Nanometrics Data Formats

Reference Guide

Nanometrics Data Formats

Reference Guide

Nanometrics Inc.
Kanata, Ontario
Canada

Nanometrics Data Formats Reference Guide

Nanometrics, Inc.
250 Herzberg Road
Kanata, Ontario, Canada K2K 2A1
Tel (613)592-6776
Fax (613)592-5929
Email info@nanometrics.ca

Part number 14602R4
Release date 2003-11-07

Contents

Chapter 1 NMXP Data Format

Data received on the serial port of the instrument are packetized in NMXP format and then these packets are embedded in standard UDP packets prior to transmission. This chapter defines the NMXP format for inbound data. It includes an overview of packet structure, a list of packet types, and descriptions of packets and bundles.

1.1 Overview

NMXP data transmission format facilitates the transfer of data along with a wide variety of status information from an instrument to a central site. The data format requires that the instrument have an accurate time source (i.e. GPS) for time tagging the data prior to transmission.

NMXP data format:

- Supports error free transmission of data using retransmission requests of bad packets.
- Is simple to implement, even on small microprocessors
- Is expandable: As new status information messages are created, they can be added to the data format without affecting the existing information.
- Supports programmable frequency for status information: Most of the status messages can be transmitted at a user defined frequency. This allows the user to tailor the ratio of data to status information. This is important on limited bandwidth or noisy transmission media.
- Provides efficient bandwidth usage

1.1.1 Overview of Protocol

Communication between the equipment and the Naqs receiver is completely stateless - there really is no protocol. When you turn on an instrument, it sends unsolicited data. Each packet is labelled with a channel-specific sequence number and also provides the sequence number of the oldest packet available for that channel. The receiver may send retransmission requests to request retransmission of certain packets (by sequence number). The instrument marks the requested packets for retransmission and sends them as bandwidth permits.

The order in which retransmitted packets are sent is not specified. HRDs and Europas send oldest retx first. The retx order is configurable in Janus and Europa-T instruments running firmware 5.60 and above.

The sequence number is an unsigned 4-byte integer which rolls over to 0 at 2^32. The protocol does not specify how the receiver should handle rollover.

1.1.2 Transport-specific wrapping

The packets described in this chapter are augmented with header bytes to facilitate transport over specific lower-level protocols.

Packets carried via serial communications are preceded by a synchronization word and followed by a 2-byte CRC, as follows:

2 bytes	synchronization word = 0xAABB
N bytes	packet payload
2 bytes	16-bit CRC

where the CRC is computed using the algorithm provided in Appendix B. The synchronization word and the CRC are sent in little-endian byte order.

Packets carried via UDP or TCP are preceded by a 12-byte header containing the following information:

4 bytes	synchronization word = 0x7ABCDE0F
4 bytes	message type = 1 for all inbound NMXP packets
4 bytes	message content length (packet length)

Note that these parameters are encoded in big-endian byte order.

1.1.3 Description of inbound packets

Data are gathered into sequenced and time stamped packets consisting of 17 byte "bundles". Each bundle is an independent collection of data. Each packet contains a word indicating the oldest packet available, and a time stamp bundle followed by n data bundles.

The number of bundles in a packet is a programmable parameter. The number of bundles is odd and has a range of 1-255. This allows the packet size to be tailored to the data link. Short packets should be used on noisy error prone data links. Packets may be the same size for the entire network, or different on each branch (a branch is connected to one RM-4 port) of the network. All instruments on a given branch must use the same packet size. Short messages must be padded out to the packet size.

Definitions:

- Inbound data: data that is being transmitted from the field stations to the central recording site
- Channel: a channel is a unique stream of information (e.g., serial port 1)
  - an instrument may transmit 1 or more channels of information
- Packet: a packet is a uniquely identifiable collection of information that is transmitted, composed of data bundles

- a packet contains information from only one channel
- inbound packets contain data, status, or configuration information

- Bundle: each bundle is an independent collection of data, for example time stamp information, status information, or data.
- Data is represented in the little endian format (Intel format) unless otherwise indicated

1.2 Inbound packet types

Inbound Packets (size = 4 + 17 +17 x (number of bundles)), where

4 bytes	Oldest packet available for a data stream
17 bytes	Packet header
17*n bytes	n bundles where n is odd

Compressed Data Packet		1
	Data Bundle	n/a
	Extended Header Bundle	0
	Null Bundle	9
State-of-Health Packet		2
	Fast State-of-Health - Obsolete	3
	Slow State of Health - Obsolete	4
	VCXO Calibration	7
	DSP Status Factory Test - Obsolete	8
	Null (indicates no more valid bundles in packet)	9
	Min-Max1	10
	Min-Max2	11
	Instrument Log (HRD / Orion)	12
	GPS Location	13
	GPS Error Bundle - Obsolete	14
	GPS Satellite Status/Reference Time Error	15
	D1 (Early) Threshold Trigger	20
	D2 (Late) Threshold Trigger	21
	D1 (Early) STA/LTA Trigger	22
	D2 (Late) STA/LTA Trigger	23
	Event	24
	RM-3 SOH	27
	RM-3 Rx Status	29
	Fast External State-Of-Health	32
	Slow External State-Of-Health	33
	Instrument SOH (generic)	34
	Orion Internal Temperature Slow SOH	35
	Orion Source Voltages Slow SOH	36
	Orion Powering Status Slow SOH	37

GPS Time Quality	39
GPS Satellite Information	40
Serial Port Map	41
Telemetry Packet Reader Errors	42
Serial Port Errors	43
Receiver Slot State	44
Transmitter Slot Error	45
Receiver Slot Error	47
Libra Instrument SOH	48
Libra Environmental SOH	49
Transmitter Address and Frequency	50
Receiver Address and Frequency	51
Burst Bundle	52
Epoch Bundle	53
Libra GPS Time Quality	54
Libra System Time Quality	55
Libra Operation State	56
Serial Data Bytes	57
Telemetry Packet Sender SOH	58
Authentication SOH	59

spare bundle numbers: 0, 1, 2, 5, 6, 16, 17, 18, 19,25, 26, 28, 30, 31, 38, 46, 69-255

Log Message Packet	5
Transparent Serial Packet	6

1.3 Packet header

1 byte	Packet type
4 bytes	Long seconds in seconds since 1970
2 bytes	packet specific
2 bytes	Instrument ID [bits 0-10 serial number, bits 11-15 model type]
4 bytes	Sequence Number
4 bytes	packet specific

The instrument ID defines the instrument type transmitting the channel of data:

0	HRD
1	ORION
2	RM-3
3	RM-4
4	LYNX
5	CYGNUS
6	EUROPA
7	CARINA
8	TimeServer

9	Trident
10	Janus
11-31	Reserved for future use

1.4 Compressed data packet

A data packet always consists of a timestamp header followed by n data bundles (where n is user defined). A timestamp bundle contains a sequence number, the time of the first sample, instrument ID (model and serial number), sample rate of packet and channel number, and the first sample.

1.4.1 Compressed data packet header

1 byte	Packet type = 1 (bit 5 = 1 indicates the packet is being retransmitted)
4 bytes	Long seconds
2 bytes	Sub-seconds in 10,000th of a second
2 bytes	Instrument ID [bits 0-10 serial number, bits 11-15 model type]
4 bytes	Sequence Number
1 byte	Sample Rate, Channel # [bits 0-2 channel number, bits 3-7 sample rate]
3 bytes	X0 (first sample) as a 24-bit signed integer (LSB first)

The instrument ID defines the instrument type transmitting the channel of data. Supported types are defined in section 1.3.

The sample rate is an enumerated value:

0	reserved	10	125 s/s
1	1 s/s	11	200 s/s
2	2 s/s	12	250 s/s
3	5 s/s	13	500 s/s
4	10 s/s	14	1000 s/s
5	20 s/s	15	25 s/s
6	40 s/s	16	120 s/s
7	50 s/s	17	240 s/s
8	80 s/s	18	480 s/s
9	100 s/s	19-31	Reserved for future use

1.4.1.1 Extended seismic data header

If the first data bundle has 0 in the compression byte, indicating that all four compressed data fields are not used, the bundle is an extended seismic data header.

1 byte	extended header = 0
4 bytes	XO (first sample, 32 bit version of the same field in main header)
1 byte	status

bit 0	channel 1 calibration in progress
bit 1	channel 2 calibration in progress
bit 2	channel 3 calibration in progress

	bit 3-7	unused
11 bytes	unused	

1.4.2 Data bundle

A data bundle contains between 4 and 16 compressed samples of data. The samples are compressed using a first difference algorithm. The data is compressed as byte, word, or long differences. Each set of four bytes contains either 4 byte differences, 2 word differences, or 1 long difference. The compression bits indicate how each set of 4 bytes is packed. For each 4 byte set there are 2 compression bits. The compression bits are packed into a byte as follows:

byte: ww xx yy zz	where the compression bits indicate:	
ww- data set 1	00	not used
xx- data set 2	01	byte difference
yy- data set 3	10	word difference
zz- data set 4	11	long difference

The format of the data bundle is as follows:

1 byte	Compression bits
4 bytes	Compressed data set 1
4 bytes	Compressed data set 2
4 bytes	Compressed data set 3
4 bytes	Compressed data set 4

1.4.3 Null bundle

This bundle is provided to pad out packets. The first occurrence of a Null bundle indicates that there is no further data in the packet. The null bundle contains no useful information. The receiver should disregard this bundle and all remaining bundles, and skip to the next packet.

1 byte	Bundle Type = 9
16 bytes	Filler

1.5 State-of-Health packets

A state-of-health packet consists of a status time stamp bundle followed by n status bundles. A status time stamp consists of a sequence number, the time (nominal time when the packet was created), instrument ID (model and serial number).

Status bundles have a general format that is outlined below:

1 byte	bundle type = xx
4 bytes	Long seconds
12 bytes	Defined by the specific bundle type

1.5.1 Status packet header bundle

1 byte	Packet type = 2 (bit 5 = 1 is for retransmit)
4 bytes	Long seconds

2 bytes	Sub-seconds in 10,000th of a second, this value always 0
2 bytes	Instrument ID [bits 0-10 serial number, bits 11-15 model type]
4 bytes	Sequence Number
1 byte	indicates test packet if (byte & 0x01 != 0)
3 bytes	Reserved for future use

1.5.2 VCXO calibration bundle

1 byte	Bundle type = 7
4 bytes	Long seconds
2 bytes	VCXO value (counts)
2 bytes	Time difference at Lock (counts, 3.84 counts = 1 microsecond)
2 bytes	Time Error (counts, 3.84 counts = 1 microsecond)
2 bytes	Frequency Error (in counts/sec (coarse lock) or counts/16 secs (fine lock))
2 bytes	Crystal temperature (counts)
1 byte	PLL Status? (1=fine locked, 2=coarse locking, 3 =temp. ref, gps off, 4=temp ref, gps on)
1 byte	GPS Status(0=3D, 1=2D, 2=1 sat, 3=search, 4= gps off, 5-6=gps error)

1.5.3 Null bundle

This bundle is provided to pad out packets. The first occurrence of a Null bundle indicates that there is no further data in the packet. The null bundle contains no useful information. The receiver should disregard this bundle and skip to the next packet.

1 byte	Bundle Type = 9
16 bytes	Filler

1.5.4 Min-Max1 bundle (Orion only)

The activity indicator provides a 1 Hz or slower filtered summary of a seismic data channel. This would be used to provide the end user with a summary of the collected data. This allows the user to quickly browse large quantities of data for events. The data may be filtered using a 5th order filter. The filter may be low pass, high pass, or band pass. In order not to lose the higher frequency information, the minimum and maximum over the interval of the filtered signal is stored. The interval is a programmable value of 1s or greater.

1 byte	Bundle type = 10
4 bytes	Long seconds
3 bytes	Filtered min. over 1st interval
3 bytes	Filtered max. over 1st interval
3 bytes	Filtered min. over 2nd interval
3 bytes	Filtered max. over 2nd interval

1.5.5 Min-Max2 bundle (Orion only)

The activity indicator provides a 1 Hz or slower filtered summary of a seismic data channel. This would be used to provide the end user with a summary of the collected data. This allows the user to quickly browse large quantities of data for events. The data may be filtered using a 5th order filter. The filter may be low pass, high pass, or band pass. In order not to lose the higher frequency information, the minimum and maximum over the interval of the filtered signal is stored. The interval is a programmable value of 1s or greater.

1 byte	Bundle type = 11
4 bytes	Long seconds
3 bytes	Filtered min. over 1st interval
3 bytes	Filtered max. over 1st interval
3 bytes	Filtered min. over 2nd interval
3 bytes	Filtered max. over 2nd interval

1.5.6 Instrument Log bundle (Orion/HRD only)

Any errors or warnings generated by the instrument are stored in this bundle. Some typical errors or warnings are GPS locked/unlocked, low battery, clock adjustments, external events, self test errors, status of disk space, duty cycle, etc.

1 byte	Bundle type = 12
4 bytes	Long seconds
2 bytes	Error code, where bits 0-11= error code, bits 12-15 = data format
2 bytes	Error Level
	ErrorLevel is a bit mapped value which is broken down as follows:
	bits 0-7 — Area (each bit identifies a separate area) - currently unused
	bits 8-10 — Processor (TCP, Aux, DSP)
	bits 11-15 — Error Level (Fatal, error, warning, info, debug)
8 bytes	Error Parameters

1.5.7 GPS Location bundle

This bundle contains the latitude and longitude of the instrument GPS antenna. This bundle has a programmable measurement frequency. The latitude and longitude is stored in IEEE floating point format.

1 byte	Bundle type = 13
4 bytes	Long seconds
4 bytes	Latitude
4 bytes	Longitude
4 bytes	Elevation

1.5.8 GPS Satellite Status/Reference Time Error bundle (Rockwell GPS-specific)

This bundle contains the status of the GPS engine's satellite tracking channels. It records the signal to noise ratio, activity, and satellite number for the five satellite tracking channels. The activity indicates whether the GPS channel is idle, searching or locked to a satellite signal. This information is very useful in diagnosing a GPS engine that is not locking.

1 byte	Bundle type = 15
4 bytes	Long seconds
2 bytes	Status bits (see Rockwell manual, contains operating mode, figure of merit)
10 bytes	GPS Satellite Channel - 2bytes per channel
	where the 2 bytes are defined:

bits 0-4	Satellite PRN code (0-31)
bits 5-7	Unused
bits 8-13	Signal to Noise Ratio (0-63)
bits 14-15	Activity 0=idle, 1 searching, 3=tracking

1.5.9 D1 (Early) Threshold Trigger bundle (Orion only)

The D1 threshold trigger bundle reports the start of a threshold trigger event. It is sent at a programmable time after the start of a trigger. It reports the start time of the trigger, along with some statistics about the trigger. The D1 trigger bundle is followed by a D2 trigger which reports the end of a trigger. The D1 bundle contains the peak amplitude, the half period of the amplitude, and the samples after trigger of the peak amplitude.

