NRC / CAR
Jari Hyyryläinen, Iiro Jantunen 11/4 2005

SSI protocol specification
Version 1.0

Owner: Jari Hyyryläinen
Scope: Communication protocol for sensor monitoring
Status: Stable

Document history:

VERSION	DATE	Editor	Change history
0.1	14/3 2003	Jari Hyyryläinen	First draft
0.2	29/4 2003	“	Wirsu partner contributions added
0.3	20/5 2003	”	As agreed in meeting 14/5 at NRC
0.4	2/10 2003	“	Delay added in command ‘A’
0.5	5/12 2003	“	As agreed in meeting 5/12 ar NRC
VERSION	**DATE**	**Editor / Contributor(s)**	
0.6	3/11 2004	Jari Hyyryläinen , Samuli Silanto, Antti Virolainen	Commands for data streaming added, error code added, ‘length’ two bytes in UART protocol,
0.7	22/12 2004	Jari Hyyryläinen, Martti Huttunen	Number of counts added into command ’O’. Command ‘U’ -observer finished- added.
0.8	14/1 2005	Jari Hyyryläinen, Dirk Trossen	’Min’ and ’max’ values added to ’N’. Threshold field added to ‘O’.
1.0	11/4 2005	Iiro Jantunen	RFID tag memory layout for SSI

Acknowledgements

The following persons have contributed heavily this document.

Name	Company
Santtu Naukkarinen	NRC
Samuli Silanto	NRC
Jussi Kaasinen	NRC
Antti Virolainen	NRC
Dirk Trossen	NRC
Iiro Jantunen	NRC
Panu Kopsala	Vaisala
Veikko Koivumaa	Suunto
Mikko Martikka	Suunto
Otto Chrons	Ionific
Kimmo Rosendahl	Ionific
Zach Shelby	University of Oulu
Martti Huttunen	University of Oulu
Matti Dahlbom	Mermit
Heikki Vesalainen	Mermit

Table of contents

1. FOREWORD

In this document is specified the SSI (Simple Sensor Interface) protocol structure. The SSI communications protocol is intented to be used to transfer data between sensor unit(s) and a terminal.

In the chapter 4 are presented as an example two alternative utilisation of the SSI protocol: point-to-point (UART) and networking (nanoIP) applications. In a point-to-point case the SSI protocol operates over a serial link (UART) connection. This can be physical (wired) or virtual wireless Bluetooth serial port. The SSI protocol also has the capability to be used over layer 3 networking protocols such as TCP/IP or nanoIP.

The criterion for SSI protocol development are:

- general purpose
- simple – minimal overhead
- small footprint on the server (sensor) side

The version of the SSI protocol is presented as ‘A.B’, where ‘A’ is the main version and ‘B’ is the minor version. The dot ‘.’ between the main and minor version number is a separator, not a decimal point.

2. REQUIREMENTS

The wireless sensor units must operate long period on a small battery. Thus the protocol must not be too complex.

The following configurations can be identified:

- The terminal can be connected into one or more sensor units through the SSI protocol
- There may be one or more sensor boards behind one communications link
- There may be one or more sensors on the sensor board

Within the reference environment the SSI protocol and the application program (client) operating on the Symbian OS must be able to:

- Find sensor devices
- Read data from sensor devices
- Send data to sensor devices

In the reference environment below sensor data is monitored over the Bluetooth virtual serial port connection.

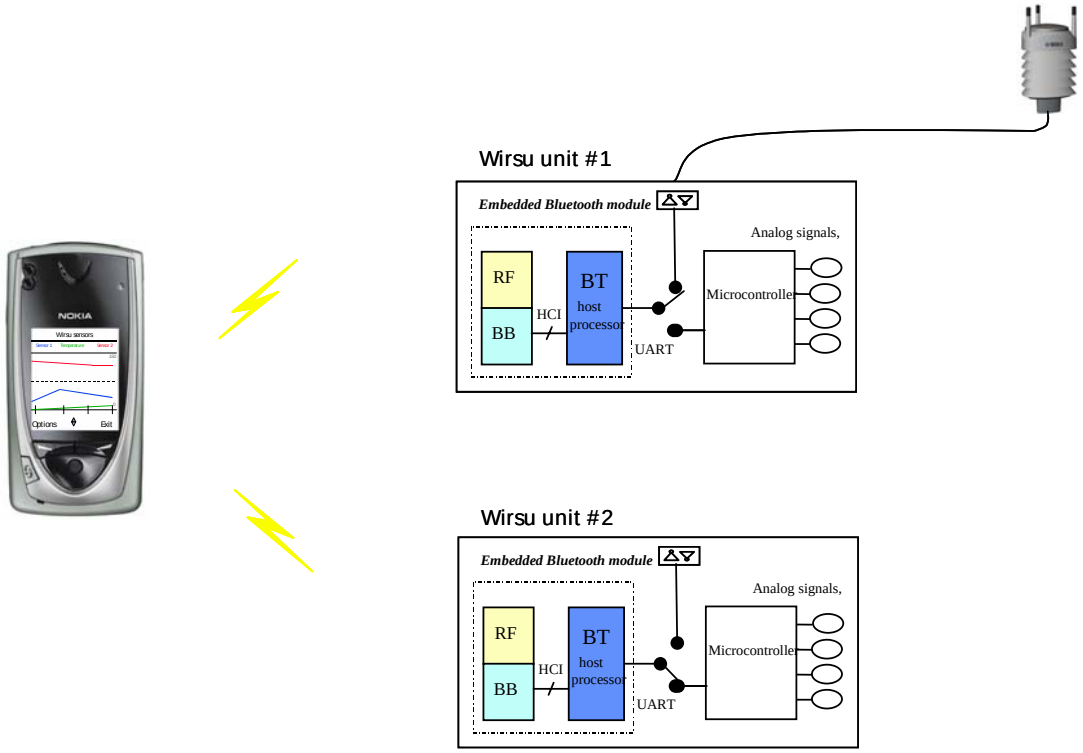


Figure 1. A reference environment configuration.

