CoLa 2.0

Protocol Specification

Author : K. Schröder, N. Falkenberg, S. Heidepriem

Printing date : 2015-10-20

Revision History

Version	Description	Date
0.10	First Draft	28.02.2013
0.20	First Revision	22.03.2013
0.30	Revised Introduction and Glossar	23.04.2013
	Restructured the Chapters	
	Added some Introductory Texts and Figures	
	Added Timing Constraints	
0.40	Spell Check	26.04.2013
0.50	Changed Parameter of SetIpParameter	12.06.2013
	Changed Parameter of FindMe	
	Added flag for CoLa Scan	
	Added InterfaceNumber to RemoteScan	
	Network Configuration in CoLa Scan modified	
0.60	Changed timeouts of CoLa Scan	24.07.2013
	Added interface number to 'J'oinNetwork and 'H'ello command	
	Changed Chapter Security to SHA256	
0.70	Corrections from some Reviews	26.09.2013
0.80	Removed Security chapter	18.11.2013
0.90	Changed structure of Device Info	05.12.2013
0.91	Removed Up-and Download commands Added the 'D'(DeviceInfo) command	18.12.2013
	Changed CoLa2 port number	

	Added Firmwareversion to DeviceInfo	
	Added handling for not reachable devices to ‘H’ command	
1.00	Changed “Sopas Hub” to “CoLa Hub” and corrected port 2014 to 2122 (Page 7)	19.12.2013
1.01	Added the ‘N’ Command to modify Remote Network	10.01.2014
	Added the ‘B’(Blink/Beep) Command	
1.02	Move changes from Errata sheet into this new document	09.10.2015
1.03	Added CoLa Serial Bus (CSB) Specification/ chapter 5.2 (not colored blue because it is unreadable).	15.10.2015

1. Content

2. Introduction

The Command Language CoLa is a protocol to monitor, maintain and use SICK sensors. The protocol specification is strongly based on the SOPAS middleware concept and the device structure defined by the SOPAS communication interface description CID.

The protocol is mostly based on other commonly known protocol specifications like TCP/IP or USB bulk transfer which will be treated as the «CoLa Transport Layer» (See chapter «CoLa Transport Layers»)

This document contains the official specification of CoLa 2.x. The original storage is in the SICK EDB with the document number »E_70754 «.

The CoLa 2.x protocol is used to access the device for the purpose of maintenance, parameterization, monitoring and in some cases for automation (transmission of measured values). Against the former CoLa specification this document rules all facets of the communication protocol. CoLa 2.x must not be used in combination with any transport protocol but the defined transport protocols defined in here.

Version 2.0 defines the usage of CoLa on USB using the bulk transfer definition and on Ethernet using the TCP/IP specification. More transport techniques will be defined with CoLa 2.1 (e.g. using a half-duplex UART line).

Before revising the CoLa specification an extensive requirements analysis was done asking representatives of all SICK divisions for needs and inconveniences. The hereby called requirements have been separated into issues which will be immediately realized in the specification (CoLa 2.0), some will be specified later (CoLa 2.1) and some will remain in the open-for-discussion state.

Design Decisions

During the requirements analysis some design decisions have been made.

- CoLa also specifies all underlying transport mechanisms or refers to specific existing specifications (e.g. TCP/IP). There is no more discretion of usage in any communication scenario.
- With CoLa 2.x most of the former options will become mandatory or will be removed from specification in favor of a more unique behavior of all CoLa devices.
- All new features in CoLa 2.0 shall be mandatory in favor of a unique set of features in all devices.
- CoLa 2.x is not a trade mark nor are any constraints given to partly use this specification, but a device must not be called supporting CoLa 2.0 or CoLa 2.1 before the CoLa 2.0 or CoLa 2.1 conformance has successfully been passed and no derivation from this specification is known.
- In most cases the CoLa specification only specifies a protocol which belongs to the layer 7 of the well-known ISO/OSI model (application layer).
- The protocol shall not implement features which are already implemented in underlying protocol.
- Hence a set of requirements are defined which have to be fulfilled by all of these «underlying protocols». These requirements have been named axiom 1… axiom n.
- These “subjacent protocols” are named «CoLa Transport Layer».
- The specification shall be scalable. It shall be adequate for complex devices with huge resources and also for very small devices.
- But very small sensors, not able to handle a multi-client environment, are out of focus. These devices are meant to support IO-Link which will also support a SOPAS/CID mapping (in future).

History of CoLa

The command language CoLa was designed in 2002 as ASCII protocol for configuration, diagnosis and measurement value transmission of SICK sensors. Big focus of this protocol was the ease of use at simple terminal software. As CoLa became more and more successful a binary variant of CoLa was invented. For this purpose a new framing was designed and data was serialized henceforth in a binary representation using the so called Motorola byte order (Big Endian).

By the time more and more features have been added to the CoLa specification like the CoLa routing (SOPAS-Hub), new data types, new commands and much more. But security and stability aspects were never improved due to compatibility reasons.

Now with this new version of the CoLa specification the compatibility requirement will be dropped in favor of an enhanced and coherent protocol specification.

Enhancements of CoLa 2.0

New Header Structure (framing)

The header (framing) has been revised. The new framing includes also the former HubAddress which becomes mandatory in all frames.

The CRC32 was omitted due to an adequate implementation in both selected CoLa Transport Layers TCP/IP and USB bulk transfer. Therefore the checksum is defined as an axiom (requirement) for all CoLa Transport Layers.

Events had become reliable (no event shall be dropped). It was a matter of simplicity to define the reliability of telegrams (retry) as part of the CoLa Transport Layer (axiom).

Henceforth a client can use a 32 bit field named «Request ID» to assign a response to their associated request.

Tight Client/Server Binding (session management)

While a CoLa server (SRT) had to do the session management implicitly, CoLa 2.x defines an explicit session management. Commands for starting and ending a session (open/close) are introduced.

A mandatory heart beat definition allows the server to detect when a client has lost the connection. This «heart beat» becomes mandatory otherwise the session automatically ends.

CoLa Hub Routing enhanced and becomes mandatory

The CoLa Hub routing has been enhanced and becomes a mandatory feature of all CoLa 2.x devices. At least the differentiation of client s while being placed behind a CoLa Hub will be a capability of all CoLa 2.x devices. (Client Handling[1])

The addresses formerly known as HubAddress will be named PortID and will be 32 bit wide which allows a separation into a high word and low word which will be used internally to route messages in the sensor (between CPUs or threads).

Strict Timing Constraints (P2P)

While SOPAS defined rather gracious timing constraints which have not been part of the CoLa specifications, CoLa 2.0 defines stricter timing constraints concerning the physical communication interface of the device.

Find and Explore Devices with One Command: CoLa Scan

The CoLa scan technology formerly known as SOPAS AutoIP becomes a mandatory part of the CoLa 2.0 specification.

[1] Devices which do not have enough resources for the client handling will have to switch to IO-Link or use the CoLa 1.0 specification.

In addition the answer of devices will contain all information the client needs to decide whether to establish a connection (all information of the so called standard block) and all communication information which are needed to establish a good and stable connection.

A client can demand a device to perform such a scan at a network connected at another port.

The communication between a searched device and the searching device is optimized according to whether broadcast is necessary or whether UDP unicast can be used.

Ethernet Transport Layer

The usage of TCP/IP with port number 2122 is specified.

USB Transport Layer

The usage of USB «bulk transfer» and end points are specified.

International Characters

Since CoLa 2.0 discontinues the ASCII protocol CoLa A and the framing supports the transport of binary data (bytes from 0…255 allowed in data), Strings will be defined to carry all kind of data. Hence the transport of UTF-8 strings is now possible.

The definition of strings in CID will be supplemented by an attribute specifying whether the content has to be treated as a 8 bit ASCII value (ISO-8859-15, 0x00 not allowed, legacy, default) or an UTF-8 String. CID properties will not be described in this specification.

Binary Large Objects

New commands for sending so called BLOBs to the device or receive them from the device are introduced:

• Upload	(from client/PC to server/device)
• Download	(from server/device to client/PC)

BLOBs are substitutes for CID-Files and are free of type information (byte array). Transmission of BLOBs is done in fragments (like writing to a file) and SRT will pass those fragments to the application without swapping of bytes.

Optimized Serialization

In certain cases serialization can be done in the device by simply streaming the content of the memory to the communication port (DMA, memcpy() …). This feature shall be used for CPUs organized in Big Endian architecture and organized in Little Endian architecture. Hence the serialization of numbers bigger than 8 bit is either transferred in Big Endian order or in Little Endian order.

The order of bytes is defined for the whole device and is part of the device information passed by the CoLa Scan (communication settings).

Request Queues

A CoLa 2.x device can receive multiple requests before having answered them. All involved queues are able to avoid an overflow of the queue.

Planned Enhancements with CoLa 2.1

This document describes the gathered requirements which were decided not to be in CoLa 2.0 and are probably part of CoLa 2.1. But a detailed requirements assessment will take place before CoLa 2.1 will be started.

It is determined that a CoLa 2.1 device can have more capabilities than a CoLa 2.0 device. But it will be compliant in so far that a CoLa 2.1 conformance test will also state the conformity for devices supporting CoLa 2.0 only (but with missing optional features)

Exclusive Access

The definition of «Exclusive Access» has not been used in the past due to a small set of problems. One is the need of rescue strategy when a client obtained the «Exclusive Access» and disappears without returning it. Another deficit for some devices is the junction between «Exclusive Access» and unlocking of the device (UserLevel).

The future definition of «Exclusive Access» will be decoupled from the UserLevel and combined with the new session management which will lead to a rescue strategy when the heart beat is missing.

Alternative Timing Constraints (remote scenario)

With CoLa 2.1 the timing constraints will be revised by means of opening the answer time diagrams for remote scenarios (larger time outs in certain cases).

UART based Transport Layer

A «CoLa Transport Layer» will be defined for a simple RS485 and RS232 line implementing techniques for all given requirements (axioms) of a «CoLa Transport Layer».

Security

In CoLa 2.1 the ease of cracking the password to unlock the device will be significantly more difficult. First the time, the device does not react on password cracking attempts, will increase with every attempt.

Second a challenge response technique is introduced to avoid replay attacks.

Although security issues like data validation, one-time-passwords and authentication by certificates will be treated and (partly) specified.

PLC Friendly Channel

PLC programs have difficulties with dynamic lengths for telegram header (addressing by name) and dynamic lengths of arrays (FlexString, FlexArray). Hence CoLa 2.1 device shall ensure a possibility for PLC programs to address all necessary Variables, Events and Methods of the device in an easy way.

Profinet Channel

It is planned to specify a CoLa channel which uses Profinet services as transport layer. This shall enable a client to connect a SICK device with CoLa using a Siemens PLC as a gateway to the device.

EtherNet/IP Channel

It is planned to specify a CoLa channel which uses EtherNet/IP services as transport layer. This shall enable a client to connect a SICK device with CoLa using a Rockwell PLC as a gateway to the device.

Leaf Access

With CoLa 2.0 and before, the Variable is treated as an atom. Using communication a Variable can only be read or written completely. Methods can be defined to access parts of a Variable (members of a Struct or of an Array) but there is no generic way of addressing branches or leafs of a very complex Variable (tree).

It has been requested to specify and implement such an addressing mechanism within CoLa 2.x but there is still a heated debate about this issue.

3. Glossary

Term / Abbrevation	Declaration	Remark
CoLa	Command Language	A protocol definition for configuration and diagnostic purpose
LSB	Least significant byte/bit	
MSB	Most significant byte/bit	
SOPAS	SICK Open Portal for Applications and Systems	A middleware concept of SICK
HB	High byte	The MSB of a 16 bit value
LB	Low byte	The LSB of a 16 bit value
HW	High word	Bit 16…31 of a 32 bit value
LW	Low word	Bit 0…15 of a 32 bit value
Little Endian	A representation of multi byte values in a byte wise represented memory storing the LSB at the top most address or transmitting the LSB first in serialized representation.	The LSB (little value) is transferred or read first. Also known as Intel format due to the standard approach at Intel circuits.
Big Endian	A representation of multi byte values in a byte wise represented memory storing the MSB at the top most address or transmitting the MSB first in serialized representation.	The MSB (big value) is transferred or read first. Also known as Motorola format due to the standard approach at Motorola circuits.
CID	Communication Interface Description	A SICK standard for devices to describe their software interfaces (over communication) in a XML file. The standard is part of the SOPAS technology.
Layer	A software abstraction of a part of the CoLa protocol where high layers are close to the application and low layers are more close to the physical interface	Regarding Ethernet the CoLa specifications is added to the “application layers” on top of TCP/IP. Hence the CoLa layers, specifically defined here, must not be compared to the ISO/OSI layer model.
CoLa Transport-Layer	An artificial notion of those layers which were not part of the former CoLa 1.0 specification but used to transport CoLa telegrams	Certain prerequisites must be met to be a valid CoLa 2.x Transport Layer. (See introduction)
SOPAS hub	SOPAS device that is able to route CoLa messages to SOPAS devices in other networks	
NoC	Number of Cascades	The number of hops a telegram must be routed until it reaches its destination.
PortID	Each connection in a SOPAS device is describes by a unique PortID (32Bit). The high word represents the interface (e.g. “Eth0”) and the low word represents the connection (e.g.	

	socket number)	
Socket	Is one connection within an one interface	
ClientID	Identifier for a client that is unique within one device	
SRT	SOPAS Runtime	The typical implementation of a CoLa Server
SRTOS	A SRT Operating Systems (message based)	The internal queue for Events and asynchronous Methods is replaced by an abstraction for the OS functions for messaging.
SU_OSAI	SOPAS User Operating System Abstraction Interface	The C module which builds the abstraction interface between SRTOS and the operating system
MTU	Maximum Transmission Unit	Maximum size of a packet that the network layer can pass (e.g. 1500 bytes)
Limited Broadcast	Destination IP Address is 255.255.255.255	Destination is always in the own network. Limited broadcast will not forwarded by a router
Directed Broadcast	Destination are the members of a certain network.	Address is a combination of destination net address and all host bits set to '1'

3.1. Description Rules

Some proper names are written with capital letters when the corresponding specification of SOPAS is referred.
E.g.: A Variable is a SOPAS Variable. A variable is a normal variable (E.g. in a C module).

Rectangular brackets are mostly used to signal that the content between those brackets is optional.
Example: "This is my [tiny] cat". Here the word "tiny" is optional.

Numbers are normally decimal numbers (base 10) unless they have a prefix of "0x".
Example: (16 == 0x10) → true.

Single quotes signal a character (byte) representing the ASCII value of the character in between.
Examples: 'A' represents the byte with the value 0x41 (65) and '2' represents the byte with the value 0x32 (50).

Sharp brackets may be used to emphasize the entity of a byte.
Examples:
<0x32> represents a byte with the value 50 (or ASCII character '2').
<cmd> represents a byte which is meant to be the "cmd" (command byte).
<'e'> represents a byte with the value 'e' (0x65 or 101).