1 byte	Bundle type = 20
4 bytes	Long seconds
2 bytes	Sub-seconds in 10,000th of a second
2 bytes	LTA value (low word of LTA which is a long, hi word in D2)
2 bytes	Half period of peak amplitude
2 bytes	Samples after trigger of peak amplitude
1 byte	Channel# (3 bits) | trigger # (5 bits)
3 bytes	Peak amplitude

1.5.10 D2 (Late) Threshold Trigger bundle (Orion only)

The D2 threshold trigger bundle reports the end of a threshold trigger event. It is sent at a programmable time after a trigger is finished. It reports the end time of the trigger, along with some statistics about the trigger. The D2 bundle contains the peak amplitude, the half period of the amplitude, and the samples after trigger of the peak amplitude for the entire trigger event.

1 byte	Bundle type = 21
4 bytes	Long seconds
2 bytes	Sub-seconds in 10,000th of a second
2 bytes	LTA value (hi word of LTA which is a long, low word in D1)
2 bytes	Half period of peak amplitude
2 bytes	Samples after trigger of peak amplitude
1 byte	Channel# (3 bits) | trigger # (5 bits)
3 bytes	Peak amplitude

1.5.11 D1 (Early) STA/LTA Trigger bundle (Orion only)

The D1 STA/LTA trigger bundle reports the start of a STA/LTA trigger event. It is sent at a programmable time after the start of a trigger. It reports the start time of the trigger, along with some statistics about the trigger. The D1 trigger bundle is followed by a D2 trigger which reports the end of a trigger. The D1 bundle contains the peak amplitude, the half period of the amplitude, and the samples after trigger of the peak amplitude.

1 byte	Bundle type = 22
4 bytes	Long seconds
2 bytes	Sub-seconds in 10,000th of a second
2 bytes	LTA value (low word of LTA which is a long, hi word in D2)
2 bytes	Half period of peak amplitude
2 bytes	Samples after trigger of peak amplitude
1 byte	Channel# (3 bits) | trigger # (5 bits)
3 bytes	Peak amplitude

1.5.12 D2 (Late) STA/LTA Trigger bundle (Orion only)

The D2 STA/LTA trigger bundle reports the end of a STA/LTA trigger event. It is sent at a programmable time after a trigger is finished. It reports the end time of the trigger, along with some statistics about the trigger. The D2 bundle contains the peak amplitude, the half period of the amplitude, and the samples after trigger of the peak amplitude for the entire trigger event.

1 byte	Bundle type = 23
4 bytes	Long seconds
2 bytes	Sub-seconds in 10,000th of a second
2 bytes	LTA value (hi word of LTA which is a long, low word in D1)
2 bytes	Half period of peak amplitude
2 bytes	Samples after trigger of peak amplitude
1 byte	Channel# (3 bits) | trigger # (5 bits)
3 bytes	Peak amplitude

1.5.13 Event bundle (Orion only)

1 byte	Bundle type = 24
4 bytes	Long Seconds
4 bytes	End Time in Long seconds
1 byte	Cause (1=external, 2=internal, 4=manual (calibration))
1 byte	Trigger Flags (1 bit per trigger, LSB = trigger 0)
6 byte	spare

1.5.14 RM-3 SOH bundle (RM-3 only)

1 byte	Bundle type = 27
4 bytes	Long seconds
4 bytes	Battery voltage (float)
4 bytes	External SOH (float)
4 bytes	Temperature (float)

1.5.15 RM-3 Rx Status bundle (RM-3 only)

1 byte	us-int8	Bundle type = 29
4 bytes	us-int32	Long seconds
1 byte	us-int8	Rx Channel = 0, 1, 2, 3
1 byte	us-int8	Rx s/n ratio (average over the interval)
2 bytes	us-int16	number of valid data packets received during interval
2 bytes	us-int16	number of valid filler packets received during interval
2 bytes	us-int16	number of bad packets (CRC error) received during interval
2 bytes	us-int16	number of packets discarded (buffer overrun) during interval
2 bytes	us-int16	spare

1.5.16 Fast External State-Of-Health bundle

1 byte	bundle type = 32
4 bytes	long seconds
4 bytes	float of calibrated fast SOH1 in volts or units
4 bytes	float of calibrated fast SOH2 in volts or units
4 bytes	float of calibrated fast SOH3 in volts or units

1.5.17 Slow External State-Of-Health bundle

1 byte	bundle type = 33
4 bytes	long seconds
4 bytes	float of calibrated slow SOH1 in volts or units
4 bytes	float of calibrated slow SOH2 in volts or units
4 bytes	float of calibrated slow SOH3 in volts or units

1.5.18 Instrument SOH bundle

1 byte	bundle type = 34
4 bytes	long seconds
4 bytes	float of battery voltage measured at PSU in volts
4 bytes	float of temperature in degrees Celsius (VCXO temp on HRD)
4 bytes	(unused; or float of radio SNR in xxxx on HRD)

1.5.19 Orion Internal Temperature Slow SOH bundle (Orion only)

1 byte	bundle type = 35
4 bytes	long seconds
4 bytes	float of the Aux interface temperature in degrees Celsius
4 bytes	float of VCXO temperature in degrees Celsius
4 bytes	float of Disk Temperature in degrees Celsius

1.5.20 Orion Source Voltages Slow SOH bundle (Orion only)

1 byte	bundle type = 36
4 bytes	long seconds

4 bytes	float external battery voltage in volts
4 bytes	float internal battery voltage in volts
4 bytes	float mains voltage in volts

1.5.21 Orion Powering Status Slow SOH bundle (Orion only)

1 byte	bundle type = 37
4 bytes	long seconds
4 bytes	float charge current in Amps
4 bytes	float HRD PSU voltage in volts
1 byte	byte of external battery status
1 byte	byte of internal battery status
1 byte	byte of mains supply status
1 byte	switch status: 0 mains supply switch 1 internal battery switch 2 external battery switch 3 aux power switch 4 heater power switch 5 charger enable switch 6 charger high/low setting

1.5.22 GPS Time Quality bundle

This contains information about duty cycling and is produced only if the GPS is duty cycled.

1 byte	bundle type = 39
4 bytes	long seconds
2 bytes	GPS on time (in seconds)
2 bytes	GPS off time during the last cycle (in seconds)
2 bytes	GPS time to lock in seconds
2 bytes	Time difference at lock in counts (divide by 3.84 to get microseconds)
2 bytes	VCXO offset (div. by 16 to get the DAC offset)
1 byte	Reason GPS turned off: 0 -PLL finished correcting time error 1 -GPS on time expired
1 byte	Final GPS mode: 0 -3D navigation 1 -2D navigation 2 -tracking 1 sat or more 3 -searching for satellites

1.5.23 GPS Satellite Information bundle

1 byte	bundle type = 40

4 bytes	long seconds	
1 byte	MillisecFlag | Channel #	
		bits 0-3 Channel # (0-15)
		bits 4-7 Millisec Flag
		1 msec from sub_frame data collection
		2 verified by a bit crossing time
		3 verified by successful position fix
		4 suspected msec error
1 byte	Acquisition Flag | PRN	
		bits 0-4 PRN
		bits 5-7 Acquisition Flag:
		0 = unlocked
		1 = search
		2 = track
1 byte	Elevation (0-255): el= value/255x90	
1 byte	Azimuth (0-255: az = value/255x360	
2 bytes	Signal Level	
6 bytes	repeat for another channel - see the 6 bytes above	

1.5.24 Serial Port Map bundle

1 byte	int8	bundle type = 41
4 bytes	long	long seconds
1 bytes	int8	index
1 bytes	int8	serial port number
2 bytes	int16	number of minutes since last packet arrived
2 bytes	int16	HRD instrument ID (see data packets)
6 bytes	-	spare

1.5.25 Telemetry Packet Reader Errors bundle

1 byte	int8	bundle type = 42
4 bytes	long	long seconds
1 bytes	int8	serial port number
3 bytes	int24	Bad Packets since startup or start of the day
3 bytes	int24	Good Packets since startup or start of the day
3 bytes	int24	Lost Packets since startup or start of the day
2 bytes	int16	Tx Packets sent by Naqs since startup or start of the day

1.5.26 Serial Port Errors bundle

1 byte	int8	bundle type = 43
4 bytes	long	long seconds
1 bytes	int8	serial port number
4 bytes	long	serial port overrun errors since startup or last reboot (continuously increases, then wraps, it is never zeroed)

4 bytes	long	serial port frame errors since startup or last reboot (continuously increases, then wraps, it is never zeroed)
3 bytes	-	spare

1.5.27 Receiver Slot State bundle

1 byte	int8	bundle type = 44
4 bytes	long	long seconds
4 bytes	int32	receiver IP address
2 bytes	int16	DQT_AGC - AGC level for quadrature tuner in units of 0.1 dB
2 bytes	int16	carrier offset in units of 10 Hz
2 bytes	int16	symbol offset in Hz
1 byte	int8	DCL_AGC - AGC level for Costas loop in units of 0.1 dB
1 byte	-	spare

1.5.28 Transmitter Slot Error bundle

1 byte	int8	bundle type = 45
4 bytes	long	long seconds
4 bytes	int32	transmitter IP address
4 bytes	int32	no. of bad packets since the start of this TDMA configuration
4 bytes	int32	no. of good packets since the start of this TDMA configuration

1.5.29 Receiver Slot Error bundle

1 byte	int8	bundle type = 47
4 bytes	long	long seconds
4 bytes	int32	receiver IP address
4 bytes	int32	no. of bad packets since the start of this TDMA configuration
4 bytes	int32	no. of good packets since the start of this TDMA configuration

1.5.30 Libra Instrument SOH bundle

1 byte	int8	bundle type = 48
4 bytes	long	long seconds
2 bytes	int16	ten MHz frequency error
2 bytes	float16	SSPB temperature
2 bytes	float16	WW temperature
2 bytes	float16	TX temperature
2 bytes	float16	battery temperature
2 bytes	—	spare

1.5.31 Libra Environment SOH bundle

1 byte	int8	bundle type = 49
4 bytes	long	long seconds
4 bytes	float	external SOH channel 1 (scaled)
4 bytes	float	external SOH channel 2

4 bytes	float	external SOH channel 3

1.5.32 Transmitter bundle

1 byte	int8	bundle type = 50
4 bytes	long	long seconds
4 bytes	int32	transmitter IP address
4 bytes	int32	transmitter frequency in hHz
4 bytes	int32	transmitter level

1.5.33 Receiver bundle

1 byte	int8	bundle type = 51
4 bytes	long	long seconds
4 bytes	int32	receiver IP address
4 bytes	int32	receiver frequency in hHz
4 bytes	—	spare

1.5.34 Burst bundle

1 byte	int8	bundle type = 52
4 bytes	long	long seconds
4 bytes	int32	transmitter IP address
1 byte	int8	bits 0-1: slot state 0 = find (sweeping for carrier) 1 = verify (has carrier, looking for data) 2 = track (receiving data) bits 2-3: burst state for most recent burst 0 = not found 1 = found CW 2 = found UW 3 = found data
3 bytes	int24	no. of good burst since the start of this TDMA configuration
3 bytes	int24	no. of bad burst since the start of this TDMA configuration
1 byte	—	spare

1.5.35 Epoch bundle

1 byte	int8	bundle type = 53
4 bytes	long	long seconds
4 bytes	int32	next epoch start time (seconds since 1970)
8 bytes	—	spare

1.5.36 Libra GPS Time Quality bundle

1 byte	int8	bundle type = 54
4 bytes	long	long seconds

2 byte	short	GPS status
		0: computing position fixes (navigating)
		1: no_time
		2: needs initializing
		3: pdop_too_high (no solution)
		8 to 11: acquiring (8 + #satellites tracked))
2 bytes	short	number of usable satellites
4 bytes	float	PDOP value
4 bytes	float	TDOP value

1.5.37 Libra System Time Quality bundle

1 byte	int8	bundle type = 55
4 bytes	long	long seconds
4 bytes	int32	system time quality:
		-10: time_unknown
		-1: time_not_good
		n >= 0: worst prediction of time error in nsec
2 bytes	int16	PLL mode:
		1: fine_lock
		2: coarse_lock
		3: no_lock
2 bytes	int16	time displacement (system time - GPS time in nanoseconds)
2 bytes	int16	time velocity
2 bytes	float16	current compensation

1.5.38 Libra Operation State bundle

1 byte	int8	bundle type = 56
4 bytes	long	long seconds
4 bytes	int32	bitfield indicating operating state:
		bit 0 (LSB): network transmission state: on = 1, off = 0
		bits 1-31: reserved for future use
8 bytes	—	spare

1.5.39 Serial Data Bytes bundle

1 byte	int8	bundle type = 57
4 bytes	long	long seconds
1 byte	int8	port number
4 bytes	int32	bytes read since the startup or the start of the day
4 bytes	int32	bytes written since the startup or the start of the day
3 bytes	—	spare

1.5.40 Telemetry Packet Sender Soh bundle

1 byte	int8	bundle type = 58

4 bytes	long	long seconds
1 byte	int8	port number
3 bytes	int24	Bad command packets received since startup (mod 10 million)
3 bytes	int24	Good command packets received since startup (mod 10 million)
3 bytes	int24	Packets transmitted since startup (mod 10 million)
2 bytes	int16	Lost packets on receive since startup (mod 10 thousand)

1.5.41 Authentication Soh bundle

1 byte	int8	bundle type = 59
4 bytes	long	long seconds
4 bytes	int32	number of CD1 subframes built since startup (mod 1 billion)
2 bytes	int16	number of subframes with invalid signature since startup (mod 10 thousand)
2 bytes	int16	number of subframes with missing status since startup (mod 10 thousand)
2 bytes	int16	number of subframes with missing data samples since startup (mod 10 thousand)
2 bytes	—	spare

1.5.42 TimeServer Instrument Soh bundle

1 byte	int8	bundle type = 60
4 bytes	long	long seconds
2 bytes	float16	measured temperature of SOH circuit
2 bytes	float16	measured supply voltage of SOH circuit
2 bytes	float16	measured bus voltage of NMXbus
2 bytes	float16	measured external analog voltage
1 byte	int8	bits 0: bus termination indicator 0 = disabled 1 = enabled bits 1-7; reserved
3 bytes	uint24	uptime (minutes); time since last reboot in minutes (mod 10 million)

1.5.43 TimeServer Time PLL Soh bundle

1 byte	int8	bundle type = 61
4 bytes	long	long seconds
3 bytes	uint24	subsecond time in fast counts; multiply by 104.17 to get ns.
1 byte	int8	bits 0-3: status 0 = initializing 1 = no time 2 = raw time 3 = approximate time 4 = measuring frequency 5-6 reserved

		7 = no lock
		8 = coarse lock
		9 = fine lock
		10 = superfine lock
		11-15 reserved
		bits 4-7: time quality
		0= < 100 ns
		1 = < 200 ns
		2 = < 500 ns
		3 = < 1 micro s
		4 = < 2 micro s
		5 = < 5 micro s
		6 = < 10 micro s
		7 = < 20 micro s
		8 = < 50 micro s
		9 = < 100 micro s
		10 = < 1 ms
		11 = < 10 ms
		12 = < 100 ms
		13 = < 1 s
		14 = < 10 s
		15 = > 10 s
4 bytes	long	measured time error (fast counts); multiply by 104.17 to get ns; rails if actual measurement is larger.
1 byte	int8	measured frequency error (0.1 ppm); multiply by 0.96 to get Hz; rails if actual measurement is larger.
3 bytes	uint24	time since GPS lock loss; time spent in current state of GPS lock loss when applicable (mod 10 million).