As a terminal can be used a Symbian based mobile phone to monitor sensor data on the sensor boards. The mobile terminal can operate also as a gateway to obtain Internet connectivity.

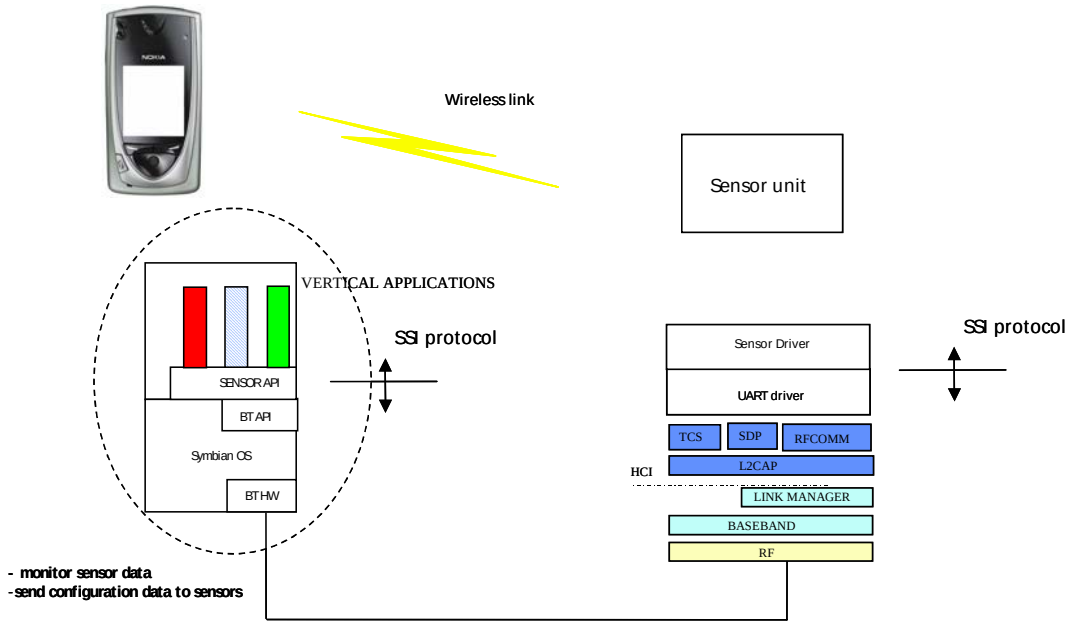


Figure 2. Application implementation example within a reference environment.

The wireless link between the server (sensor unit) and the client (terminal) can be e.g. Bluetooth, BTLEE (Bluetooth Low End Extension), UWB or any proprietary radio system.

3. SSI PROTOCOL

This chapter describes the low-level protocol for sensor devices. The SSI protocol is asynchronous and stateless. The SSI protocol command structure consists of three parts:

- a header
- a payload
- an optional CRC checksum.

The header structure and length may change from case to case, but the payload structure is always the same. Here are presented two different five byte header structures; UART (point-to-point) connection and embedded networking (nanoIP).

The SSI protocol byte order is **Big Endian** (most signifigant byte first).

3.1 SSI UART Protocol

SSI UART protocol is based on command messages as shown in figure 3 below. All messages use the same frame format with a header, a message body and an optional two byte CRC checksum.

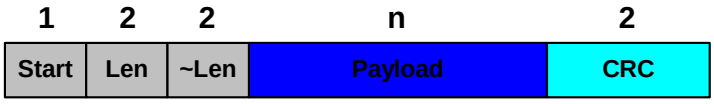


Figure 3 – SSI UART protocol command structure

Each message frame contains a 5-byte header and a varying size payload. Message header contains

- A start byte equal to 0xFE
- Length of the message in bytes
- Bitwise NOT of the length, to help identifying frame start
- Payload (Device address, Command code (a-z, A-Z), etc.)
- CRC checksum (optional)

The optional CRC checksum is present in the message frame only when the command code is written in lower case (a-z). The CRC checksum is calculated over the payload part. The algorithm for the CRC calculation is given in Appendix 1.

If the CRC is not correct, the message has to be ignored.

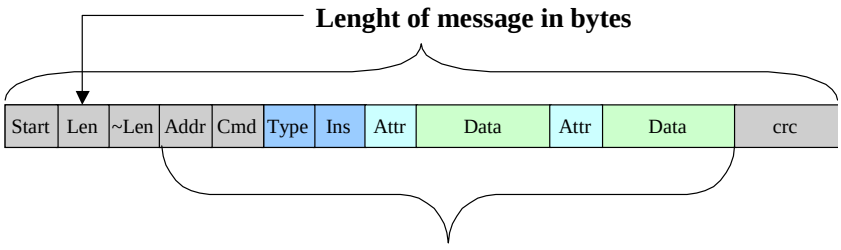


Figure 4 – An example of the SSI UART protocol message

3.2 SSI Networking Protocol

The SSI protocol also has the capability to be used over layer 3 networking protocols such as TCP/IP or nanoIP. In this chapter only nanoIP is considered, although it is identical to apply the SSI protocol to any other socket based network protocol. In this section the modification of the SSI UART protocol frame structure along with settings for its use over nanoIP are introduced. The nanoIP is a minimal networking protocol for use with very limited devices over a single subnet. The nanoIp development work is done in the University of Oulu, CWC center.