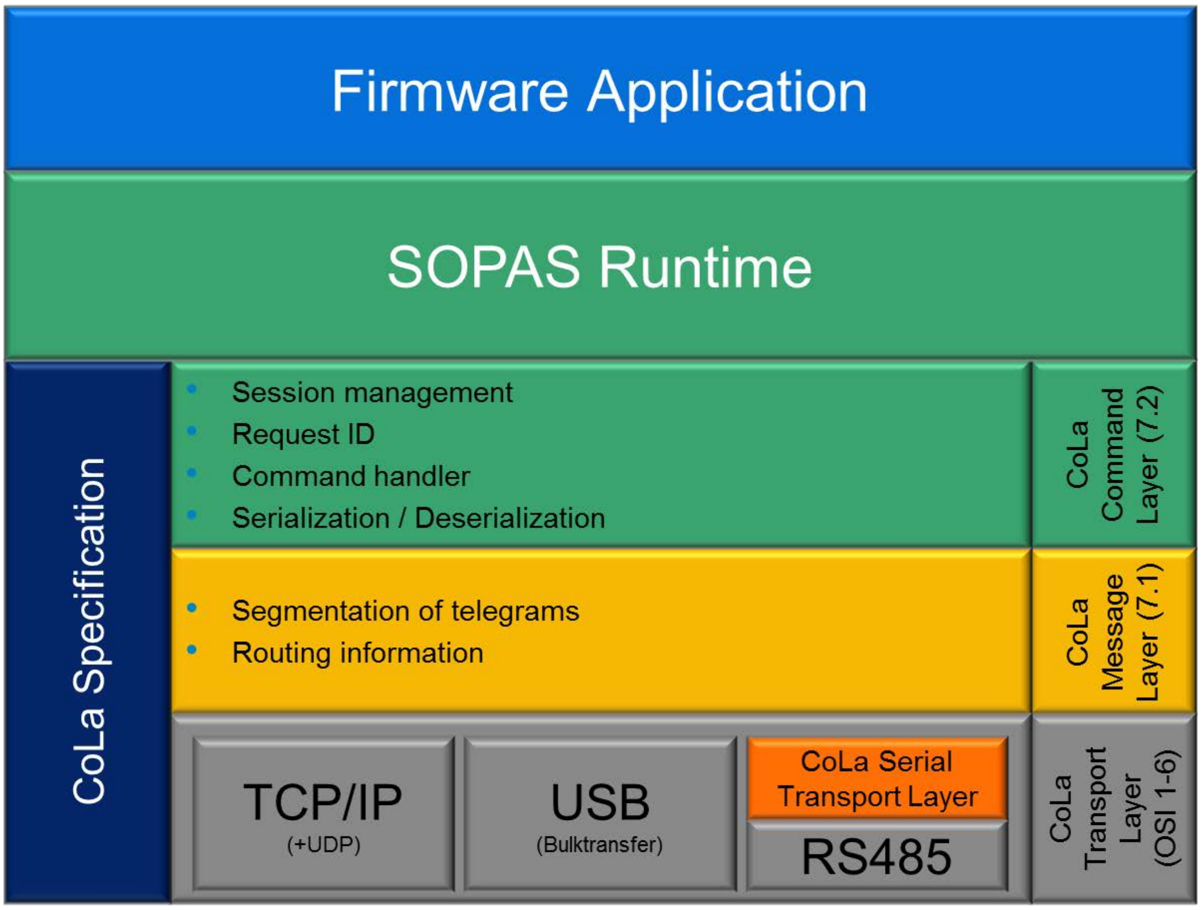
Since CoLa 2.x introduces 2 different rules of serialization sometimes defined byte by byte. Instead the name of the value is enclosed in round parenthesis beginning with the definition of the SOPAS type.

Example: (Uint SVIdx) represents the serialization of the 2 bytes of the value "SVIdx". Or (UDInt XYZ) represents the serialization of the 4 bytes of value of XYZ. If XYZ is 0x12345678, the BigEndian serialization is <0x12><0x34><0x56><0x78> and the LittleEndian serialization is <0x78><0x56><0x34><0x12>.

To signal a blank in a telegram, the blank is replaced by an underscore "_". An underscore in this document is a placeholder for a blank (Chr(32)). In the description of the framing, a blank is represented by the _ character (underscore).

4. Overview

Against the former CoLa specification the specification of CoLa 2.x includes all layers from SOPAS Runtime down to specifying the usage of well-known standards like TCP/IP or a RS485. The following figure shows an overview of layers of the given or planned connection possibilities.



The CoLa Transport Layer is responsible to transport a stream of bytes (8 bit) from one communication partner to another. It was not part of the CoLa 1.0 specification.

The CoLa Message Layer is responsible to intersect the stream of bytes into telegrams and if needed route the stream of one telegram to a certain port within the device or to another device. This layer corresponds to the binary framing specification in conjunction if the SOPAS-Hub specification of the CoLa 1.0 specification.

The CoLa Command Layer specifies the commands with serialization of data (former focus of CoLa 1.0) extended by an explicit session management a Request ID to re-assign responses to the correct request. As a matter of optimization the serialization was extended by defining a second order of bytes: Beside the former big Endian byte order, the little Endian byte order may be used in certain devices.

Although not shown in the figure above, an enhanced version of the SOPAS AutoIP protocol belongs to the CoLa specification as a mandatory feature since CoLa 2.0. This feature is described in the chapter CoLa Scan.

4.1. Total Telegram Format

As a short introduction and for having an overview (the “big picture”) the following picture shows the content of a CoLa 2.x telegram of both CoLa specific layers «CoLa Message Layer 7.1» & «CoLa Command Layer 7.2».

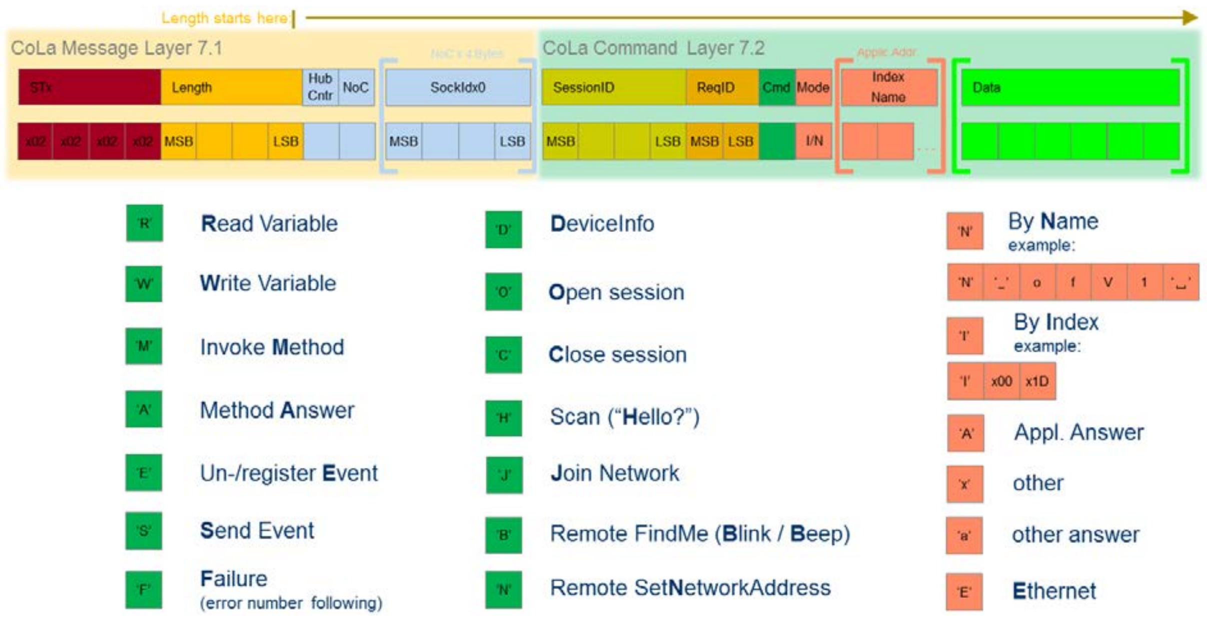


Illustration: CoLa 2.0 Standard Telegram

STx, Length and the CoLa Hub addressing fields (light blue, =HubCnt+NoC+SockIdx[n]) belong to the CoLa Message Layer.

SessionID, ReqID, Cmd, Mode, Index/Name and "Data" belong to the CoLa Command Layer.

The "Data" field starts at an address which is aligned to 32-Bit, when data and method addressing by index is chosen.

5. CoLa Transport Layers

There are several possible "CoLa Transport Layers". Several preconditions must be met by transport layers that shall be used for transporting CoLa 2.0 telegrams. These preconditions are:

- Axiom 1: **Flow Control**
  - All CoLa devices (servers and clients) can send data at any time.
  - For half duplex connections some kind of flow control (collision detection or collision avoidance) is provided.
  - Sending functions (API) have to block in case of all queues are full.
  - Timing surveillance
  - Packets must arrive in correct order
- Axiom 2: **Data corruption**
  - The content of telegrams is secured by a checksum equivalent to 1 CRC32 per 2kByte.
- Axiom 3: **Loss of data**
  - The transport layer cares for a secure transmission of every frame. (acknowledgement)
  - In case of data corruption or loss of a frame, the frame is retransmitted automatically.

- Axiom 4: **Fragmentation**
  - A CoLa transport layer automatically transmits data in fragments when required
- Axiom 5: **Direct Addressing (e.g. on a RS485 bus)**
  - If needed, the transport layer handles addressing of CoLa devices.
  - A CoLa transport layer defines a maximum size for one such fragment

Because some transport layers (e.g. serial interface) do not meet these requirements, they have to be implemented for those transport layers individually. CoLa 2.0 is purely a layer 7 application level protocol and assumes that proper lower layers are available, which provides, at least, the above described mechanisms.

Implementation detail: Please note that CoLa2 can and will stream telegrams that are not complete (fragmented). The CoLa Command Layer can get fragmented telegrams and is able to handle it correctly. If connections are multiplexed, this could theoretically lead to corrupted data at the destination port (if several clients send to the same port at the same time) therefore the ColaRouter fetches a unique Mutex for a destination PortId. This Mutex must be provided by the sensor developer and must be valid across the complete system (even in a multiprocessor system). Please see the CoLa2 training documents for details.

5.1. TCP/IP

Ports to use:

- 2122: CoLa 2.0 TCP port for CoLa 2.0 (The implementations on both server and client should allow for a different port than this to prevent network/firewall problems)

Connections with high latency:

For handling connections with high latency it is advisable to send CoLa commands (and replies) without waiting for the answers. This is possible because CoLa2 has a request ID field that can be used to associate requests and corresponding replies.

Note: On a TCP transport layer this means that many commands will be sent in one packet, thus avoiding the latency for every single command. The MTU must not be configured below 1492 bytes (standard).

Most CoLa commands are very short (only a few bytes). On connections with high latency this leads to unnecessarily long response times, as every single command will be answered separately and the next command will only be sent as soon as the previous command was answered. Sending commands in blocks will make communication much faster (up to factor 100).

Ethernet stacks have an optimization feature that will pack many small messages together to one bigger packet. This will lead to the effect that ACKs will be delayed. The client is waiting for the ACK before sending the next command. To avoid the collection of telegrams the socket option "TCP_NODELAY" can be enabled. This will result in a much faster communication between client and server.

The client implementations must send CoLa commands "en block" whenever possible.

5.2. RS232, RS422 and RS485 CoLaSerialBus (CSB)

CoLa Serial Bus (CSB) is a protocol below the CoLa Messaging Layer (7.1) and fulfilling all the axioms given for CoLa 2 communication.

The CoLa Serial Bus specification is based on the Modbus RTU specification hence it is possible to use a CoLa CSB device in a RS485 network with Modbus RTU devices. There is a master and several slaves. Each slave has a slave address and the master initiates all communication. The master controls which slave is asked and how often.

Since there are several use cases the strategy when asking which slave is not specified in this specification. Only a certain communication interval time is specified in idle mode due to the energy consumption on the slave side.

Not all features that are described in this document are mandatory. For point to point serial connections (like RS232) some of the features are optional. Optional features will be described as optional in the respective chapters.

5.2.1. CRC16 - Cyclic Redundancy Check

The CRC16 in CoLa Serial Bus is implemented using pre calculated polynomial lookup tables.

The tables that are used are:

```
//High Order Byte Table (Table of CRC values for high-order byte)
static const unsigned char ausiCrcHi[256] = {
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40,
    0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40, 0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41,
    0x00, 0xC1, 0x81, 0x40, 0x01, 0xC0, 0x80, 0x41, 0x01, 0xC0, 0x80, 0x41, 0x00, 0xC1, 0x81, 0x40
};
//Low Order Byte Table (Table of CRC values for low-order byte)
static const unsigned char ausiCrcLo[256] = {
    0x00, 0xC0, 0xC1, 0x01, 0xC3, 0x03, 0x02, 0xC2, 0xC6, 0x06, 0x07, 0xC7, 0x05, 0xC5, 0xC4, 0x04,
    0xCC, 0x0C, 0x0D, 0xCD, 0x0F, 0xCF, 0xCE, 0x0E, 0x0A, 0xCA, 0xCB, 0x0B, 0xC9, 0x09, 0x08, 0xC8,
    0xD8, 0x18, 0x19, 0xD9, 0x1B, 0xDB, 0xDA, 0x1A, 0x1E, 0xDE, 0xDF, 0x1F, 0xDD, 0x1D, 0x1C, 0xDC,
    0x14, 0xD4, 0xD5, 0x15, 0xD7, 0x17, 0x16, 0xD6, 0xD2, 0x12, 0x13, 0xD3, 0x11, 0xD1, 0xD0, 0x10,
    0xF0, 0x30, 0x31, 0xF1, 0x33, 0xF3, 0xF2, 0x32, 0x36, 0xF6, 0xF7, 0x37, 0xF5, 0x35, 0x34, 0xF4,
    0x3C, 0xFC, 0xFD, 0x3D, 0xFF, 0x3F, 0x3E, 0xFE, 0xFA, 0x3A, 0x3B, 0xFB, 0x39, 0xF9, 0xF8, 0x38,
    0x28, 0xE8, 0xE9, 0x29, 0xEB, 0x2B, 0x2A, 0xEA, 0xEE, 0x2E, 0x2F, 0xEF, 0x2D, 0xED, 0xEC, 0x2C,
    0xE4, 0x24, 0x25, 0xE5, 0x27, 0xE7, 0xE6, 0x26, 0x22, 0xE2, 0xE3, 0x23, 0xE1, 0x21, 0x20, 0xE0,
    0xA0, 0x60, 0x61, 0xA1, 0x63, 0xA3, 0xA2, 0x62, 0x66, 0xA6, 0xA7, 0x67, 0xA5, 0x65, 0x64, 0xA4,
    0x6C, 0xAC, 0xAD, 0x6D, 0xAF, 0x6F, 0x6E, 0xAE, 0xAA, 0x6A, 0x6B, 0xAB, 0x69, 0xA9, 0xA8, 0x68,
    0x78, 0xB8, 0xB9, 0x79, 0xBB, 0x7B, 0x7A, 0xBA, 0xBE, 0x7E, 0x7F, 0xBF, 0x7D, 0xBD, 0xBC, 0x7C,
    0xB4, 0x74, 0x75, 0xB5, 0x77, 0xB7, 0xB6, 0x76, 0x72, 0xB2, 0xB3, 0x73, 0xB1, 0x71, 0x70, 0xB0,
   0x50, 0x90, 0x91, 0x51, 0x93, 0x53, 0x52, 0x92, 0x96, 0x56, 0x57, 0x97, 0x55, 0x95, 0x94, 0x54,
   0x9C, 0x5C, 0x5D, 0x9D, 0x5F, 0x9F, 0x9E, 0x5E, 0x5A, 0x9A, 0x9B, 0x5B, 0x99, 0x59, 0x58, 0x98,
   0x88, 0x48, 0x49, 0x89, 0x4B, 0x8B, 0x8A, 0x4A, 0x4E, 0x8E, 0x8F, 0x4F, 0x8D, 0x4D, 0x4C, 0x8C,
   0x44, 0x84, 0x85, 0x45, 0x87, 0x47, 0x46, 0x86, 0x82, 0x42, 0x43, 0x83, 0x41, 0x81, 0x80, 0x40
};
```

All bytes of a received telegram are used for the calculation, including the header bytes.

```
UInt calcCRC16(USInt *pusiBytes, UInt uiNumberOfBytes)
{
  USInt usiCrcHi = 0xff;  // init high order CRC byte
  USInt usiCrcLo = 0xff;  // init low order CRC byte
  USInt usiIndex;         // index für Tabellen Lookup

  while (uiNumberOfBytes--)
  {
      usiIndex   = usiCrcLo ^ *pusiBytes++ ;
      usiCrcLo = usiCrcHi ^ ausiCrcHi[usiIndex] ;
      usiCrcHi = ausiCrcLo[usiIndex] ;
  }

  return ((((unsigned short)usiCrcHi) << 8) | usiCrcLo) ;
}
```

5.2.2. Baud Rates

There are 4 different baud rates defined for CoLa 2.1 Serial Bus (CSB): One mandatory baud rate (19k2) and three optional and faster baud rates. The baud rates are listed in the table below.

