full. Bit 5 – The instrument is currently buffering data (baud rate too low to support real-time data transfer). Bit 6 – The instrument has run out of internal memory and has stopped collecting data. Data transmission of the buffered data will continue until the internal buffer has been emptied. Bit 7 – Internal processing error detected. Retrieve the system log from the instrument when making a support request.
2	ID	2	Command or Data Record Type
2	dataSize	4	Amount of data in data block
2	Checksum	6	Word-wise checksum of this header
Total Size 8 Bytes			

Set baud rate	
Execute command **BR** **Hex** 5242	**Description** Sets the instrument baud rate. **Response** Protocol header with **ID=0606 dataSize = 0** **Parameter to be sent** Z dataSize = 4 **Parameter structure**

	z = 4 byte binary baud rate Supported baud rates are 9600 19200 38400 57600 115384 234375 460800 937500 1250000
	Example A5 00 52 42 04 00 87 F8 1C 4E 0E 00 Command for setting baud rate to 937500 (0x000E4E1C) baud followed by the response A5 00 06 06 00 00 37 BC
Reference	**Note** The baud rate is stored in the instrument only when an SB command is sent. This will ensure that the communication can be restored by powering down the instrument. 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 configuration and sample rate or internal buffering of the data stream will occur in the instrument.

Save baud rate	
Execute command **SB** **Hex** **4253**	**Description** Saves the currently set baud rate
	Response Protocol header with **ID=0606 dataSize = 0**
	Parameter to be sent None
	Example A5 00 42 53 04 00 73 09

	followed by the response A5 00 06 06 00 00 37 BC
Reference	**Note** The baud rate is stored in the instrument only when an SB command is sent. This will ensure that the communication can be restored by powering down the instrument. 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 configuration and sample rate or internal buffering of the data stream will occur in the instrument.

Sets user configuration data	
Execute command CC **Hex** 4343	**Description** Set the current user configuration from the instrument. It is recommended that, before altering any configuration elements, the configuration structure be read with the GC command first. Given that the firmware will alter the configuration sent to match internal capabilities, it is also recommended that the instrument configuration be read directly after being set to get the actual values used.
	Response Protocol header with **ID=0606 dataSize=0** on success or **ID=1515 dataSize=length of error string** followed by the error string on failure
	Parameter **None**
	Example A5 00 43 43 B8 00 2C FA B8 00 04 02 00 00 04 01 00 03 04 34 00 00 00 00 00 80 BB 44 00 00 C8 42 10 27 00 00 00 00 00 00 64 00 00 00 0A 00 00 00 00 00 00 00 35 00 96 00 1C 00 0A 00 24 00 90 01 00 00 84 3F 00 00 20 42 … **Response** A5 00 06 06 00 00 37 BC
Reference **Chapter on** **Data Structures**	**Note** Some lines in the above example have been removed for clarity.

Read user configuration data	
Execute command GC **Hex** 4347	**Description** Read the current user configuration from the instrument. **Response** Protocol header with **ID=0606 dataSize=184** followed by User configuration (184 bytes) **Parameter** **None** **Example** A5 00 43 47 00 00 74 FD **Response** A5 00 06 06 B8 00 EF BC B8 00 04 02 00 00 04 01 00 03 04 34 00 00 00 00 00 80 BB 44 00 00 C8 42 10 27 00 00 00 00 00 00 64 00 00 00 0A 00 00 00 00 00 00 00 35 00 96 00 1C 00 0A 00 24 00 90 01 00 00 84 3F 00 00 20 42 00 00 B8 0B C1 0F 20 05 1E 00 90 01 EE 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 2C 00 02 01 01 00 00 00 00 00 20 41 32 00 78 00 46 00 0A 00 00 00 84 3F 00 00 48 42 00 60 F4 42 28 00 00 00 …
Reference **Chapter on** **Data Structures**	**Note** Some lines in the above example have been removed for clarity.

Read hardware configuration data	
Execute command GP	**Description** Read the currently used hardware configuration from the instrument.
Hex 5047	**Response** Header with **ID=0606 dataSize=104** followed by hardware configuration (104 bytes)
	Parameter None
Reference **Chapter on** **Data Structures**	**Note**

Read head configuration data	
Execute command **GH** **Hex** **4847**	**Description** Read the currently used head configuration from the instrument **Response** Header with **ID=0606 dataSize=224** followed by head configuration (224 bytes) **Parameter** None
Reference **Chapter on** **Data Structures**	**Note**

Get identification string	
Execute command **ID** **Hex** **4449**	**Description** Read the identification string from the instrument. **Response** Header with **ID = 0606, dataSize = ID Length** followed by an ASCII string **Parameter** None **Response example** A5 00 06 06 0C 00 43 BC 56 45 43 54 49 49 2D 50 72 6F 66 69 65 corresponding to VECTII-Profi

Start measurement	
Execute command **ST** **Hex** **5453**	**Description** Immediately starts a measurement using the current configuration of the instrument. **Response** Header with **ID = 0606, dataSize = 0** **Parameter** None
Reference	**Note** 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.

4.2 In Measurement Mode

Enter command mode	
Execute command **MC** **Hex** **434d**	**Description** Preceded by a break command, this command is sent to force the instrument to exit **Measurement mode** and enter **Command mode.** **Response** Header with **ID = 0606, dataSize = 0** **Parameter** None
Reference	**Note** The **MC** command must be sent within 10 seconds of the break being sent. Otherwise the measurement will continue. Within 2 seconds of **AckAck** being sent, the instrument will enter **Command mode**

4.3 Configuration

The configuration structure is described in Section 5.1. There are three sections to the configuration: Doppler configuration, bottom check configuration, adaptive ping interval configuration.

4.3.1 Doppler User Configuration

The main configuration structure is used for configuring the velocity sampling. It contains both elements that are set by the user and elements that are returned by the instrument. Most of the settings are explained in the description of the configuration structure. Note that all pad / unused bytes must be set to 0 for compatibility reasons.

The version number returned during a "get configuration" operation should always be checked to ensure that it matches the expected version of your controller. This will allow new firmware to be recognized and accommodated for as required.

In terms of profile range, the user sets the cellSize, nCells and cellStart parameters and the instrument returns the actual cell size and cell start in cellSizeSelected and cellStartSelected. The speedOfSound variable is set by the user when calcSpeedOfSound is 0 and calculated and returned if 1.

nTranducers is always returned as 4 and nFrequencies is always returned as 1.

pingInterval / extendedPingInterval and ensemblePingPairs are calculated and returned based upon the value the calcPingInterval.

horizontalVelocityRange and verticalVelocityRange are calculated and returned based upon the calculated ping intervals and user setting for velocityRange.

rawSampleSeparation should be set to 0 (continuous pings) to provide the best data collection. In "unwrap" extended velocity mode, this value is calculated. (Don't forget to reset the value back to 0 if you swap between unwrap mode and dual PRF mode.)

Bottom check and dynamic ping interval selection use the embedded bottom and beam sub-structures.

4.3.2 Bottom Check Configuration

Bottom check is enabled by setting config.bottom.enable to 1. The additional variables within the sub structure are explain in section 5.1. The cellSizeSelected, minRangeSelected, and maxRangeSelection values are the calculated values actually used by the instrument that are closest to those set by the user in cellSize, minRange, and maxRange. Note that the bottom check sample rate can be *at most* 1/3 of the velocity sample rate (to a maximum of 10Hz).

4.3.3 Adaptive Ping Interval Configuration

Adaptive ping interval calculations use the embedded beam sub-structure.

To enable adaptive ping interval calculations, set config.calcPingInterval to 3 set config.beam.enable to 1. For "Adaptive once" set config.beam.sampleRate to 0. For dynamic adaptive and bottom check, set beam.sampleRate to some value that is much less than bottom.sampleRate.

The software application uses the following settings:

```
config.beam.enable = 1;
config.beam.cellSize = 40;
config.beam.minRange = 30;
config.beam.maxRange = 1000;
config.beam.gainReductiondB = 0;
```

4.3.4 Using a Configuration Command File

MIDAS (the Nortek software used to operate the Vectrino Profiler) contains a feature that allows the user to create a binary configuration command file that can be sent directly to the instrument. The command file contains both the protocol header and user configuration structure. To use the file, the user supplied software must first connect to the instrument. A code snippet showing how to read and send the file is included (full source for how to connect to the instrument and for the `port.write` and `readBlock` commands are included in Chapter 7).

```
pBinaryFile = fopen(binaryFileName, "rb");
if (binaryFileName == NULL) {
        printf("Configuration file %s could not be opened.", binaryFileName);
        endit();
}

char cfg[256];

/* Get file size */
fseek(pBinaryFile, 0L, SEEK_END);
unsigned int fsize = ftell(pBinaryFile);
fseek(pBinaryFile, 0L, SEEK_SET);

if (fsize != fread(cfg, 1, fsize, pBinaryFile)) {
        printf("File read failed");
        endit();
}

fclose(pBinaryFile);
port.write(cfg, fsize);
rx_length = BUFF_SIZE;

if (readBlock(serBuff, &rx_length, 1) == VECTRINOPROFILER_ID_SUCCESS) {
        printf("Configuration accepted.\n\n");
} else {
        printf("Configuration Error: %s\n\n", serBuff);
}
```

5 FIRMWARE DATA STRUCTURES

This section describes the data structures that are used for the Vectrino Profiler. All structures are prefixed with the protocol header.

5.1 COMMAND MODE STRUCTURES

Protocol Header

Size	Name	Offset	Description
1	Sync	0	0xA5 (hex)
1	Refer	1	Bits 0 – 3: The lower 4 bits are reserved for internal use. In measurement mode, the upper four bits contain status information. Bit 4 – The instrument is buffering and the internal memory is almost full. Bit 5 – The instrument is currently buffering data (baud rate too low to support real-time data transfer). Bit 6 – The instrument has run out of internal memory and has stopped collecting data. Data transmission of the buffered data will continue until the internal buffer has been emptied. Bit 7 – Internal processing error detected. Retrieve the system log from the instrument when making a support request.
2	ID	2	Command or Data Record Type
2	dataSize	4	Amount of data in data block
2	checksum	6	Word-wise checksum of this header
Total Size 8 Bytes			

Hardware Configuration

Retrieved using the “GP” command

Size	Name	Offset	Description
14	SerialNo	0	instrument type and serial number
2	Config	14	board configuration: bit 0: Recorder installed (0=no, 1=yes) bit 1: Compass installed (0=no, 1=yes)
2	Frequency	16	board frequency [kHz]
2	PICversion	18	FPGA version number
2	HWrevision	20	Hardware revision
2	RecSize	22	Recorder size (*65536 bytes)
14	Spare	24	Unused
16	FW Version	38	DSP firmware version
16	FWRepoVersion	54	DSP Firmware build version
32	FWdate	70	Date that firmware was built
2	Checksum	102	= b58c(hex) + sum of all words in structure
Total Size 104 Bytes			

Head Configuration

Retrieved using “GH” command

Size	Name	Offset	Description
1	Sync	0	a5 (hex)
1	Id	1	04 (hex)
2	Size	2	size of structure in number of words (1 word = 2 bytes)
2	Config	4	head configuration: bit 0: Pressure sensor (0=no, 1=yes) bit 1: Magnetometer sensor (0=no, 1=yes) bit 2: Tilt sensor (0=no, 1=yes) bit 3: Tilt sensor mounting (0=up, 1=down)
2	Frequency	6	head frequency (kHz)
2	Type	8	head type
12	SerialNo	10	head serial number
176	System	22	system data
22	Spare	198	spare
2	NBeams	220	number of beams
2	Checksum	222	= b58c(hex) + sum of all bytes in structure

Total Size 224 Bytes

Bottom Check Configuration

The bottom check configuration is embedded in the User Configuration structure (following)

Size	Name	Offset	Description
2	size	0	Size of this data structure (in bytes: 44)
1	version	2	Structure version (3)
1	supported	3	1 if license supports bottom check
1	Enable	4	1 = enable, 0 = disable
1	rangeCompAmp	5	Compensate for range dependent attenuation (1). No compensation (0).
2	ProbeCheck	6	Enables / Disables probe check operation. Only relevant in the beamCheck structure (see below).
4	sampleRate	8	Sample rate (float: Hertz)
2	minRange	12	Requested minimum profiling range (mm)
2	maxRange	14	Requested maximum profiling range (mm)
2	nCells	16	Number of cells in profile
2	cellSize	18	Requested cell size (0.1mm)
4	cellSizeSelected	20	Actual cell size (float: in mm)
4	minRangeSelected	24	Actual minimum profiling range (Float: mm)
4	maxRangeSelected	28	Actual maximum profiling range (Float: mm)
2	gainReductiondB	32	Amount of gain reduction (dB)
10	pad	34	Unused

User Configuration

Set with the “CC” command. Retrieved with the “GC” command.