1.5.44 TimeServer M12 GPS Soh bundle

1 byte	int8	bundle type = 62
4 bytes	long	long seconds
1 byte	int8	bits 7-5: tracking mode indicator
		2 = Bad Geometry
		3 = Acquiring Satellites
		4 = Position Hold
		5 = Propagate Mode
		6 = 2D Fix
		7 = 3D Fix
		bit 4: autosurvey mode
		0 = false
		1 = true
		bit 3: insufficient visible satellites
		0 = false

		1 = true
		bits 2-1: antenna status
		0 = ok
		1 = overcurrent
		2 = not connected
		3 = n/a
		bit 0: engine powered
		0 = not powered
		1 = powered
1 byte	int8	number of visible satellites
1 byte	int8	number of tracked satellites
1 byte	int8	UTC offset (s); difference between UTC and GPS time frame
2 bytes	short	clock bias (ns) of GPS engine
2 bytes	ushort	frequency bias (Hz) of GPS engine
2 bytes	float16	receiver temperature (deg C) on GPS engine
2 bytes	float16	measured antenna voltage (V)

1.5.45 NMXbus Master Soh bundle

1 byte	int8	bundle type = 63
4 bytes	long	long seconds
2 bytes	ushort	instrument id
3 bytes	uint24	number of slot requests received; (mod 10 million)
3 bytes	uint24	number of slot permits issued; (mod 10 million)
3 bytes	uint24	number of slot denials issued; (mod 10 million)
1 byte	—	spare

1.5.46 NMXbus Request Soh bundle

1 byte	int8	bundle type = 64
4 bytes	long	long seconds
2 bytes	ushort	instrument id
3 bytes	uint24	number of slot requests sent; (mod 10 million)
3 bytes	uint24	number of slot permits received; (mod 10 million)
3 bytes	uint24	number of slot denials received; (mod 10 million)
1 byte	—	spare

1.5.47 NMXbus Rx Soh bundle

1 byte	int8	bundle type = 65
4 bytes	long	long seconds
4 bytes	long	Rx good packets; number of good bus messages received (mod 1 billion)
4 bytes	long	Rx bytes; number of bytes received (mod 1 billion)
2 bytes	short	Rx buffer overrun; number of Rx FIFO overruns (mod 10,000)

2 bytes	short	HDLC errors; number of HDLC errors; CRC, abort or other (mod 10,000)

1.5.48 NMXbus Tx Soh bundle

1 byte	int8	bundle type = 66
4 bytes	long	long seconds
4 bytes	long	Tx good packets; number of good bus messages sent (mod 1 billion)
4 bytes	long	Tx bytes; number of bytes transmitted (mod 1 billion)
2 bytes	short	Tx buffer underrun; number of Tx FIFO underruns (mod 10,000)
2 bytes	short	discarded packets; e.g. Due to collisions or defers (mod 10,000)

1.5.49 NMXbus Device List Soh bundle

1 byte	int8	bundle type = 67
4 bytes	long	long seconds
2 bytes	ushort	instrument ID of device 1
2 bytes	ushort	instrument ID of device 2
2 bytes	ushort	instrument ID of device 3
2 bytes	ushort	instrument ID of device 4
2 bytes	ushort	instrument ID of device 5
2 bytes	ushort	instrument ID of device 6

1.5.50 Trident PLL Status Soh bundle

1 byte	int8	bundle type = 68
4 bytes	long	long seconds
2 bytes	ushort	current state
		0 = INIT (not digitizing)
		1 = TIME (correcting time error)
		2 = ACQ0
		3 = TRK1
		4 = TRK2
		5 = TRK3
		6 = TRK4
2 bytes	ushort	DAC counts; value to DAC to control VCXO (ranges from 0 to 4096)
4 bytes	float	time error (micro s); relative to TimeServer (+ve indicates Trident ahead)
4 bytes	float	temperature (deg C)

1.6 Log message packet

This packet contains log messages from a Nanometrics instrument. It consists of a timestamp header and up to 119 bytes (7 bundles) of formatted log message. Note that HRDs send log bundles (see section 1.5.6 on page 8) rather than log message packets.

1.6.1 Log message packet format

1 byte	us-int8	Packet type = 5 (bit 5 = 1 is for retransmit)
4 bytes	us-int32	Long seconds
2 bytes	us-int16	spare
2 bytes	us-int16	Instrument ID [bits 0-10 serial number, bits 11-15 model type]
4 bytes	us-int32	Sequence Number
2 bytes	us-int16	Error Number
1 byte	char	Error Severity (D, V, I, W, E, F) which means: debug, verbose, information, warning, error, fatal
1 byte	spare	
119 bytes	char	Error message (text message)

1.7 Transparent serial packet

A transparent serial packet contains a time stamp header followed by N data bytes (where N is user defined subject to N = 17 * k, where k is an integer, 1 <= k <= 28). k is typically 15, which gives N = 255. The time stamp bundle contains a sequence number, the time of the first sample, instrument ID, channel number, and the number of valid payload bytes, M.

Transparent serial packets are normally sent when the packet is full (M = N). However, the packet sender may be configured to send partial packets after a time out (i.e., if a specified time has passed since the first byte of the packet was received). In this case, M < N, and the last N - M bytes should be discarded. Partial packets are always padded out to full length.

1.7.1 Transparent serial packet format

1 byte	Packet type = 6 (bit 5 = 1 is for retransmit)
4 bytes	Long seconds
2 bytes	Sub-seconds in 10,000th of a second
2 bytes	Instrument ID [bits 0-10 serial number, bits 11-15 model type]
4 bytes	Sequence Number
2 bytes	Number of bytes of payload data
1 byte	Channel number (port number)
1 byte	spare
N bytes	binary serial data

1.7.2 Authentication information

Authentication information is carried in generic Transparent Serial Packets, with the following format. Whereas Transparent Serial Packet header is in least significant byte (LSB)-first order, for compatibility with NMXP protocol, the payloads are in most significant byte (MSB)-first order.

1 byte	packet type = 6
4 bytes	nominal frame time (seconds since 1970)
2 bytes	frame time subseconds = 0

2 bytes	instrument ID
4 bytes	packet sequence number
2 bytes	N = number of bytes of payload data
1 byte	channel number = seismic_channel + 16
1 byte	spare
N bytes	payload

For CD1.0, N = 60, in MSB-first order

The 60 byte payload contains a 4-byte internal header:

1 byte	version number = 0
2 bytes	number of actual samples signed
1 byte	header length = 56

plus 56 bytes of subframe information, as defined in IDC-3.4.2 Rev. 0.1, Table 6:

40 bytes	DSA signature
8 bytes	time of first sample (IEEE 8-byte float)
4 bytes	number of samples in subframe
4 bytes	status bytes

For CD1.1, N = 148, in MSB-first order

The 148 byte payload contains a 4-byte internal header:

1 byte	version number = 1
2 bytes	number of actual samples signed
1 byte	header length = 144

plus 144 bytes of subframe information, as defined in IDC-3.4.3 Rev. 0.2, Table 10:

24 bytes	channel description
20 bytes	time of first sample
4 bytes	subframe time length
4 bytes	number of samples in subframe
4 bytes	channel status size
32 bytes	channel status data, as defined in IDC-3.4.3 Rev. 0.2, Table 22
4 bytes	data size = 4 * number of samples
4 bytes	subframe count = 0
4 bytes	authentication key ID
4 bytes	authentication size = 40
40 bytes	authentication value (DSA signature)

Chapter 2 NMX Alert Format

Nanometrics Janus, Europa, and Libra family instruments can be enabled to send Alert frames to notify users of certain important state changes. These frames are send via either unicast or multicast UDP to an alert handler which forwards appropriate messages to a list of subscribers via email or other transport. Currently, the only alert handler which has been implemented by Nanometrics is AlertMailer, which forwards alert messages via email. For further information, see the AlertMailer reference manual.

2.1 Overview

Each Alert frame is issued in response to an important state change on the instrument, such as loss of contact with a VSat remote instrument, or supply voltage leaving the acceptable range. A corresponding message is issued when the condition is corrected.

Alert frames are sent via UDP to the alert destination address and port defined in the internet section of the instrument configuration. These packets are not acknowledged by the alert receiver, and there is no mechanism for requesting retransmission of an Alert frame.

2.2 Alert frame format

Each message contains information identifying the message source, error class and severity, and time of occurrence. It also contains a format string and a string of arguments describing the error condition in detail. The arguments and format string are sent separately to allow reformatting of the message by the alert forwarding software. Details of this argument replacement method are provided below.

The frame format is as follows:

4 byte int	packet identifier = 0x7ABCDE0F
4 byte int	message type = 320
4 byte int	message content length = 20 + sum of string lengths
4 byte int	Message severity
8 byte int	Message time (UT, milliseconds)
String	sourceID = name of the module which generated the alert
String	classID = type name of the alert message

String	format string - default format string for the message
String	arguments (first character is delimiter)

All integers are sent in big-endian byte order.

Each string is encoded as a 2-byte int (string length) followed by an array of ascii bytes (not zero-terminated).

2.2.1 Argument substitution

Each alert message type may contain a number of instance-specific arguments indicating, for example, a station name, earthquake magnitude, or other information. Alert format strings follow a simple but powerful convention which allows these arguments to be included anywhere, in any order, in the reformatted message. Arguments are indicated by special character sequences beginning with %, with defined tags as described in Table 2-1.

This method is very flexible, since it allows each argument to be referenced an arbitrary number of times in any order. This allows constructing of both complete and abbreviated messages, and accommodating different grammatical conventions which may be associated with different languages.

Table 2-1 Argument tags

Tag*	Description
%1, %2, ...%9	These represent the corresponding element from the argument list
%s	The sourceID of the module which generated the message
%m	The message type, or classID
%p	The message priority or severity
%a	The entire argument string, concatenated together, separated by a delimiting character (usually /)
%t	The time at which the message was generated
%r	A carriage return
%%	The % sign

* Unrecognized or invalid tags will be displayed in the formatted message as “*”.

2.3 Definition of Alert Messages

2.3.1 Alert Messages Generated by AlertMailer

2.3.1.1 AlertSystemUp

This message indicates that the AlertMailer system has started.

sourceID	AlertMailer
classID	AlertSystemUp
format string	Nmx Alert system is now running%r\

Components online: %1%r\
Components offline: %2%r

arguments /arg1/arg2

where arg1 = sourceIds of components which are online, and
arg2 = sourceIds of components which are offline

2.3.1.2 AlertSystemDown

This message indicates that the AlertMailer system is shutting down.

sourceID	AlertMailer
classID	AlertSystemDown
format string	Nmx Alert system is shutting down
arguments	none

2.3.1.3 ComponentOffline

This message indicates that AlertMailer is not receiving from the specified component.

sourceID	AlertMailer
classID	ComponentOffline
format string	No message received from %1 for %p minutes
arguments	/arg1

where arg1 = sourceId of component which is being reported offline

2.3.1.4 ComponentOnline

This message indicates that AlertMailer has started receiving from a component that was previously offline.

sourceID	AlertMailer
classID	ComponentOnline
format string	Started receiving from %1 after outage of more than %p minutes
arguments	/arg1

where arg1 = sourceId of component which just came online

2.3.1.5 MailErr

This message indicates that AlertMailer was unable to send a mail message.

sourceID	AlertMailer
classID	MailErr
format string	Failure to forward alert type: %1%rError message: %2
arguments	/arg1/arg2

where arg1 = classId of message for which send failed, and
arg2 = error string from system

2.3.2 Alert Messages Generated by NaqsServer

2.3.2.1 NaqsAlive

This message indicates that NaqsServer has just started.

sourceID	NaqsServer (or configured ID)
classID	NaqsAlive
format string	NaqsServer is now running.%r\
	Currently receiving from %1 / %2 instruments %r\
	The following instruments are online: %3 %r\
	The following instruments are offline: %4 %r
arguments	/arg1/arg2/arg3/arg4

where arg1 = number of instruments which are currently online
arg2 = total number of instruments in NaqsServer configuration
arg3 = list of stations which are currently online
arg4 = list of stations which are currently offline

2.3.2.2 NaqsStatus

This message is issued once per hour, giving current status.

sourceID	NaqsServer (or configured ID)
classID	NaqsStatus
format string	NaqsServer hourly status report.%r\
	Currently receiving from %1 / %2 instruments %r\
	The following instruments are online: %3 %r\
	The following instruments are offline: %4 %r
arguments	/arg1/arg2/arg3/arg4

where arg1 = number of instruments which are currently online
arg2 = total number of instruments in NaqsServer configuration
arg3 = list of stations which are currently online
arg4 = list of stations which are currently offline

2.3.2.3 NaqsReport

This message is issued once per day, giving uptime summary status for past 24 hours.

sourceID	NaqsServer (or configured ID)
classID	NaqsReport
format string	NaqsServer daily status report.%r\
	Uptime: %1 %r\
	Packets received today: %2 (retx %3 %%) %r\
	Received data from %4 / %5 instruments. %r\
	Received data from the following instruments: %6 %r\
	The following instruments remained offline: %7
arguments	/arg1/arg2/arg3/arg4/arg5/arg6/arg7

where arg1 = elapsed time since NaqsServer was last started
arg2 = number of packets received in past 24 hours (or since startup)
arg3 = percentage of packets received which were retransmitted packets
arg4 = number of instruments from which packets were received in past 24 hours
arg5 = total number of instruments in NaqsServer configuration
arg6 = list of stations from which packets were received in the past 24 hours
arg7 = list of stations from which NO packets were received in the past 24 hours

2.3.2.4 RbfOpenFail

This message is issued if one or more ringbuffers cannot be opened properly when NaqsServer starts up.

sourceID	NaqsServer (or configured ID)
classID	RbfOpenFail
format string	File open failed on %1 / %2 ringbuffers. See Naqs log.
arguments	/arg1/arg2

where arg1 = number of ringbuffers which could not be opened
arg2 = number of ringbuffers which NaqsServer attempted to open