Baud rate	Time of a byte	Baud Rate : 115k2	Baud Rate : 19k2	Mandatory
115k2	86,806 µs	1:1	6:1	No
57k6	173,611 µs	1:2	3:1	No
38k4	260,417 µs	1:3	2:1	No
19k2	**520,833 µs**	**1:6**	**1:1**	**Yes**

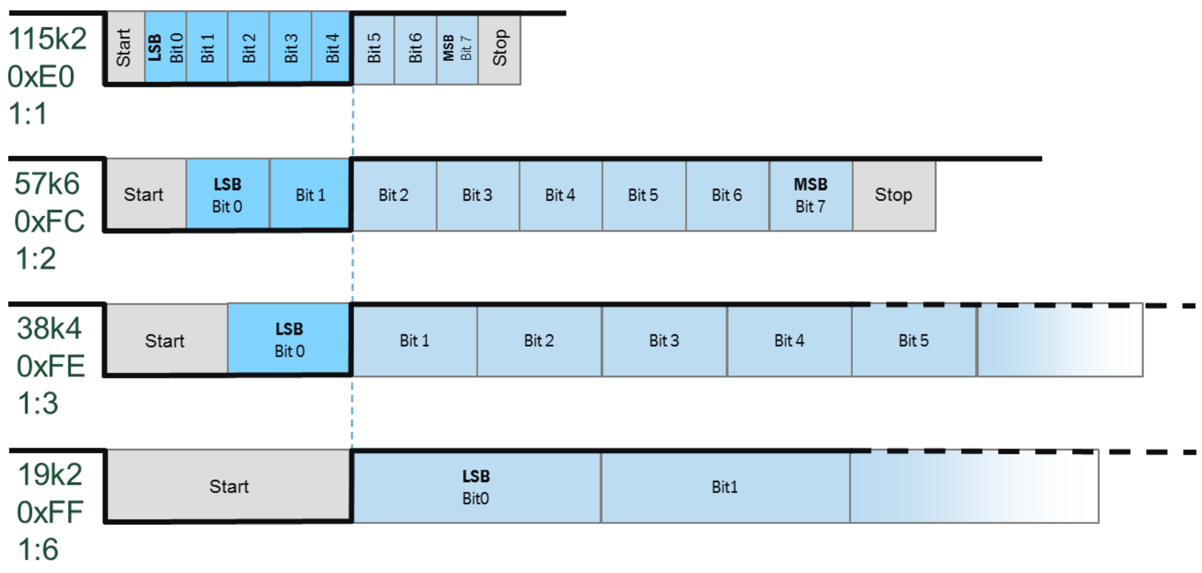
The baud rates have been selected carefully that certain bytes in one baud rate are compliant to another byte in other baud rates.

When sending a 0xFF at the baud rate of 19k2 (or 0xE0 at 115k2) it can be detected in the other baud rates as shown in the table below:

The byte 0xE0 @ baud rate	=compliant @ 115k2	=compliant @ 57k6	=compliant @ 38k4	=compliant @ 19k2
115k2	**0xE0**	**0xFC**	**0xFE**	**0xFF**
57k6	n/c	0xE0	0xF8	0xFE
38k4	n/c	n/c	0xE0	0xFC
19k2	n/c	n/c	n/c	0xE0

The important thing is that sending a 0xFF at a baud rate of 19k2 can be understood at all other baud rates. (Someone at 57k6 "hears" a 0xFC, someone at 38k4 "hears" a 0xFE, ...)

The figure below shows the relative timings of the byte 0xE0 to explain these relations.

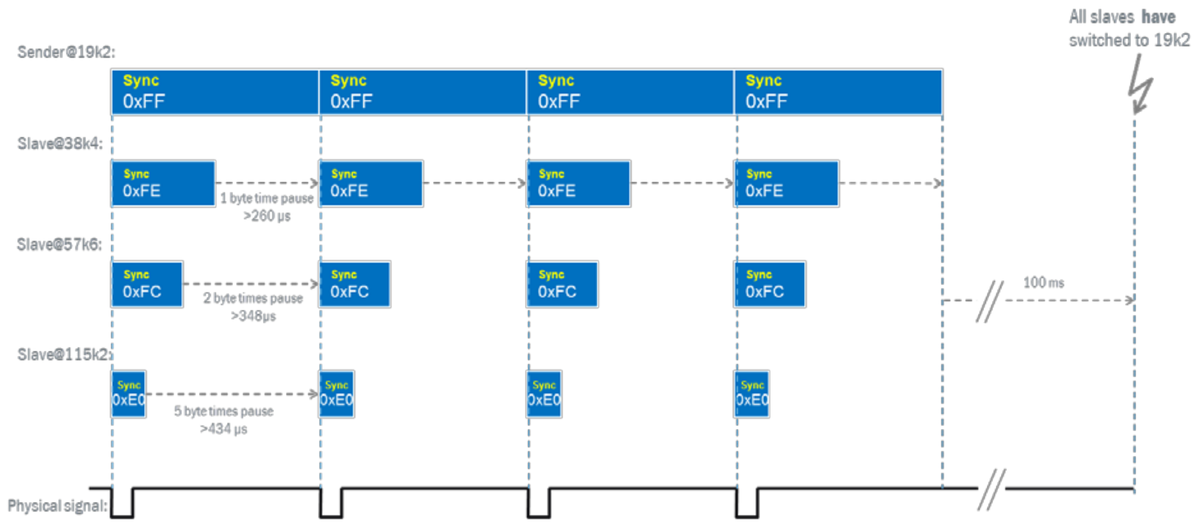


Not all devices have to support all baud rates but the baud rate 19k2 is mandatory for all slaves. The master should support all 4 baud rates.

Synchronize Baud Rate

When not knowing the baud rate, the master can send 0xFF @ 19k2 which is a valid character in all other baud rates. For robustness reasons, the master sends this byte four times and then waits for 100ms. During these 100ms every slave has to switch temporarily to 19k2 which must be supported by all slaves.

The following figure shows the timing of a synchronization procedure to 19k2:



The synchronizing master sends 4 times a 0xFF at the baud rate of 19k2. The physical signal of this sequence is shown at the bottom of the figure above.

A slave configured to 38k4 receives 4 times a 0xFE with a timing gap of about 1 byte in between. The 4 bytes will be received as one packet (gap is below inter byte timeout of 3,5 bytes)

A slave configured to 57k6 receives 4 times a 0xFC with a timing gap of about 2 bytes in between. The 4 bytes will be received as one packet (gap is below inter byte timeout of 3,5 bytes)

A slave configured to 115k2 receives 4 times a 0xE0 with a timing gap of about 5 bytes in between.
Attention: The reception handler will most times treat these 4 bytes as 4 packets since the inter byte timeout of 3,5 bytes is exceeded. As the inter byte timeout can be adjusted by a slave, it is also possible that the 4 bytes arrive in one packet. Implementations must allow for this.

After the reception of this sequence (dependent on the own baud rate) the slaves have 100 ms time to switch to the baud rate of 19k2. This switch shall be volatile which means that resetting the device will return the device to the previously configured baud rate.

Now, the next step of the master will be scanning the slaves (see chapter “Scan”, “H” command) with the baud rate of 19K2. After the master successfully found all slaves, it will determine an appropriate baud rate for the serial bus and switch all slaves to that baud rate (see “B” command).

5.2.3. Protocol Definition

The protocol is completely compliant to Modbus RTU. It uses the possibility of Modbus to define own (manufacturer specific) functions (commands). The manufacturer specific command ‘C’ (0x43) has been chosen based on the C of “CoLa”.

Packet discrimination

The discrimination of packets is done using the time between bytes. The time between two bytes within a packet shall be minimized as possible to 0 µs but must not extend the time of 1 byte. After the transmission of a telegram a pausing time of minimum 3,5 times of a byte have to be waited before sending the next byte where this next byte is typically sent by the other party.

The following table shows the timings of bytes, allowed gaps between two bytes and the pausing time after a packet:

Baudrate	Time of a byte	Tolerable time between two bytes within a packet	Idle time after a packet = 3,5 times the time of a byte
115k2	**86,806 µs**	86,8 µs	303,8 µs
57k6	**173,611 µs**	173,6 µs	607,6 µs
38k4	**260,417 µs**	260,4 µs	911,5 µs
19k2	**520,833 µs**	520,8 µs	1,8229 ms

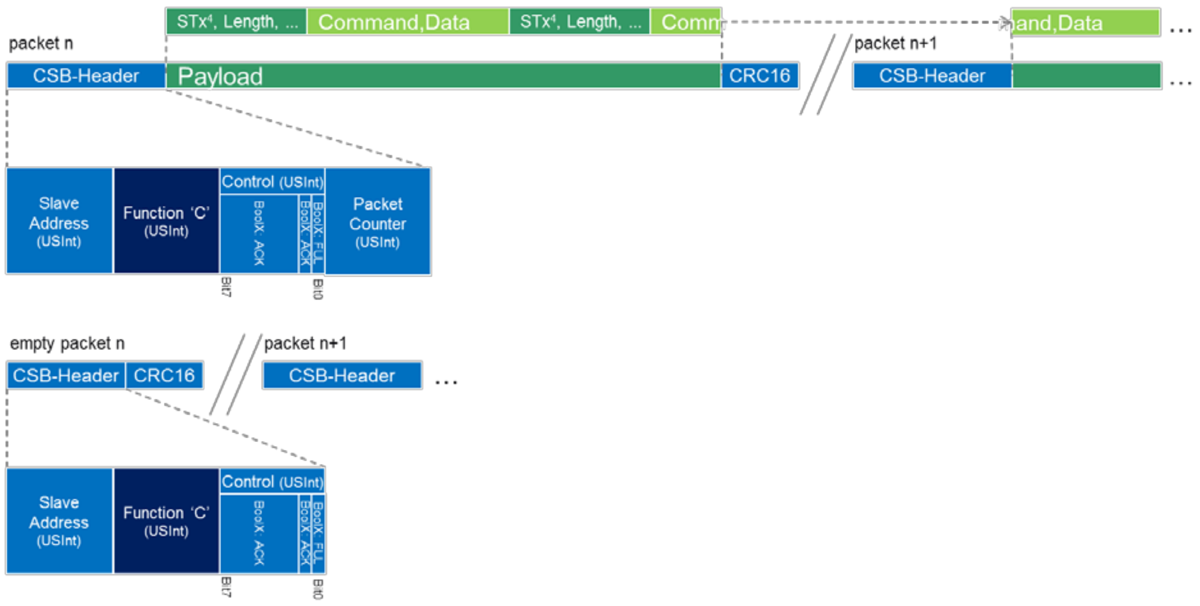
Packet Structure

As mentioned before the protocol is compliant to Modbus RTU and so is the packet structure. There are different kinds of addressing and different functions defined for CSB:

- Direct Unicast Packets
  - Exchange for CoLa 2.1 message data — Function C
- Indirect Unicast Packets
  - FindMe — Function F
  - SetAddress — Function A
- Broadcast
  - Scan (1st time) — Function H
  - Scan (follow up) — Function h
  - SetBaudRate — Function B

Direct Unicast Packets

The following figure shows the structure of direct unicast packets with (upper chart) and without (lower chart) payload. A direct unicast packet is being sent from the master to ONE slave or from ONE slave to the master.



The CSB information is shown in different shapes of blue and the payload is shown in different in green. An example of the payload is shown at the top of the figure in different shapes of green where the first packet (packet n) contains a complete CoLa 2 message and the beginning of a message which is continued in the second packet (packet n+1).

The following table shows and describes the fields of the directed packet structure in detail.

Field	Byte position	Bit Position	Description
Slave Address	0		The slave address addresses one slave to which the packet is sent or the address of the sending slave (when sent to the master)
Function **'C' = 67**	1		Manufacturer specific Modbus function 67
ACK	2	0	This flag signals that the last packet was received with a correct checksum (CRC16). If this flag is cleared (=NACK) it is signaled that the last packet has to be repeated. When sending the first packet, the ACK flag is set.
FUL	2	1	This flag signals that the counter part is currently not able to receive payload in packets (due to full buffers). If this flag is set, the next packet must not contain any payload.
Reserved	2	2…7	All bits are cleared (0)
Packet Counter (optional)	3		The packet counter is increased with every new packet. When 0xFF is reached the counter starts from 0 again. It helps to determine at the receiver's site whether a packet was repeated or is a new packet. If no payload is given, the Packet Counter is not transmitted.
Length	4,5		The complete length of this packet, including header and CRC16
Payload (optional)	6… 3+LOP		The payload may be a fragment of a message and must be concatenated with payload(s) of the previous/ successive packet(s).
CRC16	Hi: 6+LOP Lo: 7+LOP		The CRC16 checksum over the whole packet (including and beginning from the slave address) excluding the CRC16 itself. See Modbus specification at http://www.modbus.org

The length field is redundant for the sensor implementation, because typically sensors can use the specified inter byte timeout for receiving. On PC based clients with (no real time) operating systems, this may and will cause problems. Packets will not be received completely, except when very huge inter byte timeouts are used, which leads to slow communication.