Size	Name	Offset	Description
2	size	0	Size of this data structure (in bytes:184)
1	version	2	Structure version (6)
1	CoordSystem	3	coordinate system (1=XYZ, 2=BEAM)
1	calcSpeedOfSound	4	Speed of sound (0=user,

			1=calculated internally)
1	syncType	5	0=None 1=On start 2=On measure 3=Master (Vectrino) 4=Master (Other)
1	nTransducers	6	Number of receive transducers (always 4)
1	nFrequencies	7	Number of Tx frequencies (always 1)
1	calcSampleRate	8	Must be set to 0 (Maximize ensemble count)
1	calcPingInterval	9	Ping interval algorithm 1 – Minimum ping interval to achieve range + dual PRF extended range 2 – Maximum ping interval to achieve ambiguity velocity + dual PRF extended range 3 – Adaptive ping interval + dual PRF extended range
1	powerLevel	10	1:Low Minus 2:Low 3:High Minus 4:High
1	internalMemory	11	Amount of internal memory available for buffering (in MB)
4	pad1	12	Pad bytes (4)
4	speedOfSound	16	Speed of sound (Float: in m/s)
4	sampleRate	20	Sample rate (Float: in Hertz)
12	Internal Use	24	not used
2	Pulse Length	36	Tx pulse length (in units of 0.1mm)
8	Internal Use	38	Not used
2	pingInterval	46	Interval between pings (us) (calculated)
2	ensemblePingPairs	48	Number of ping pairs in an ensemble (calculated)
2	cellSize	50	Requested cell size (0.1mm)
2	nCells	52	Number of cells in a profile
2	cellStart	54	Range to first cell (0.1mm)
4	cellSizeSelected	56	Cell size actually used (float: mm)
4	cellStartSelected	60	Range to first cell actually used (float: mm)
2	Salinity	64	Salinity in 0.1ppt

2	velocityRange	66	Ambiguity velocity (in mm/s)
2	horizontalVelocityRange	68	Horizontal velocity range in XYZ coordinates (calculated in mm/s)
2	verticalVelocityRange	70	Calculated vertical velocity range in ZYA coordinates (calculated in mm/s)
2	extendedPingInterval	72	Extended Velocity Range effective ping interval in us calculated: 0 is disabled). Actual ping intervals produced are dependent upon the selected extended range mode.
2	minCalibratedRange	74	Minimum supported XYZ calibration range (0.1mm)
2	maxCalibratedRange	76	Maximum supported XYZ calibration range (0.1mm)
2	rawSampleSeparation	78	Separation time between samples within an ensemble (first ping to first ping in us). Should be set to 0 (continuous pings). In unwrap extended range mode this is calculated (first coarse ping to first coarse ping in us).
2	Internal use	80	Must be 0
2	Reserved	82	Future use
2	Reserved	84	Future use
2	velocityExponent	86	Exponent of scale of raw velocity measurement (-3 = mm/s, -4 = 0.1m/s)
4	Pad3	88	Spare
44	bottomCheck	92	Bottom Check Configuration (see above)
44	beamCheck	136	Beam Check Configuration (see above)
2	Pad4	180	Unused
2	Checksum	182	=b58c (hex) + sum of all words in structure
Total Size 184 Bytes			

Probe Profile Calibration

The profile calibration consists of a series of 4x4 matrices used to convert from the beam co-ordinate data into XYZ co-ordinates (this is done

automatically within the instrument when XYZ co-ordinates are selected in the configuration). The profile calibration is retrieved with the "GR" command. It consists of a header section followed by a cell structure for each cell in the profile.

To create the 4x4 transformation matrix for each cell in the profile, divide each element by the header **scaleFactor** to produce a floating point number. To convert from beam co-ordinates to XYZ co-ordinates for a particular cell, perform the following matrix operation:

$$\begin{vmatrix} M1 & M2 & M3 & M4 \\ M5 & M6 & M7 & M8 \\ M9 & M10 & M11 & M12 \\ M13 & M14 & M15 & M16 \end{vmatrix} x \begin{vmatrix} V\ Beam\ 1 \\ V\ Beam\ 2 \\ V\ Beam\ 3 \\ V\ Beam\ 4 \end{vmatrix} = |Vx \quad Vy \quad Vz1 \quad Vz2|$$

Size	Name	Offset	Description
2	hdrChecksum	0	=b58c (hex) + sum of all words in structure (starting with the size)
2	Size	2	Size of this data structure (in bytes:72)
1	version	4	Structure version (1)
3	Reserved	5	Reserved
4	startRange	8	Start of calibrated range (0.1mm)
4	endRange	12	End of calibrated range (0.1mm)
4	cellSize	16	Length of each cell (0.1mm)
4	nCells	20	Number of cells in the profile
2	scaleFactor	24	Scaling factor to apply to matrix elements (matrix value = matrix element / scaleFactor)
2	cellsChecksum	26	=b58c (hex) + sum of all words in the cell structures
2	cellElemSize	28	Size of each cell structure
12	serialNo	30	Probe serial number (ASCII)
30	Reserved	42	Reserved
40 * nCells	Cell matrices	72	Data block containing a cell structure for each cell

Size	Name	Offset	Description
2	Size	0	Size of the cell data structure (in bytes:40)
2	startRange	2	Start range for this cell (0.1mm)
2	cellSize	4	Cell size (0.1mm)
2	Reserved	6	Start of calibrated range (0.1mm)
16 * 2	Matrix element	8	Matrix element (scaled by the header scaleFactor).

5.2 Measurement Mode Structures

When in measurement mode, the Vectrino Profiler will output data records. Each data record is prefaced by a protocol header containing the data ID (type) for that record. The following types may be produced (depending upon configuration and licensing):

00 50 Velocity Data Header

00 51 Velocity Data

00 61 Bottom Check Data

00 62 Beam Check Data

In addition to data records, the Vectrino Profiler also issues "information headers" describing two operating conditions within the instrument

01 00 Collection Stopped

01 01 Internal Memory Buffer Full

The Collection Stopped ID will be issued if an error condition occurs during data collection. This will be followed by an error string which can be read to determine the cause of the error.

The Internal Memory Buffer Full ID will be issued if the instrument is buffering data and the internal memory becomes full. Data collection will be stopped at this point since no further memory is available.

Velocity Data Header

Data ID = 0x0050

Size	Name	Offset	Description
2	checksum	0	=b58c(hex) + sum of all words in structure
1	status	2	Bits 0,1,2= number of beams Bit 3 = Internal memory approaching full Bit 4 = Processing error: Data possibly dropped Bit 5 = Internal buffering is occurring
1	headerOnly	3	0 if noise profile data is included, 1 if only the header is included
4	timeStamp	4	Time stamp (relative to start of collection in 100us)
2	nCells	8	Number of cells in the profile
2	pingInterval1	10	First ping interval in us
2	pingInterval2	12	Second ping interval in us (equal to first if dualPRF is disabled)
2	horizontalVelocityRange	14	Horizontal velocity range in XYZ coordinates (mm/s)
2	verticalVelocityRange	16	Vertical velocity range in XYZ coordinates (mm/s)
2	temperature	18	Temperature (0.01 Celsius)
2	soundSpeed	20	Speed of sound (0.1m/s)
2	adaptiveStatus	22	0 = OK 2 = failed
2*nCells	NoiseAmplitude Profile Beam 1	24	Noise amplitude beam 1 (linear counts)
2*nCells	NoiseAmplitude Profile Beam 2	24+2*nCells	Noise amplitude beam 2 (linear counts)
2*nCells	NoiseAmplitude Profile Beam 3	24+4*nCells	Noise amplitude beam 3 (linear counts)
2*nCells	NoiseAmplitude Profile Beam 4	24+6*nCells	Noise amplitude beam 4 (linear counts)
nCells	NoiseCorrelation Profile	24+8*nCells	Noise Correlation Beam

	Beam 1		1 (0-255 = 0-100%)
nCells	NoiseCorrelation Profile Beam 2	24+9*nCells	Noise Correlation Beam 2 (0-255 = 0-100%)
nCells	NoiseCorrelation Profile Beam 3	24+10*nCells	Noise Correlation Beam 3 (0-255 = 0-100%)
nCells	NoiseCorrelation Profile Beam 4	24+11*nCells	Noise Correlation Beam 4 (0-255 = 0-100%)
Total Size	**24 + 12*nCells Bytes**		

Velocity Data

Data ID = 0x0051

Size	Name	Offset	Description
2	checksum	0	=b58c(hex) + sum of all words in structure
1	Status	2	Bits 0,1,2= number of beams Bit 4 = Internal memory approaching full Bit 5 = Processing error: Data possibly dropped Bit 6 = Internal buffering is occurring
1	exponent	3	Velocity exponent (-3 = mm/s, -4 = 0.1 mm/s)
4	timeStamp	4	Time stamp (relative to start of collection in 100us)
2	nCells	8	Number of cells in the profile
2	temperature	10	0.01 Celsius
2	soundSpeed	12	Speed of sound (in 0.1m/s)
2	pingPairs	14	Number of ping pairs in the ensemble
2*nCells	Velocity profile B1/X	16	velocity beam1 or X (mm/s or 0.1mm/s: See configuration velocityExponent)
2*nCells	Velocity profile B2/Y	16+2*nCells	velocity beam2 or Y (mm/s or 0.1mm/s: See configuration velocityExponent)
2*nCells	Velocity profile B3/Z1	16+4*nCells	velocity beam3 or Z1 (mm/s or 0.1mm/s: See configuration velocityExponent)
2*nCells	Velocity profile B4/Z2	16+6*nCells	velocity beam4 or Z (mm/s or 0.1mm/s: See configuration

			velocityExponent)
2*nCells	Amp B1	16+8*nCells	amplitude beam1 (counts)
2*nCells	Amp B2	16+10*nCells	amplitude beam2 (counts)
2*nCells	Amp B3	16+12*nCells	amplitude beam3 (counts)
2*nCells	Amp B4	16+14*nCells	amplitude beam4 (counts)
nCells	Corr B1	16+16*nCells	correlation beam1 (0-255 = 0-100%)
nCells	Corr B2	16+17*nCells	correlation beam2 (0-255 = 0-100%)
nCells	Corr B3	16+18*nCells	correlation beam3 (0-250 = 0-100%)
nCells	Corr B4	16+19*nCells	correlation beam4 (0-250 = 0-100%)
nCells/4	DQ B1	16+20*nCells	Data Quality beam 1 (2 bits per cell) – Currently unused
nCells/4	DQ B2	16+20.25*nCells	Data Quality beam 1 (2 bits per cell) – Currently unused
nCells/4	DQ B3	16+20.5*nCells	Data Quality beam 1 (2 bits per cell) – Currently unused
nCells/4	DQ B4	16+20.75*nCells	Data Quality beam 1 (2 bits per cell) – Currently unused
1	Pad (if needed)	16+21*nCells	A pad byte is added to make the structure size an even number. This byte should be ignored.
Total Size 16 + 21*nCells Bytes (+1 if this calculation is odd)			