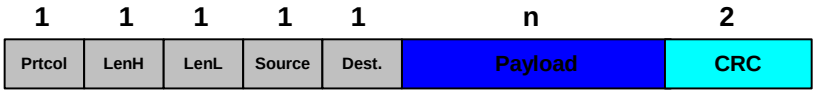


Figure 5 - SSI nanoUDP message format

In the nanoIP case the five byte header consists of

- A protocol and flag byte – for nanoIP usage only
- LenH – msb byte of the payload and CRC length
- LenL – lsb byte of the payload and CRC length
- Source port number
- Destination port number (0x28)

When a node sends a packet using the SSI protocol over nanoIP, the SSI protocol frame will be encapsulated in a nanoUDP header (5 bytes), which contains the length and 40 (0x28) for the source and destination ports.

4. PAYLOAD

Within the SSI protocol structure the payload section does not depend upon the type of the connection. The first byte of the payload is an address -'Addr'- and the second byte a command –'Cmd'. These are present within every SSI payload structure. The other parts depend upon the command.

4.1 Address field

The address field is required to separate multiple sensor devices on a single communication device. Address field is always present and one byte long.

Address field is always a hexadecimal number (0x00 – 0xFF). If the field value is '?' (0x3F) with the 'Query' command it means 'wildcard'.

4.2 Command field

The command field is one byte long thus allowing 255 different alternatives. Because each command has two variants (with and without a CRC checksum) altogether 127 different commands are possible. The commands has be chosen so that the command can be understood as an ASCII character, but this is not necessary. Within each command the payload field structure is the same.

SSI protocol defines the following 18 different commands as seen in Table 1. Six commands are sent always from the terminal (client) to the sensor unit (server). Five commands are sent always from the sensor unit to the terminal. One command - 'F' – 'Free data for custom purposes' can be sent either by the client or server side.

Note! It is strongly advised, that the command 'Free data for custom purposes' is not used.

Each command has two variants, one without and another with a CRC checksum. A bit-mask of 0x20 is used to determine whether a CRC is being used. Effectively this means that if the command byte is in lower-case (a-z), a CRC is part of the message.

Table 1 - SSI command messages

Command byte	Dir.	Description
Q,q (0x51,0x71)	->	Query
A,a (0x41,0x61)	<-	Query reply
C,c (0x43,0x63)	->	Discover sensors
N,n (0x4E,0x6E)	<-	Discovery reply
Z,z (0x5A,0x7A)	->	Reset Wirsu device
G,g (0x47,0x67)	->	Get configuration data for a sensor.
X,x (0x58,0x78)	<-	Configuration data response
S,s (0x53,0x73)	->	Set configuration data for a sensor
R,r (0x52,0x72)	->	Request sensor data
V,v (0x56,0x76)	<-	Sensor data response
D,d (0x44,0x64)	<-	Sensor response with one byte status field
O,o (0x4F,0x6F)	->	Create sensor observer
Y,y (0x59,0x79)	<-	Observer created
K,k (0x4B,0x6B)	<->	Delete sensor observer / listener
U,u (0x55,0x75)	<->	Observer / listener finished
L,l (0x4C,0x6C)	<-	Request sensor listener
J,j (0x4A,0x6A)	->	Sensor listener created
E,e (0x45, 0x65)	<->	Error
F,f (0x46, 0x66)	<->	Free data for custom purposes

Every command is directed at a sensor, a set of sensors or a sensor device. Sensors are identified by a 16-bit value – Sensor Id. Sensor devices are identified by the address.
Direction (Dir.) identifies the command direction: '->' means from terminal to the sensor unit and '<-' vice versa.

The group of commands

Q – guery
A – guery reply
C – sensor discovery
N – discovery reply
Z – reset
G - get sensor configuration
S – set sensor configuration

are used to find and configure sensor units utilizing SSI-protocol.

The group of commands

R – request sensor data
V – data response
D – data response with status field

are used to read sensor data infrequently. For data streaming purposes are defined commands

O – create sensor observer
Y – observer created
K – delete observer
U – observer finished
L – request sensor listener
J – sensor listener created.

On the chapters below are described the SSI commands combined with the possible other fields. Here is seen the 'Payload' part only.

4.2.1 Query – 'Q'

1	1
Addr	Q/q

or

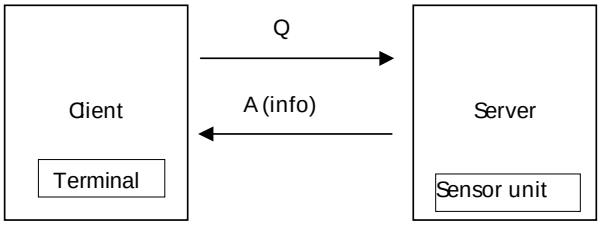
1	1
?	Q/q

The 'Query' command is sent by the terminal to find out:

- Are there any sensor devices present?
- What SSI protocol version they are using?
- How long is the input buffer?
- How long delay there must be between successive messages?

The sensor units must be identified using the SSI protocol. The address field value '?' is a wildcard.

Total length of this command structure is two bytes.



4.2.2 Query response – 'A'

1	1	2	2	2	2
Addr	A/a	Version	buffer size	Delay	Reserve 2

The 'Query response' command is sent as a response to the client. The sensor unit sends a response 'Query response' with the address, protocol version (2 bytes), buffer size (2 bytes) and Delay (2 bytes). A 2 byte field 'Reserve 2' is for the future usage.