To mitigate this effect, the length field for the 'C' command can be used. Clients can receive until they have the amount of bytes indicated by the length field. The broadcast commands are very short and do not need a length field, they always can be read with the inter byte timeout mechanism without problems, even on Microsoft Windows.

Indirect Unicast Packets

There are some packets (commands) which are necessary to transmit before all slaves have assigned valid slave addresses. For this purpose the indirect unicast packets are designed. The indirect unicast packets are logically sent as broadcast to all slaves but the function code is followed by a unique identifier if the device.

Unique Device Identifier (UDI)

The Unique Device Identifier is composed of the vendor ID, the device name (family name) and the unique serial number of the device (within the family). The following table shows the content of the UDI:

Field	Type	Byte position (where fix)	Description
Vendor ID	USInt	Next	The ID of the vendor[2]. **`<0><1>`** (=0x0001) for all devices of the SICK AG.
Device Name	FlexString20	Next	The name of the device (family). Must be unique within the vendor's portfolio.
Serial Number	FlexString20	Next	The serial number (as assigned by the production line) Must be unique within the device family.

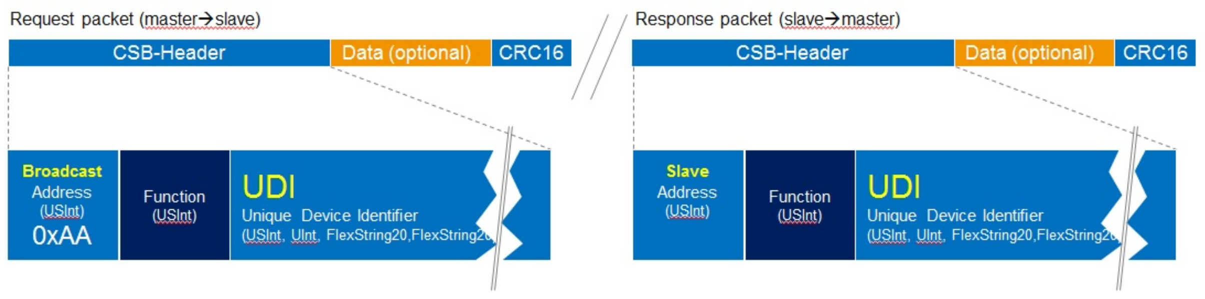
Structure of an indirect unicast package

Indirect Unicast Packets are sent to the broadcast address. The function code is immediately followed by the Unique Device Identifier. Every slave device has to receive and compare the packet until it can decide that the UDI is not its own UDI. The UDI is also enclosed in the slave's response where the broadcast address is replaced by the slave's current address.

Remember: Due to address conflicts it is possible that a slave is addressed directly by another slave's response (with the same address). Hence a function code for indirect unicast package in conjunction with a normal address must be interpreted as a packet to the master.

Due to that indirect unicast packages are used as single commands, the packets are also treated and named as requests and responses.

The following figure shows the structure of an indirect unicast package (generalized).



[2] Currently only SICK AG is defined

FindMe()

The FindMe packet is a command to let the slave device flash, beep or do something similar to indicate the physical device to the user which is currently addressed (by the UDI, the Unique Device Identifier).

The FindMe feature is optional for point to point serial connections, where the endpoints are known.

The following table shows the packet in detail.

Field	Request	Response	Description
Address	0xAA	Slave Address	
Function	'F'	No response	Manufacturer specific Modbus function 70 This function must not be treated as a request when address is not the broadcast address.
UDI	slave's UDI	No response	See specification of UDI above
Data	Ø	Ø	
CRC16	UInt (Hi-Byte first)	No response)	The CRC16 checksum over the whole packet (including and beginning from the slave address) excluding the CRC16 itself. See Modbus specification at http://www.modbus.org

Initialize()

The Initialize packet is a command to let the slave device clear its CSB send buffers.

If a previously issued command tried to read a huge variable and the connection is reset, while the data was not yet transferred, the data will still be in the buffers. The client has a means to clear these buffers by issuing the Initialize command.

Field	Request	Response	Description
Address	0xAA	Slave Address	
Function	'I'	No response	Manufacturer specific Modbus function 70 This function must not be treated as a request when address is not the broadcast address.
UDI	slave's UDI	No response	See specification of UDI above
Data	Ø	Ø	
CRC16	UInt (Hi-Byte first)	No responce	The CRC16 checksum over the whole packet (including and beginning from the slave address) excluding the CRC16 itself. See Modbus specification at http://www.modbus.org

SetNewAddress()

The SetNewAddress packet is a command to temporarily change the slave address to a new value. The packet must not be accepted after a payload has been received by a slave. Hence a master has to set the new address as a first action (after searching the device / scan) after the device reset. Furthermore the new slave address is stored volatile which leads to the 'old' slave address after a power reset of the slave device.

The SetNewAddress feature is optional for point to point serial connections, where the endpoints are known.

The following table shows the packet in detail.

Field	Request	Response	Description
Address	0xAA	Slave Address	
Function	'A'	'A'	Manufacturer specific Modbus function 65 This function must not be treated as a request when address is not the broadcast address.
UDI	slave's UDI	slave's UDI	See specification of UDI above
Data	USInt newAddress	USInt currentAddress (same as new Address on success)	The new (desired) slave address is transmitted in the requesting packet. The slave answers with the same packet (new address) in case of success. If the slave received a valid payload before or the desired new address is invalid, the device sends the 'old' as current address as an error feedback.
CRC16	UInt (Hi-Byte first)	UInt (Hi-Byte first)	The CRC16 checksum over the whole packet (including and beginning from the slave address) excluding the CRC16 itself. See Modbus specification at http://www.modbus.org

Temporarily setting the slave address is used to connect the device. Layer 7 functions shall be used to finally set the slave address nonvolatile.

Broadcast Packets

There are two broadcast commands which are sent by master to all slaves. One is called "**H**ello" and used for scanning the bus or the so far unknown communication interface. The other one is called "Set**B**audrate" and is used to switch the baud rate of all slaves.

Hello Packet

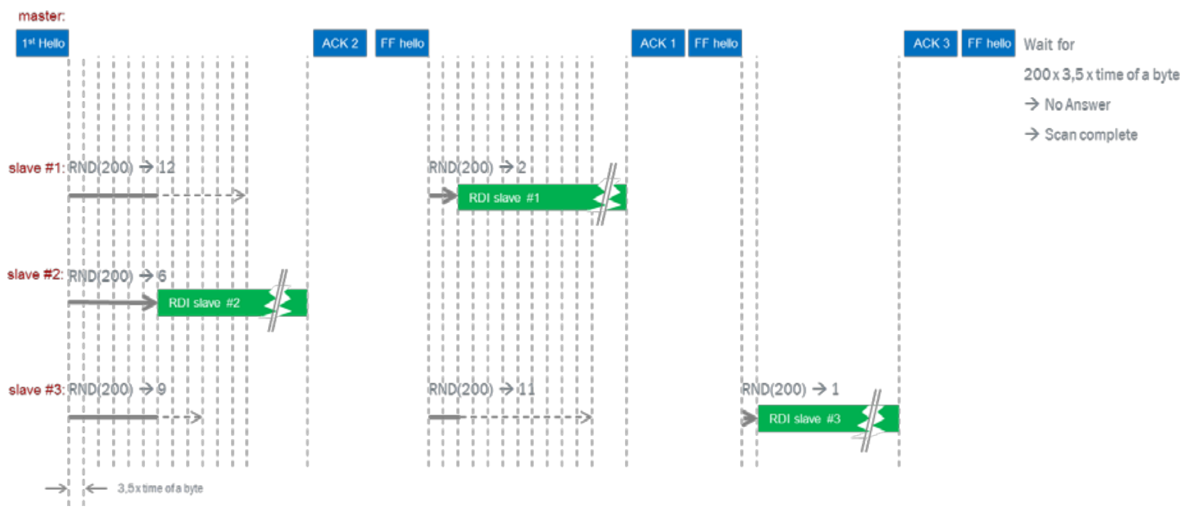
The sequence below describes the procedure of scanning a CoLa Serial Bus with the broadcast packet named "Hello":

1. Master sends a "1st Hello" packet (Function 'H'). If the packet is a follow up Hello packet (find next slave), the master sends the hello packet with the function value 'h' ("Follow up (FF) Hello").
2. Each slave generates an **integer random** value between 1 and 200.
3. This random value is multiplied with 3,5 times of the time of a byte (baud rate dependent). The resulting value is named **answering slot** and has to be waited before sending the answer.
4. During the answering slot the slave listens to the bus whether other slaves generated a smaller random value answering earlier. If a byte is received (from another slave), the slave aborts its intention to answer and waits for another packet from the master.
5. If no other slave answered during the waiting time, the slave begins to send its answer. The answer is a special packet called "Reduced Device Info".
6. If two slaves have answered (same random value, same answering slot) the CRC will most probably be wrong (since the answers will be different) which leads to that the master will repeat the procedure from the top.
7. If the CRC was correct the mast will send an ACK (direct unicast packet ACK and without payload) to the address which was given in the answer from the slave.
8. If no answer was received during 200 x 3,5 x time of a byte the master assumes that no more slaves are present.

The slaves remember whether they have answered once and stop reacting on the "Follow Up Hello" (FF hello → 'h') packet. A "1st Hello" packet resets this memory of all slave leading to a "new scan".

The following figure shows a timing chart of one master and 3 slaves which are scanned with the Hello packet.

Due to graphical convenience the generated random values (RND(200)) are limited to small values here.



It is important to distinguish between the first Hello (Function 'H') which resets the state of remembering of all slaves whether they have answered already or not. Further the slaves reset their statet I-have-answered state when not receiving any Hello packet for more than 10 seconds.

Time a slave remembers that it has already answered a "Hello" packet
10 seconds since the last Hello packet was received

The following table shows the baud rate dependent timings:

Baudrate	Time of a Byte =10/BaudRate	Multiplier for RND() = TimeOfByte*3,5	Max wait time = 200*MultiplierForRND
115k2	**86,806 µs**	304 µs	60 ms
57k6	**173,611 µs**	600 µs	120 ms
38k4	**260,417 µs**	911 µs	182 ms
19k2	**520,833 µs**	1,83 ms	366 ms

The scan needs the following packet types which are defined in detail below

- First Hello packet
- Follow Up Hello packet
- Reduced Device Information
- Acknowledge

The following table defines the **First Hello packet**.

Field	Byte position	Bit Position	Description
Slave Address **= 0xAA**	0		The slave address addresses all slave (broad cast)
Function **'H' = 72**	1		Manufacturer specific Modbus function 72
CRC16	Hi: 2 Lo: 3		The CRC16 checksum over the whole packet (including and beginning from the slave address) excluding the CRC16 itself. See Modbus specification at http://www.modbus.org

The following table defines the **Follow Up Hello packet**.

Field	Byte position	Bit Position	Description
Slave Address **= 0xAA**	0		The slave address addresses all slave (broad cast)
Function **'h' = 104**	1		Manufacturer specific Modbus function 104
CRC16	Hi: 2 Lo: 3		The CRC16 checksum over the whole packet (including and beginning from the slave address) excluding the CRC16 itself. See Modbus specification at http://www.modbus.org

The following table defines the **Reduced Device Information packet**.

Field	Type	Byte position (where fix)	Description
Slave Address	USInt	0	The slave address informs about the address of the sending slave
Function **'D' = 68**	Char	1	Manufacturer specific Modbus function 68
UDI **Unique Device Identifier**	USInt FlexString20 FlexString20	Next	See UDI specification of indirect unicast packets
MTU-Size	UInt	Next	Max number of bytes (payload) of a packet (slave attribute)
Energy saving **Poll Interval**	UInt	Next	Slave tells master if polling shall be paused when possible. 0 means: poll as fast as possible.
Supported Baud Rates	SCont	Next	BoolX Bit0 : 115200 supported BoolX Bit1 : 57600 supported BoolX Bit2 : 38400 supported BoolX Bit3 : 19200 supported UInt4 Bit4…Bit7 : reserved (must be 0)
CRC16	Hi: 2 Lo: 3		The CRC16 checksum over the whole packet (including and beginning from the slave address) excluding the CRC16 itself. See Modbus specification at http://www.modbus.org

The Reduced Device Info is acknowledged by the master (in case of correct CRC) with an empty direct unicast package to the sending slave.

Remember that several slaves may share the slave address at this moment. Hence the slave may only suppose that it has successfully transmitted its Reduced Device Info when it receives the acknowledge **and** has actually sent the Reduced Device Info recently.

Since the Reduced Device Information contains unique information, it can be excluded that the CRC16 is correct when two devices send its Reduced Device Info in the same slot.

Switch Baud Rate

The Function code 'B' is used to switch the baud rate. It may be used in conjunction with direct unicast addressing but it makes no sense. The following table defines the packet for switching the baud rate (and for the later defined objection).