2.3.2.5 RbfWriteFail

This message is issued when there is an error writing to a ringbuffer.

sourceID	NaqsServer (or configured ID)
classID	RbfWriteFail
format string	Ringbuffer write failure on %1, rc = %2, count = %p.
arguments	/arg1/arg2

where arg1 = name of channel for which write failed
arg2 = integer error code from program

2.3.2.6 RbfWriteOk

This message is issued when a ringbuffer write succeeds after previously being failed.

sourceID	NaqsServer (or configured ID)
classID	RbfWriteOk
format string	Ringuffer write succeeded on %1 after %p failures.
arguments	/arg1

where arg1 = name of channel

2.3.2.7 InstrumentOffline

This message is issued when Naqs has not received any data for 10 minutes from one or more instruments that were previously online.

sourceID	NaqsServer (or configured ID)
classID	InstrumentOffline
format string	NaqsServer has stopped receiving from %1%r\
	Currently receiving from %2 / %3 instruments %r\

	The following instruments are online: %4 %r\
	The following instruments are offline: %5 %r
arguments	/arg1/arg2/arg3/arg4/arg5

where arg1 = list of stations which have just gone offline
arg2 = number of instruments which are currently online
arg3 = total number of instruments in NaqsServer configuration
arg4 = list of stations which are currently online
arg5 = list of stations which are currently offline

2.3.2.8 InstrumentOnline

This message is issued when Naqs starts receiving from one or more instruments that were previously offline.

sourceID	NaqsServer (or configured ID)
classID	InstrumentOnline
format string	NaqsServer has started receiving from %1%r\
	Currently receiving from %2 / %3 instruments %r\
	The following instruments are online: %4 %r\
	The following instruments are offline: %5 %r
arguments	/arg1/arg2/arg3/arg4/arg5

where arg1 = list of stations which have just gone offline
arg2 = number of instruments which are currently online
arg3 = total number of instruments in NaqsServer configuration
arg4 = list of stations which are currently online
arg5 = list of stations which are currently offline

2.3.2.9 NaqsEvent

This message is issued when the event-detection module detects a seismic event.

sourceID	NaqsServer (or configured ID)
classID	NaqsEvent
format string	Naqs detected seismic event \
	%r Start time: %1 \
	%r Duration (seconds): %2 \
	%r Trigger type: %3 \
	%r Number of triggers: %4 \
	%r Peak Sta/Lta: %p \
	%r Stations: %5 %r
arguments	/arg1/arg2/arg3/arg4/arg5

where arg1 = time of the earliest trigger in the event (UT)
arg2 = duration of the event in seconds (until last de-trigger)
arg3 = trigger type from Naqs.stn file
arg4 = number of triggers in this event
arg5 = list of stations or channels which triggered for this event

2.3.3 Alert Messages Generated by each Carina Instrument (CARxxx)

2.3.3.1 VSatShutdown

This message is issued when Carina is shutdown manually (via shutdown command, entering TEST mode, etc.).

sourceID	Instrument ID (CARxxx)
classID	VSatShutdown
format string	Carina %s transmission shutdown (%1)
arguments	/arg1

where arg1 = list of reasons of shutdown

2.3.3.2 VSatResume

This message is issued when Carina transmission resumes.

sourceID	Instrument ID (CARxxx)
classID	VSatResume
format string	Carina %s transmission resumed
arguments	none

2.3.3.3 VSatTxOutage

This message is issued when Carina has NOT transmitted for N minutes, where N = 2, 5 and 20.

sourceID	Instrument ID (CARxxx)
classID	VSatTxOutage
format string	Carina %s has NOT transmitted for past %p minutes(%1)
arguments	/arg1

where arg1 = list of reasons of shutdown

2.3.3.4 VSatTxOk

This message is issued when Carina transmission resumes.

sourceID	Instrument ID (CARxxx)
classID	VSatTxOk
format string	Carina %s transmission resumed after %1 minutes
arguments	/arg1

where arg1 = number of minutes of outage

2.3.3.5 VSatSelfRxOutage

This message is issued when Carina has NOT received its own transmission for N minutes, where N = 2, 5 and 20.

sourceID	Instrument ID (CARxxx)
classID	VSatSelfRxOutage
format string	Carina %s has NOT received its own transmission for past %p minutes

arguments	none

2.3.3.6 VSatSelfRxOk

This message is issued when Carina self-reception resumes.

sourceID	Instrument ID (CARxxx)
classID	VSatSelfRxOk
format string	Carina %s now receiving from itself
arguments	none

2.3.3.7 VSatRxOutage

This message is issued when Carina has NOT received from VSat xx for N minutes, where N = 2, 5 and 20.

sourceID	Instrument ID (CARxxx)
classID	VSatRxOutage
format string	No packets received from %1 for %p minutes
arguments	/arg1

where arg1 = sourceId of remote which is not being received

2.3.3.8 VSatRxOk

This message is issued when Carina starts receiving data from a VSat after an outage.

sourceID	Instrument ID (CARxxx)
classID	VSatRxOk
format string	Now receiving from %1 after outage of %2 minutes
arguments	/arg1/arg2

where arg1 = sourceId of remote which was offline
arg2 = number of minutes of outage

2.3.4 Alert messages generated by each Cygnus, Janus, or Europa

2.3.4.1 PowerWarn

This message is issued when supply voltage enters RED zone (using thresholds from user interface).

sourceID	Instrument ID (e.g. EUR123)
classID	PowerWarn
format string	Battery voltage %1 Volts
arguments	/arg1

where arg1 = value of voltage

2.3.4.2 PowerOk

This message is issued when supply voltage enters GREEN zone.

sourceID	Instrument ID

classID	PowerOk
format string	Battery voltage %1 Volts
arguments	/arg1

where arg1 = value of voltage

2.3.4.3 SohWarn

This message is issued when an SOH reading enters the RED zone.

sourceID	Instrument ID
classID	SohWarn
format string	SOH %1 reading out of range(%2)
arguments	/arg1/arg2

where arg1 = name of SOH channel
arg2 = value of SOH reading

2.3.4.4 SohOk

This message is issued when SOH enters GREEN zone.

sourceID	Instrument ID
classID	SohOk
format string	SOH %1 reading OK(%2)
arguments	/arg1/arg2

where arg1 = name of SOH channel
arg2 = value of SOH reading

Chapter 3 Private Data Streams

NaqsServer provides online access to time-series, serial data, triggers, and state-of-health data via TCP subscription. The Stream Manager subsystem of NaqsServer acts as a data server; it accepts connections and data requests from client programs and forwards the requested data to each client program in near-realtime. This chapter describes version 1.4 of the protocol and data formats required for a client program to request, receive and interpret online data. C-language source code for a sample datastream client program is provided with NaqsClient as dsClient.cpp (also in Appendix A, “Data Stream Client”). See also the NaqsServer acquisition software manual pages.

3.1 Data stream types

The following data stream types are currently supported:

- Time-series data
- State-of-health data
- Transparent serial data
- Triggers
- Events

Time-series data may be requested in compressed or uncompressed format. Compressed data are in the original packet format received from the digitizer; uncompressed data are transmitted as 32-bit integer values. State-of-health data and transparent serial data are always transmitted in compressed format.

Compressed data may be requested as either raw or buffered streams:

- Raw stream: All packets (both original and retransmitted) are forwarded in the same order that they are received from the instrument by NaqsServer. Packets may be missing, duplicated, or out of order, but are received with minimal delay.
- Buffered stream: Short-term-complete data stream. Packets for each channel are guaranteed to be in chronological order, with short gaps filled by retransmitted packets.

NaqsServer maintains buffers of recent time-series, serial and state-of-health packets. Optionally, these buffered packets may be included at the beginning of a requested stream, before NaqsServer begins to send real-time data. Effectively, this enables the client program to request a stream which begins several packets in the past.

Triggers are detected using short-term-complete data; therefore trigger messages are always sent immediately when the trigger is detected. Similarly, events packets are always sent as soon as they are created.

In the current version, each client program (i.e. each socket connection) may subscribe to one or more data type. Client programs may subscribe to all data channels of a given type, or to any subset of the available channels. Each packet contains data for a single channel, and contains a key or name to identify the channel.

3.2 Subscription protocol

Every client program must implement the communication protocol summarized by the following steps. Italics indicate specific message types. Message formats are given in section 3.3.

1. Open a socket to the Stream Manager, using the stream manager port specified in the NaqsServer configuration. The default port is 28000.
2. Send a *Connect* message to Stream Manager.
3. Receive the *Channel List* from the Stream Manager. This is a list of the time-series and state-of-health channels available from the server.
4. (optional) Send a *Request Pending* message every few seconds until the request is ready. Once a connection is made, the client has 30 seconds to send a Request message before Stream Manager times out the connection. If the client application needs time to organize a packet request (for instance, if the client application must wait for user input), the client can send a *Request Pending* message to ensure the connection stays open. Each time Stream Manager receives a *Request Pending* message, it restarts its 30 seconds count down.
5. Send an *AddChannels* message to Stream Manager. The initial receipt of a *AddChannels* by Stream Manager stops the 30 second count down, and creates a new subscription for the channels indicated. Any subsequent *AddChannels* messages received by Stream Manager are treated as edits to the subscription.
6. Repeat until finished: receive and handle packets from Stream Manager, and (optionally) send a new *AddChannels* message (step 5) or a new *RemoveChannels* message whenever desired to change the subscription. The client should be prepared to process both *Error* messages and messages of the subscribed data type(s).
7. Send a *Terminate Subscription* message to Stream Manager to cancel the subscription when finished.
8. Close the socket.

3.3 Message formats

Each message consists of a 12-byte header and a variable length data content field. The header provides the type and length of the content. The client application should read the header first, then read the content after determining its type and length.

3.3.1 Client messages

Client messages are subscription protocol messages sent by the client to Stream Manager.

3.3.1.1 Connect

The purpose of the *Connect* message is to prove to Stream Manager that a valid client is requesting a connection. It has no content.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 100
4 byte int	message content length = 0

3.3.1.2 Request Pending

The *Request Pending* message is sent to reassure Stream Manager that the client is still alive and intends to make a request eventually. It has no content.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 110
4 byte int	message content length = 0

3.3.1.3 Terminate Subscription

A *Terminate Subscription* message is typically sent to Stream Manager by the client to indicate the end of the request. However, Stream Manager may also end the connection if an error occurs. There are currently 3 *Terminate Subscription* message types defined:

- Normal Shutdown: sent by the client to indicate the end of a subscription
- Error Shutdown: sent by Stream Manager to indicate that a fatal error has occurred
- Timeout Shutdown: sent by Stream Manager to indicate that client has not sent a subscription in the allotted time.

The string message portion of a *Terminate Subscription* message is used by Stream Manager to provide a more detailed description of why the connection was terminated. It is not necessary for a client to include a string message in any *Terminate Subscription* messages that it sends.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 200
4 byte int	message content length = 4 + N

Content:

4 byte int	Reason for termination 1 = Normal shutdown 2 = Error shutdown 3 = Timeout shutdown
N byte string	String message (none if N = 0)

3.3.2 AddChannels messages

Clients send an *AddChannels* message to subscribe to certain data streams. An *AddChannels* message contains the following information:

1. Data type requested (time-series, state-of-health, or triggers).
2. Channels requested (given by an array of channel indices, or keys).
3. Delay for the short-term completion buffer (buffered streams only).
4. Parameters for the decimation filter (time-series only).
5. Parameters for buffered packet request (time-series, serial and state of health data only)

An *AddChannels* message can be used in two ways: to add channels to a subscription, or to edit the parameters for a currently subscribed channel. If Stream Manager receives an *AddChannels* message which includes a channel already in the subscription, it checks if parameters in the *AddChannels* message (the short-term completion and decimation parameters if they apply) are different from the current parameters for the channel. If the parameters are different, the old values are discarded for the channel, and packets from that channel are processed using the new instructions.

The current version of Stream Manager recognizes the five types of Add Channels messages described below.

3.3.2.1 AddTime-SeriesChannels

The *Add Time-SeriesChannels* message is used to request time-series packets. Time series streams can be buffered to ensure that packets for each channel are output in chronological order. When packets are missed, Stream Manager will wait a specified period of time for the gap to be filled by retransmitted packets. Specifying a completion time of 0 will guarantee that packets are in chronological order, without waiting for missed data.

Time series streams can be requested in three different output formats: compressed data at the original sample rate, uncompressed data at the original sample rate, or uncompressed data decimated using Stream Manager's built in decimating FIR filters. Uncompressed time-series data is typically sent in packets of fixed length (one second duration). However, missing incoming data may cause Stream Manager to output a shorter packet.

An *AddDataChannels* message can also include a request to receive packets from the Naqs buffer of recent packets, before receiving the stream of new packets. This, effectively, moves the start of the stream several packets into the past.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 120
4 byte int	message content length = 16 + 4 * N, where N = number of channels

Content:

4 byte int	number of channels requested = N (use 0 to request all channels)

N * 4 byte int	Channel key for each requested channel (same as in Channel List message)
4 byte int	Short-term-completion time = s , -1 <= s <= 300 seconds (-1 indicates no short-term completion)
4 byte int	Output format -1 = compressed packets 0 = uncompressed packets, original sample rate 0 < r = requested output sample rate
4 byte int	Buffer flag 0 = do not send buffered packets for these channels 1 = send buffered packets for these channels

3.3.2.2 AddSohChannels

The *Add SohChannels* message is used to request state-of-health packets. Like time-series streams, state-of-health streams can be buffered to ensure that packets for each channel are output in chronological order. When packets are missed, Stream Manager will wait a specified period of time for the gap to be filled by retransmitted packets. Specifying a completion time of 0 will guarantee that packets are in chronological order, without waiting for missed data. A completion time of -1 instructs Stream Manager to make no attempt to output packets in chronological order.

Like time series streams, state-of-health streams can also be requested to include a buffer of recent packets at the beginning of the stream, thereby moving the start of the stream several packets into the past.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 121
4 byte int	message content length = 12 + 4 * N, where N = number of channels

Content:

4 byte int	number of channels requested = N (use 0 to request all channels)
N * 4 byte int	Channel key for each requested channel (same as in Channel List message)
4 byte int	Short-term-completion time = s , -1 <= s <= 300 seconds (-1 indicates no short-term completion)
4 byte int	Buffer flag 0 = do not send buffered packets for these channels 1 = send buffered packets for these channels

3.3.2.3 AddSerialChannels

The *Add SerialChannels* message is used to request transparent serial data packets. Like time-series streams, transparent serial streams can be buffered to ensure that packets for each channel are output in chronological order. When packets are missed, Stream Manager will wait a specified period of time for the gap to be filled by retransmitted packets. Specifying a completion time of 0 will guarantee that packets are in chronological order, without waiting for missed data. A completion time of -1 instructs Stream Manager to make no attempt to output packets in chronological order.