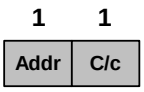
Protocol version is described so that first byte is the main version and the second byte the minor version. As an example the version '0.70' is 0x0046.

Buffer size is the length of input buffer in bytes.

Delay is the delay value between successive messages in milliseconds. Zero value means that the messages can be sent immediately after each other.

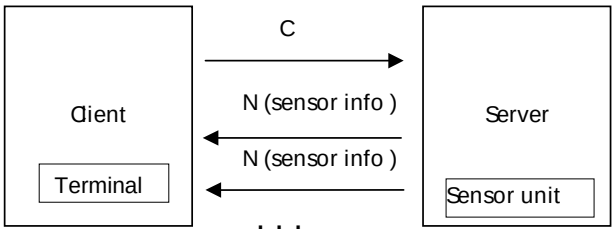
Total length of the 'query reply' is 10 bytes.

4.2.3 Discover sensors – 'C'

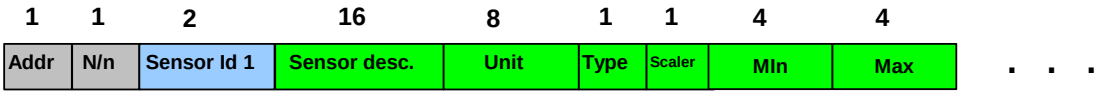


This is a *Discovery* command. The addressed device replies with one or more `N´ command(s) identifying the sensors it has.

Total length of the discover sensors command is two bytes.



4.2.4 Discovery reply – 'N'



Discovery Reply message contains information about one or more sensors. Each sensor is identified with

- Sensor Id – 2bytes
- Description – 16 byte ASCII
- Unit – 8 byte ASCII
- Type – 1 byte
- Scaler - signed 1 byte
- Min – minimum sensor reading value
- Max – maximum sensor reading value

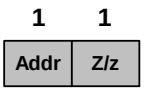
Type	Description
0x00	4-byte floating point format
0x01	Signed 32-bit integer
0x02	Non displayable configuration sensor Id

The scaler defines the number of viewed decimals for floating point numbers and 10's exponent multiplier for integers.

A special sensor id of 0xFFFF indicates the end of discovery replies for that particular address. However the client application should not be dependent upon waiting the reply, and should act asynchronously.

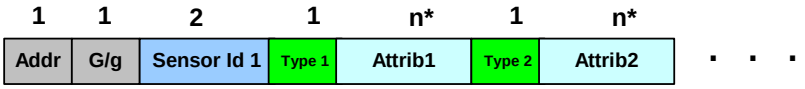
The minimum length of the discovery reply is 38 bytes if the information of only one sensor is transferred.

4.2.5 Reset Sensor device – 'Z'

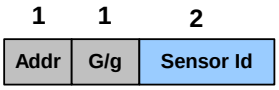


The *Reset* command causes the sensor device to reset itself to a specified initial stage.

4.2.6 Get configuration data - 'G'

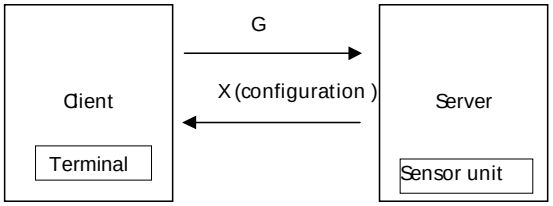


or

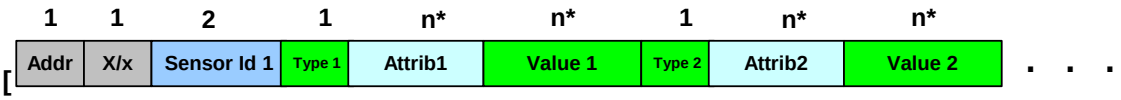


The latter means that all the sensor configuration attributes has to be sent to the client. The attribute length (n*) depends upon the type definition, see 5.2.7.
Only the high nibble of the type is used, the lower nibble must be set to 0x0.

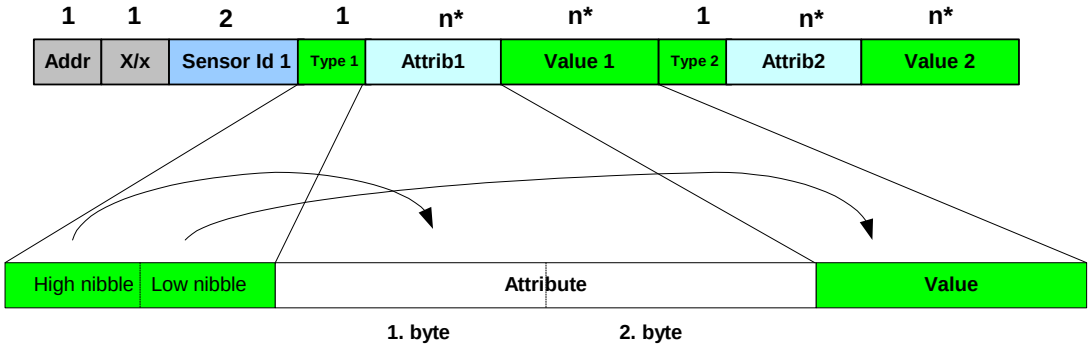
Response command is 'X'.



4.2.7 Configuration data response –'X'



The high nibble of the type defines the attribute and the low nibble the data value (4 bytes, when integer).