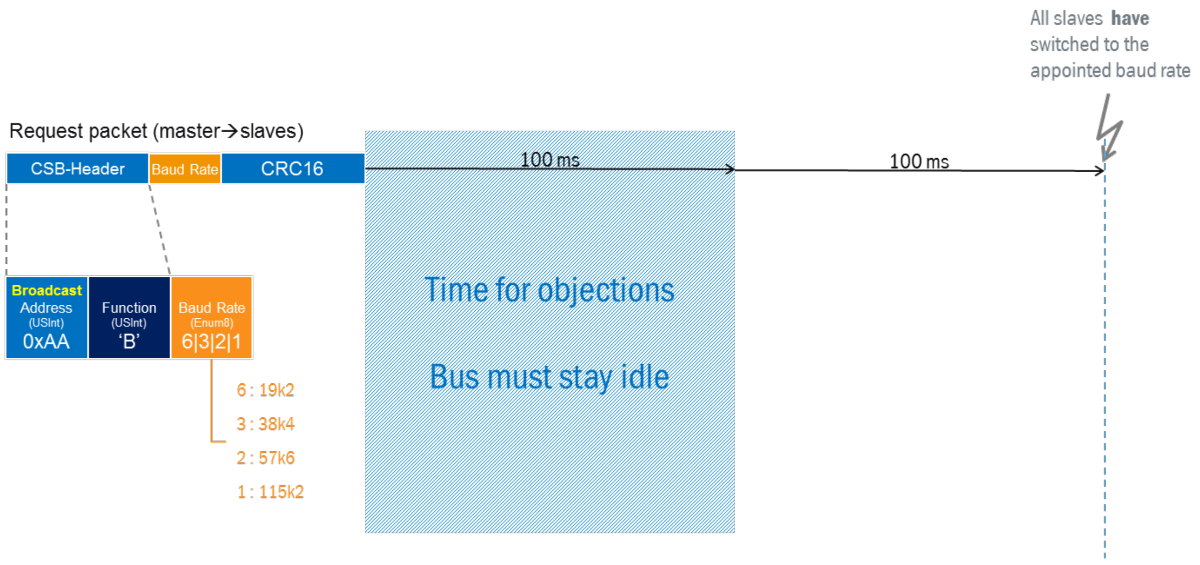
Field	Type	Byte position (where fix)	Description
Slave Address	USInt	0	Usually all slaves are addressed using the broad cast address 0xAA.
Function **'B' = 66**	Char	1	Manufacturer specific Modbus function 66
Baud Rate	Enum8	2	The master sends the baud rate change request to all slaves. If a slave does not support this baud rate, it responds with this packet placing the most likely baud rate at this position. Enum8 values are: 1 : 115k2 2 : 57k6 3 : 38k4 6 : 19k2
CRC16	Hi: Lo:	3 4	The CRC16 checksum over the whole packet (including and beginning from the slave address) excluding the CRC16 itself. See Modbus specification at http://www.modbus.org

After this packet has been sent/received, the bus shall stay idle for 100 ms. These idle time is named time-for-objections and may be used by any slave which cannot switch to the new baud rate to send an objection packet. (see below)

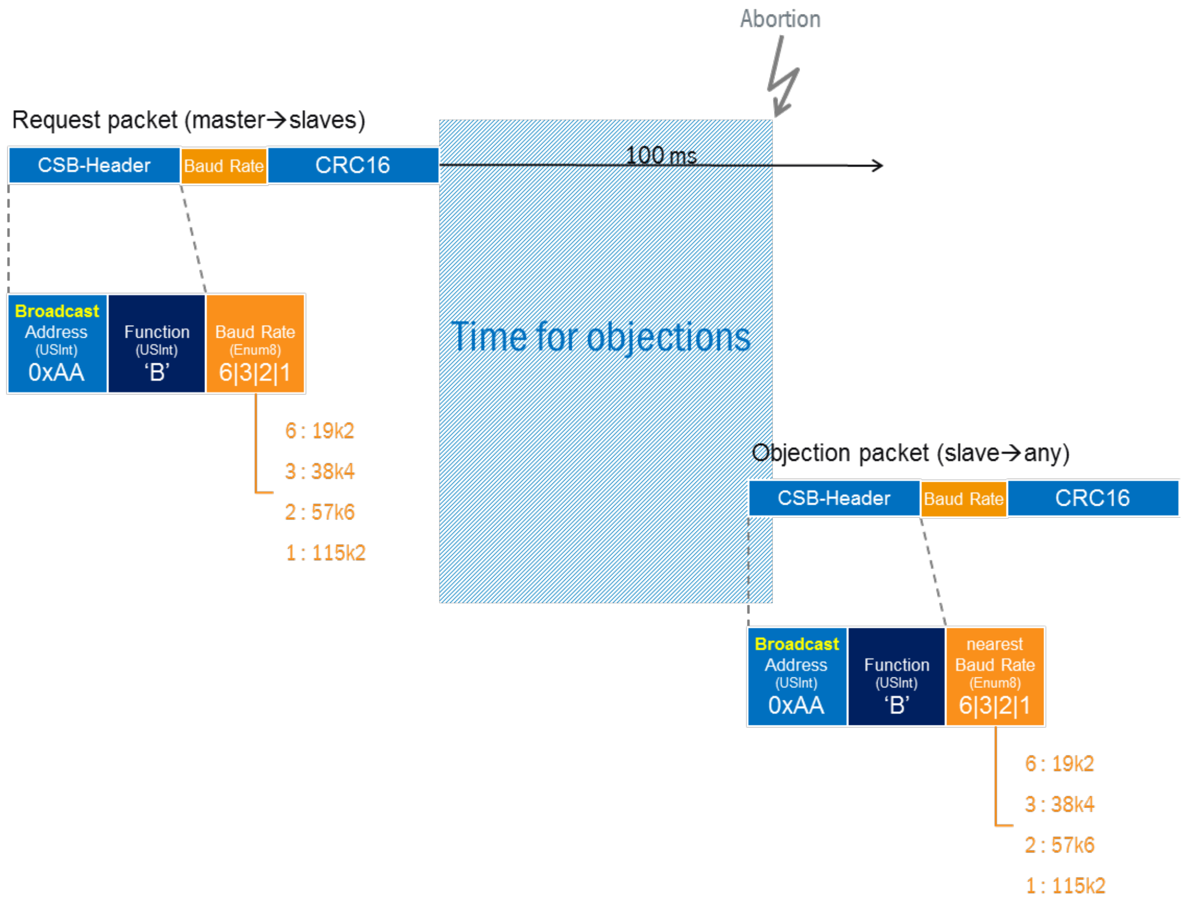
When the time for objections has passed, all slaves have a time of another 100 ms to switch to the new baud rate temporarily. This means that resetting the device will let the device return to the previously configured baud rate.

Permanently changing the baud rate (storing in nonvolatile memory) is meant to be done on layer 7 protocol regarding security constraints.

The following figure shows the timing diagram of a successful change of the baud rate of all connected slaves (without any objections).


All slaves have switched to the appointed baud rate
Request packet (master→slaves)
CSB-Header
Baud Rate
CRC16
100 ms
100 ms
Time for objections
Bus must stay idle
Broadcast Address (USInt) 0xAA
Function (USInt) 'B'
Baud Rate (Enum8) 6|3|2|1
6 : 19k2
3 : 38k4
2 : 57k6
1 : 115k2

The following figure shows the timing diagram of an attempt to change the baud rate of all connected slaves but one slave rejects the desired baud rate (since it is not supported).



When any slave answers anything (there may be several slaves answering at the same time), the procedure must be aborted regardless what has been received. Even framing errors shall lead to an abortion.

An objection shall not be arbitrarily: A baud rate must not be rejected by a slave if it is marked as a supported baud rate in the RDI (reduced device info).

5.2.4. Fullfilment of the CoLa Axioms

The job of the CoLa Serial Bus specification is to fulfill the requirements of the CoLa transport layers (axioms) on a serial half duplex line including RS485, RS422 and RS232.

5.2.5. Layer 1 specification

CSB uses RS485 and RS422 as half-duplex communication carrier and RS232 treating it like a half-duplex communication carrier.

Connectors are not specified.

There are 4 possible baud rates listed in the table below. Other baud rates are not allowed.

Baudrate	Time / byte	Idle time between 2 packets
115k2	**86,8 µs**	303,8 µs
57k6	**173,6 µs**	607,6 µs
38k4	**260,41 µs**	911,5 µs
19k2	**520,8 µs**	1,82 ms

The UART settings are: 8 bit data, no parity, 1 stop bit:

UART Settings :	**8 bit, no parity, 1 stop bit (8N1)**

CSB Flow Control

All communication is initiated by the master. The master sends one packet to a slave which “immediately” sends a packet as a response.

Attention: A packet is not a telegram or message. A packet contains stream of data. This may be one telegram, serveral telegrams where the last and / or the first telegram may be incomplete.

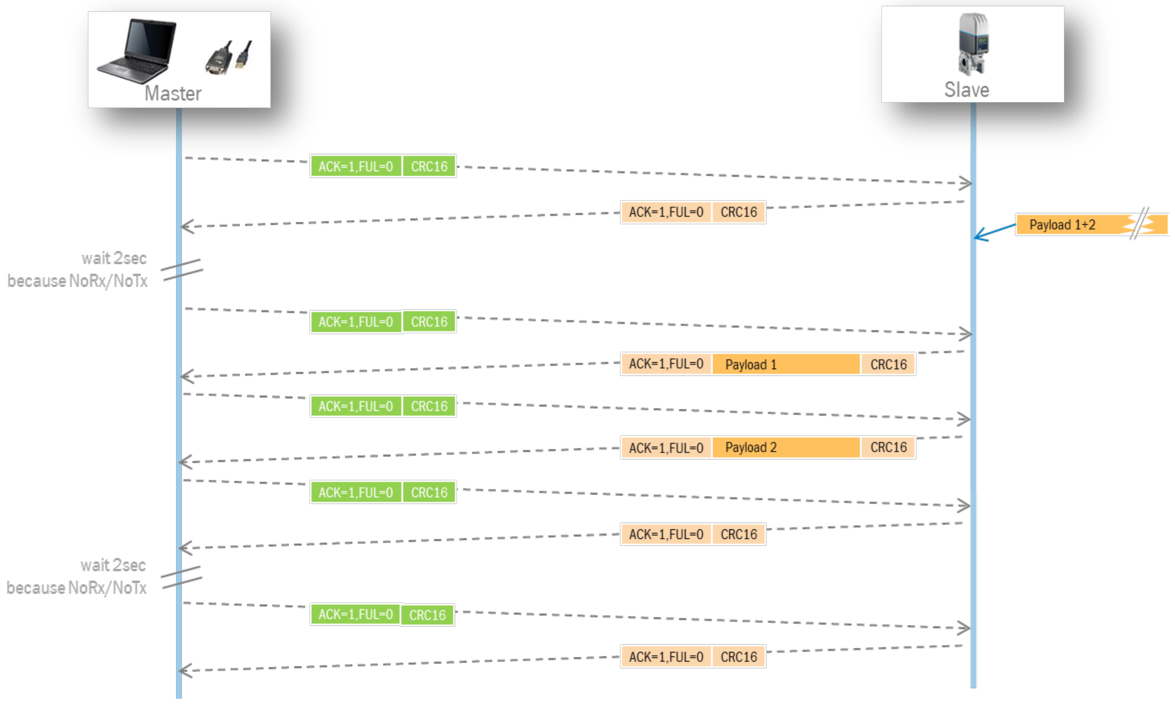
Sending from master to slave

When a master wants to send something, the master packs the stream into one or several packets and sends them one after each other to the slave while after each packet the master waits for the responding packet where the responding packets may contain data or may be empty.

Receiving from slave to master

The slave must not initiate any communication. Hence the master has to send a packet to the slave which allows or forces the slave to respond with a packet. This packet contains data which has to be transmitted to the master.

The following figure shows the transmission of something like an Event from the slave (not directly caused by a request from the master).



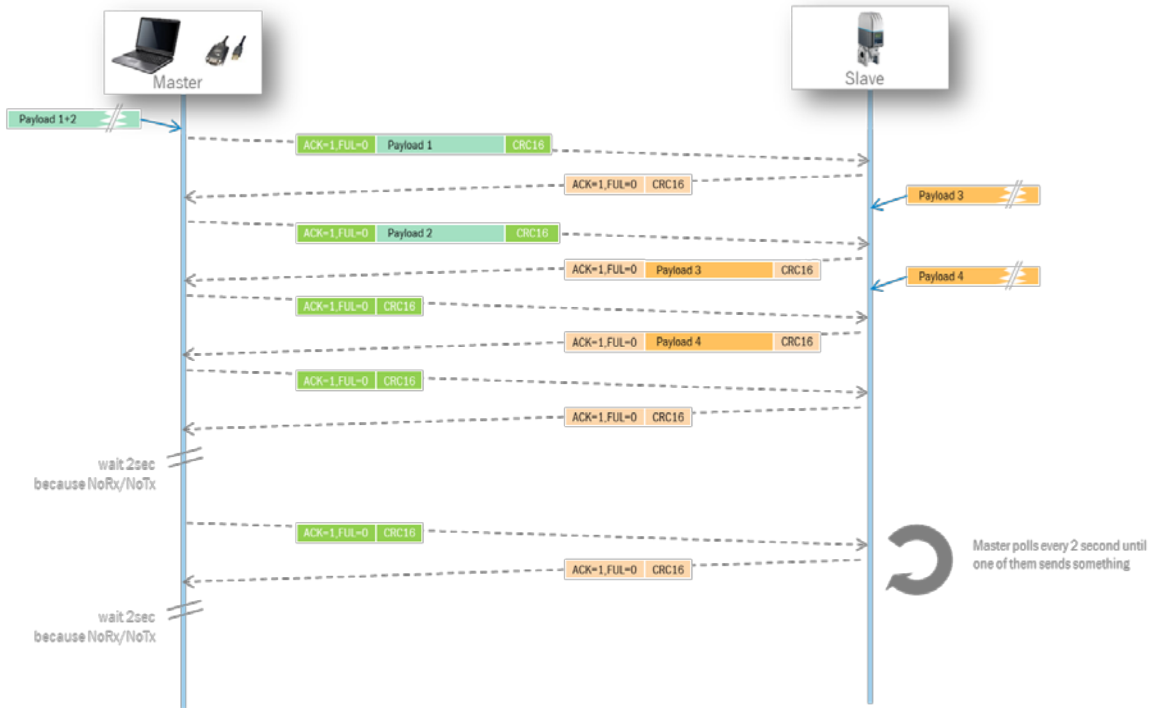
Poll Intervall

When the master believes that the device wants to send some data, it may poll the slave device without any delay. The default strategy for believing whether the device wants to send something is: When the last telegram is empty, the slave device has nothing to send.

The strategy to believe in principle that the slave could want to send some data is not allowed.

When the master wants to send data, it also may send all packets without an additional interval time between two exchanges of packet.

The following figure shows a sequence diagram of a master and a slave with more details (FUL-bit, ACK-bit, CRC) where at the end payload in both directions is empty which leads to that the master waits 2 seconds between every polling packet.

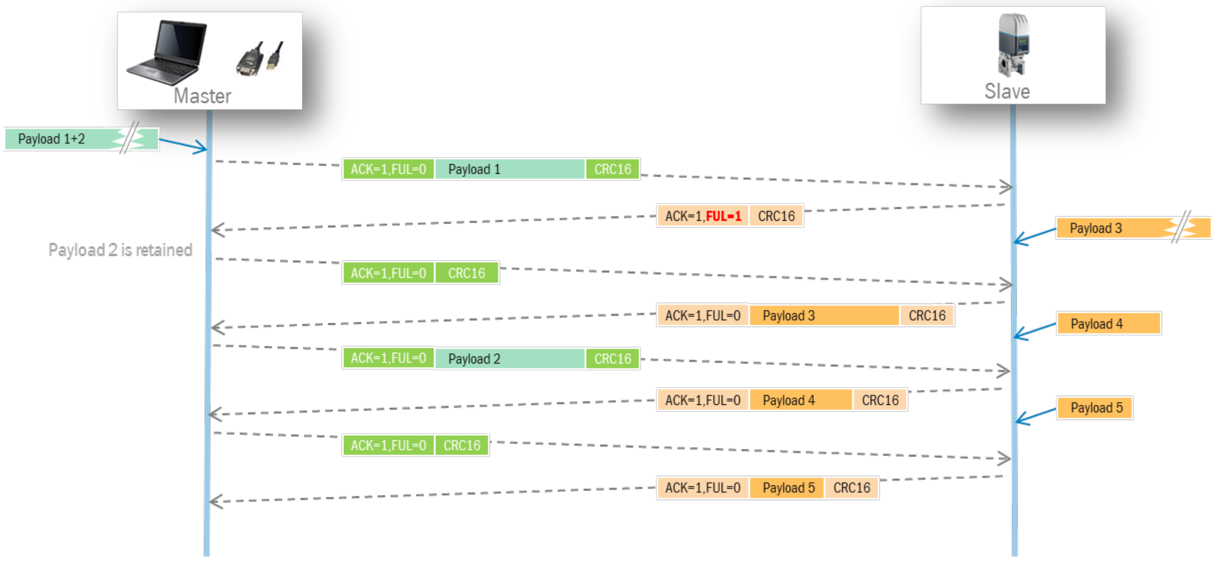

Master
Slave
Payload 1+2
ACK=1,FUL=0
Payload 1
CRC16
Payload 3
Payload 2
Payload 4
wait 2sec because NoRx/NoTx
Master polls every 2 second until one of them sends something

Idle time poll intervall: e.g. 2 seconds

Note: the poll interval can be adjusted by each slave. They tell their preferred polling interval to the master in the UDI. The diagram shows a polling interval of 2 seconds (energy saving).