Like time series streams, transparent serial streams can also be requested to include a buffer of recent packets at the beginning of the stream, thereby moving the start of the stream several packets into the past.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 124
4 byte int	message content length = 12 + 4 * N, where N = number of channels

Content:

4 byte int	number of channels requested = N (use 0 to request all channels)
N * 4 byte int	Channel key for each requested channel (same as in Channel List message)
4 byte int	Short-term-completion time = s , -1 <= s <= 300 seconds (-1 indicates no short-term completion)
4 byte int	Buffer flag 0 = do not send buffered packets for these channels 1 = send buffered packets for these channels

3.3.2.4 AddTriggerChannels

The *AddTriggerChannels* message is used to request trigger packets. Currently Stream Manager ignores any channel keys indicated in the *AddTriggerChannels* message, responding instead by sending trigger packets for all channels in the network. Consequently, any attempt to edit the channels in an existing subscription by sending an *AddTriggerChannels* message with a different list of channel keys will provoke a warning message from Stream Manager.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 122
4 byte int	message content length = 4 + 4 * N, where N = number of channels

Content:

4 byte int	number of channels requested = N (use 0 to request all channels)
N * 4 byte int	Channel key for each requested channel (same as in Channel List message)

3.3.2.5 AddEvents

This message is used to request event packets. There are no parameters to an *AddEvents* message; Stream Manager sends event packets for every event that it identifies. Consequently, editing of an event subscription is not allowed, and any *AddEvents* message sent after the initial request will provoke an error.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 123

4 byte int	0

3.3.3 Remove Channels messages

A client sends a *RemoveChannels* message to remove some or all channels from its subscription. A *RemoveChannels* message contains the following information:

1. Data type of the channels to remove (time-series, state-of-health, or triggers).
2. Channels to be removed (given by an array of channel indices, or keys).

If some of the channels indicated in the *RemoveChannels* message are not in the current subscription, or do not correspond to the data type indicated in the message, Stream Manager will respond with an error message stating that there were some channels it was unable to remove.

3.3.3.1 RemoveTimeSeriesChannels

The *RemoveTimeSeriesChannels* message is used to remove time-series channels from the client's subscription.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type =130
4 byte int	message content length = 4 + 4 * N, where N = number of channels

Content:

4 byte int	number of channels to remove = N (use 0 to remove all channels)
N * 4 byte int	Channel key for each requested channel (same as in Channel List message)

3.3.3.2 RemoveSohChannels

The *RemoveSohChannels* message is used to remove state-of-health channels from the client's subscription.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type =131
4 byte int	message content length = 4 + 4 * N, where N = number of channels

Content:

4 byte int	number of channels to remove = N (use 0 to remove all channels)
N * 4 byte int	Channel key for each requested channel (same as in Channel List message)

3.3.3.3 RemoveSerialChannels

The *RemoveSerialChannels* message is used to remove transparent serial channels from the client's subscription.

Header:

4 byte int	Signature = 0x7ABCDE0F

4 byte int	message type =134
4 byte int	message content length = 4 + 4 * N, where N = number of channels

Content:

4 byte int	number of channels to remove = N (use 0 to remove all channels)
N * 4 byte int	Channel key for each requested channel (same as in Channel List message)

3.3.3.4 RemoveTriggerChannels

The *RemoveTriggerChannels* message is used to remove trigger channels from the client's subscription. Currently, Stream Manager maintains an all-or-nothing policy toward trigger subscriptions. A remove-all *RemoveTriggerChannels* message (number of channels = 0) will cause Stream Manager to delete all trigger channels from the subscription. Any *RemoveTriggerChannels* message with a non-empty list of channel keys will provoke an *Error* message from Stream Manager.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type =132
4 byte int	message content length = 4 + 4 * N, where N = number of channels

Content:

4 byte int	number of channels to remove = N (use 0 to remove all channels)
N * 4 byte int	Channel key for each requested channel (same as in Channel List message)

3.3.3.5 RemoveEvents

The *RemoveEvents* message is used to unsubscribe from the event packet stream.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type =133
4 byte int	0

3.3.4 Server messages

These are subscription protocol messages sent by Stream Manager to the client.

3.3.4.1 Channel List

A *Channel List* message contains a list of the data streams available. Each stream is identified by an unique integer key (which encodes digitizer/channel information) and an 11-character ASCII string which encodes the station and channel name. The channel list contains time-series, serial and state-of-health channels. The data type for a given channel can be determined from the channel key using the following formula: type = ((key >> 8) & 0xff).

Header:

4 byte int	Signature = 0x7ABCDE0F

4 byte int	message type = 150
4 byte int	message content length = 4 + N * 16

Content:

4 byte int	number of channels available = N

N 16-byte channel info bundles of the form:

{

4 byte int	Channel key = ((ID << 16) | (type << 8) | channel) where ID is the full instrument serial number type is the data subtype: 1 = time series 2 = state of health 6 = transparent serial channel is the data channel number (0 to 5)
12 bytes	zero-terminated channel name string (e.g. STN01.BHZ).

}

3.3.4.2 Error

Error messages are sent to the client by Stream Manager to indicate some error condition - usually in response to an invalid request from the client.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 190
4 byte int	message content length = N

Content:

N byte string	Error message

3.3.5 Data messages

3.3.5.1 Compressed Data, Soh or Transparent Serial Packets

This message contains data in the original compressed format generated by the Nanometrics instrument. Details of the packet contents are given in Chapter 1, “NMXP Data Format”.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	Data type = 1
4 byte int	Data content length = variable

Content:

4 byte int	Oldest sequence number
N bytes	N byte compressed data packet

3.3.5.2 Decompressed Data Packets

This message contains decompressed time-series data in fixed-length blocks (usually one second).

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	Data type = 4
4 byte int	Data length = 20 + 4 * N, where N = number of samples

Content:

4 byte int	Channel key (same as in Channel List message)
8 byte double	Time of first sample (seconds since January 1, 1970)
4 byte int	Number of samples in this packet = N.
4 byte int	Sample rate (samples per second).
N * 4 byte int	Samples as 32-bit integers

3.3.5.3 Trigger Packet

Trigger messages contain information on triggers detected by the Naqs internal Sta/Lta trigger-detection system.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	Data type = 5
4 byte int	Data length = 56

Content:

12 bytes	Station-channel name as zero-terminated string.
4 byte int	The trigger TypeID from the naqs.stn file
8 byte double	The trigger-on time, that is, the time at which the trigger criterion was met (seconds since Jan 1, 1970).
4 byte float	The duration of the trigger in seconds.
4 byte float	The LTA (long-term average) value at the trigger-on time.
4 byte float	The LTA (long-term average) value at the trigger-off time.
4 byte float	The peak STA (short-term average) value during this trigger.
4 byte float	The maximum peak-to-peak signal during this trigger in counts.
4 byte float	The time in seconds from trigger-on to the beginning of the maximum peak-to-peak signal.
4 byte float	The half period of the maximum peak-to-peak signal in seconds (time between reversals).
4 byte int	The trigger phase: 0 message generated at trigger-on 1 early-report message generated typically 1 second after trigger-on 2 complete-report message generated at trigger-off

3.3.5.4 Event Packet

Event messages contain information on events detected by the Naqs internal Event Associator subsystem.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	Data type = 6
4 byte int	Data length = 28 + N*12

Content:

4 byte int	The event phase: 0 event on, sent when the event is first detected 1 event on, sent at the end of the user-defined coincidence window (see the Event Associator section in the naqs.ini file). 2 event off, sent when all channels have stopped triggering, or when the event has expired
4 byte int	The type of triggers included in the event. The trigger type is defined by the DetectorType TypeID parameter in the naqs.stn file.
8 byte double	The start time of the event, in seconds since January 1, 1970.
8 byte double	The event duration, in seconds.
4 byte int	N = The number of data channels included in this event.
N*12 bytes	zero-terminated station-channel name for each of the channels included in the event.

Chapter 4 Data Access Protocol

The Nanometrics DataServer provides local and remote access to Nanometrics seismic, serial, and state-of-health data via TCP/IP. This chapter defines version 1.0 of the protocol and data formats required for a client program to request, receive and interpret Nanometrics data. See also the NaqsServer acquisition software manual pages.

4.1 Data types

The following data types are currently supported:

- Time-series data
- State-of-health data
- Transparent serial data
- Triggers
- Events

Data of each type (except events) may be requested by channel, start time, and end time. Event data (which are not channel-specific) may be requested by start and end time. Time-series, state of health, and transparent serial data are sent in the original compressed format received from the data-acquisition instrument. Triggers are sent in a summary form which includes the channel name, trigger time, and duration. Event data include the time and duration. For all data types, data are normally sent in chronological order.

The DataServer also provides data-availability information of two types:

- Channel list: a list of the available channels
- Precis list: a list of available channels, including the start and end time of data available on each channel

Compressed data are tagged with a 4-byte integer key which identifies the channel. The channel list and precis list provide a cross-reference from channel keys to ASCII channel names.

4.2 Subscription protocol

Every client program must implement the communication protocol summarized by the following steps. Italics indicate specific message types. Message formats are given in the next section.

1. Open a socket to the DataServer, using the port number specified in the DataServer configuration (typically 28002).
2. Read the connection time from the socket as a 4-byte integer.
3. Send a *ConnectRequest* (encoding the connection time) to the DataServer.
4. Wait for a *Ready* message from the DataServer.
5. Send a *Request* message of the appropriate type to request data from the DataServer.
6. Receive and process response messages from the DataServer, until receiving a *Ready* message. The *Ready* message indicates the end of data for the last request, and indicates that the DataServer is ready for another request. Each request may elicit 0 or more response messages.
7. Repeat steps 5 and 6 for each data request.
8. (optional) Send a *Terminate* message indicating that you are about to close the connection. Do NOT wait for a *Ready* message.
9. Close the socket.

Note (1) The *ConnectRequest* message is used to provide basic logon security. The first message sent by the client must be a *ConnectRequest* from an authorized user; otherwise, DataServer will close the connection.

(2) The DataServer will process one request at a time, and send a *Ready* message when it is ready for the next request. Sending a request while the server is still processing the previous request will cancel the previous request. The DataServer will send a *Ready* message to indicate end-of-data for the previous request, then start processing the new request.

(3) The DataServer will process any number of requests (one at a time) over a single connection, provided that the connection remains active. A connection will be closed if it becomes inactive (if no request is received for 20 seconds following a *Ready* message).

(4) If no data are available for a certain request, the DataServer will simply return a *Ready* message.

(5) The DataServer will close the connection if it receives an improperly formatted request, or a request of an unknown or unsupported type.

4.3 Client message and request types

As summarized above, all communication under this protocol is initiated by the client; the DataServer simply sends data in response to a client request. Version 1.0 of this protocol supports the eleven request types listed in Table 4-1.

Table 4-1 Client request types for Data Access Protocol v1.0

Request Type	Purpose	Server Response*
ConnectRequest	Initiates the connection	*Ready*
RequestPending	Keeps the connection alive but does not request data	*Ready*
CancelRequest	Cancels the last request (typically a long data request)	*Ready*. The client will receive two *Ready* messages - one for the request being cancelled, and one for the CancelRequest itself.
TerminateMessage	Indicates that the client is about to close the connection	Closes the connection
ChannelListRequest	Requests a list of channels available from the server	*ChannelList* - a list of channel names and associated channel keys
PrecisListRequest	Requests a list of channels and time intervals available from the server	*PrecislList* - a list of channels with start and end time of available data
ChannelInfoRequest	Requests supplementary information for a specified channel	*ChannelHeader* - brief information about the requested channel
DataSizeRequest	Requests information about the volume of data available for a specified channel and time interval (this is useful in order to pre-allocate storage space for the data to be received)	*DataSize* - an estimate of the packet size and number of packets available
DataRequest	Requests data for a specified data channel and time interval. This may be used to request time-series, state of health, or transparent serial data.	N *CompressedData* packets. N may be zero if no data are available for the specified channel and time interval.
TriggerRequest	Requests seismic trigger data for a specified data channel (or set of channels) and time interval	N *NaqsTrigger* packets. N may be zero if no data are available for the specified channels and time interval.
EventRequest	Requests seismic event data for the specified time interval	N *NaqsEvent* packets. N may be zero if no data are available for the specified time interval.

* In all cases except *TerminateMessage*, the DataServer will send a *Ready* message to indicate the end-of-data for a given request.

4.4 Message formats

Each message consists of a 12-byte header and a variable length data content field. The header provides the type and length of the content. The client application should read the header first, then read the content after determining its type and length.

4.4.1 Request messages

Requests are messages sent by the client to DataServer.

4.4.1.1 ConnectRequest

The purpose of the *ConnectRequest* is to initiate the connection and to authenticate the client requesting the connection.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 206
4 byte int	message content length = 24

Content:

12 byte string	User name (maximum 11 characters), zero terminated.
4 byte int	Data Access Protocol version (currently 0)
4 byte int	The time in seconds since Jan. 1, 1970 (UT) that the connection was opened. This should normally be the same value as that sent by the DataServer when the socket is first opened.
4 byte int	32-bit CRC computed for the username, protocol version, connection time, and password. This enables the username and password to be verified without sending the password with the message.

4.4.1.2 RequestPending

A *RequestPending* message is sent to keep the connection active (and open). It has no content.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 110
4 byte int	message content length = 0

4.4.1.3 CancelRequest

A *CancelRequest* message is sent to cancel the previous request. It has no content.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 205
4 byte int	message content length = 0

4.4.1.4 TerminateMessage

A *TerminateMessage* may be sent by either DataServer or the client to indicate that the connection is about to be closed. There are currently 3 message types defined:

- Normal Shutdown: sent by the client to indicate that it is disconnecting
- Error Shutdown: sent by DataServer to indicate that a fatal error has occurred

- Timeout Shutdown: sent by DataServer to indicate that an inactive connection is being closed.

A *TerminateMessage* may also include a brief ASCII string to provide a more detailed explanation of why the connection is being closed.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 200
4 byte int	message content length = 4 + N

Content:

4 byte int	Reason for termination 1 = Normal shutdown 2 = Error shutdown 3 = Timeout shutdown
N byte string	String message (none if N = 0)

4.4.1.5 ChannelListRequest

A *ChannelListRequest* is sent to request the list of channels available from the server. It has no content.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 209
4 byte int	message content length = 0

4.4.1.6 PrecisListRequest

A *PrecisListRequest* is sent to request the list of channels and time intervals available from the server. It provides fields to allow the client to request information for a subset of channels.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 203
4 byte int	message content length = 12

Content:

4 byte int	Instrument ID for which data are requested (or -1 for all instruments).
4 byte int	Data type for which data are requested (1 for time series, 2 for state of health, 6 for transparent serial, or -1 for all types).
4 byte int	Channel for which data are requested (or -1 for all channels).