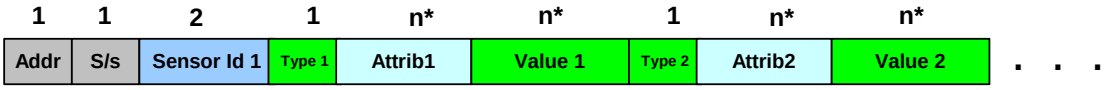
On the example above the type value is '0x21'. The attribute field is 2 bytes ASCII and the value field is 1 byte ASCII.

Nibble	Description	Note
0000	Null field	
0001	ASCII 1	
0010	ASCII 2	
0011	ASCII 4	All unused bytes in the tail are filled with 0x00.
0100	ASCII 8	
0101	ASCII 16	
0110	ASCII 32	
0111	ASCII n	'n' is transferred on the 1. byte of the field in question
1000	Integer divider 1	All integer values are 2 bytes
1001	Integer divider 10	
1010	Integer divider 100	
1011	Integer divider 1,000	
1100	Integer divider 10,000	
1101	Integer divider 100,000	
1110	Integer divider 1,000,000	
1111	4-byte floating point format	

When ASCII data is transferred on the attribute and value fields, unused bytes in the tail are filled with 0x00.
When a valid number of ASCII characters is transferred (nibble value 0x7), the number 'n' is on the first byte of the attribute or value field. The value 'n' is presented in binary (0xXX).

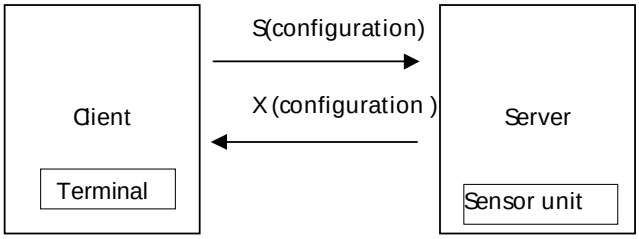
All integer values are 2 bytes.

4.2.8 Set configuration data - 'S'

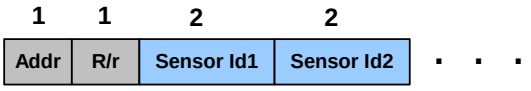


The *Set Configuration Data* command sets one or more sensor configuration attributes a new value. Each attribute is identified with two byte sensor Id. The format of the attribute and value (integer, floating point) is specified on the type field. See 5.2.7.

Response command is 'X' with the actual values of the sensor configuration attributes after the setting.



4.2.9 Request Sensor Data - 'R'

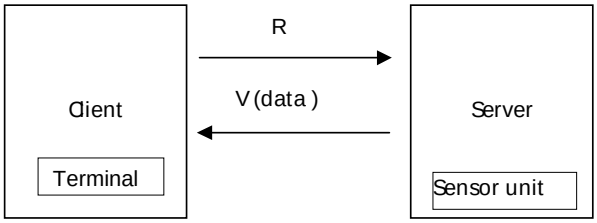


or

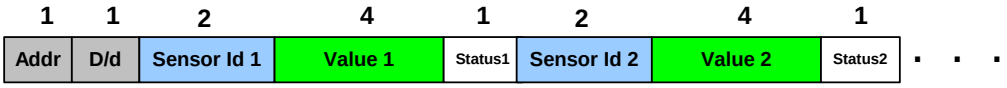
The *Request Sensor Data* command requests the values of one or more sensors. Each sensor is identified with a 16-bit Sensor Id field. The latter means that all the sensor values has to be sent.
Response command is 'V' with the (4-byte) sensor values or ‘D’ where a status field is included.

4.2.10 Sensor Data Response – 'V'

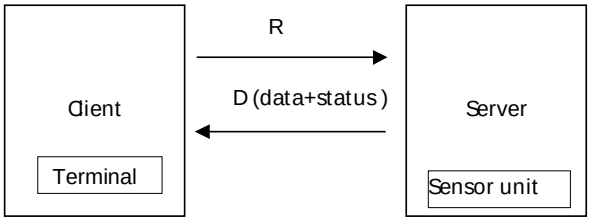
This command is used as a response to 'R' command from the client. The message body contains data for one or more sensors. For each sensor four byte data value is returned. The unit, data type and scaler of the returned value are defined in Discovery reply 'N'.



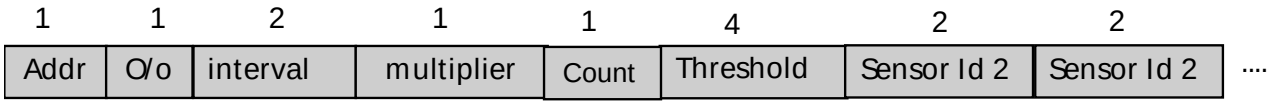
4.2.11 Sensor Data Response – 'D'



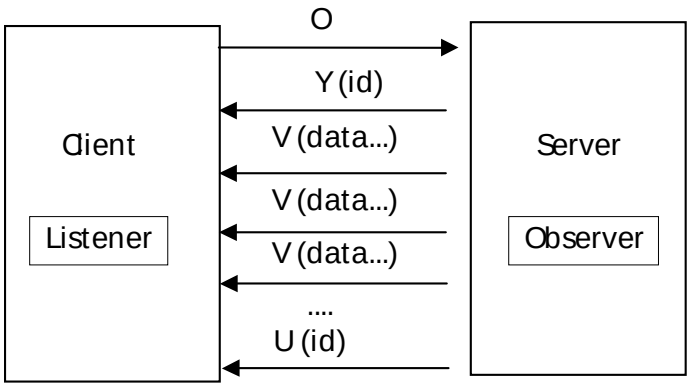
The 1 byte status field is application specific and the protocol does not care about it.