Buffer control

When the input buffer of a slave is full (or is close to be full) the "FUL" flag in the outgoing packet is set which leads to that the counterpart has to retain the payload until the flag is cleared again. The following figure shows a sequence diagram with the same payload situation but with the circumstance that the input buffer of the slaves is temporarily full (after payload 1 was received).



After sending the payload 1 the master receives a FUL=1 from the slave. Hence the master retains the payload 2. With the next packet exchange (the master sent an empty packet) the master receives a FUL=0 again and can now send the retained payload 2 with its next packet.

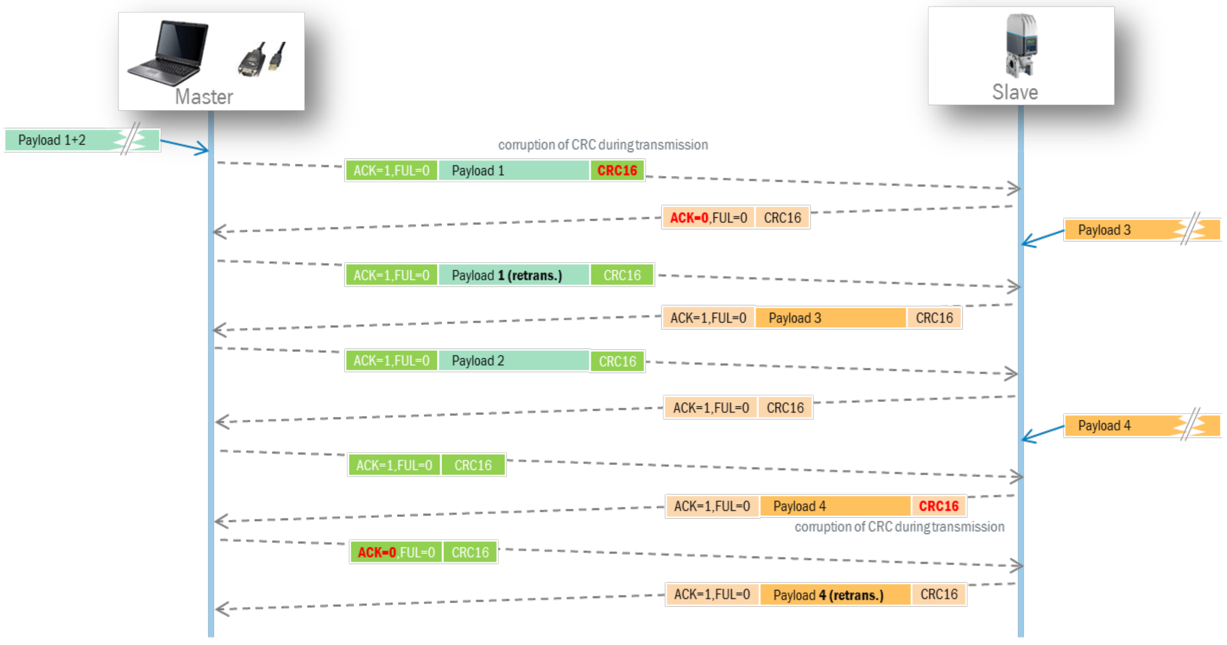
Due to that payload 2 was sent later the diagram also shows that payload 4 is now split into payload 4 and payload 5 due to the later cause.

Data corruption

The data is transmitted in so called packets. A packet is a container containing fragments or multiple telegrams. Each packet has a CRC16 at the end which is built over the whole telegram excluding the CRC16 itself.

When a packet was transmitted correctly (CRC16 matches) the counterpart transmits an ACK with its next packet. In case the packet was corrupted (CRC16 does not match) the counterpart transmits no ACK (called NACK) with its next packet and the same packet will be repeated immediately. Hence a sender may free the buffer of a packet he sent not until the ACK=1 was received.

The following figure shows a sequence diagram with the same payloads as assumed before but with a corruption of the packet during transmission (CRC16 mismatch).


Master
Slave
Payload 1+2
corruption of CRC during transmission
ACK=1,FUL=0 | Payload 1 | CRC16
ACK=0,FUL=0 | CRC16
Payload 3
ACK=1,FUL=0 | Payload 1 (retrans.) | CRC16
ACK=1,FUL=0 | Payload 3 | CRC16
ACK=1,FUL=0 | Payload 2 | CRC16
ACK=1,FUL=0 | CRC16
Payload 4
ACK=1,FUL=0 | CRC16
ACK=1,FUL=0 | Payload 4 | CRC16
corruption of CRC during transmission
ACK=0,FUL=0 | CRC16
ACK=1,FUL=0 | Payload 4 (retrans.) | CRC16

Loss of data

If some bytes of a packet are lost on the transmission line it is assumed that the CRC16 will not match anymore and hence the packet will be repeated.

If total packets from master to slave or from slave to master get lost, the master will miss the responding packet from the slave. After 5 seconds without an packet reply from the slave, the master repeats the last packet with a cleared ACK flag (NACK).

Reply Packet Timeout : 5 seconds

Since the master cannot tell whether the master packet was lost or the replying slave packet the slave has to decide whether the packet is repeated or not by surveying the packet numbers.

Offline detection

After 3 times sending a packet to a slave without getting a correct reply packet, the master shall treat this device as being offline (not reachable) and inform the application about the offline state. The information to the application shall distinguish between No-Answers and Corrupted-Answers.

Fragmentation

Arbitrary data streams may be transmitted using the CSB transport layer. The data stream is split into packages which size may differ between 0 bytes up to MTU-Size bytes. The MTU-Size is specified by the device and may vary from 64 bytes up to a baud rate dependant value which is shown in the table below:

Baudrate	Possible MTU values	Max time of one (gap free) packet	Worst case possible cycle time[3]
115k2	**64...6144**	533,9 ms	2,22 s
57k6	**64...3072**	534,4 ms	2,22 s
38k4	**64...2048**	534,9 ms	2,22 s
19k2	**64...1024**	536,5 ms	2,22 s

Remember: The MTU is not the size of a packet or the payload of a packet. It is the device specific value of the maximum size of the payload of a packet.

Modbus compliant MTU

When communicating in a Modbus environment (with pure Modbus devices) the MTU must not exceed the maximum length of the payload concerning the Modbus specification: 250 bytes (256 incl. header/CRC).

Baudrate	Possible MTU values	Max time of one (gap free) packet	Worst case possible cycle time
115k2	**64...250**	22,2 ms	169 ms
57k6	**64...250**	44,4 ms	257,8 ms
38k4	**64...250**	66,7 ms	346 ms
19k2	**64...250**	133,3 ms	613 ms

[3] Worst case assumptions: 1) A gap of 1 byte times is given between all bytes. 2) The master starts sending the next packet 40ms after the complete reception of the slave packet. 3) The slave begins sending its packet 40ms after the complete reception of the master packet. 4) The telegram overhead of a packet is 6 byte.

Direct Addressing

Each packet starts with the 8 bit (1 byte) address.

The following addresses are not allowed or should be avoided:

Hex-Value	Recommandation	Binary representation	Reason
0xFF	**Not allowed**	b1111 1111	Used for baud rate sync.
0xFE	**Not allowed**	b1111 1110	Used for baud rate sync.
0xFC	**Not allowed**	b1111 1100	Used for baud rate sync.
0xF8	**Not recommended**	b1111 1000	Similar to baud rate sync. bytes
0xF0	**Not recommended**	b1111 0000	Similar to baud rate sync. bytes
0xE0	**Not allowed**	b1110 0000	Used for baud rate sync.
0xC0	**Not recommended**	b1100 0000	Similar to baud rate sync. bytes
0x80	**Not recommended**	b1000 0000	Similar to baud rate sync. bytes
0x00	**Not recommended**	b0000 0000	Similar to baud rate sync. bytes
0xAA	**Not allowed**	b1010 1010	Broadcast address

5.2.6. Summary of Function Codes (Modbus custom functions)

Function code	Dec. value	Addressing type	Function
‘C’	**67**	direct unicast	Transmit payload (**C**oLa messages)
‘B’	**66**	broadcast, (direct unicast)	Change **b**aud rate
‘F’	**70**	indirect unicast	**F**indMe
‘H’	**72**	Broadcast	First **H**ello (scan for slaves)
‘h’	**104**	Broadcast	Follow Up **H**ello (scan for slaves)
‘A’	**65**	indirect unicast	Change Slave **A**ddress
‘D’	**68**		

5.3. USB

SOPAS USB devices have to support at least “USB Full-Speed” (12 Mbit/s) and have to implement the “vendor specific” SOPAS USB device class. This class requires the “vendor specific” SOPAS USB interface to be the default interface (interface number 00h) within the default configuration.

The SOPAS USB interface consists of a bulk in endpoint at address 1 and a bulk out endpoint at address 2. A SOPAS USB device descriptor must have a unique USB product ID (idProduct), the device version (bcdDevice) specified as binary coded decimal number, a reasonable manufacturer and product string (iProduct) and a serial number string (iSerialNumber) unique among all devices with the same product ID.

Please see EDB Document “SOPAS USB SPECIFICATION” EDB # E_93608 and EDB Document “SICK USB Devices” EDB # E_95007.

6. CoLa Message Layer

While the CoLa Transport Layer only transports a stream of bytes (without start and end of a telegram) the CoLa Message Layer is responsible to determine the start and end of a telegram. Additionally it contains a specification of routing a telegram over CoLa devices into different networks.

This Layer is named «CoLa Message Layer 7.1» because it is a layer within the application layer of the ISO OSI communication model.

Due to the fact that CoLa telegrams are streamed through the CoLa devices a separation into two different layers is not very useful:

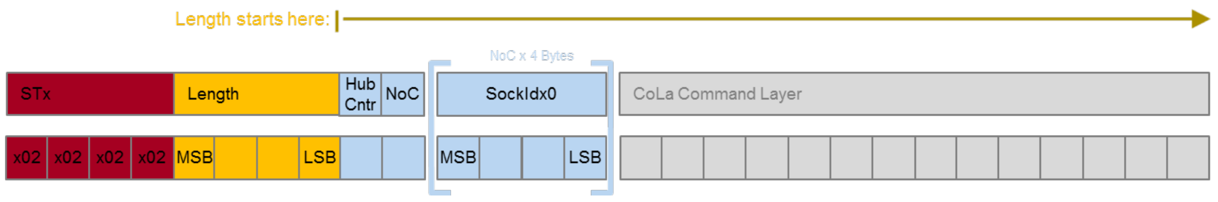
Note:

When the implementation of the CoLa Message Layer detects, that the telegram has to be routed to somewhere else, sending the telegram starts immediately (=”Cut-through”. Compare to ”store and forward”) The state machine of a separated “Routing Layer” would be very dependent of the state machine of layer responsible for detecting start and end of telegrams.

The serialization of the CoLa-Message-Layer is defined firmly in big Endian order (high byte first or also known as Motorola format)

6.1. Telegram Structure

The following figure shows the telegram structure of the CoLa Message Layer:



STx Start of Text (STx)

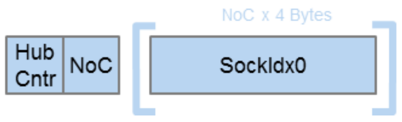
Every telegram starts with four times the byte 0x02 named STx. In theory these four bytes would not be needed (obsolete), because the length of a field is enough to determine when a new telegram starts.

In practice it was shown that those four STx bytes are very helpful when client and server lost their synchronization and have to find a new telegram start. It is redundant information, but helps to make the protocol more stable.

Length

Length

After the four STx bytes follow 4 bytes (in Big Endian format) that indicate the length of the telegram in bytes. The first byte that is calculated for the length is the first byte after the length field itself.



Hub Addressing Fields:

The <HubCntr> byte is a counter for hub addressing. If a remote connection is used this counter is increased in every gateway that is passed en route to the remote host. For local connections this field is always a binary zero.

The byte <NoC> indicates the direction (0=from client to server, 1=from server to client) in Bit7 and the number of cascades in Bit0..Bit2.

The number of cascades is the total number of gateways that have to be passed until the destination host can be reached. This means that only 7 cascades are allowed. For local connections this byte is always a binary zero.

The most significant bit indicates the direction the telegram is routed. If the bit is zero, it is a request and if it is a binary one it is a response.

If <NoC> is not a binary zero, then remote address(es) will follow. Each remote address consists of four bytes (Big Endian format) that represent a connection identifier SocketIdx. There are as many SocketIdx fields as the value of the number of cascades field indicates.

If the <NoC> field holds a binary zero, there will be no SocketIdx fields.

For a detailed description of the hub addressing mechanism please refer to the example in section **Fehler! Verweisquelle konnte nicht gefunden werden.**.

6.2. Routing Mechanism

The so called CoLa Hub Routing is optional. A directly connected CoLa 2.0 server (connected without hubrouting) can be addressed without any hub addresses. In this case only the <HubCntr> and <NoC> fields do exist, both with value zero.

If the field hub Control (<NoC>) is not null, it is an indicator that hub addressing is used in the telegram and it must be routed.

The cascading depth (also number of “SockIdx” fields) is encoded in the 3 least significant bits of the field <NoC>. This means that a maximum number of 7 cascades are allowed.

The SOPAS hub needs to know which of the addresses is meant for it: It has to know its position within the cascading. For this reason, a hub counter (HubCntr) is passed within the addressing data which is initialised with zero by the requesting client and incremented with every SOPAS hub which passes the request to the next CoLa Hub or simple CoLa device.

In order for the receiver to see the sender’s address and not its own address, the receiver address is successively replaced by the sender address. Every CoLa Hub routes the telegram to the SockIdx where the hub counter points to (start counting from 0), exchanges this SockIdx with the one the telegram came from and after that it increments the hub counter.

When the hub counter is equal to the number of cascades field the telegram has reached his destination.

The destination devices sets bit 7 of the NoC field and handles the request. This bit in the NoC field is an indication for the hub devices that the message is on the way back to the client. This means that the hub device now needs to decrement the hub counter and after that is done the telegram is routed to the to the SockIdx where the hub counter points to and this SockIdx is exchanged with the one the telegram came from.