4.4.1.7 ChannelInfoRequest

A *ChannelInfoRequest* is sent to request supplementary information for a specified channel.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 226
4 byte int	message content length = 8

Content:

4 byte int	Channel key of the channel for which data are requested.
4 byte int	Type defines the type of data being requested (currently ignored).

4.4.1.8 DataSizeRequest

A *DataSizeRequest* is sent to request the packet size and number of packets that would be sent in response to a given *DataRequest*. This message is useful in cases where it is desired to pre-allocate memory or storage space for the requested data.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 229
4 byte int	message content length = 12

Content:

4 byte int	Channel key for which data are requested.
4 byte int	Start time of the interval for which data are requested, in seconds since January 1, 1970 (UT).
4 byte int	End time of the interval for which data are requested, in seconds since January 1, 1970 (UT).

4.4.1.9 DataRequest

A *DataRequest* is sent to request data for a specified channel and time interval. It may be used to request any type of data: time-series, state of health, or transparent serial.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 227
4 byte int	message content length = 12

Content:

4 byte int	Channel key for which data are requested.
4 byte int	Start time of the interval for which data are requested, in seconds since January 1, 1970 (UT).
4 byte int	End time of the interval for which data are requested, in seconds since January 1, 1970 (UT).

4.4.1.10 TriggerRequest

A *TriggerRequest* is sent to request seismic trigger data for a specified channel and time interval.

Header:

4 byte int	Signature = 0x7ABCDE0F

4 byte int	message type = 231
4 byte int	message content length = 12

Content:

4 byte int	Channel key for which data are requested (use key = 0 to request triggers for all channels).
4 byte int	Start time of the interval for which data are requested, in seconds since January 1, 1970 (UT).
4 byte int	End time of the interval for which data are requested, in seconds since January 1, 1970 (UT).

4.4.1.11 EventRequest

An *EventRequest* is sent to request seismic event data for a specified time interval.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 232
4 byte int	message content length = 16

Content:

4 byte int	Start time of the interval for which data are requested, in seconds since January 1, 1970 (UT).
4 byte int	End time of the interval for which data are requested, in seconds since January 1, 1970 (UT).
8 byte float	Minimum event amplitude requested. The amplitude scale is server-dependent and does not necessarily correspond to event magnitude. DataServer 1.00 considers all events to have amplitude 1.0.

4.4.2 Response messages

The following messages are sent to the client by DataServer in response to requests:

4.4.2.1 ReadyMessage

A *ReadyMessage* is sent to the client by DataServer to indicate that it has sent all data available for the previous request, and is ready for the next request. It has no content.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 208
4 byte int	message content length = 0

4.4.2.2 ChannelList

A *ChannelList* message contains a list of the data channels available. Each stream is identified by an unique integer key (which encodes digitizer/channel information) and an 11-character ASCII string which encodes the station and channel name. The channel list contains time-series, serial and state-of-health channels. The data type for a given channel can be determined from the channel key using the following formula: type = ((key >> 8) & 0xff).

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 150
4 byte int	message content length = 4 + N * 16

Content:

4 byte int	number of channels available = N

N 16-byte channel info bundles of the form:

{

4 byte int	Channel key = ((ID << 16) | (type << 8) | channel) where ID is the full instrument serial number type is the data subtype: 1 = time series 2 = state of health 6 = transparent serial channel is the data channel number (0 to 5)
12 bytes	zero-terminated channel name string (e.g. STN01.BHZ).

}

4.4.2.3 PrecisList

A *PrecisList* contains a list of the data channels available, plus the start and end time for the available data for each channel. It is similar to a *ChannelList*, but provides more information. It will include only channels for which data are available on the server.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 253
4 byte int	message content length = 4 + N * 24

Content:

4 byte int	number of channels available = N

N 24-byte channel info bundles of the form:

{

4 byte int	Channel key = ((ID << 16) | (type << 8) | channel) where ID is the full instrument serial number type is the data subtype: 1 = time series 2 = state of health 6 = transparent serial channel is the data channel number (0 to 5)
12 bytes	zero-terminated channel name string (e.g. STN01.BHZ).
4 byte int	Start time of the data available for this channel, in seconds since January1,1970 (UT).
4 byte int	End time of the data available for this channel, in seconds since January1,1970 (UT).

}

4.4.2.4 ChannelHeader

A *ChannelHeader* is sent in response to a *ChannelInfoRequest*. It contains supplementary information for the specified channel.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 256
4 byte int	message content length = 28

Content:

4 byte int	Channel key for this channel.
12 bytes	zero-terminated channel name string (e.g. STN01.BHZ).
12 bytes	zero-terminated network name string

4.4.2.5 DataSize

A *DataSize* message is sent in response to a *DataSizeRequest*. It contains the packet length for data on the requested channel, plus an estimate of the number of packets that would be sent in response to a *DataRequest*.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 257
4 byte int	message content length = 12

Content:

4 byte int	Channel key for which data were requested.
4 byte int	Length in bytes of data packets for this channel.
4 byte int	Estimated (maximum) number of packets available for the requested interval.

4.4.2.6 NaqsEvent

A *NaqsEvent* message contains information about a single seismic event. DataServer may send any number of *NaqsEvent* messages in response to an *EventRequest*.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 260
4 byte int	message content length = 24

Content:

8 byte float	Time of the event in seconds since January 1, 1970 (UT).
8 byte int	Duration of the event in seconds.
8 byte int	Amplitude of the event. The amplitude scale is server- dependent which may or may not correspond to event magnitude. DataServer 1.00 considers all events to have amplitude 1.0.

4.4.2.7 NaqsTrigger

A *NaqsTrigger* message contains information about a trigger on a single channel. Triggers indicate changes in signal energy which may result from a seismic event. DataServer may send any number of *NaqsTrigger* messages in response to a *Trigger-Request*.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	message type = 259
4 byte int	message content length = 32

Content:

4 byte int	Channel key for the channel on which trigger was detected.
12 bytes	Zero-terminated channel name string (e.g. STN01.BHZ).
8 byte float	Time of the trigger in seconds since January 1, 1970 (UT).
8 byte int	Duration of the event in seconds.

4.4.2.8 CompressedData

CompressedData messages contain data in the original compressed format received from the data-acquisition instrument. Details of the internal compressed packet format are given in Chapter 1, “NMXP Data Format”.

Header:

4 byte int	Signature = 0x7ABCDE0F
4 byte int	Data type = 1
4 byte int	Data content length = variable

Content:

4 byte int	Oldest sequence number
N bytes	N byte compressed data packet

Chapter 5 Tagged File Format

The Nanometrics Tagged File Format is used by both Nanometrics Binary Pick Files and Nanometrics Y-files. It is designed to be quick and easy to read and write, especially using the C language. The tagged format allows the format to be extended without breaking backwards or forwards compatibility. This chapter describes tagged file data types, tag types, and tag formats.

5.1 Overview

Tagged files are divided into records. Each record starts with a tag indicating the type of data in the record. Following the tag is the data. Each tag has the offset in bytes to the next tag so that if a program reading the file does not understand the type of data indicated by the tag, it can skip over it to the next tag. This preserves forward compatibility since older programs can read newer versions of the format as they can simply ignore any new records. This also preserves backward compatibility since newer programs can skip out-of-date records and look for the newer, replacement records.

There is no padding or alignment of the records in a tagged file. Each tag or block of data is written immediately following the last byte of the last record.

Each tagged file starts with a tag indicating the type of file. This tag has no data attached to it. It is immediately followed by the first data tag.

5.2 Data types

The descriptions of the file formats use the following data types:

CHAR	signed 8 bit character
UCHAR	unsigned 8 bit character
SHORT	signed 16 bit integer
USHORT	unsigned 16 bit integer
LONG	signed 32 bit integer
ULONG	unsigned 32 bit integer
FLOAT	IEEE 32 bit floating point number
DOUBLE	IEEE 64 bit floating point number
REALTIME	DOUBLE containing the number of seconds since January 1, 1970
BOOL16	a 16 bit boolean value (integer) - either 0 (FALSE) or 1 (TRUE)

PTR	a 32 bit integer -- unused externally and should always be 0
UNIQUEID	a unique 32 bit integer that identifies an instance of a record

If there is an array of a data type this is indicated by square brackets containing the number of elements in the array; for example, CHAR Name[13] indicates that Name is a character array containing 13 elements.

Some of the fields contain a string of characters. A string is defined as an array of characters. There are two types of strings used in the data files: zero terminated and blank padded:

- Zero terminated strings (called ASCIIZ) are compatible with the C definition of a string. That is, an array of characters ending with an ASCII 0 (not the “0” character).
- Blank padded strings (called BLANKPAD) are used when the entire array of characters must be printable. In this case there is no terminating zero. Every character in the array must be a printable so if an array entry is not used by the text it must be set to the space character.

The PTR field is only used internally by programs and never holds valid data. These fields should be set to zero when writing a tagged file and should be ignored when reading.

The UNIQUEID field is used to uniquely identify instances of records. In some cases a record needs to be associated with another record. This is done by assigning a unique number to the Self field of the other record and then using this number in the first record. For example, an event record has to indicate which of its many solution records is the preferred one. It does this by giving the unique ID of the preferred solution PreferredSolution field for the event. Each solution has a Self field with a unique number -- the solution whose Self field matches the PreferredSolution field is the preferred solution.

5.3 Tag format

UCHAR	Format
UCHAR	Magic
USHORT	Type
LONG	NextTag
LONG	NextSame
LONG	Spare

Format	This is the byte order format for this data. Use the letter “I” for Intel format data (little endian) or the letter “M” for Motorola (big endian) format
Magic	This is a unique number that allows programs to check that this a valid tag. This number must be 31.
Type	This is the type of data attached to this tag. It must be one of the predefined tag types listed below.
NextTag	NextTag is the offset in bytes from the end of this tag to the start of the next tag. That means, the offset is the size of the data attached to this tag.
NextSame	NextSame is the offset in bytes from the end of this tag to the start of the next tag with the same type. If zero, there is no next tag with the same type.

Spare	Spare is added to pad the size of the tag to an even sixteen bytes. Also available for future use. Should always be zero.

5.4 Tag types

The list below gives the tag types which have been defined so far. See Chapter 6, “Y-File Format”, for examples of usage.

0	TAG_Y_FILE
1	TAG_STATION_INFO
2	TAG_STATION_LOCATION
3	TAG_STATION_PARAMETERS
4	TAG_STATION_DATABASE
5	TAG_SERIES_INFO
6	TAG_SERIES_DATABASE
7	TAG_DATA_INT32
8	TAG_PICK_FILE
9	TAG_UNASSOCIATED_PICKA
10	TAG_CRUSTAL_MODEL
11	TAG_CRUSTAL_LAYER
12	TAG_EVENTA
13	TAG_MAGNITUDE
14	TAG_PICKA
15	TAG_SOLUTION
16	TAG_HYPO_PARAMETERS
17	TAG_ASSOCIATION
18	TAG_STN_LOC_PARAMETERS
19	TAG_HYPO_STN_PARAMETERS
20	TAG_LOC_STN_PARAMETERS
21	TAG_LOC_PARAMETERS
22	TAG_X_FILE
23	TAG_DATA_STEIM
24	TAG_EVENT_COMMENTS
25	TAG_SOLUTION_COMMENTS
26	TAG_STATION_RESPONSE
27	TAG_PICKB
28	TAG_EVENTB
29	TAG_UNASSOCIATED_PICKB
30	TAG_RINGBUFFER_FILE
31	TAG_RINGBUFFER_INFO
32	TAG_RINGBUFFER_INDEX
33	TAG_RINGBUFFER_DATA
34	TAG_LOGBUFFER_FILE
35	TAG_LOGBUFFER_INFO
36	TAG_LOGBUFFER_DATA

37	TAG_SOHBUFFER_FILE
38	TAG_SOHBUFFER_INFO
39	TAG_SOHBUFFER_LABEL
40	TAG_SOHBUFFER_CALIB
41	TAG_SOHBUFFER_DATA
42	TAG_SKIP_DATA
43	TAG_END_MARKER

Chapter 6 Y-File Format

This chapter defines the format used in the Nanometrics Y-file format version 5. It includes a description of the physical format of the file and a description of the meaning of each field in the file. A Y-file is an instance of a tagged file; see also Chapter 5, “Tagged File Format”.

A Y-file always contains only one series of continuous data. If there a is break in the data, then you will need more than one Y-file to hold the data.

6.1 File format

The first tag in a Y-file must be the TAG_Y_FILE tag. This must be followed by the following tags, in any order:

TAG_STATION_INFO
TAG_STATION_LOCATION
TAG_STATION_PARAMETERS
TAG_STATION_DATABASE
TAG_SERIES_INFO
TAG_SERIES_DATABASE

The following tag is optional:

TAG_STATION_RESPONSE

Each tag must be followed by the data associated with the tag. See below for a description of the data for each tag.

The last tag in the file must be a TAG_DATA_INT32 tag. This tag must be followed by an array of LONG's. The number of entries in the array must agree with what was described in the TAG_SERIES_INFO data.

6.2 Field descriptions

6.2.1 TAG_STATION_INFO

UCHAR	Update[8]
STNID	StationID
UCHAR	NetworkID[51] (ASCIIZ)

UCHAR	SiteName[61] (ASCIIZ)
UCHAR	Comment[31] (ASCIIZ)
UCHAR	SensorType[51] (ASCIIZ)
UCHAR	DataFormat[7] (ASCIIZ)
Update	This field is only used internally for administrative purposes. It should always be set to zeroes.
StationID	StationID is the identification name of the station in SEED format. This uses a sub-record called STNID which is described above.
NetworkID	This is some descriptive text identifying the network.
SiteName	SiteName is some text identifying the site.
Comment	Comment is any comment for this station.
SensorType	SensorType is some text describing the type of sensor used at the station.
DataFormat	DataFormat is some text describing the data format recorded at the station.

6.2.1.1 Station ID

UCHAR Station[5]	(BLANKPAD)
UCHAR Location[2]	(BLANKPAD)
UCHAR Channel[3]	(BLANKPAD)
Station	Station is the five letter SEED format station identification.
Location	Location is the two letter SEED format location identification.
Channel	Channel is the three letter SEED format channel identification.

6.2.2 TAG_STATION_LOCATION

UCHAR	Update[8]
FLOAT	Latitude
FLOAT	Longitude
FLOAT	Elevation
FLOAT	Depth
FLOAT	Azimuth
FLOAT	Dip
Update	This field is only used internally for administrative purposes. It should always be set to zeroes.
Latitude	Latitude is the latitude in degrees of the location of the station. The latitude should be between -90 (South) and +90 (North).
Longitude	Longitude is the longitude in degrees of the location of the station. The longitude should be between -180 (West) and +180 (East).
Elevation	Elevation is the elevation in meters above sea level of the station.
Depth	Depth is the depth in meters of the sensor.
Azimuth	Azimuth is the azimuth of the sensor in degrees clockwise.
Dip	Dip is the dip of the sensor. 90 degrees is defined as vertical right way up.