4.2.12 Create sensor observer – 'O'



- interval – data interval in milliseconds, 2 bytes integer
- multiplier – 10's exponent multiplier for rate, signed 1 byte
- count – number of sensor data messages (V) to be sent
- threshold – required sensor value change



Command create sensor observer asks server to create data observer for set of sensors identified with Sensor Id numbers. Server replies with command 'Y' and observer identification number (1 byte integer). Observer starts to send sensor data with Sensor Data Response 'V'. Client takes care that is has set a listener for sensor data when create observer command has been sent.

The **interval** is given in *milliseconds*. If the interval is zero, data has been send when new data is valid in observer. **Multiplier** is signed integer defining the actual data sending interval for the observer – observer_interval.

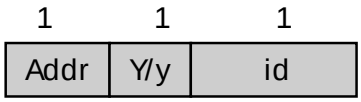
Observer_interval (ms) = interval (ms) x 10 $^{\pm multiplier}$

The **count** (one byte integer) informs the number of data response messages (V) the server has to send. If the count is '0xFF', the data response messages are sent continuously and the process is stopped with the 'Kill sensor observer' command (K).

Threshold (four bytes) determines that the sensor value is only sent when the change compared to the last reading exceeds the value given in threshold. The threshold is an absolute value. The data format (4-byte floating point or integer) is the same as with the data reading associated to the sensor. See command 'N' – discovery reply.

The server ends the data streaming session by sending the command 'Observer finished' (U) to the client.

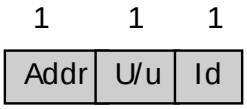
4.2.13 Observer created – ‘Y’



- id – Observer identification number, 1 byte integer

When server creates the data observer it sends acknowledgement followed by observer id number.

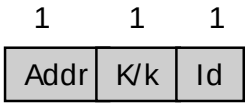
4.2.14 Observer finished – ‘U’



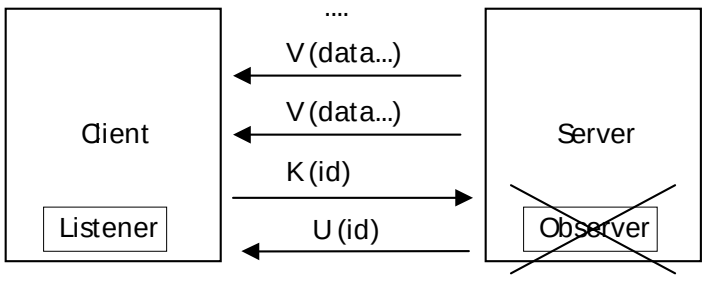
id – Observer identification number, 1 byte integer

The server sends ‘Observer finished’ message to the client when the data streaming process has ended.

4.2.15 Kill sensor observer – ‘K’



- id – Observer identification number, 1 byte integer

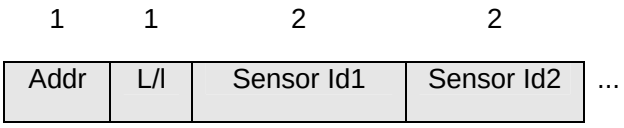


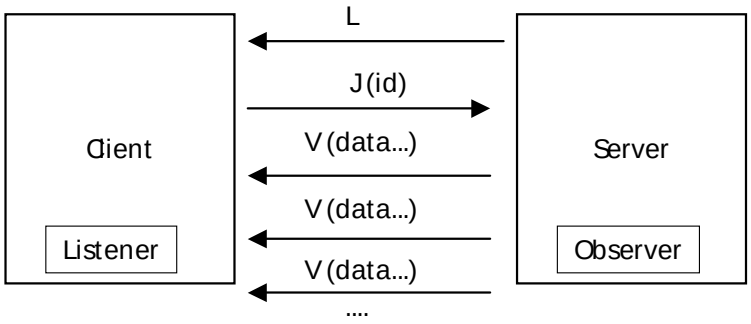
Kill sensor observer command deletes the observer bound with observer id. The client takes care to delete the listener for particular observer. Server acknowledges by sending ‘Observer finished’ message to the client.

If server sends ‘K’ command to the client then server is requesting client to kill sensor listener with given Id. Client confirms the listener deletion by sending command ‘U’ to the server.

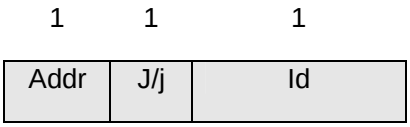
4.2.16 Request sensor listener – 'L'

Server can request client to create a sensor listener for coming data as is case 'Create sensor observer'. The server should know on which rate data would be send to the client. When client has set a listener for incoming data it will send acknowledgement Listener created - 'J' to server. After acknowledgement server can start to send data to the client.





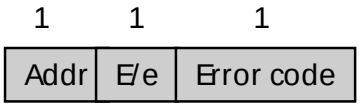
4.2.17 Sensor listener created – 'J'



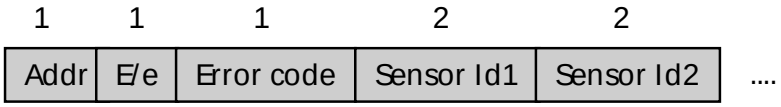
- id – listener identification number, 1 byte integer

Acknowledgement from client to server that sensor listener has been created.

4.2.18 Error – 'E'



or



Error code can be send anytime from server to client or from client to server. If specific sensor that causes error is known the sensor id can be added in error message.