Bottom Check Data

Data ID = 0x0061

Size	Name	Offset	Description
2	checksum	0	=b58c(hex) + sum of all words in structure
1	status	2	Bits 0,1,2= number of beams Bit 4 = Internal memory approaching full Bit 5 = Processing error: Data possibly dropped Bit 6 = Internal buffering is occurring
1	pad	3	Unused
4	Timestamp	4	Time stamp (relative to start of

			collection in 100us)
2	nCells	8	Number of cells in profile
4	distanceToBottom	10	Distance to bottom (Float: in mm)
4	rangeStart	14	Range to first amplitude cell (Float: in mm)
4	resolution	18	Cell size resolution (Float: in mm)
2*nCells	Amplitude	22	Amplitude profile (if licensed: in linear counts)
2*nCells	curveFit	22+2*nCells	Curve fit to amplitude data (if licensed: in linear counts)
Total Size 22 + 4*nCells Bytes			

Beam/Probe Check Data

Beam check data ID = 0x0062

Probe check data ID = 0x0063

Size	Name	Offset	Description
2	checksum	0	=b58c(hex) + sum of all words in structure
1	status	2	Bits 0,1,2= number of beams Bit 4 = Internal memory approaching full Bit 5 = Processing error: Data possibly dropped Bit 6 = Internal buffering is occurring
1	nBeams	3	Number of beams in profile (4 for Vectrino Profiler)
4	Timestamp	4	Time stamp (relative to start of collection in 100us)
2	nCells	8	Number of cells in profile
4	rangeStart	10	Range to first amplitude cell (Float: in mm)
4	resolution	14	Cell size resolution (Float: in mm)
2*nCells	Amplitude Beam 1	18	Amplitude profile of Beam 1 (in linear counts)
2*nCells	Amplitude Beam 2	18 +2*nCells	Amplitude profile of Beam 2 (in linear counts)
2*nCells	Amplitude Beam 3	18 +4*nCells	Amplitude profile of Beam 3 (in linear counts)
2*nCells	Amplitude Beam 4	18 +6*nCells	Amplitude profile of Beam 4 (in linear counts)
2*nCells	detectedPeaks	18+8*nCells	Amplitude peaks detected by the peak detection algorithm (in linear counts)
Total Size 18 + 10*nCells Bytes			

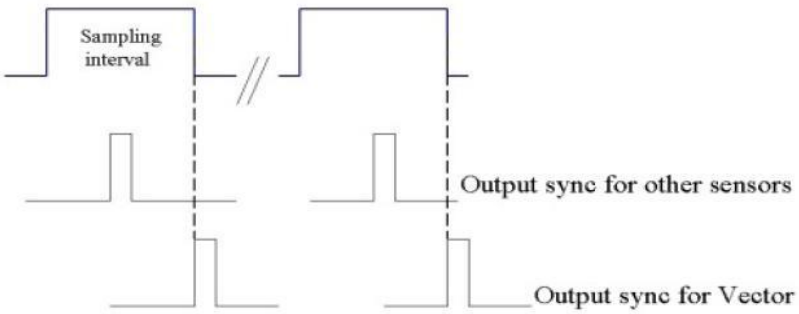
6 USE WITH OTHER INSTRUMENTS

6.1 SYNCHRONIZING WITH OTHER INSTRUMENTS

6.1.1 VECTOR

Sync out

The Vector can be synchronized with other instruments via the SyncIn and SyncOut. The synchronization option requires that the correct wiring harness is installed in your system. The SyncOut signal consists of 1.95 ms long, 3.3V pulses that can be configured for two different schemes of output synchronization. The selection of the type of SyncOut signal is made on the Advanced tab in deployment planning. If Output Sync for other sensor is selected, the SyncOut pulse will be output in the middle of each velocity-sampling interval. If Output Sync for Vector is used, the SyncOut pulse will be output at the completion of each sampling interval. In addition one SyncOut pulse will be output when the sampling of velocities is started.



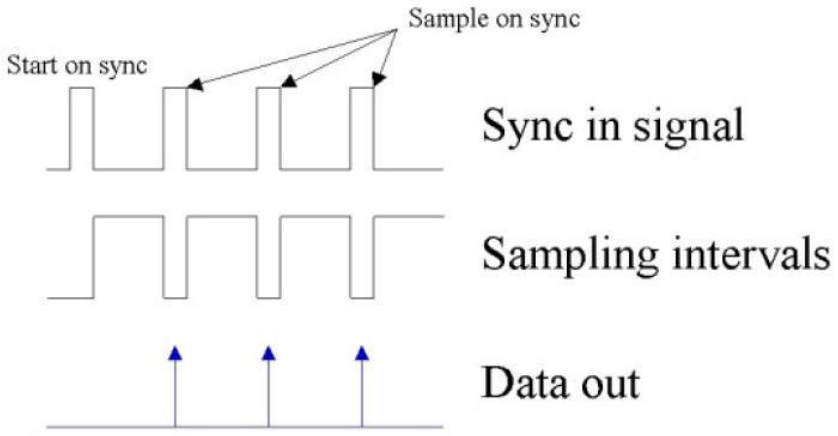
Sync In

The SyncIn signal permits external control of the sampling. There are three possible modes of operation that can be set for Input Sync:

• **No Sync.** In this mode, the Vector ignores the SyncIn signal and data collection starts under software control only.

• **Start on Sync.** In this mode, the Vector starts data collection on the rising edge of the SyncIn signal. Sampling of velocities then proceeds at the set sampling rate. After data collection is started the SyncIn is ignored.

• **Sample on Sync.** In this mode, the Vector outputs a sample after every rising edge of SyncIn. To use this mode, the Start on Sync mode must be used as well. Therefore, the first rising edge starts the averaging process for the first sample only. The first data sample is output on the second rising edge of SyncIn. The output data at each rising edge of SyncIn will correspond to an average since the previouis rising edge of the SyncIn. The Vector must be configured with the setup software for a sampling rate that is equal to or higher than the sampling rate that will be used. For example, if the signal on the SyncIn input to the Vector is generated to correspond to a sampling rate of 25 Hz, the Vector should be configured in software for a sampling rate of 32 Hz. In most cases it will be sufficient to synchronize different Vectors using Start on Sync. Instruments shipped after November 2000 are fitted with a real time clock with an accuracy of ±1 min/year over a temperature range of 0–40 °C. Over a burst period of for example 1 hour, the maximum clock drift between two Vectors will then be 13.7 ms. Since they will operate at the same temperature the clock drift is likely to be even smaller. At 8 Hz sampling rate this will be 1/10 of a sampling interval over one hour of measurements.



Output of the Vector system data (compass, tilt and temperature) is not synchronized between different Vectors through the use of the SyncIn signal. Instead, the start on the sync edge that is present in both synchronization modes is used to start the 1 Hz output of the system data. Each Vector will then output its system data based on its internal real time clock. This implies that the number of system data outputs may vary slightly from one instrument to another. Note that this is not a problem for the synchronization of the velocity measurements. The compass and tilt data are used internally in the Vector to ensure correct transformation of the velocities to earth coordinates when ENU is selected as coordinate system. In detail, each Vector configured for **Start on Sync** will output system data in a certain order at the start of continuous measurement and at the start of each burst when configured for burst measurements. **The system data are output in the following order**:

1 Output header containing noise measurements and date/clock.

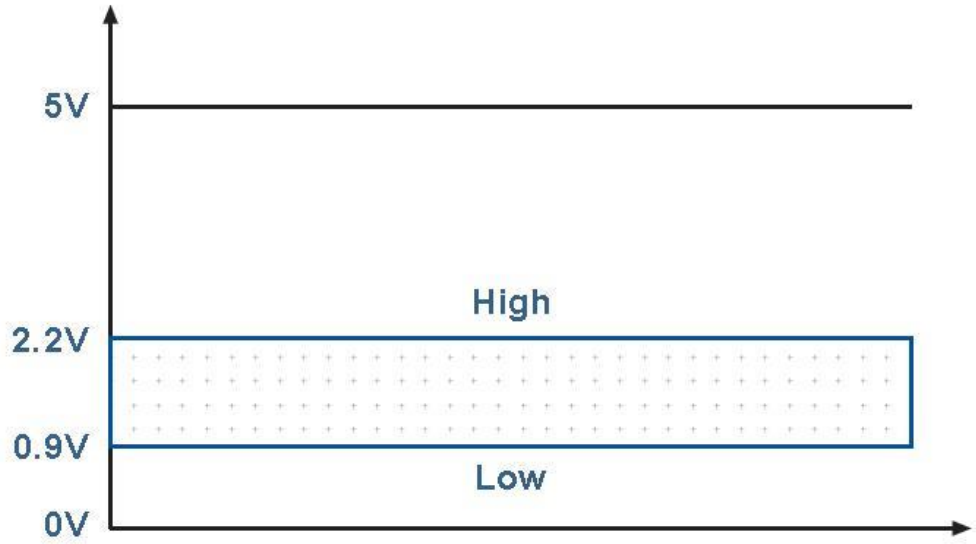
2 Output one sample of system data.

3 Wait until external trigger is detected and then start to output system data at a 1 Hz rate based on the local clock. Velocity data will be output according to the configured synchronization scheme.

Specifications of Signal Levels

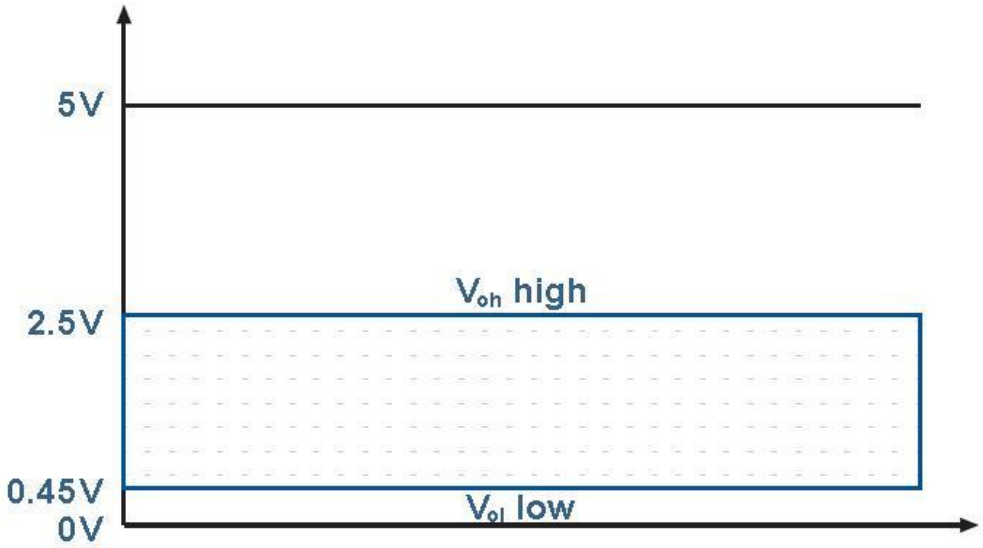
The SyncIn input voltage must be between 0 V and 5.0 V. The SyncIn is a Schmitt Trigger input with a pulldown resistor of 100kΩ to ground. The input threshold values for the SyncIn are:

- Vt+ input positive threshold, 2.2 V
- Vt- input negative threshold, 0.9 V



The output voltage levels for the SyncOut are:

- Voh high level output voltage, min 2.5 V
- Vol low level output voltage, max 0.45 V



There is spike protection on both signal ports but there is no filtering on the input port. It is important to consider noise issues (ground-loops, etc) as noise may cause an unwanted start on sync trigger.