Examples of a Hub Addressing setup, via one and two CoLa Hubs:

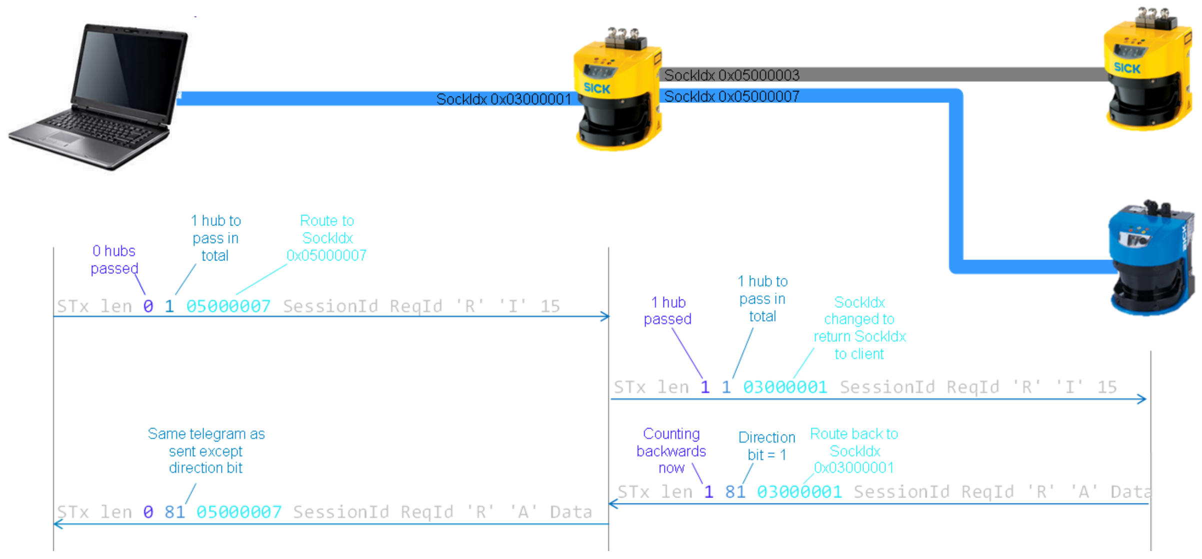

SockIdx 0x03000001
SockIdx 0x05000003
SockIdx 0x05000007
0 hubs passed
1 hub to pass in total
Route to SockIdx 0x05000007
STx len 0 1 05000007 SessionId ReqId 'R' 'I' 15
1 hub passed
1 hub to pass in total
SockIdx changed to return SockIdx to client
STx len 1 1 03000001 SessionId ReqId 'R' 'I' 15
Same telegram as sent except direction bit
STx len 0 81 05000007 SessionId ReqId 'R' 'A' Data
Counting backwards now
Direction bit = 1
Route back to SockIdx 0x03000001
STx len 1 81 03000001 SessionId ReqId 'R' 'A' Data

Illustration: A CoLa2.0 Client passes a telegram to a CoLa 2.0 server, using hub addressing via one hub

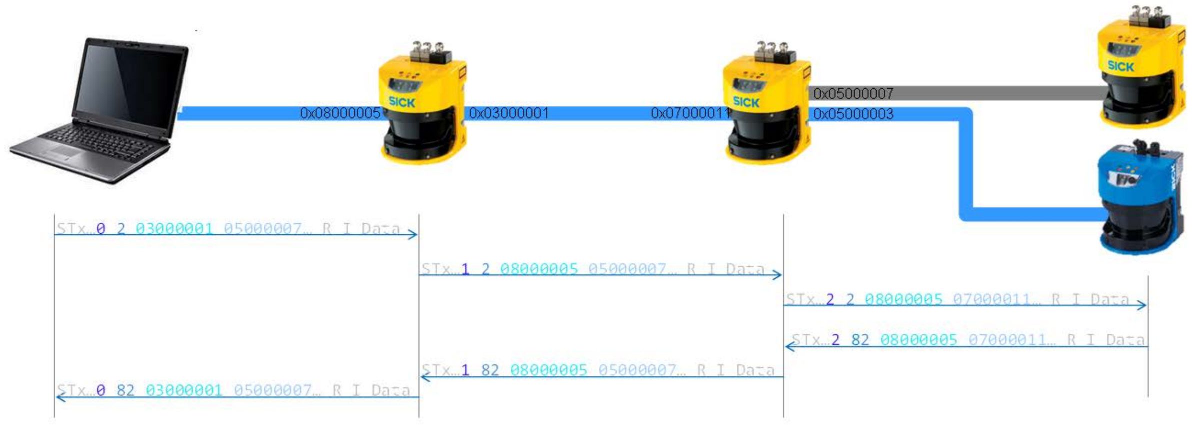

0x08000005
0x03000001
0x07000011
0x05000007
0x05000003
STx...0 2 03000001 05000007... R I Data
STx...1 2 08000005 05000007... R I Data
STx...2 2 08000005 07000011... R I Data
STx...2 82 08000005 07000011... R I Data
STx...1 82 08000005 05000007... R I Data
STx...0 82 03000001 05000007... R I Data

Illustration: A CoLa2.0 Client passes a telegram to a CoLa 2.0 server, using hub addressing via two hubs

Note: Please see the remarks below.

Remarks on the HubAddresses:

Name	Meaning
HubCounter	How many hubs have been passed already?
NoC	
Bit7: DIR	Direction (0= from client to server, 1=from server to client)
Bit3..Bit6	Reserved
Bit0..Bit2: NoC	How many hubs have to be passed in total?
SockIdx0 **SockIdx1** **...** **SockIdxN**	The number of addresses must correspond to the value of Number of Cascades. **Applies to requests (DIR=0):** With every passing of a hub the corresponding target address (e.g. SockIdx in the first hub) will be replaced by the given source address. **Applies to responses (DIR=1):** With every passing of a hub the corresponding source address will be used to route the message back to the originator of it and will be replaced by the given target address to finally restore the original destination address.

7. CoLa Command Layer

This Layer is named «CoLa Command Layer 7.2» because it is a layer within the application layer of the ISO OSI communication model and placed above of the «CoLa Message Layer 7.1».

The serialization of the CoLa Command Layer is not uniformly defined. The numbers before the «Cmd» byte are serialized always in big Endian sequences (high byte first, also known as Motorola format).

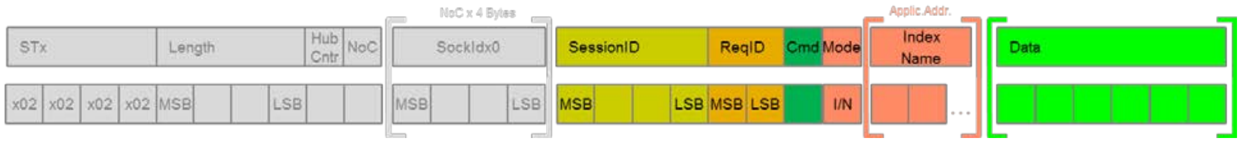
The serialization of the CoLa Message Layer behind the «Cmd» byte is variably defined in either

- big Endian order (high byte first, also known as Motorola format) or
- little Endian order (low byte first, also known as Intel format).

The Byte Order after the «Cmd» byte has to be unique for a device and must be indicated in the so called DeviceInfo described in chapter 8.2.2.

7.1. Telegram Structure

The following figure shows the telegram structure of the CoLa Command Layer:



SessionID

Session ID

The 4 Byte Session ID will be generated by the server / device. Therefore a (pseudo) random function will be needed; the device developer will have to provide this function for SOPAS Runtime (SRT).

During the session open handshake, the session ID must be set to 0.

In SOPAS Runtime an explicit session handling, with corresponding session data will be established. The Session ID will be the "Key" for the session data.

Note:

For correct session handling (especially the termination of session after a connection loss) a heartbeat mechanism is required. Only with a heartbeat sessions can be reliably terminated (See chapter "Heartbeat"). The timeout period is parameterized during the session open handshake.

Only with an explicit session handling both sides, server and client, are always able to decide if a session is still valid and active. For detailed description of the session management see chapter session management.

ReqID

Request ID:

The Request ID is used to associate answers to the respective requests. The client always generates the Request ID. The client must not use the same ID multiple times.

Note:

One possibility is for example to increment every Request ID by 1 in a sequence of requests that share the same session. But a Request ID is not coercive related to a session.

Command:

A Cola command as described in chapter CoLa Commands.

Cmd	Mode	Index/Name

Note:

Commands do not start with the letter <'s'> anymore, like they did in CoLaB.

The <Cmd> byte is an identifier which command is issued (e.g. "R" stands for the command "Read Variable"). The <Mode> Byte usually indicates whether addressing by Index or by Name is used, followed by 2 bytes Index or n bytes Name. After that follows the command data, but only if the command needs data.

7.1.1. Session Management

A client has to establish a session and keep it alive by sending any valid telegram every ***n*** seconds. If ***n*** exceeds the specified timeout the session will time out after the specified timeout value (30 seconds in a hubrouting scenario).

A local session is established when the client is directly connected to the device on which the session shall be established (addressing or routing completely done by one CoLa Transport protocol).

A remote session is established when a client cannot directly address the device (using only one CoLa Transport Layer instance). In this case a CoLa device in the middle (CoLa router) is asked to establish the connection on the remote device (placed behind the routing "Hub" device).

Local Session

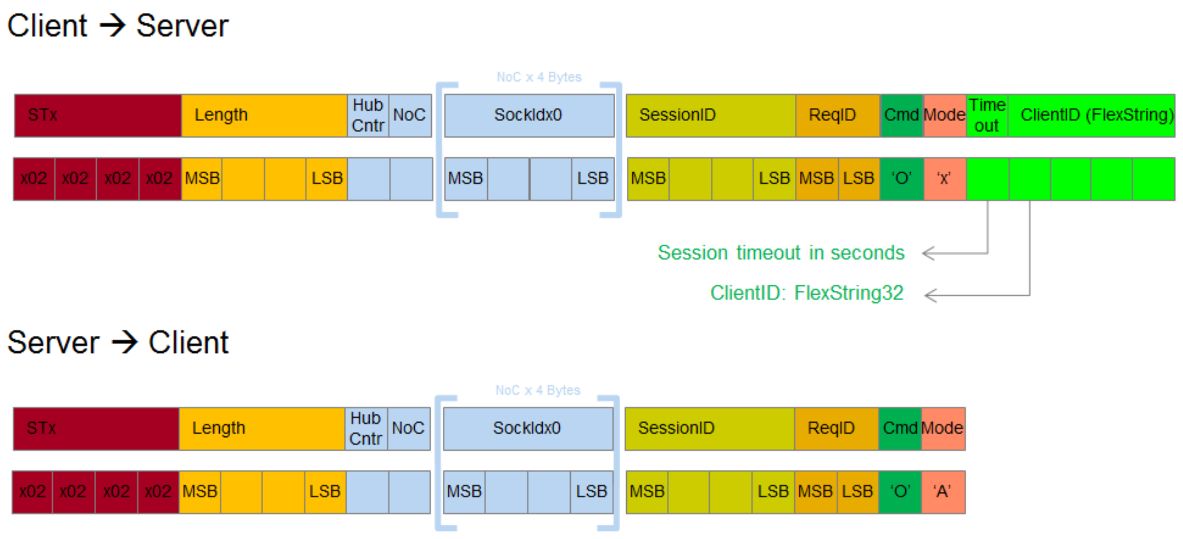


Illustration: Local open session command and reply

In CoLa 2.0 all commands require that a session is available. A session must be explicitly requested by the client.

The client asks for a session to be started by issuing the "Open Session" Command "O" with the following parameters:

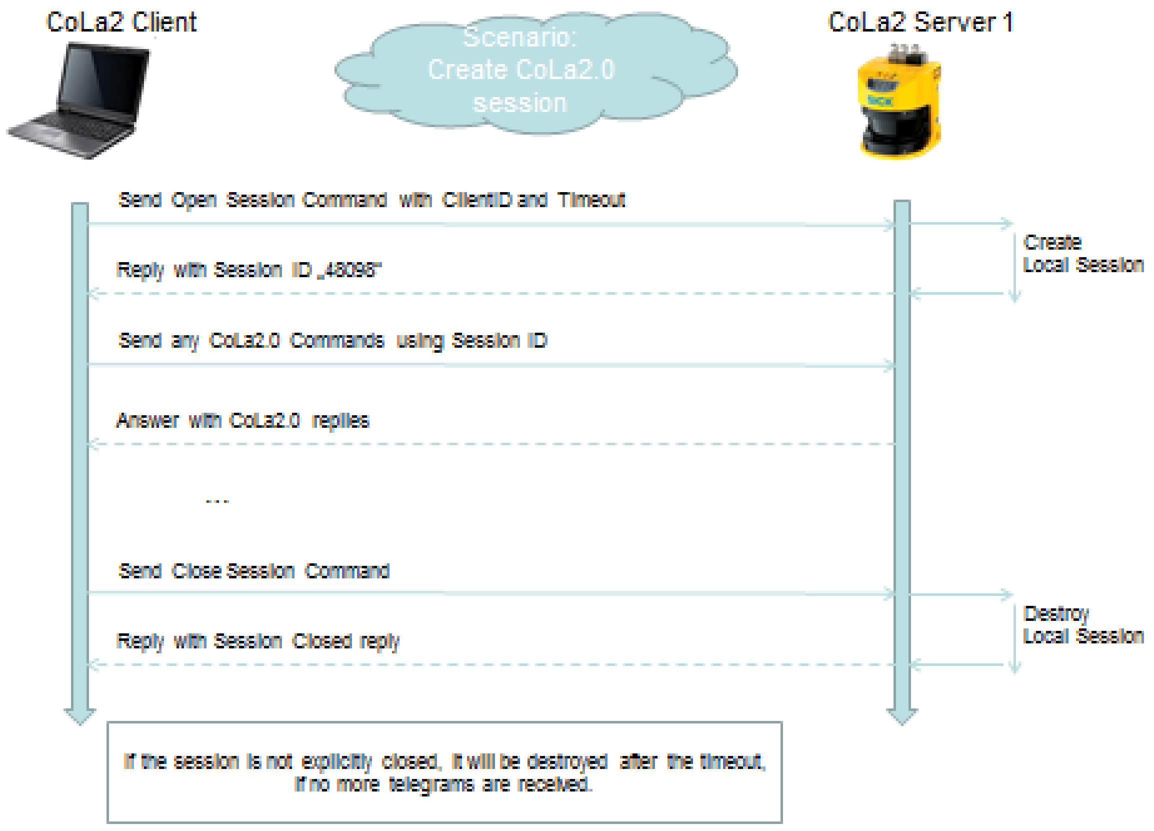
- Timeout: 1 Byte specifying the heartbeat / timeout interval in seconds, which shall be used for this session. (Will be ignored when Session is routed via a CoLa Hub)
- Client ID: a Flexstring with a maximum width of 32 Bytes. The connecting client should give a describing string here. (Example: "SOPAS ET/deagm10000"). The server will store this ID in the session data.

The server responds with "Open Session acknowledged" and fills the SessionID field with a random generated Session ID. If no more sessions can be opened by the server, it replies with an errorcode (Sopas_Error_SESSION_NORESOURCES 0x0021).

.

Recommendation: build the ClientId like in the example above. Which is ClientName or User/Hostname.