General error codes:

Error code	Name	Description
0x00	Error	Generic error
0x01	Unsupported command	Client / host doesn't support command/feature
0x02	Wrong sensor ID	
0x03	Calibration failure	Some sensor devices has calibration failures
0x04	Data rate too high	Device doesn't support so high data rate
0x05	Unable to create observer	Server cannot create the sensor observer
0x06	Unable to create listener	Client cannot create the sensor listener

Other codes are application/sensor specific.

4.2.19 Free data – 'F'

The *Free data* command can be used to extend the protocol safely for custom purposes.

4.3 Sensor Id

Sensor Id field is used to identify different sensors on the sensor unit. The sensor unit is identified by the 'Addr' field. Typically there is only one sensor unit under each communication link, but multiple sensor units are possible.

4.4 Attribute field

The attribute field identifies different attributes for a certain Sensor Id. See chapters 4.2.7 and 4.2.8

4.5 Field values

In this section are defined the default values for different fields.

4.5.1 Sensor Id

Sensor Id field specify the sensor unit for a special address.

The value '0xFFFF' is reserved for Wirsu protocol usage. When the two most significant bits are 'one' (0xC000) The sensor id field is defined as 'group' and 'instance' hierarchy. The group mask is 0x3F00 and the instance mask 0x00FF.

5. RFID TAG COMPATIBILITY WITH SSI

An RFID tag cannot support an SSI server. For reading SSI compatible RFID sensor tags, one should have a virtual SSI server on the terminal device, which reads the appropriate sections of the memory of the RFID tag to answer the queries of the SSI client. In this section, the layout of the memory for SSI compatibility is defined.

The memory layout of a RFID tag is defined in the ISO 18000-4 standard (see Section 5.1). Mode 1 (Passive backscatter RFID system) is used. After the data written by the manufacturer, there is room for user data (here the sensor value and information, see Section 5.2). The possibility of adding variable length configuration data by the sensor owner is discussed in Section 5.3.

5.1 Manufacturer data

	Bytes	Field name	Written	Locked
8	0x00 – 0x07	Tag ID	Manufacturing	Manufacturing
4	0x08 – 0x09	Tag manufacturer	Manufacturing	Manufacturing
	0x0A – 0x0B	Tag hardware type	Manufacturing	Manufacturing
6	0x0C	Tag memory layout: Embedded Application Code	Manufacturing or application	As required by application
	0x0D – 0x11	Tag memory layout: Tag Memory Map Allocation		
	0x12 –	User data	Application	As required

Table 1. Mode 1 Tag memory map

There should be code **0x53** (for ASCII "S") in address 0x0C (Embedded Application Code / Tag memory layout) to define that the memory layout is SSI compliant, i.e., as the next section defines.

5.2 Sensor data

Here is assumed, that **only one sensor** is connected to the RFID tag. Sensor data is written to the "User data" section in the Tag memory (from address 0x12).

Sensor value has to be written to the EEPROM addresses 0x12 – 0x15. On the addresses 0x16 – 0x3B are sensor specific data, which is written by the 'sensor owner' and must not be changed by the application. The data is here assembled to 8 byte blocks for the convenience of using ISO 18000-4 commands.
The status of the sensor is written in byte address 0x19, where the bit 0 indicates the status of the sensor data. As the sensor is powered up by the reader, the status bit is zero. After the sensor has written the data in the byte address 0x12 – 0x15, the status bit is toggled to one. This is the indication to the reader that the sensor value is valid. After the reader has read the sensor data, it should overwrite the status bit with zero.

	Byte address	**Field name**	**Type**	**Description**	**Example**
8	0x12 – 0x15	Sensor value	4 byte HEX	Variable sensor value	0x00000014
	0x16	Type	1 byte HEX	Describes the type of the Sensor value (0x00 = floating point, 0x01 = signed integer…)	0x01
	0x17	Multiplier	1 byte HEX		0x00
	0x18	Status	1 byte HEX	Sensor status. Bit 0 indicates if the sensor value (0x12 – 0x15) is valid data (bit 0 = 1) or not yet valid (bit 0 = 0). Seven bits are free for future use.	0x01
	0x19	Empty	1 byte	For future use. Could be, e.g., number of sensors on RFID tag.	
16	0x1A – 0x29	Sensor description	16 byte ASCII	Constant sensor description	"Temperature sensor"
8	0x2A – 0x31	Unit	8 byte ASCII	Constant unit description. All unused bytes in the tail are filled with 0x00.	"C "
8	0x32 – 0x35	Minimum value	4 byte HEX	Minimum value the sensor can provide.	0x00000000
	0x36 – 0x3B	Maximum value	4 byte HEX	Maximum value the sensor can provide.	0x00000064
4	0x3C	Activate sensor	1 byte	Bits 0 – 7 will be written by the reader to request the tag to write the sensor data	0x01
	0x3D – 0x3F	Sensor control	3 bytes	Not defined yet. Could be, e.g., time needed before sensor value is valid.	
n	0x40 –	Configuration data	n bytes	Optional, see next section.	

Table 2. Sensor data mapping to the Tag memory

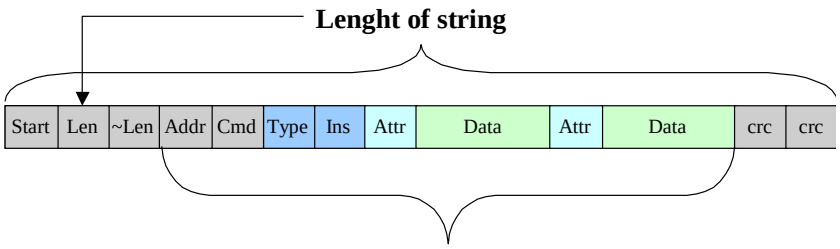
5.3 Optional configuration data

Here is an example configuration data memory layout based on the configuration data on SSI specification. The data addresses are variable due to the variable length of the data. Type fields could contain information on the length and type of both attributes and data, as shown in Chapter 4.2.7 "Configuration data response".