Example 1 Running three Vectors with synchronized sampling of velocities in continuous mode.

Connect the SyncOut line from one Vector to the SyncIn line on the two other Vectors. Connect the ground cables together. Choose Output Sync for Vector for the first Vector. This will be the master. In the setup for the two other Vectors that will be slaves, check both boxes in the Input Sync (Start on sync and Sample on sync). With the rest of the setup identical for the three instruments start the two Vector slaves first. When the master Vector then is started, it will trigger the start of the other two Vectors.

Example 2 Running three Vectors with synchronized sampling of velocities in burst mode.

Connect the SyncOut line from one Vector to the SyncIn line on the two other Vectors. Connect the ground cables together. Choose Output Sync for Vector for the first Vector. This will be the master. In the setup for the two other Vectors that will be slaves, check both boxes in the Input Sync (Start on sync and Sample on sync). Use identical setup for the rest of the configuration parameters for all three instruments. Synchronize the clocks in all the Vectors. Start Recorder Deployment for the master Vector setting the deployment time to the time when you want the instrument to start (for example 4.00.00 p.m. or 16.00.00 – exact time format depends on your computer's settings). Start Recorder Deployment for the two slave Vectors setting the deployment time to the deployment time of the master Vector minus 10 seconds (in this example 3.59.50 p.m.). The two slave Vectors will now wake up in each burst 10 seconds ahead of the master Vector. After outputting the burst header and one set of system data they will then wait for the start on sync trigger from the master Vector so that all three instruments will start the data acquisition simultaneously (in

this example at 4.00.00 p.m.). The two slaves will then continue taking data at the identical rate of the master Vector as they receive the sample on sync triggers from the master Vector.

The same procedure can of course be used if only synchronized startup is required. The only difference is that the Sample on sync box in the Input Sync configuration must be left unchecked.

Example 3 Starting three Vectors simultaneously from another instrument

Connect the SyncIn signals from the three Vectors together with the sync output line from the instrument providing the start on sync signal. Ground all the four instruments together. Using identical and desired setups for the three Vectors, start all of them with the Start on sync option configured. Generate the start on sync trigger from the other instrumentation.

6.1.2 VECTRINO

Your Vectrino can be synchronized with other instruments via a pair of sync lines. To avoid noise problems and to make cabling easier, the two sync lines are RS485 which is balanced and bi-directional. The sync lines are labeled Sync- and Sync+.

Synchronizing Multiple Vectrinos

1 Connect all the Sync+ lines together and all the Sync- lines together. All instruments should share a common ground.

2 Select one instrument as the master and the rest as slaves (Input sync) in the software.

3 Start data collection in all slaves before starting the master. The slaves will then wait for the first sync pulse from the master before sampling commences. If you use **Sample on sync**, all instruments should be configured with the same sampling rate.

Synchronizing From a TTL Source

TTL signal levels can be mixed with the RS485 levels of the Vectrino: Connect your sync pulse to Sync+ and connect Sync- to a constant voltage, 1.5 volts for instance. This provides a defined transition when the TTL signal changes, which is not the case if you ground Sync-.

When Vectrino functions as master, the Sync+ line can be connected to the TTL input and the two systems must be connected so they share common ground.

How Synchronization Works

Output Sync (operating as the master) transmits a pulse on RS485 with a duration of 40 µs. SyncOut for Vectrino transmits the pulse by the end of the sampling interval and a single pulse at the beginning of the first sampling interval to start off everything. For other sensors, SyncOut transmits a pulse at the middle of the sampling interval. *Input Sync* triggers on any edge, rising or falling, on RS485. After each trig the input is discarded for the next 64 µs.

RS485 levels:

These levels define the signal the Vectrino reads for Input Sync:

High level: Sync+ > Sync- by 200 mV

Low level: Sync+ < Sync- by 200 mV

Noise problems:

If you encounter noise problems, try terminating the *Sync+* line at the TTL input by connecting a 120 ohm resistor in series with a 1nF capacitor between the *Sync+* line and ground. The best solution for external sync is using an RS485 as the input/output device.

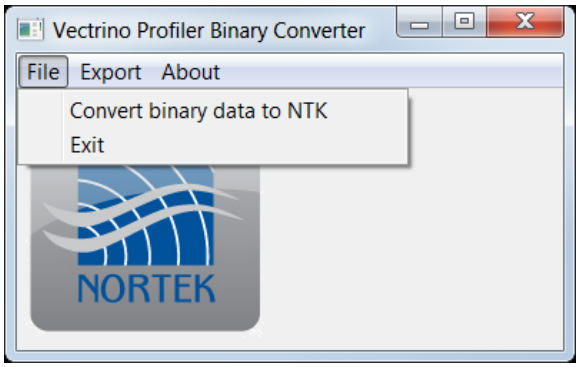
Note: This implies that a Vectrino can operate as a master for a Vector, but the opposite is not possible.

Example: Vectrino master and Vector slave

1. Connect ground of the two systems together
2. Connect sync+ (sync2) on the Vectrino to SyncInput on the Vector
3. Configure the Vectrino with the desired velocity range, 30 Hz sampling rate and select output **sync for Vectrino** and check the box for **master**
4. Configure the Vector with the desired velocity range, continuous sampling, the next higher available sample rate (so 32 Hz in this example) and check the boxes for **start on sync** and **sample on sync**
5. Select the file name for disk recording in both software instances and make sure you also **start disk recording** in both software instances at this point.
6. Press Start Data Collection in the **Vector** software
7. Press Start Data Collection in the **Vectrino** software
8. The measured data will now be shown on the display and stored to file, synchronized to each other.

7 CONVERSION UTILITY

A conversion utility (vProToNTK.exe) which allows raw Vectrino Profiler data collected directly from the serial port to be converted into the NTK file format. From there the data can be exported to either Matlab or ASCII format using the export functions. The data file collected must have the binary configuration records included in the file before the raw data records in order for the conversion utility to work. The configuration records must include a command header with the ID set to the equivalent "get" command (e.g. "GC" for the instrument configuration or "GH" for the head configuration; see the sample code following). As with the main acquisition application, the associated l4j.ini file (vProToNTK.l4j.ini) can be modified to increase the amount of memory used by the JVM to reduce the number of split files produced by the Matlab export function (see the section on "Increasing the amount of memory available to the application").



8 SAMPLE CODE

For your convenience, sample code showing integration with the Vectrino Profiler is included here.

The following examples are provided:

- Class definition for serial port operations
- Decoding the data structures and interacting with the instrument
- Structure definitions

This code also shows how to create a binary data output file that can be converted into the Nortek .NTK format using the included vProToNTK.exe utility that can be found in the installation directory of the software.

8.1 Serial Port Definition

```
class serialPort {
private:
	HANDLE handle;
	char errMsg[ERR_SIZE];

public:
	serialPort();
	bool open(int port);
	void close(void);
	bool configure(int32_t baudrate, int8_t bytesize, int32_t fparity, int8_t parity,
			int8_t stopbits);
	int read(char *buffer, DWORD min_bytes, DWORD max_bytes);
	bool timeouts (int readinterval, int readmult, int readconst,
							int writemult, int writeconst);

	int write(char *buffer, DWORD bytes);
	bool sendBreak(int ms);
	void flushIn(void);
	virtual ~serialPort();
};
```