6.2.3 TAG_STATION_PARAMETERS

UCHAR	Update[16]
REALTIME	StartValidTime
REALTIME	EndValidTime
FLOAT	Sensitivity
FLOAT	SensFreq
FLOAT	SampleRate
FLOAT	MaxClkDrift
UCHAR	SensUnits[24] (ASCIIZ)
UCHAR	CalibUnits[24] (ASCIIZ)
UCHAR	ChanFlags[27] (BLANKPAD)
UCHAR	UpdateFlag
UCHAR	Filler[4]
Update	This field is only used internally for administrative purposes. It should always be set to zeroes.
StartValidTime	This is the time that the information in these records became valid.
EndValidTime	This is the time that the information in these records became invalid.
Sensitivity	Sensitivity is the sensitivity of the sensor in nanometers per bit.
SensFreq	This is the frequency at which the sensitivity was measured.
SampleRate	This is the number of samples per second. This value can be less than 1.0. (i.e. 0.1)
MaxClkDrift	This is the maximum drift rate of the clock in seconds per sample.
SensUnits	This is some text indicating the units in which the sensitivity was measured.
CalibUnits	This is some text indicating the units in which calibration input was measured.
ChanFlags	Text indicating the channel flags according to the SEED definition.
UpdateFlag	This flag must be “N” or “U” according to the SEED definition.
Filler	Pads out the record to satisfy the alignment restrictions for reading data on a SPARC processor.

6.2.4 TAG_SERIES_DATABASE, TAG_STATION_DATABASE

UCHAR	Update[8]
REALTIME	LoadDate
UCHAR	Key[16]
Update	This field is only used internally for administrative purposes. It should always be set to zeroes.
LoadDate	LoadDate is the date the information was loaded into the database.
Key	Key is a unique key that identifies this record in the database.

6.2.5 TAG_SERIES_INFO

UCHAR	Update[16]
REALTIME	StartTime
REALTIME	EndTime

ULONG	NumSamples
LONG	DCOffset
LONG	MaxAmplitude
LONG	MinAmplitude
UCHAR	Format[8] (ASCIIZ)
UCHAR	FormatVersion[8] (ASCIIZ)
Update	This field is only used internally for administrative purposes. It should always be set to zeroes.
StartTime	This is start time of the data in this series.
EndTime	This is end time of the data in this series.
NumSamples	This is the number of samples of data in this series.
DCOffset	DCOffset is the DC offset of the data.
MaxAmplitude	MaxAmplitude is the maximum amplitude of the data.
MinAmplitude	MinAmplitude is the minimum amplitude of the data.
Format	This is the format of the data. This should always be “YFILE”.
FormatVersion	FormatVersion is the version of the format of the data. This should always be “5.0”

6.2.6 TAG_STATION_RESPONSE

UCHAR	Update[8]
UCHAR	PathName[260]
Update	This field is only used internally for administrative purposes. It should always be set to zeroes.
PathName	PathName is the full name of the file which contains the response information for this station.

Appendix A Data Stream Client

The file dsClient.c is an example DataStream client program. The purpose of this code is to demonstrate how to communicate with the NaqsServer datastream service. It is written for Windows 95 or NT, but may easily be modified to run on other platforms.

dsClient connects to the datastream service, requests data for a single channel, and prints out some information about each data packet received. It can request and receive time-series, state-of-health, or serial data.

The requested channel name, and the host name and port name for the datastream service, are input as command-line parameters. By default, the program connects to port 28000 on the local machine. Note that all data received from the datastream server are in network byte order (most-significant byte first), except for compressed data packets. Compressed data packets are forwarded without modification from the originating instrument; these packets are ordered least significant byte (LSB)-first.


```
/* Includes -------------------------------------------------------------*/

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>

#include <winsock.h>
#include <winbase.h>
#include <ctype.h>

/* Definitions ----------------------------------------------------------*/

// default host and port for datastream server
#define DEFAULT_HOST          "localhost"
#define DEFAULT_PORT          28000

#define INVALID_INET_ADDRESS  INADDR_NONE

// first 4 bytes of all messages
#define NMX_SIGNATURE 0x7abcde0f
```

```
// defines the message types
#define CONNECT_MSG        100
#define CHANNEL_LIST       150
#define ERROR_MSG          190
#define TERMINATE_MSG      200
#define COMPRESSED_DATA      1

// time series
#define TIMSER_TYPE          1
#define TIMSER_ADD_REQ     120

// state of health
#define SOH_TYPE             2
#define SOH_ADD_REQ        121

// transparent serial
#define SERIAL_TYPE          6
#define SERIAL_ADD_REQ     124

// macro to determine data type from key
#define dataType(key) ((key >> 8) & 0xFF);

// used to indicate a valid return
#define SOCKET_OK          0

#define KEY_NOT_FOUND     -1

// maximum time between connection attempts (seconds)
#define SLEEPMAX 10

/* Structures ------------------------------------------------------------*/

// for documentation on message structures see the NaqsServer manual

// Header for all messages
struct MessageHeader
{
  unsigned long signature;
  unsigned long type;
  unsigned long length;
};

// Request for time series data (single channel)
struct DataAddRequest
{
  long numChannels;
  long channel;
  long stcDelay;
  long format;
  long sendBuffers;
};

// Request for soh or serial data (single channel)
struct AddRequest
{
  long numChannels;
  long channel;
  long stcDelay;
  long sendBuffers;
};

// The key/name info for one channel
struct ChannelKey
{
  long key;
```

```
  char name[12];
};

// A channel list structure
struct ChannelList
{
  unsigned long length;
  ChannelKey    channel[200];
};

/* Variables ------------------------------------------------------------*/

// use a static ChannelList to keep it simple
static ChannelList channelList;

/***************************************************************************

  Function:  initSockets

  Purpose:   initializes sockets for Windows

 -------------------------------------------------------------------------*/
static int initSockets()
{
  WORD wVersionRequested = MAKEWORD(1, 1);
  WSADATA wsaData;

  int err = WSAStartup(wVersionRequested, &wsaData);

  if (err != 0)
    // Tell the user that we couldn't find a useable
    // winsock.dll.
    return 1;

  // Confirm that the Windows Sockets DLL supports 1.1.
  // Note that if the DLL supports versions greater than 1.1
  // in addition to 1.1, it will still return 1.1 in wVersion
  // since that is the version we requested.

  if ( LOBYTE( wsaData.wVersion ) != 1 ||
       HIBYTE( wsaData.wVersion ) != 1 )
  {
    // Tell the user that we couldn't find a useable winsock.dll.
    WSACleanup();
    return 1;
  }

  // The Windows Sockets DLL is acceptable.
  return 0;
}

/***************************************************************************

  Function:  addressString

  Purpose:   makes a dotted string for an IP address

 -------------------------------------------------------------------------*/
static char* dottedString(unsigned long addr)
{
  static char buffer[32];
  unsigned char* paddress = (unsigned char*) &addr;
  sprintf(buffer, "%u.%u.%u.%u",
                (unsigned int) paddress[0],
                (unsigned int) paddress[1],
```

```
                  (unsigned int) paddress[2],
                  (unsigned int) paddress[3]);
  return buffer;
}


/***************************************************************************
  Function:  getAddressOfHost

  Summary:   Gets the internet address for the given host.
             First tries to interpret hostname as a dotted string.
             If that fails, it tries to interpret it as a host name.

  Parameters:
    hostname  - the name of the host
    paddress  - the internet address (returned)

  Return value:
    TRUE on success
    FALSE for unknown host

-------------------------------------------------------------------------*/
static int getAddressOfHost (char* hostname, unsigned long* paddress)
{
  struct hostent *pHost = NULL;

  if (!hostname)
  {
    *paddress = INVALID_INET_ADDRESS;
    return FALSE;
  }

  // First try to interpret name as dotted decimal, then as a host name.

  *paddress = inet_addr (hostname);

  if (*paddress == INVALID_INET_ADDRESS)
    pHost = gethostbyname(hostname);

  else
    pHost = gethostbyaddr((char*) paddress, sizeof(unsigned long), AF_INET);

  if (pHost != NULL)
  {
    memcpy(paddress, pHost->h_addr, (size_t) pHost->h_length);
    return TRUE;
  }

  else
  {
    *paddress = INVALID_INET_ADDRESS;
    return FALSE;
  }
}

/***************************************************************************

  Function:  openSocket

  Purpose:   opens a socket and connects

 ------------------------------------------------------------------------*/
static int openSocket(unsigned long hostAddress, int portNum)
{
  static int sleepTime = 1;
```

```
  int isock = 0;
  struct sockaddr_in psServAddr;

  while(1)
  {
    isock = socket (AF_INET, SOCK_STREAM, 0);
    if (isock < 0)
    {
      printf ("dsClient: Can't open stream socket\n");
      exit(12);
    }

    /* Fill in the structure "psServAddr" with the address of server
       that we want to connect with */
    memset (&psServAddr, sizeof(psServAddr), 0);
    psServAddr.sin_family = AF_INET;
    psServAddr.sin_addr.s_addr = hostAddress;
    psServAddr.sin_port = htons((unsigned short) portNum);
    printf("attempting to connect to %s port %d\n",
             dottedString(hostAddress), portNum);

    if (connect(isock, (struct sockaddr *)&psServAddr, sizeof(psServAddr)) >= 0)
    {
      sleepTime = 1;
      printf ("Connection established, path number=%i\n", isock);
      return isock;
    }
    else
    {
      printf("Trying again later...Sleeping\n");
      closesocket (isock);
      Sleep (1000 * sleepTime);
      sleepTime *= 2;
      if (sleepTime > SLEEPMAX)
        sleepTime = SLEEPMAX;
    }
  }
}

/***************************************************************************

  Function:  s_send

  Purpose:  sends a message and computes rc

  Return:   rc = SOCKET_OK on success
            rc = SOCKET_ERROR on error

 -------------------------------------------------------------------------*/
static int s_send(int isock, void* data, int length)
{
  int sendCount = send(isock, (char*) data, length, 0);

  if (sendCount != length)
    return SOCKET_ERROR;

  return SOCKET_OK;
}

/***************************************************************************

  Function:  s_recv

  Purpose:  receives a message and computes rc
```

```
  Return:   rc = SOCKET_OK on success
            rc = SOCKET_ERROR on error

 ------------------------------------------------------------------------*/
static int s_recv(int isock, void* data, int length)
{
  int recvCount = recv(isock, (char*) data, length, 0);

  if (recvCount != length)
    return SOCKET_ERROR;

  return SOCKET_OK;
}

/**************************************************************************

  Function:  sendHeader

  Purpose:  sends a MessageHeader to the server

 ------------------------------------------------------------------------*/
static int sendHeader(int isock, int type, int length)
{
  int sendCount = 0;

  MessageHeader msg;
  msg.signature = htonl(NMX_SIGNATURE);
  msg.type      = htonl(type);
  msg.length    = htonl(length);

  return s_send(isock, &msg, sizeof(msg));
}

/**************************************************************************

  Function:  receiveHeader

  Purpose:  receives a MessageHeader from the server

 ------------------------------------------------------------------------*/
static int receiveHeader(int isock, MessageHeader* pmsg)
{
  int rc = s_recv(isock, pmsg, sizeof(MessageHeader));

  if (rc == SOCKET_OK)
  {
    pmsg->signature = ntohl(pmsg->signature);
    pmsg->type      = ntohl(pmsg->type);
    pmsg->length    = ntohl(pmsg->length);

    if (pmsg->signature != NMX_SIGNATURE)
      rc = SOCKET_ERROR;
  }

  return rc;
}

/**************************************************************************

  Function:  sendConnectMessage

  Purpose:  sends a Connect message to server

 ------------------------------------------------------------------------*/
static int sendConnectMessage(int isock)
```

```
{
  return sendHeader(isock, CONNECT_MSG, 0);
}

/***************************************************************************

  Function:  requestTypeChannel

  Purpose:  requests one channel of serial or soh data

 -------------------------------------------------------------------------*/
static int requestTypeChannel(int isock, int channel, int type)
{
  AddRequest request;
  int sendCount = 0;

  int rc = sendHeader(isock, type, sizeof(request));

  if (rc == SOCKET_OK)
  {
    request.numChannels = htonl(1);
    request.channel     = htonl(channel);
    request.stcDelay    = htonl(0);
    request.sendBuffers = htonl(0);
    rc = s_send(isock, &request, sizeof(request));
  }

  return rc;
}

/***************************************************************************

  Function:  requestSerialChannel

  Purpose:  requests one channel of serial data

 -------------------------------------------------------------------------*/
static int requestSerialChannel(int isock, int channel)
{
  printf("Requesting serial channel 0x%8.8x\n", channel);
  return requestTypeChannel(isock, channel, SERIAL_ADD_REQ);
}

/***************************************************************************

  Function:  requestSohChannel

  Purpose:  requests one channel of SOH data

 -------------------------------------------------------------------------*/
static int requestSohChannel(int isock, int channel)
{
  printf("Requesting soh channel 0x%8.8x\n", channel);
  return requestTypeChannel(isock, channel, SOH_ADD_REQ);
}

/***************************************************************************

  Function:  requestDataChannel

  Purpose:  requests one channel of time series data

 -------------------------------------------------------------------------*/
static int requestDataChannel(int isock, int channel)
{
```

```
  DataAddRequest request;
  int rc = 0;

  printf("Requesting time series channel 0x%8.8x\n", channel);

  rc = sendHeader(isock, TIMSER_ADD_REQ, sizeof(request));

  if (rc == SOCKET_OK)
  {
    request.numChannels = htonl(1);
    request.channel     = htonl(channel);
    request.stcDelay    = htonl(0);
    request.format      = htonl(-1);
    request.sendBuffers = htonl(0);
    rc = s_send(isock, &request, sizeof(request));
  }

  return rc;
}

/***************************************************************************

  Function:  requestChannel

  Purpose:  requests one channel of any type

 -------------------------------------------------------------------------*/
static int requestChannel(int isock, int channel)
{
  int type = dataType(channel);
  if (type == TIMSER_TYPE)
    return requestDataChannel(isock, channel);
  else if (type == SOH_TYPE)
    return requestSohChannel(isock, channel);
  else
    return requestSerialChannel(isock, channel);
}

/***************************************************************************

  Function:  receiveChannelList

  Purpose:  receives a ChannelList from the server

 -------------------------------------------------------------------------*/
static int receiveChannelList(int isock, ChannelList* plist, int length)
{
  int ich = 0;
  int recvCount = recv(isock, (char*) plist, length, 0);

  if (recvCount != length)
    return SOCKET_ERROR;

  plist->length = ntohl(plist->length);
  if ((unsigned) length != 4 + plist->length * sizeof(ChannelKey))
  {
    printf("wrong number of channels in Channel List\n");
    return SOCKET_ERROR;
  }

  for (ich = 0; ich < (int) plist->length; ich++)
  {
    plist->channel[ich].key = ntohl(plist->channel[ich].key);
    printf("channel %s has key 0x%8.8x\n", plist->channel[ich].name,
                                           plist->channel[ich].key);
```

```
  }

  return SOCKET_OK;
}

/***************************************************************************

  Function:  lookupChannel

  Purpose:  looks up a channel in the ChannelList

 -----------------------------------------------------------------------*/
static int lookupChannel(char* name, ChannelList* plist)
{
  int length = plist->length;
  int ich = 0;

  for (ich = 0; ich < length; ich++)
  {
    if (stricmp(name, plist->channel[ich].name) == 0)
      return plist->channel[ich].key;
  }

  return KEY_NOT_FOUND;
}

/***************************************************************************

  Function:  receiveError

  Purpose:  receives an Error message from the server

 -----------------------------------------------------------------------*/
static int receiveError(int isock, int length)
{
  int rc = 0;

  if (length > 0)
  {
    char* buffer = (char*) malloc(length);
    rc = s_recv(isock, buffer, length);
    if (rc == SOCKET_OK)
      printf("%s\n", buffer);
    free(buffer);
  }
  return rc;
}

/***************************************************************************

  Function:  receiveTermination

  Purpose:  receives a Terminate message from the server

 -----------------------------------------------------------------------*/
static int receiveTermination(int isock, int length)
{
  int reason = 0;
  int rc = s_recv(isock, &reason, 4);

  if (rc == SOCKET_OK)
  {
    printf("Connection closed by server, reason = %d\n", ntohl(reason));

    if (length > 4)
```

```
      rc = receiveError(isock, length - 4);
  }
  return rc;
}

/***************************************************************************

  Function:  flushBytes

  Purpose:  receives and discards some bytes from the server

 -------------------------------------------------------------------------*/
static int flushBytes(int isock, int length)
{
  int rc = 0;

  if (length > 0)
  {
    char* buffer = (char*) malloc(length);
    rc = s_recv(isock, buffer, length);
    free(buffer);
  }
  return rc;
}

/***************************************************************************

  Function:  processData

  Purpose:  processes compressed data from the server

 -------------------------------------------------------------------------*/
static void processData(char* buffer, int length)
{
  int   bundles    = (length - 21) / 17;
  int   type       = buffer[4];
  int   oldestSeq  = 0;
  int   sequence   = 0;
  int   timeSecs   = 0;
  short timeFrac   = 0;
  short instrument = 0;
  short channel    = 0;
  short byteCount  = 0;

  double pktTime = 0;


  // copy the header contents into local fields
  // note these are little endian (LSB first)
  memcpy(&oldestSeq,  &buffer[0], 4);
  memcpy(&timeSecs,   &buffer[5], 4);
  memcpy(&timeFrac,   &buffer[9], 2);
  memcpy(&instrument, &buffer[11], 2);
  memcpy(&sequence,   &buffer[13], 4);
  memcpy(&byteCount,  &buffer[17], 2);
  memcpy(&channel,    &buffer[19], 2);


  pktTime = timeSecs + 0.0001 * timeFrac;

  // print out header and/or data for different packet types

  if (type == TIMSER_TYPE)
  {
    channel = buffer[17] & 0x07;
```

```
    printf("Rx time series inst %u:%d seq %6u time %.4f bundles %d\n",
            instrument, channel, sequence, pktTime, bundles);
  }

  else if (type == SOH_TYPE)
  {
    printf("Rx SOH inst %u:soh seq %6u time %.4f bundles %d\n",
            instrument, sequence, pktTime, bundles);
  }

  else if (type == SERIAL_TYPE)
  {
    char* data = &buffer[21];
    int ix;

    printf("Rx serial data inst %u:%d seq %6u time %.4f bytes %d\n",
            instrument, channel, sequence, pktTime, byteCount);

    // make a printable version of the data
    for (ix = 0; ix < byteCount; ix++)
    {
      if (!isprint(data[ix]))
        data[ix] = '.';
    }

    // and print it out
    fwrite(data, 1, byteCount, stdout);
    printf("\n");
  }

  else
  {
    printf("unrecognized data type: %d\n", type);
  }
}

/***************************************************************************

  Function:  receiveData

  Purpose:  receives compressed data from the server

 -------------------------------------------------------------------------*/
static int receiveData(int isock, int length)
{
  int rc = 0;

  if (length > 0)
  {
    char* buffer = (char*) malloc(length);
    rc = s_recv(isock, buffer, length);
    if (rc == SOCKET_OK)
      processData(buffer, length);
    free(buffer);
  }
  return rc;
}

/***************************************************************************

  Function:  receiveMessage

  Purpose:  receives message from the server,
            requests channelName if it gets a ChannelList
```

```
 ----------------------------------------------------------------------*/
static int receiveMessage(int isock, char* channelName)
{
  /* receive a message header */
  MessageHeader header;
  int rc = receiveHeader(isock, &header);

  if (rc != SOCKET_OK)
    return rc;

  /* receive whatever message is incoming */
  if (header.type == CHANNEL_LIST)
  {
    rc = receiveChannelList(isock, &channelList, header.length);

    if (rc == SOCKET_OK)
    {
      int key = lookupChannel(channelName, &channelList);
      if (key == KEY_NOT_FOUND)
      {
        printf("Channel %s not found in channel list\n", channelName);
        exit(1);
      }
      rc = requestChannel(isock, key);
    }

    return rc;
  }

  // if it is an Error message, receive it and print it out
  else if (header.type == ERROR_MSG)
  {
    return receiveError(isock, header.length);
  }

  // if it is a Terminate message, receive it, print it,
  // and return SOCKET_ERROR to exit loop
  else if (header.type == TERMINATE_MSG)
  {
    receiveTermination(isock, header.length);
    return SOCKET_ERROR;
  }

  // if it is data, receive it
  else if (header.type == COMPRESSED_DATA)
  {
    return receiveData(isock, header.length);
  }

  // if it is anything else, just read it to keep in sync
  else
  {
    printf("Unrecognized message, type = %d, length = %d\n",
            header.type, header.length);
    return flushBytes(isock, header.length);
  }
}

/***********************************************************************

  Function:  main

  Purpose:  does everything

 ----------------------------------------------------------------------*/
```

```
void main (int argc, char* argv[])
{
  int   portNum     = DEFAULT_PORT;
  char* serverName  = DEFAULT_HOST;
  char* channelName = NULL;
  unsigned long hostAddress;

  int isock;
  int rc = 0;
  channelList.length = 0;

  printf("dsClient v1.0 - Sample datastream client program\n");
  printf("Copyright (C) 1999 Nanometrics, Inc.\n\n");

  /* get server address & port from command line. */
  if (argc < 2)
  {
    printf("Usage:  dsclient channel [host [port]]\n");
    exit(1);
  }

  channelName = argv[1];

  if (argc >= 3)
    serverName = argv[2];

  if (argc >= 4)
    portNum = atoi(argv[3]);

  printf("Starting dsclient with the following options:\n");
  printf("  datastream host:  %s\n", serverName);
  printf("  datastream port:  %d\n", portNum);
  printf("  data channel:     %s\n", channelName);

  // initialize sockets
  if (initSockets() != 0)
  {
    printf("Cannot initialize Winsock DLL\n");
    exit(1);
  }

  // get the IP address for the host string
  if (!getAddressOfHost (serverName, &hostAddress))
  {
    printf("Cannot resolve host name:  %s", serverName);
    exit(1);
  }
  else
  {
    printf("Datastream server host: %s (%s)\n",
            serverName, dottedString(hostAddress));
  }

  /* Main loop */
  for (;;)
  {
    /* open a TCP socket */
    isock = openSocket(hostAddress, portNum);

    /* send the Connect message */
    rc = sendConnectMessage(isock);

    while (rc == SOCKET_OK)
      rc = receiveMessage(isock, channelName);
```

```
          printf("lost connection!\n");
          closesocket (isock);
          Sleep (3000);
        }
      }
```