Byte address	**Field name**	**Type**	**Description**
A1	Type 1	1 byte	First half-byte could describe Attribute 1, second half-byte Value 1.
(A1+1) – (A1+N1)	Attribute 1	N1 bytes	Type could be ASCII, integer or float.
(A1+1+N1) – (A1+N1+M1)	Value 1	M1 bytes	Type could be ASCII, integer or float.
A2 = A1+N1+M1+1	Type 2	…	…

APPENDIX I CRC CALCULATION ALGORITHM

The CRC is calculated over the Payload section, see the figure below.



Data included in CRC calculation

CRC-Computation

All operations are assumed to be on 16 bit unsigned integers. The least significant bit is on the right. The algorithm is:

Initialize the CRC to Zero. For each character beginning with the address up to the lenght end of the string but not including the two bytes of CRC itself.

```
{
   Set the CRC equal to the exclusive OR of the character and itself
   for count =1 to 8
   {
      if the least significant bit of the CRC is one
      {
         right shift the CRC one bit
         set CRC equal to the exclusive OR of 0xA001 and itself
      }
      else
      {
         right shift the CRC one bit
      }
   }
}
```

APPENDIX II REQUIREMENTS FOR NANOIP IMPLEMENTATION

The SSI protocol also has the capability to be used over layer 3 networking protocols such as TCP/IP or nanoIP. In this chapter only nanoIP is considered, although it is identical to apply the SSI protocol to any other socket based network protocol. In this section the modification of the SSI UART protocol frame structure along with settings for its use over nanoIP are introduced. In addition the networking architecture for using SSI over nanoIP is explored.

The modifications needed to modify the protocol in Chapter 4 for use over nanoIP are straightforward. The changes include the following:

1. Removal of the **start**, **len** and **~len** fields from the frame.
2. The **addr** field is still included in case there are multiple identical sensor sets on the same device.
3. NanoIP is used for providing length and address information.
4. Standard socket **port 40** is used for identifying the SSI protocol.
5. NanoUDP is used by default.

When a node sends a packet using the SSI protocol over nanoIP, the SSI protocol frame will be encapsulated in a nanoUDP header (5 bytes) which contains the length and 40 for the source and destination ports.

A reference network architecture for WIRSU project

The basic internal architecture of nodes is shown in Figure 1. The SSI protocol runs over the socket interface of the nanoIP stack in all types of nodes. In addition, a WIRSU application programmers interface (API) is employed in browing devices.

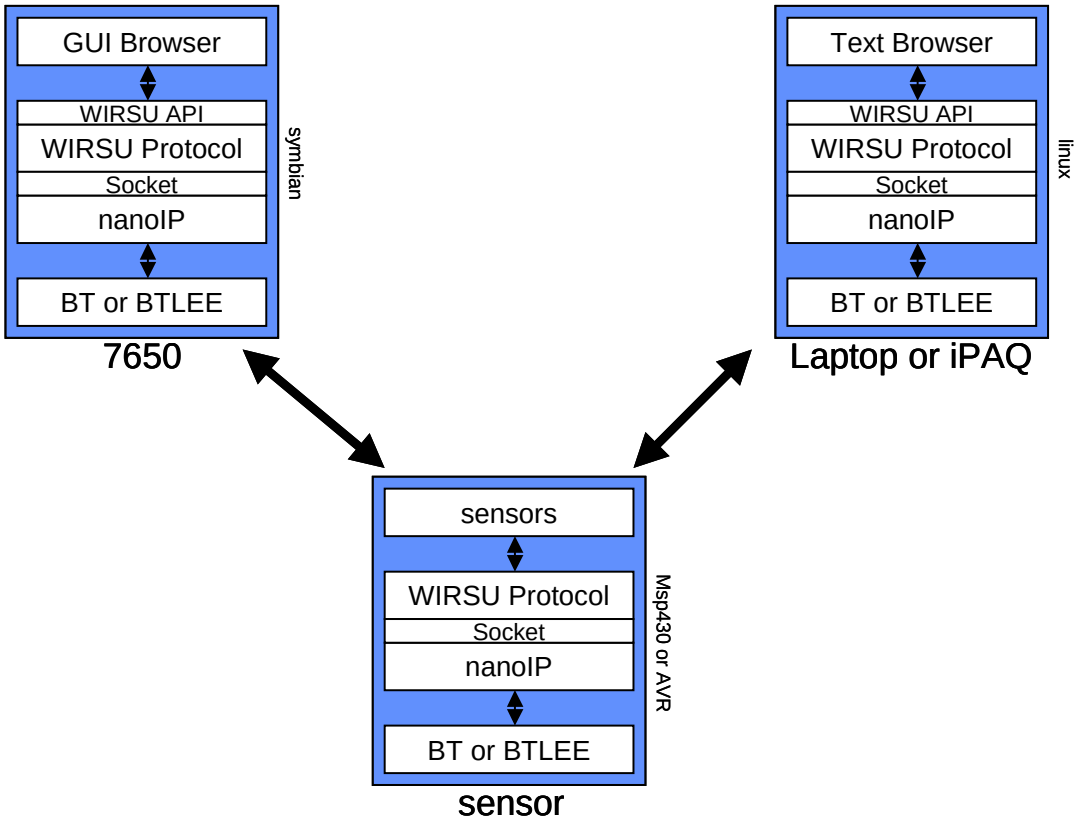


Figure 1. Architecture of phase III demo devices