8.2 Interacting with the Instrument and Decoding Data Structures

```
/**
 * @file VectrinoProfilerData.c
 *
 * Sample configuration and control program for the Vectrino Profiler
 * Velocimeter
 *
 * @author R.G.A. Craig
 * @par Edit History
 * - rcraig 13-JUN-2012 Doxygen comments added.
 * - rcraig 3-JULY-2014 Various updates (including saving a binary file)
 *
 * @par Copyright &copy; 2013-2014 Nortek Scientific Acoustics Group Inc.
 *                                 All rights reserved.
 *
 ************************************************************************/
#include <iostream>
using namespace std;
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <sys/time.h>
#include <windows.h>
#include <math.h>
#include <ctype.h>

#include "serialPort.h"
#include "VectrinoProfilerCust.h"


// Buffer used for instrument communications
#define BUFF_SIZE (1024*10)
char serBuff[BUFF_SIZE];

serialPort port;

tVectrinoProfilerConfig cfg;

// Baud rates to check during connection to the VectrinoII
int rates[] = { 9600, 19200, 38400, 57600, 115200, 230400, 460800, 937500, 1250000 };
#define DEFAULT_BAUD   937500

#define VPROFILER_NBEAMS                      4

#define COMMAND_FAILED         -1
#define COMMAND_SUCCEEDED      0

FILE *pBinaryFile;
```

```
///*****************************************************************************
//  Function: checkSum
//
/// Computes a checksum given a data buffer and the number of bytes in the
/// buffer
///
/// @param [in] buff  Pointer to the data
/// @param [in] nbytes Number of bytes in the buffer
///
/// @return The calculated checksum
///*****************************************************************************
static short checkSum(char *buff, int n)
{
	int i;
	short *p_short = (short *) buff;
	short check_sum = (short) 0xb58c;

	for (i = 0; i < n / 2; i++) {
		check_sum += p_short[i];
	}

	return (check_sum);
}

///*****************************************************************************
//  Function: resync
//
/// Reads the serial port a byte at a time until a valid protocol header is
/// received
///
/// @param[out] p_hdr  Header read in after re-sync
///
/// @return 0 if successful, 1 on failure
///*****************************************************************************
int resync(tCommandHeader *p_hdr)
{
	while (port.read((char *) p_hdr, 1, 1) == 1) {
		if (p_hdr->sync == SYNC_BYTE) {
			if (port.read(((char *) p_hdr) + 1, sizeof(tCommandHeader) - 1, sizeof(tCommandHeader)
- 1)
					== sizeof(tCommandHeader) - 1) {
				if (p_hdr->checksum == checkSum((char *) p_hdr, (sizeof(tCommandHeader) - 2)))
{
					return (1);
				} else {
					unsigned char *p_char = (unsigned char *) (p_hdr) + 1;
					for (unsigned int i = 0; i < sizeof(tCommandHeader) - 1; i++) {
						if (p_char[i] == SYNC_BYTE) {
							printf("S...\n\n");
						}
					}
				}
			}
		}
	}

	return (0);
}

///*****************************************************************************
//  Function: readBlock
//
/// Reads a full block (header + any associated data) from the serial port
/// @param[out] rx_data       Data read after the protocol header
/// @param[out] rx_length     Length of data block received
/// @param[out] do_resync     Synchronize the data stream to read a valid header
///
/// @return The ID contained in the header on success or 0x1515 on failure
///*****************************************************************************
static tCommandHeader readHdr;
```

```
int readBlock(char *rx_data, unsigned int *rx_length, int do_resync)
{
	int bytes;

	memset(&readHdr, 0, sizeof(readHdr));

	if (port.read((char *) &readHdr, 1, 1) == 0) {
		printf("No data on port.\n");
		return (VECTRINOPROFILER_ID_ERR);
	}
	if (readHdr.sync != SYNC_BYTE && do_resync) {
		printf("Unsynced\n");
		if (!resync(&readHdr)) {
			printf("Resync failed\n");
			return (VECTRINOPROFILER_ID_ERR);
		}
	} else {
		if ((bytes = port.read(((char *) &readHdr + 1), sizeof(readHdr) - 1, sizeof(readHdr) - 1)) !=
sizeof(readHdr) - 1) {
			printf("Read port failed (%d bytes read)\n", bytes);
			return (VECTRINOPROFILER_ID_ERR);
		}
	}

	short csum = checkSum((char *) &readHdr, (sizeof(readHdr) - 2));
	if (readHdr.checksum != csum) {
		// Flush port and return error. */
		port.flushIn();
		printf("Header checksum failed. Expected %d. Got %d. \n", readHdr.checksum, csum);
		return (VECTRINOPROFILER_ID_ERR);
	}

	if (readHdr.dataSize > 0) {
		if (readHdr.dataSize <= *rx_length) {
			if ((*rx_length = port.read((rx_data), readHdr.dataSize, readHdr.dataSize)) !=
readHdr.dataSize) {
				printf("Read extra failed (Got %d bytes, expected %d bytes)\n", *rx_length,
readHdr.dataSize);
				port.flushIn();
				return (VECTRINOPROFILER_ID_ERR);
			}
		} else {
			printf("Data size too large %d vs %d for ID %08x\n", readHdr.dataSize, *rx_length,
readHdr.ID);
			// Flush port.
			port.flushIn();
			return (VECTRINOPROFILER_ID_ERR);
		}
	} else {
		*rx_length = 0;
	}

	return (readHdr.ID);

}

///************************************************************************
//  Function: Send a command to the instrument
//
/// Reads a full block (header + any associated data) from the serial port
/// @param[in] id                    The command ID to be transmitted
/// @param[in] tx_data               Data to be transmitted after the protocol header
/// @param[in] tx_length       Length of data block to transmit
/// @param[out] rx_data        Data read after the protocol header
/// @param[out] rx_length      Length of data block received
/// @param[out] resync Synchronize the data stream to read a valid header
///
/// @return The ID contained in the header on success or 0x1515 on failure
///************************************************************************
int sendCommand(uint16_t id, char *tx_data, int tx_length, char *rx_data, unsigned int *rx_length, int
resync)
{
	tCommandHeader hdr;
```

```
	hdr.ID = id;
	hdr.refer = 0;
	hdr.dataSize = tx_length;
	hdr.sync = SYNC_BYTE;
	hdr.checksum = checkSum((char *) (&hdr), (sizeof(hdr) - 2));

	if (port.write((char *) (&hdr), sizeof(hdr)) != sizeof(hdr)) {
		return (VECTRINOPROFILER_ID_ERR);
	}
	if (tx_length > 0) {
		if (tx_length != port.write(tx_data, tx_length)) {
			return (VECTRINOPROFILER_ID_ERR);
		}
	}
	return (readBlock(rx_data, rx_length, resync));
}


///*************************************************************************
//  Function: Writes a data block to an already opened binary data file
//
/// @param[in] id                       The command ID to be used in the protocol header
/// @param[in] pData           Pointer to data to be written after the header
/// @param[in] length          Length of data to be written
///
/// @return 0x0606 on success or 0x1515 on failure
///*************************************************************************
int writeData(uint16_t id, char *pData, unsigned int length)
{
	tCommandHeader hdr;
	if (pBinaryFile == NULL) {
		return(-1);
	}

	hdr.ID = id;
	hdr.refer = 0;
	hdr.dataSize = length;
	hdr.sync = SYNC_BYTE;
	hdr.checksum = checkSum((char *) (&hdr), (sizeof(hdr) - 2));

	if (fwrite((void *) (&hdr), 1, sizeof(hdr), pBinaryFile) != sizeof(hdr)) {
		return (VECTRINOPROFILER_ID_ERR);
	}
	if (length > 0) {
		if (length != fwrite((void *)pData, 1, length, pBinaryFile)) {
			return (VECTRINOPROFILER_ID_ERR);
		}
	}
	return (VECTRINOPROFILER_ID_SUCCESS);
}

///*************************************************************************
//  Function: switchMeasurement
//
/// Switch the instrument into measurement mode
///
/// @return 0 on success -1 on failure
///*************************************************************************
int switchMeasurement(void)
{
	unsigned int length = BUFF_SIZE;

	if (sendCommand((int) VECTRINOPROFILER_COMMAND_MODE_MEASURE, NULL, 0, serBuff, &length,
			1) == VECTRINOPROFILER_ID_SUCCESS) {
		return (COMMAND_SUCCEEDED);
	} else {
		return (COMMAND_FAILED);
	}
}

///*************************************************************************
//  Function: switchCommand
//
/// Switch the instrument into command mode.
```

```
/// @param[out] response      Response string returned by instrument
/// @param[out] resp_length   Length of the response string
/// @param[in] resync         Whether or not to re-sync the input stream
/// @return 0 on success -1 on failure
///************************************************************************
int switchCommand(char *response, unsigned int *resp_length, int resync)
{

       int org_length = *resp_length;
       int res;

       port.write((char *) "@@@@@@", 6);
       Sleep(10);
       port.write((char *) VECTRINOPROFILER_BREAK_STRING, strlen(VECTRINOPROFILER_BREAK_STRING));

       Sleep(10);
       port.flushIn();
       if ((res = sendCommand(VECTRINOPROFILER_COMMAND_MODE_COMMAND, NULL, 0, response, resp_length, resync))
                       == VECTRINOPROFILER_ID_SUCCESS) {
               return (COMMAND_SUCCEEDED);
       } else {
               /* Possibly a data record.  Try reading another block to see if it's the "success" block.*/
               while (res == VECTRINOPROFILER_ID_VEL) {
                       *resp_length = org_length;
                       res = readBlock(response, resp_length, resync);
               }
               if (res == VECTRINOPROFILER_ID_SUCCESS) {
                       return (COMMAND_SUCCEEDED);
               } else {
                       return (COMMAND_FAILED);
               }
       }
}

///************************************************************************
//  Function: setConfig
//
/// Sets the configuration of the instrument
///
/// @param[in] config  Instrument configuration structure
/// @return NULL on success or an error string on failure
///************************************************************************
char * setConfig(tVectrinoProfilerConfig *config)
{
       unsigned int rx_length = BUFF_SIZE;
       config->version = VECTRINOPROFILER_CONFIG_VERSION;
       config->size = sizeof(tVectrinoProfilerConfig);
       config->checksum = checkSum((char *) (config), (sizeof(tVectrinoProfilerConfig) - 2));

       if ((sendCommand(VECTRINOPROFILER_COMMAND_SET_CFG, (char *) (config), sizeof(tVectrinoProfilerConfig),
serBuff,
                       &rx_length, 1)) == VECTRINOPROFILER_ID_SUCCESS) {
               return (NULL);
       } else {
               return (serBuff);
       }
}

///************************************************************************
//  Function: getConfig
//
/// Gets the current configuration of the instrument
///
/// @param[in] config  Instrument configuration structure
/// @return NULL on success or an error string on failure
///************************************************************************
char * getConfig(tVectrinoProfilerConfig *config)
{
       unsigned int rx_length = BUFF_SIZE;
       char *errString = NULL;

       if ((sendCommand(VECTRINOPROFILER_COMMAND_GET_CFG, NULL, 0, serBuff, &rx_length, 1)) ==
VECTRINOPROFILER_ID_SUCCESS) {
               if (rx_length == sizeof(tVectrinoProfilerConfig)) {
```

```
                        tVectrinoProfilerConfig *pcfg = (tVectrinoProfilerConfig *) (serBuff);
                        if (pcfg->version == VECTRINOPROFILER_CONFIG_VERSION) {
                                if (checkSum((char *) pcfg, (sizeof(tVectrinoProfilerConfig) - 2)) == pcfg-
>checksum) {
                                        *config = *pcfg;
                                } else {
                                        errString = (char *) "Configuration checksum mismatch";
                                        port.flushIn();
                                }
                        } else {
                                errString = (char *) "Configuration version mismatch";
                                port.flushIn();
                        }
                } else {
                        errString = (char *) "Incompatible firmware load";
                        port.flushIn();
                }
        } else {
                errString = serBuff;
        }
        return (errString);
}

///************************************************************************
//  Function: getID
//
/// Gets the instrument ID
///
/// @return NULL on success or an error string on failure
///************************************************************************
char * getID(void)
{
        unsigned int rx_length = BUFF_SIZE;
        serBuff[0] = 0;
        sendCommand(VECTRINOPROFILER_COMMAND_GET_ID, NULL, 0, serBuff, &rx_length, 1);
        serBuff[rx_length] = 0;
        return (serBuff);
}

///************************************************************************
//  Function: changeBaudRate
//
/// Changes the baud rate of the instrument and re-configures the
/// serial port to use the new baud rate.
///
/// @return NULL on success or an error string on failure
///************************************************************************
char * changeBaudRate(int rate, int portnum)
{
        char *errString = NULL;

        unsigned int rx_length = BUFF_SIZE;

        if ((sendCommand(VECTRINOPROFILER_COMMAND_SET_BAUDRATE, (char *) (&rate), 4, serBuff, &rx_length, 1))
                        == VECTRINOPROFILER_ID_SUCCESS) {
                errString = NULL;
                port.close();
                Sleep(100);

                port.open(portnum);
                port.flushIn();
                if (!port.configure(rate, 8, FALSE, NOPARITY, ONESTOPBIT)) {
                        return (char *) ("Serial port configure failed\n");
                }
        } else {
                errString = serBuff;
        }
        return (errString);

}

/* Zeroes out the averaged velocity data */
void initVelocities(int nCells, int nBeams, float avgVelocities[])
{
```

```
	for (int b = 0; b < nBeams * nCells; b++) {
		avgVelocities[b] = 0;
	}
}

///*************************************************************************
//  Function: collectData
//
/// Sample routine showing how to read data from the Profiler in measurement
/// mode.
/// param[in] full_time               Length of time to collect data (s)