Appendix B Serial Packet CRC

To detect serial transmission errors, each packet sent by serial port is preceded by a 2-byte synchronization word (0xAABB) and followed by a 2-byte CRC.

Nanometrics instruments use the 16-bit CRC-CCITT as the CRC polynomial. The CRC is computed using a reflected CRC algorithm, using 0 as the inital value of the CRC. The CRC is computed over the entire message (including the synchronization word), then appended to the message without modification. On receive, the CRC computed over the entire message (including the synchronization word and the CRC bytes) should be zero. Packets for which the receive CRC is not zero are discarded.

The serial transmission algorithm (including computation of CRC) is as follows:

```
unsigned short ausCrcTable[256] =
{
  0x0000, 0x1189, 0x2312, 0x329B, 0x4624, 0x57AD, 0x6536, 0x74BF, 0x8C48, 0x9DC1,
  0xAF5A, 0xBED3, 0xCA6C, 0xDBE5, 0xE97E, 0xF8F7, 0x1081, 0x0108, 0x3393, 0x221A,
  0x56A5, 0x472C, 0x75B7, 0x643E, 0x9CC9, 0x8D40, 0xBFDB, 0xAE52, 0xDAED, 0xCB64,
  0xF9FF, 0xE876, 0x2102, 0x308B, 0x0210, 0x1399, 0x6726, 0x76AF, 0x4434, 0x55BD,
  0xAD4A, 0xBCC3, 0x8E58, 0x9FD1, 0xEB6E, 0xFAE7, 0xC87C, 0xD9F5, 0x3183, 0x200A,
  0x1291, 0x0318, 0x77A7, 0x662E, 0x54B5, 0x453C, 0xBDCB, 0xAC42, 0x9ED9, 0x8F50,
  0xFBEF, 0xEA66, 0xD8FD, 0xC974, 0x4204, 0x538D, 0x6116, 0x709F, 0x0420, 0x15A9,
  0x2732, 0x36BB, 0xCE4C, 0xDFC5, 0xED5E, 0xFCD7, 0x8868, 0x99E1, 0xAB7A, 0xBAF3,
  0x5285, 0x430C, 0x7197, 0x601E, 0x14A1, 0x0528, 0x37B3, 0x263A, 0xDECD, 0xCF44,
  0xFDDF, 0xEC56, 0x98E9, 0x8960, 0xBBFB, 0xAA72, 0x6306, 0x728F, 0x4014, 0x519D,
  0x2522, 0x34AB, 0x0630, 0x17B9, 0xEF4E, 0xFEC7, 0xCC5C, 0xDDD5, 0xA96A, 0xB8E3,
  0x8A78, 0x9BF1, 0x7387, 0x620E, 0x5095, 0x411C, 0x35A3, 0x242A, 0x16B1, 0x0738,
  0xFFCF, 0xEE46, 0xDCDD, 0xCD54, 0xB9EB, 0xA862, 0x9AF9, 0x8B70, 0x8408, 0x9581,
  0xA71A, 0xB693, 0xC22C, 0xD3A5, 0xE13E, 0xF0B7, 0x0840, 0x19C9, 0x2B52, 0x3ADB,
  0x4E64, 0x5FED, 0x6D76, 0x7CFF, 0x9489, 0x8500, 0xB79B, 0xA612, 0xD2AD, 0xC324,
  0xF1BF, 0xE036, 0x18C1, 0x0948, 0x3BD3, 0x2A5A, 0x5EE5, 0x4F6C, 0x7DF7, 0x6C7E,
  0xA50A, 0xB483, 0x8618, 0x9791, 0xE32E, 0xF2A7, 0xC03C, 0xD1B5, 0x2942, 0x38CB,
  0x0A50, 0x1BD9, 0x6F66, 0x7EEF, 0x4C74, 0x5DFD, 0xB58B, 0xA402, 0x9699, 0x8710,
  0xF3AF, 0xE226, 0xD0BD, 0xC134, 0x39C3, 0x284A, 0x1AD1, 0x0B58, 0x7FE7, 0x6E6E,
  0x5CF5, 0x4D7C, 0xC60C, 0xD785, 0xE51E, 0xF497, 0x8028, 0x91A1, 0xA33A, 0xB2B3,
  0x4A44, 0x5BCD, 0x6956, 0x78DF, 0x0C60, 0x1DE9, 0x2F72, 0x3EFB, 0xD68D, 0xC704,
  0xF59F, 0xE416, 0x90A9, 0x8120, 0xB3BB, 0xA232, 0x5AC5, 0x4B4C, 0x79D7, 0x685E,
  0x1CE1, 0x0D68, 0x3FF3, 0x2E7A, 0xE70E, 0xF687, 0xC41C, 0xD595, 0xA12A, 0xB0A3,
  0x8238, 0x93B1, 0x6B46, 0x7ACF, 0x4854, 0x59DD, 0x2D62, 0x3CEB, 0x0E70, 0x1FF9,
  0xF78F, 0xE606, 0xD49D, 0xC514, 0xB1AB, 0xA022, 0x92B9, 0x8330, 0x7BC7, 0x6A4E,
  0x58D5, 0x495C, 0x3DE3, 0x2C6A, 0x1EF1, 0x0F78
};
#define CrcUpdate(usCrc,ubByte) \
  ((usCrc) >> 8) ^ ausCrcTable [((usCrc) & 0xff) ^ (ubByte)]
```

```
SendByte (ubByte)
{
  usCrc = CrcUpdate(usCrc,ubByte);
  UscTx = ubByte ^ ubScramble;
}

SendWord (usWord)
{
  SendByte (usWord >> 0);
  SendByte (usWord >> 8);
}

SendLong (ulLong)
{
  SendByte (ulLong >> 0);
  SendByte (ulLong >> 8);
  SendByte (ulLong >> 16);
  SendByte (ulLong >> 24);
}

SendMsg (pubData)
{
  usCrc = 0;
  SendByte (ubSync1);
  SendByte (ubSync2);
  SendLong (ulOldestSequenceNumber);
  for (us = 0; us < usNumberMsgByte, us ++)
    SendByte (pubData [us]);
  usCrc2 = usCrc;
  SendWord (usCrc2);
}
```

```
RecvByte ()
{
   ubByte = UscRx ^ ubScramble;
   usCrc = CrcUpdate (usCrc, ubByte);
   return ubByte;
}
RecvWord ()
{
   usWord = RecvByte ();
   usWord |= RecvByte () << 8;
   return usWord;
}
RecvLong ()
{
ulLong = RecvByte ();
ulLong |= RecvByte () << 8;
ulLong |= RecvByte () << 16;
ulLong |= RecvByte () << 24;
return ulLong;
}
```