/// param[in] discard_time     Discard all data at the start for this time period
/// param[out] avg_velocities  Averaged velocities from data read
///
/// @return 0 on success, -1 if no samples read.
///*************************************************************************
int collectData(float full_time, float discard_time, float avgVelocities[])
{
	int id;
	unsigned int rx_length;
	struct timeval tcurr, told, tstart;
	int csum;
	double elapsed_time;
	int nsamples = 0;
	int announce = 1;
	int nCells = 0;
	int init = 0;
	double vscale;
	int nBeams;

	vscale = pow(10.0, cfg.velocityExponent);

	gettimeofday(&tstart, NULL);
	elapsed_time = 0;
	while (elapsed_time < full_time) {

		rx_length = BUFF_SIZE;
		id = readBlock(serBuff, &rx_length, 1);

		gettimeofday(&tcurr, NULL);
		elapsed_time = (tcurr.tv_sec - tstart.tv_sec) + (tcurr.tv_usec - tstart.tv_usec) * 1.e-6;

		if (elapsed_time > announce) {
			printf("Collected %g seconds of data (%d samples)\n", elapsed_time, nsamples);
			announce++;
		}
		told = tcurr;
		if (readHdr.refer & VECTRINOII_HDR_STATUS_LOW_MEM) {
			// Running out of internal memory.
		}

		if (readHdr.refer & VECTRINOII_HDR_STATUS_BUFFERING) {
			// Transfer rate slower than internal collection rate.  Data is being buffered in
			// internal memory
		}

		if (readHdr.refer & VECTRINOII_HDR_STATUS_STOPPED_OOM) {
			// Internal data collection has stopped.  The internal data queue
			// has been filled.

		}

		if (readHdr.refer & VECTRINOII_HDR_STATUS_INT_ERROR) {
			// Internal error occurred.

		}

		switch (id) {
			case VECTRINOPROFILER_ID_VEL:
			{
				tVectrinoProfilerVelData *pVelS = (tVectrinoProfilerVelData *) serBuff;
				csum = checkSum((serBuff + 2), (rx_length - 2));
				if (csum != pVelS->checksum) {
					printf("Checksum mismatch %d vs %d\n\n", csum, pVelS->checksum);
```

```
                                        break;
                                }

                                if (((pVelS->status & 0x7) != nBeams) || (pVelS->nCells != nCells)) {
                                        printf("ERROR: Number of cells / beams in velocity record don't match
velocity header record");
                                        break;
                                }

                                writeData(VECTRINOPROFILER_ID_VEL, serBuff, rx_length);

                                if (pVelS->status & VECTRINOPROFILER_STATUS_MEM_FLAG) {
                                        // Running out of internal memory.  Data collection will be stopped
                                        // shortly.
                                }

                                if (pVelS->status & VECTRINOPROFILER_STATUS_BUFFERING_FLAG) {
                                        // Transfer rate slower than internal collection rate.  Data is being
buffered in
                                        // internal memory
                                }

                                if (pVelS->status & VECTRINOPROFILER_STATUS_DR_FLAG) {
                                        // The per sample processing time was exceeded.  Data may have been
dropped.
                                        // This is a very unusual occurrence.
                                }

//                              int npts = nBeams * nCells;
                                // Data is output in row major format.
                                tVelData *p_v = (tVelData *) (serBuff + sizeof(tVectrinoProfilerVelData));
//                              tAmpData *p_a = (tAmpData *) (p_v + npts);
//                              tCorrData *p_c = (tCorrData *) (p_a + npts);
//                              uint8_t *p_dq = (uint8_t *) (p_c + npts);

                                if (pVelS->timeStamp * 1.e-4 > discard_time) {
                                        for (int b = 0; b < nBeams; b++) {
                                                for (int c = 0; c < nCells; c++) {
                                                        avgVelocities[b * nCells + c] += vscale * p_v[b *
nCells + c];
                                                }
                                        }
                                        nsamples++;
                                }
                        }

                        break;

                        case VECTRINOPROFILER_ID_VEL_HEADER:
                        {
                                printf("Velocity header received\n");
                                tVectrinoProfilerVelHeader *pVelH = (tVectrinoProfilerVelHeader *) (serBuff);
                                csum = checkSum((serBuff + 2), (rx_length - 2));
                                if (csum != pVelH->checksum) {
                                        printf("Checksum mismatch %d vs %d\n\n", csum, pVelH->checksum);
                                        break;
                                }
                                writeData(VECTRINOPROFILER_ID_VEL_HEADER, serBuff, rx_length);
                                nCells = pVelH->nCells;
                                nBeams = (int) (pVelH->status & 0x7);

                                if (!init) {
                                        initVelocities(nCells, nBeams, avgVelocities);
                                }

                                if (pVelH->headerOnly != 0) {
//                                      int streams = pVelH->status & 0x7;
//                                      int bins = pVelH->nCells;
//                                      int npts = streams * bins;
//                                      tAmpData *pNoiseAmp = (tAmpData *) (serBuff +
sizeof(tVectrinoProfilerVelHeader));
//                                      tCorrData *pNoiseCorr = (tCorrData *) (pNoiseAmp + npts);
                                }
                        }
```

```
                        break;

                        case VECTRINOPROFILER_ID_BOTTOMCHECK:
                        {
                                tBottomCheckData *pBottom = (tBottomCheckData *) (serBuff);
                                csum = checkSum((serBuff + 2), (rx_length - 2));
                                if (csum != pBottom->checksum) {
                                        printf("Checksum mismatch %d vs %d\n\n", csum, pBottom->checksum);
                                        break;
                                }
                                writeData(VECTRINOPROFILER_ID_BOTTOMCHECK, serBuff, rx_length);
//                              tAmpData *pBottProfile = (tAmpData *) (serBuff + sizeof(tBottomCheckData));
//                              tAmpData *pCurveFit = (tAmpData *) (pBottProfile + pBottom->nCells);
                        }
                        break;

                        case VECTRINOPROFILER_ID_BEAMCHECK:
                        {
                                tBeamCheckData *pBeam = (tBeamCheckData *) (serBuff);
                                csum = checkSum((serBuff + 2), (rx_length - 2));
                                if (csum != pBeam->checksum) {
                                        printf("Checksum mismatch %d vs %d\n\n", csum, pBeam->checksum);
                                        break;
                                }
                                writeData(VECTRINOPROFILER_ID_BEAMCHECK, serBuff, rx_length);
                                printf("Beam check received %d %g %g\n", pBeam->nCells, pBeam->binResolution,
pBeam->rangeStart);
//                              tAmpData *pBeamProfiles = (tAmpData *) (serBuff + sizeof(tBeamCheckData));
//                              tAmpData *pDetectedPeaks = (tAmpData *) (pBeamProfiles + pBeam->nBeams *
pBeam->nCells);
                        }
                        break;

                        case VECTRINOPROFILER_ID_PROBECHECK:
                        {
                                tBeamCheckData *pProbeCheck = (tBeamCheckData *) (serBuff);
                                csum = checkSum((serBuff + 2), (rx_length - 2));
                                if (csum != pProbeCheck->checksum) {
                                        printf("Checksum mismatch %d vs %d\n\n", csum, pProbeCheck->checksum);
                                        break;
                                }
                                writeData(VECTRINOPROFILER_ID_PROBECHECK, serBuff, rx_length);
                                printf("Probe check received %d %g %g\n", pProbeCheck->nCells, pProbeCheck-
>binResolution,
                                                pProbeCheck->rangeStart);
//                              tAmpData *pProbeCheckProfiles = (tAmpData *) (serBuff +
sizeof(tBeamCheckData));
                        }
                        break;

                        case VECTRINOPROFILER_ID_BUFFER_FULL:
                                writeData(VECTRINOPROFILER_ID_BUFFER_FULL, serBuff, 0);
                                printf("Internal memory buffer full...\n\n");
                                break;
                        case VECTRINOPROFILER_ID_STOPPED:
                                writeData(VECTRINOPROFILER_ID_STOPPED, serBuff, 0);
                                printf("Collection stopped by instrument...\n\n");
                                break;
                        default:
                                port.flushIn();
                                printf("ERROR id=%08x\n\n", id);
                                break;
                }
        }

        if (nsamples > 0) {
                for (int n = 0; n < nCells * nBeams; n++) {
                        avgVelocities[n] /= nsamples;
                }
                return (COMMAND_SUCCEEDED);
        } else {
                return (COMMAND_FAILED);
        }
```

```
}

// Pause for a period of time before exiting.
void endit(void)
{
	for (int i = 0; i < 10; i++) {
		Sleep(1000);
	}
	if (pBinaryFile != NULL) {
		fclose(pBinaryFile);
		pBinaryFile = NULL;
	}
	exit(0);
}

int main(int argc, char *argv[])
{
	unsigned int rx_length;
	int portnum = 5;
	int baud = DEFAULT_BAUD;
	char *res;
	char *binaryFileName = NULL;
	char c;
	float tacq = 30.F;

	printf ("Collecting data");
	while ((c = getopt (argc, argv, "p:b:f:t:")) != -1) {
		switch (c) {
			case 'p':
				portnum = atoi(optarg);
				printf(" on comms port %d", portnum);
				break;
			case 'b':
				baud = atoi(optarg);
				printf(" at port speed %d", baud);
				break;
			case 'f':
				binaryFile = optarg;
				printf (" into binary file %s", binaryFileName);
				break;
			case 't':
				tacq = atof(optarg);
				printf(" for %g seconds", tacq);
				break;
			default:
				break;
		}
	}

	printf(".\r\n\r\n");

	if (!port.open(portnum)) {
		printf("Serial port open failed\n");
		endit();
	}

	unsigned int rt = 0xFFFFFFFF;
	if (!port.configure(baud, 8, FALSE, NOPARITY, ONESTOPBIT)) {
		printf("Serial port configure failed\n");
		endit();
	}

	if (binaryFileName == NULL) {
		pBinaryFile = NULL;
	} else {
		pBinaryFile = fopen(binaryFileName, "wb");
	}

	port.timeouts(50, 10, 1000, 10, 1000);
	rx_length = BUFF_SIZE;
	port.flushIn();

	bool connected = (COMMAND_SUCCEEDED == switchCommand(serBuff, &rx_length, 0));
```

```
	while (!connected && ++rt < (sizeof(rates) / sizeof *rates)) {
		port.close();
		Sleep(100);

		port.open(portnum);
		port.flushIn();
		baud = rates[rt];
		printf("Trying rate %d...", baud);
		if (!port.configure(baud, 8, FALSE, NOPARITY, ONESTOPBIT)) {
			printf("Serial port configure failed\n");
			endit();
		}
		if (COMMAND_SUCCEEDED == switchCommand(serBuff, &rx_length, 0)) {
			connected = true;
		} else {
			printf("Failed \n");
		}
	}

	if (!connected) {
		printf("\nNo instrument detected.\n\n");
		endit();
	}

	// Print out the ID string generated by the switch to command mode. */
	printf(serBuff);
	printf("\nConnected to %s\n", getID());

	// Get the configuration to initialize all of the configuration elements
	// before re-configuring.
	if ((res = getConfig(&cfg)) != NULL) {
		printf("Error: %s\n", res);
		endit();
	}

	if ((res = setConfig(&cfg)) != NULL) {
		printf("Error: %s\n", res);
		endit();
	}

	if (rt != 0xFFFFFFFF) {
		char *res = changeBaudRate(DEFAULT_BAUD, portnum);
		if (res != NULL) {
			printf("Change baud rate failed %s\n", res);
		}
		if (COMMAND_SUCCEEDED == switchCommand(serBuff, &rx_length, 0)) {
			if ((sendCommand(VECTRINOPROFILER_COMMAND_SAVE_BAUDRATE, serBuff, 0, serBuff,
&rx_length, 1))
					!= VECTRINOPROFILER_ID_SUCCESS) {
				printf("Save baudrate failed.\n\n");
			}
			connected = true;

		} else {
			connected = false;
		}
	}

	if (!connected) {
		printf("No instrument detected after baud rate change.\n\n");
		endit();
	}

	printf("Vectrino-Profiler found.  Current baud rate is %d Baud\n\n", baud);

	/* Write the configuration, head configuration, product configuration and profile calibration
information
	 *  into the binary data file. */
	writeData(VECTRINOPROFILER_COMMAND_GET_CFG, (char *)(&cfg), sizeof(tVectrinoProfilerConfig));


	rx_length = BUFF_SIZE;
	if (sendCommand(VECTRINOPROFILER_COMMAND_GET_HEAD, NULL, 0, serBuff, &rx_length, 1) ==
VECTRINOPROFILER_ID_SUCCESS) {
```

```
		writeData(VECTRINOPROFILER_COMMAND_GET_HEAD, serBuff, rx_length);
	}

	rx_length = BUFF_SIZE;
	if (sendCommand(VECTRINOPROFILER_COMMAND_GET_PROD, NULL, 0, serBuff, &rx_length, 1) ==
VECTRINOPROFILER_ID_SUCCESS) {
		writeData(VECTRINOPROFILER_COMMAND_GET_PROD, serBuff, rx_length);
	}

	rx_length = BUFF_SIZE;
	if (sendCommand(VECTRINOPROFILER_COMMAND_GET_PROFCALIB, NULL, 0, serBuff, &rx_length, 1) ==
VECTRINOPROFILER_ID_SUCCESS) {
		writeData(VECTRINOPROFILER_COMMAND_GET_PROFCALIB, serBuff, rx_length);
	}

	float *avgVelocities;
	avgVelocities = (float *) malloc(sizeof(float) * VPROFILER_NBEAMS * 100);
	rx_length = BUFF_SIZE;

	if (COMMAND_SUCCEEDED != switchMeasurement()) {
		printf("Switch into measurement mode failed.");
		endit();
	}

	collectData(tacq, 0.0, avgVelocities);

	rx_length = BUFF_SIZE;

	if (COMMAND_SUCCEEDED == switchCommand(serBuff, &rx_length, 1)) {
		printf("Average velocities collected \n\n");
	} else {
		printf("NO COMMAND MODE\n\n\n");
	}
	if (pBinaryFile != NULL) {
		fclose(pBinaryFile);
	}
	return 0;
}
```

8.3 STRUCTURE DEFINITIONS

```
#ifndef _VECTRINOPROFILERCUST_
#define _VECTRINOPROFILERCUST_

#include <stdint.h>

/* 2 byte aligned packing required.*/
#pragma pack(2)

#define SYNC_BYTE 0xA5

typedef struct {
	uint8_t sync;   // Synchronization byte (0xA5)
	uint8_t refer;  // Unused.
	uint16_t ID;            // Command / Data record ID
	uint16_t dataSize;      // Size of data to follow this header
	int16_t checksum;       // Word-wise checksum of this header
} tCommandHeader;

#define VECTRINOPROFILER_ID_ERR               0x1515
#define VECTRINOPROFILER_ID_SUCCESS    0x0606

#define COMMANDVALUE(a)       ((uint16_t)a[1] + (uint16_t)(a[0] << 8))
#define VECTRINOPROFILER_COMMAND_MODE_MEASURE          COMMANDVALUE("ST")      // Start measurement
#define VECTRINOPROFILER_COMMAND_MODE_COMMAND          COMMANDVALUE("MC")      // Command mode
#define VECTRINOPROFILER_COMMAND_GET_PROD                      COMMANDVALUE("GP")      // Get product
configuration
```

```
#define VECTRINOPROFILER_COMMAND_GET_HEAD                    COMMANDVALUE("GH")      // Get head
configuration
#define VECTRINOPROFILER_COMMAND_GET_PROFCALIB        COMMANDVALUE("GR")      // Get profile calibration
#define VECTRINOPROFILER_COMMAND_SET_CFG                     COMMANDVALUE("CC")      // Set configuration
#define VECTRINOPROFILER_COMMAND_GET_CFG                     COMMANDVALUE("GC")      // Get configuration
#define VECTRINOPROFILER_COMMAND_GET_ID                              COMMANDVALUE("ID")      // Get
instrument ID

#define VECTRINOPROFILER_COMMAND_SET_BAUDRATE         COMMANDVALUE("BR")      // Set baud rate
#define VECTRINOPROFILER_COMMAND_SAVE_BAUDRATE        COMMANDVALUE("SB")      // Save baud rate

#define VECTRINOPROFILER_BREAK_STRING         "K1W%!Q"

#define VECTRINOPROFILER_COORDS_EARTH         0
#define VECTRINOPROFILER_COORDS_XYZ           1
#define VECTRINOPROFILER_COORDS_BEAM          2
#define VECTRINOPROFILER_COORDS_PHASE         3

#define VECTRINOPROFILER_SYNC_NONE                   0
#define VECTRINOPROFILER_SYNC_START                  1
#define VECTRINOPROFILER_SYNC_MEASURE                2
#define VECTRINOPROFILER_SYNC_MASTER_VECTRINO        3
#define VECTRINOPROFILER_SYNC_MASTER_OTHER           4

#define VECTRINOPROFILER_POWER_LOW_MINUS             1
#define VECTRINOPROFILER_POWER_LOW                   2
#define VECTRINOPROFILER_POWER_HIGH_MINUS            3
#define VECTRINOPROFILER_POWER_HIGH                  4

#define VECTRINOPROFILER_MAX_FREQUENCIES             4
#define VECTRINOPROFILER_CONFIG_VERSION              6
#define VECTRINOPROFILER_BOTTOMCONFIG_VERSION        3

#define EXPORT(a,b,c)
#define EXPORTG(a,b,c,g)

#define VECTRINOPROFILER_ID_BUFFER_FULL              0x0100  // Indicates that internal buffering of
collected data
                                                             // has been exhausted and that data collection
is halting
#define VECTRINOPROFILER_ID_STOPPED                  0x0101  // Indicates that data collection was halted
                                                             // The reason is contained in the body of the
message

// Status bits reported in data records
#define VECTRINOPROFILER_STATUS_MEM_FLAG             0x08    // Indicates less than 10 sample buffers
remain in

      // internal memory buffer
#define VECTRINOPROFILER_STATUS_DR_FLAG              0x10    // Indicates "out of processing time" (data
possibly DRopped) problem
#define VECTRINOPROFILER_STATUS_BUFFERING_FLAG       0x20    // Indicates that internal buffering of data
is occurring



// Status bits reported in first four bits of header refer field
#define VECTRINOII_HDR_STATUS_LOW_MEM         0x10    // Indicates < 250K remains in data buffer
#define VECTRINOII_HDR_STATUS_BUFFERING       0x20    // Internal buffering of data is occurring
#define VECTRINOII_HDR_STATUS_STOPPED_OOM     0x40    // Data collection has stopped because the internal
data queue is full
#define VECTRINOII_HDR_STATUS_INT_ERROR              0x80    // Internal error has been detected


typedef struct {
      uint16_t size;                                         EXPORT(N, "", "")
      uint8_t version;                                EXPORT(N, "", "")
      uint8_t supported;                                     EXPORTG(Y, "", "1 = Profiled bottom check /
Dynamic adaptive check 2 = Bottom check / Static adaptive check", G)
      uint8_t enable;                                        EXPORTG(Y, "", "1 = Check enabled", B)
      uint8_t rangeCompensateAmp;                   EXPORTG(Y, "", "1 = Compensate for range dependent
attenuation in amplitude", B)
      uint8_t probeCheck;                                    EXPORT(N, "", "")
      uint8_t pad1;                                          EXPORT(N, "", "")
```

```
	float sampleRate;                                       EXPORTG(Y, "Hz", "Bottom check sample rate",
B)
	uint16_t minRange;                                      EXPORTG(Y, "mm", "Requested minimum range", B)
	uint16_t maxRange;                                  EXPORTG(Y, "mm", "Requested maximum range", B)
	uint16_t nCells;                                    EXPORTG(Y, "", "Number of cells to collect", B)
	uint16_t cellSize;                                  EXPORTG(Y, "0.1mm", "Requested cell sample length", B)
	float cellSizeSelected;                             EXPORTG(Y, "mm", "Selected cell sample length", G)
	float minRangeSelected;                             EXPORTG(Y, "mm", "Selected range to first cell", G)
	float maxRangeSelected;                             EXPORTG(Y, "mm", "Selected range to last cell", G)
	uint16_t gainReductiondB;                           EXPORTG(Y, "dB", "Amplifier gain reduction", B)
	uint16_t pad2[5];
} tBottomCheckConfig;

typedef struct {
	uint16_t size;                                      EXPORT(N, "","")
	uint8_t version;                                    // Version number of this structure.
	uint8_t coordSystem;                                EXPORTG(Y, "", "0=Earth 1=XYZ 2=Beam 3=Phase", B)
	uint8_t calcSpeedOfSound;                           EXPORTG(Y, "", "Speed of sound (0=fixed,
1=calculated)", B)

	uint8_t syncType;                                   EXPORTG(Y, "", "0=None 1=On start 2=On measure
3=Master (Vectrino) 4=Master (Other)", B)

	uint8_t nTransducers;                               EXPORTG(Y, "", "Number of receive transducers in
instrument", G)
	uint8_t nFrequencies;                               EXPORTG(Y, "", "Number of simultaneous frequencies
transmitted", G)
	uint8_t calcSampleRate;                             EXPORT(N, "", "")
	uint8_t calcPingInterval;                           EXPORTG(Y, "", "1:Minimum 2:maximum 3:adaptive.
Extended mode unwrap if bit 4=1, dual PRF otherwise", B)
	uint8_t powerLevel;                                 EXPORTG(Y, "", "1:Low- 2:Low 3:High- 4:High", B)
	uint8_t internalBufferMemory;                       EXPORTG(Y, "MB", "Internal memory available for
buffering data", G)
	uint8_t pad1[4];

	// 4 byte aligned
	float speedOfSound;                                     EXPORTG(Y, "m/s", "Speed of sound", B)
	float sampleRate;                                       EXPORTG(Y, "Hz", "Ensemble sample rate", B)
	uint16_t freq[VECTRINOPROFILER_MAX_FREQUENCIES];        EXPORT(Y, "kHz", "Transmit frequencies")
	uint8_t amp[VECTRINOPROFILER_MAX_FREQUENCIES];                  EXPORT(Y, "%", "Amplitude of each
pulse")
	uint16_t pulseLength[VECTRINOPROFILER_MAX_FREQUENCIES]; EXPORTG(Y, "0.1mm", "Transmit pulse length",
B)
	uint16_t blankingInterval;              EXPORT(Y, "us", "")
	uint16_t pingInterval;                  EXPORTG(Y, "us", "Time interval between pings", G)
	uint16_t ensemblePingPairs;             EXPORTG(Y, "", "Number of ping pairs to average together to
produce a measurement", G)
	uint16_t cellSize;                                      EXPORTG(Y, "0.1mm", "Requested cell sample
length", B)
	uint16_t nCells;                        EXPORTG(Y, "", "Number of cells to collect", B)
	uint16_t cellStart;                                     EXPORTG(Y, "0.1mm", "Range to first cell", B)
	float cellSizeSelected;                 EXPORTG(Y, "mm", "Selected cell sample length", G)
	float cellStartSelected;                EXPORTG(Y, "mm", "Selected range to first cell", G)
	uint16_t salinity;                      EXPORTG(Y, "0.1ppt", "", B)
	uint16_t velocityRange;                 EXPORTG(Y, "mm/s", "Velocity ranges from + to - this value",
B)
	uint16_t horizontalVelocityRange;       EXPORTG(Y, "mm/s", "", G)
	uint16_t verticalVelocityRange;         EXPORTG(Y, "mm/s", "", G)
	uint16_t extendedPingInterval;          EXPORTG(Y, "us", "Extended velocity range ping interval (0 is
disabled)", G)
	uint16_t minCalibratedRange;            EXPORTG(Y, "0.1mm", "Minimum supported XYZ calibrated range",
G)
	uint16_t maxCalibratedRange;            EXPORTG(Y, "0.1mm", "Maximum supported XYZ calibrated range",
G)
	uint16_t internaluse1;                  EXPORT(N, "", "")
	uint16_t internaluse2;                  EXPORT(N, "", "")
	uint16_t nCoarseCells;                  EXPORTG(Y, "", "Number of cells in the coarse lag
measurement", G)
	uint16_t coarseCellStart;               EXPORTG(Y, "0.1mm", "Start range to first cell in coarse lag
measurement", G)
	int16_t velocityExponent;               EXPORTG(Y, "","Raw velocity exponent (-3 = mm/s, -4 =
0.1mm/s)",G)
	uint16_t pad3[2];
	tBottomCheckConfig bottom;              EXPORT(Y, "", "")
```

```
      tBottomCheckConfig beam;               EXPORT(Y,"","")
      uint16_t pad4;

      // Word-wise checksum of the contents of this
      // structure (not including the checksum)
      int16_t checksum;                                      EXPORT(N, "", "")
} tVectrinoProfilerConfig;

typedef int16_t tVelData;
typedef uint16_t tAmpData;
typedef uint8_t tCorrData;
typedef uint8_t tDataQuality;

#define VECTRINOPROFILER_ID_VEL_HEADER        0x0050
typedef struct {
      int16_t checksum;               // Word-wise checksum
      uint8_t status;                 // Instrument status and data size:
                                      //     status & 0x7  = number of beams
                                      //     status & 0xF8 = status bits
      uint8_t headerOnly;             // If headerOnly, then no SNR data follows
      uint32_t timeStamp;             // Time stamp in units of 0.1 ms.  This is a
                                      // relative to the time that acquisition started
      uint16_t nCells;                //
      uint16_t pingInterval1;         // Ping separation of first ping pair (us)
      uint16_t pingInterval2;         // Ping separation of alternate ping pair (us)
      uint16_t horizontalVelocityRange;       // mm/s
      uint16_t verticalVelocityRange;         // mm/s
      int16_t temperature;            // Temperature (0.01 Celsius)
      uint16_t soundSpeed;            // Speed of sound (0.1m/s)
      uint16_t adaptiveStatus;        // Status of adaptive calculation (if enabled)

/*    Data added after header (if not "headerOnly")
      tAmpData noiseAmplitude[nBeams][nCells];        // In linear counts
      tCorrData  noiseCorrelation[nBeams][nCells];  // 255 = 100%
 */
} tVectrinoProfilerVelHeader;

/* Note:  The instrument always returns amplitude in linear units.  Conversion to
 dB is done by the acquisition software. */

#define VECTRINOPROFILER_ID_VEL                       0x0051
typedef struct {
      int16_t checksum;               // Word-wise checksum of the contents of this
                                                          // structure (not including the checksum)
      uint8_t status;                 // Instrument status and data size
                                      //     status & 0x7  = number of beams
                                      //     status & 0xF8 = status bits
      uint8_t pad;
      uint32_t timeStamp;             // Time stamp in units of 0.1 ms.  This is the
                                      // time relative to the when acquisition started
      uint16_t nCells;                // Number of cells in the profile
      int16_t temperature;            // Temperature (0.01 Celsius)
      uint16_t soundSpeed;            // Speed of sound (0.1 m/s)
      uint16_t pingPairs;             // Number of ping pairs averaged together to produce this sample

/*    Data added after header
      int16_t  velocity[nBeams][nCells];      // In mm/s
      uint16_t amplitude[nBeams][nCells];     // In linear counts
      uint8_t  correlation[nBeams][nCells];   // 255 = 100%
      uint8_t  data_quality[nBeams][nCells/4];  // Not currently used
 */
} tVectrinoProfilerVelData;

#define VECTRINOPROFILER_ID_BOTTOMCHECK               0x0061
typedef struct {
      int16_t checksum;               // Word-wise checksum of the contents of this
                                      // structure (not including the checksum)
                                      // checksum is at the START of this structure
                                      // to account for variable sized arrays at end.
      uint8_t status;                 // Instrument status
      uint8_t pad;                    //
      uint32_t timeStamp;             // Time stamp in units of 0.1 ms.  This is a
                                      // relative to the time that acquisition was
                                      // started.  Note that this will wrap every ~5 days.
```

```
      uint16_t nCells;
      float distanceToBottom;         // Interpolated distance to bottom (mm).
      float rangeStart;               // Start range of amplitude data (mm)
      float binResolution;            // Resolution of each bin in (mm)

/*  Data added after header: Instrument dependent.
      uint16_t amplitude[nCells];     // In linear counts
      uint16_t curveFit[nCells];      // In linear counts
 */
} tBottomCheckData;

#define VECTRINOPROFILER_ID_BEAMCHECK          0x0062
#define VECTRINOPROFILER_ID_PROBECHECK 0x0063

typedef struct {
      int16_t checksum;               // Word-wise checksum of the contents of this
                                      // structure (not including the checksum)
                                      // checksum is at the START of this structure
                                      // to account for variable sized arrays at end.
      uint8_t status;                 // Instrument status
      uint8_t nBeams;                 //
      uint32_t timeStamp;             // Time stamp in units of 0.1 ms.  This is a
                                      // relative to the time that acquisition was
                                      // started.  Note that this will wrap every ~5 days.
      uint16_t nCells;
      float rangeStart;               // Start range of amplitude data (mm)
      float binResolution;            // Resolution of each bin in (mm)
/*  Data added after header: Instrument dependent.
      uint16_t amplitude[nBeams][nCells];     // In linear counts
      uint16_t detectedPeaks[nCells];         // In linear counts
 */
} tBeamCheckData;

#define VECTRINOPROFILER_VERSION_PROBE_PROFILE_CALIBRATION 1
typedef struct {
      /* Size is included here as well as header to allow code generation tools
       * to easily generate the read/write routines. */
      uint16_t size;                  EXPORT(N, "", "")
      uint16_t startRange;            EXPORT(Y, "0.1mm", "Applicable start range for this cell")
      uint16_t cellSize;              EXPORT(Y, "0.1mm", "Cell size ")
      uint16_t pad;                   EXPORT(N, "", "")
      int16_t matrix[16];             EXPORT(Y, "", "Calibration matrix")
} tProbeCellCalibration;

#define VECTRINOPROFILER_PROBE_DOWNLOOKING_STEM               1
#define VECTRINOPROFILER_PROBE_DOWNLOOKING_CABLE      2
#define VECTRINOPROFILER_PROBE_SIDELOOKING_STEM               3
#define VECTRINOPROFILER_PROBE_SIDELOOKING_CABLE      4
#define VECTRINOPROFILER_PROBE_FIELD                          5

#define PD_MAX_HEADSERIALNO         12

typedef struct {
      /* Checksum goes first to ensure it's in the same place all the time for future versions
       of the structure. */
      int16_t hdrChecksum;            EXPORT(N, "", "")
      uint16_t size;                  EXPORT(N, "", "")
      uint8_t version;                EXPORT(Y, "", "Calibration version")
      uint8_t probeType;              EXPORT(Y, "", "")
      uint8_t pad[2];                 EXPORT(N, "", "")
      uint32_t startRange;            EXPORT(Y, "0.1mm", "Start of calibrated range")
      uint32_t endRange;              EXPORT(Y, "0.1mm", "End of calibrated range")
      uint32_t cellSize;              EXPORT(Y, "0.1mm", "Sampling length per cell")
      uint32_t nCells;                EXPORT(Y, "", "Number of transform matrices (one per cell)")
      uint16_t scaleFactor;           EXPORT(N, "", "")
      int16_t cellsChecksum;          EXPORT(N, "", "")
      uint16_t cellElemSize;          EXPORT(N, "", "")
      char serialNo[PD_MAX_HEADSERIALNO];     EXPORT(Y, "","Probe Serial Number")
      uint16_t reserved[15];          EXPORT(N, "", "")

/*  Transformation matrix data:
       tProbeCellCalibration beamToXYZ [nCells];
*/
```

```
} tProbeProfileCalibration;

#define PD_MAX_SERIALNO             14

#define VECTRINOPROFILER_ID_HEADCONF   0x04
typedef struct {
       unsigned char cSync;            EXPORT(N, "", "")  // sync = 0xa5
       unsigned char cId;              EXPORT(N, "", "") // identification = 0x04
       unsigned short hSize;           EXPORT(N, "", "") // total size of structure (words)
       unsigned short hConfig;       // head configuration:
       unsigned short hFrequency;      EXPORT(Y, "kHz", "Transducer Frequency")
       unsigned short hType;         // head type
       char acSerialNo[PD_MAX_HEADSERIALNO];
       short hBeamAngles[4];
       short hBeamToXYZ[16];           EXPORT(Y,"", "Beam to XYZ transformation matrix (scaled by 4096)")
       short hSpare0;                  EXPORT(N, "", "")
       unsigned short hSpare1;         EXPORT(N, "", "")
       short hCompAlignUp[9];
       short hCompAlignDown[9];
       short hTiltAlignUp[4];
       short hTiltAlignDown[4];
       unsigned short hPressCalib[4];
       short hTempCalib[4];
       short hTiltCalibUp[8];
       short hCompCalib[16];
       short hDatum[4];
       short hTiltRange;
       unsigned short hTiltScale;
       unsigned short hCompScale;
       unsigned short hTempScale;
       unsigned short hSampPos;      // sampling position (Vectrino) (Standard probe = ca 65 counts, Field
probe = ca 125 counts)
       unsigned short hProbeType;    // 3/2-d orientation (Vectrino)
       unsigned short hDistOffset;   // distance offset (mm) (Vectrino)
       unsigned short hSpare2[7];              EXPORT(N, "", "")
       unsigned short hProbeScale;   // probe scale factor (Vectrino)
       unsigned short nBeams;        // number of beams
       short hChecksum;                        EXPORT(N, "", "")
} tPdHeadConf;

#define PD_MAX_SERIALNO             14
#define PD_MAX_HEADSERIALNO         12
#define PD_MAX_FWVERSION            16
#define PD_MAX_FWDATE                           32

#define VECTRINOPROFILER_ID_PRODCONF                    0x05    // hardware production / configuration data
typedef struct {
       char acSerialNo[PD_MAX_SERIALNO];       // instrument type and serial number
       unsigned short hConfig;                 EXPORT(Y,"", "Board configuration: Bit 0 - Recorder installed
Bit 1 - Compass Installed Bit 2 - Compass Installed")
       unsigned short hFrequency;              EXPORT(Y,"kHz", "Board transmit frequency")
       unsigned short hPICversion;             // PIC Code version number
       unsigned short hHWrevision;             // Hardware revision
       unsigned short hRecSize;                EXPORT(Y,"64K Bytes", "Internal Recorder Size")
       unsigned short hSpare[7];               EXPORT(N, "", "")
       char cFWversion[PD_MAX_FWVERSION];      EXPORT(Y, "", "Firmware Version")
       char cFWRepoVersion[PD_MAX_FWVERSION];  EXPORT(Y, "", "Repository Revision")
       char cFWdate[PD_MAX_FWDATE];            EXPORT(Y, "", "Date firmware was built")
       short hChecksum;                        EXPORT(N, "", "")
} tPdProdConf;

#pragma pack()

#endif
```