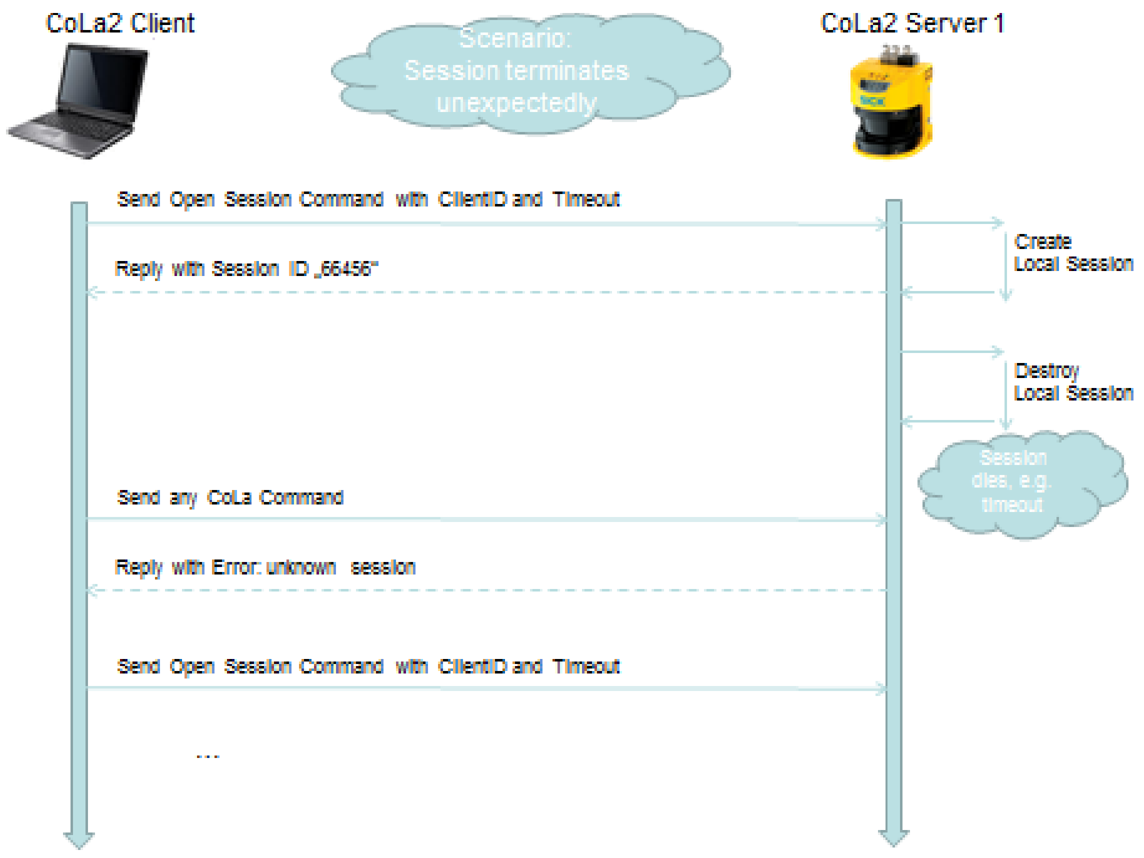
The following sequence diagram shows how local sessions are used:


CoLa2 Client
Scenario: Create CoLa2.0 session
CoLa2 Server 1
Send Open Session Command with ClientID and Timeout
Create Local Session
Reply with Session ID „48098"
Send any CoLa2.0 Commands using Session ID
Answer with CoLa2.0 replies
...
Send Close Session Command
Destroy Local Session
Reply with Session Closed reply
If the session is not explicitly closed, it will be destroyed after the timeout, if no more telegrams are received.

The diagram shows how a CoLa2 Client can open a session with a CoLa2 Server.

1. The CoLa2 Client issues an "Open Session" command to the CoLa2 Server. Parameters are ClientId and SessionTimeout
2. CoLa2 Server creates a session, generates a SessionId and acknowledges the successful creation of the session
3. Now, using this SessionIDd the CoLa2 Client can issue more CoLa2 commands until it finally closes the session

The next sequence diagram shows how broken connections are handled:

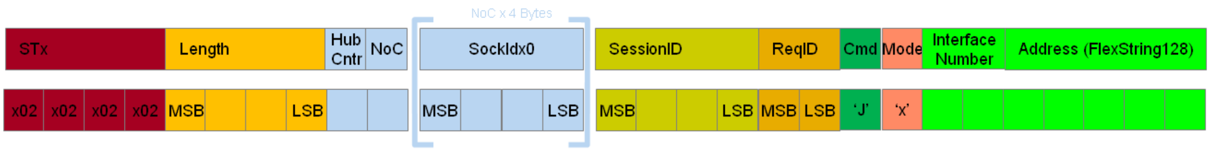

CoLa2 Client
Scenario: Session terminates unexpectedly
CoLa2 Server 1
Send Open Session Command with ClientID and Timeout
Create Local Session
Reply with Session ID „66456"
Destroy Local Session
Session dies, e.g. timeout
Send any CoLa Command
Reply with Error: unknown session
Send Open Session Command with ClientID and Timeout
...

The diagram shows what happens when a session on a CoLa2 Server gets accidentally destroyed:

1. The CoLa2 Client opens a session with CoLa2 Server (already described in previous diagram)
2. The session times out before any more commands arrive
3. CoLa2 Client issues a CoLa2 command, using the SessionId from 1.
4. CoLa2 Server replies that the session does not exist.
5. CoLa2 Client continues with 1.

Remote Session

Client → Server



Server → Client

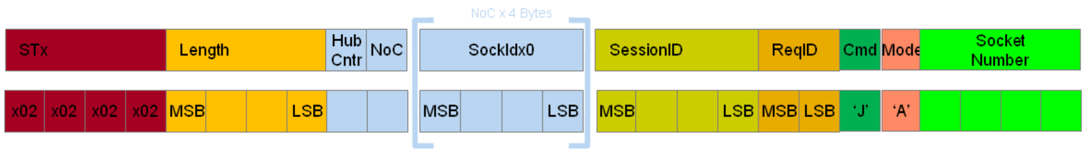


Illustration: Remote open session command ("J"oin Network) and reply

The client asks for a connection to be established by issuing the "Join Network" Command. The parameters are:

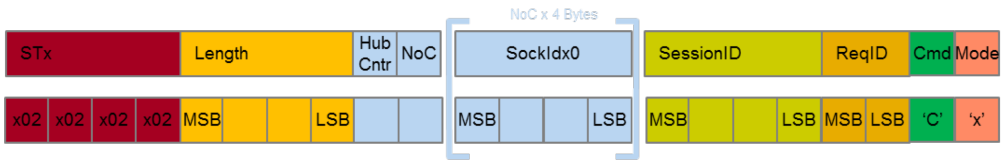
- Interface Number: A UInt value representing the number of the Interface the client wants to connect.
- Address Field: a FlexString with a maximum length of 128 Bytes. It holds the remote address the client wants to connect to. He got this address from a previously issued Scan ("H"ello) command.

Note: The remote session command does not really open a session. It only notifies a gateway (SOPAS Hub) that a connection to the given address will be needed. The SOPAS Hub / Gateway will open a connection to that address (if necessary) and tell the client the corresponding hub address.

Then the client must open a local session on the destination hub with the "normal" Open Session command.

Local and Remote Sessions can both be closed with the same command:

Client → Server



Server → Client

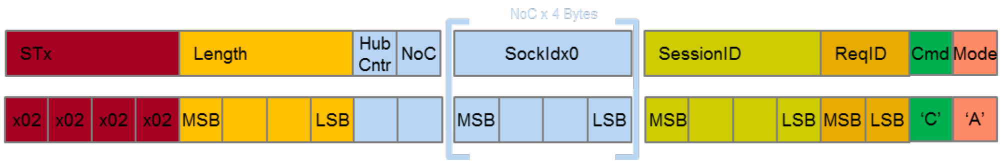
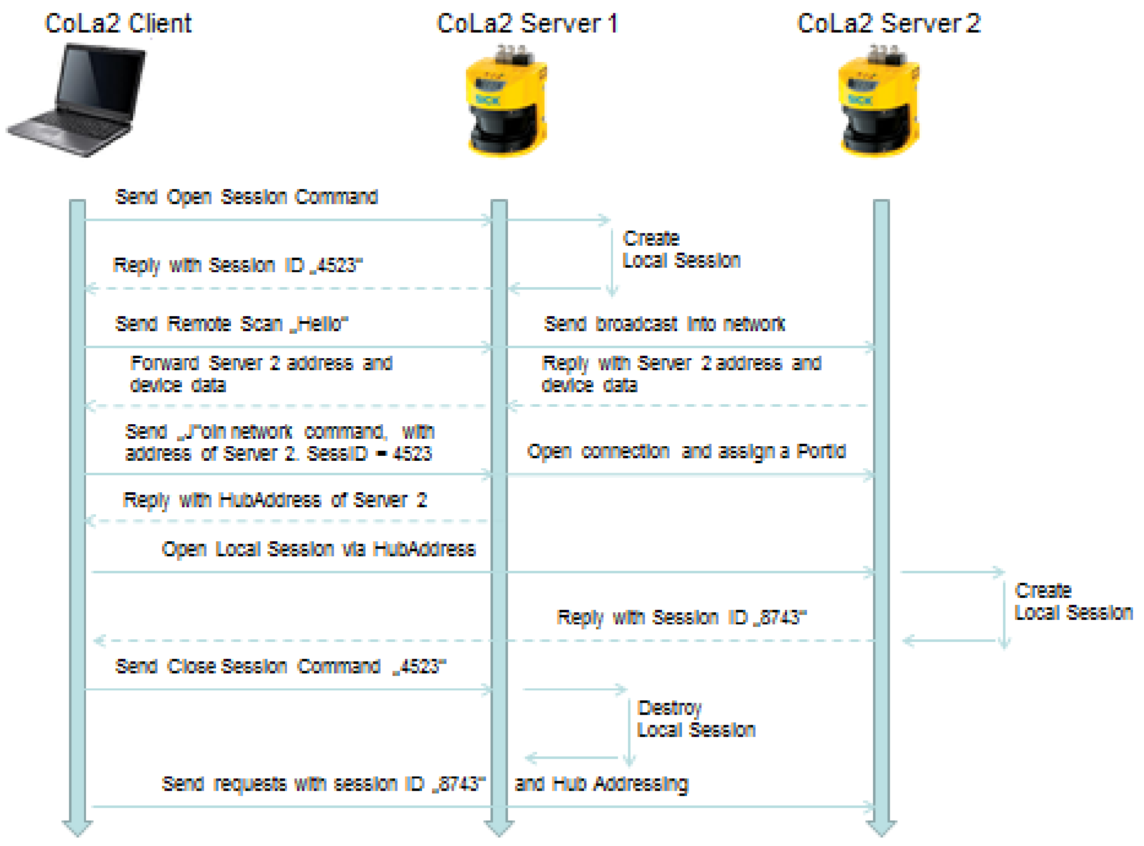


Illustration: Close session command and reply

A session can be explicitly closed with the Close Session "C" command. The receiver of that command acknowledges the command.

If the sessions are not explicitly closed by the peer, using this command, they must be locally closed after the timeout expires.

The following sequence diagram shows a remote session:


CoLa2 Client
CoLa2 Server 1
CoLa2 Server 2
Send Open Session Command
Create Local Session
Reply with Session ID „4523"
Send Remote Scan „Hello"
Send broadcast into network
Forward Server 2 address and device data
Reply with Server 2 address and device data
Send „J"oin network command, with address of Server 2. SessID = 4523
Open connection and assign a PortId
Reply with HubAddress of Server 2
Open Local Session via HubAddress
Create Local Session
Reply with Session ID „8743"
Send Close Session Command „4523"
Destroy Local Session
Send requests with session ID „8743"
and Hub Addressing

The above diagram shows how a CoLa2 Client can open a session with CoLa2 Server which is connected via hubrouting to another CoLa2 Server (remote session).

1. CoLa2 Client opens a session to CoLa2 Server 1. CoLa2 Server 1 creates a local session and replies using the corresponding session id.
2. CoLa2 Client performs a remote scan ("Hello" command). CoLa2 Server 1 performs a network scan and sends the reply telegrams to CoLa2 client.
3. CoLa2 Client tells CoLa2 Server 1 to open a connection to CoLa2 Server 2 by issuing a "Join Network" command. CoLa2 Server 1 opens a connection to CoLa2 Server 2 and sends the corresponding PortId to CoLa2 Client.
4. CoLa2 Client opens a session with CoLa2 Server 2 using the PortId for hubrouting. The session with CoLa2 Server 1 can now be closed (if not, it will time out). Hubrouting does not need a session with CoLa2 Server 1.

7.1.2. Session Timeouts / Heartbeat

The device (and the client) can detect closed connections and terminate the sessions. Even if the device developer forgot the CloseSession() call. Or if the connection loss was undetectable by Software (e.g. cable unplugged). This can be done by closing sessions that had no data transmissions during the timeout interval.

Note: Clients should implement a "heartbeat" mechanism to prevent the timeout.

The heartbeat does not need a protocol enhancement.

The heartbeat does not need a dedicated command or variable which is polled. Every request can be used.

The connection is reset and the session is terminated if no communication from the session peer was received for a specified time.

The specified time is always 30 seconds when a session is routed through SOPAS hubs (the addressing fields are used). If it is a local connection, the specified time is defined by a field in the session start "O" command.

7.2. CoLa Commands

The following commands are defined.

Overview of commands:

Command	Addressing by ...	Request	Response	Remark
Read	index	RI(Uint SVIdx)	RA(Uint SVIdx) (Data)	
	name	RN _(Name)_	RA_(Name)_(Data)	The name is enclosed in blanks!
Write	index	WI(Uint SVIdx) (Data)	WA(Uint SVIdx)	
	name	WN_(Name)_(Data)	WA_(Name)_	The name is enclosed in blanks!
Invoke Method	index	MI(Uint SMIdx) [(Pars)]	[MA(Uint SMIdx)] AI(Uint SMIdx) [(RetV)]	The MA only acknowledges the starting of the (asynchronous) Method. It may be omitted. Finally the AI / AN signals are the result of the method. Parameters (Pars) and return values (RetV) may be omitted corresponding to the CID definition.
	Name	MN_(Name)_ [(Pars)]	[MA_(Name)_] AN_(Name)_ [(RetV)]	

Command	Addressing by ...	Request	Response	Remark
Register Event	index	**EI(Uint SEIdx) [(State)]**	**EI(Uint SEIdx)State)**	The request **EI** / **AN** optionally carries the (new) state of the event (registered:1 or not registered:0). Hence omitting this state in the request can be used to query the current state.
	Name	**EN_(Name)_[(State)]**	**EN_(Name)_(State)**	
Send Event	index		**SI(Uint SEIdx) [(Data)]**	The data may be omitted corresponding to the CID definition.
	Name		**SN_(Name)_ [(Data)]**	
Open Session	Local/ remote	**Ox(Data)***	**OA**	*****=see Session
Close Session	local	**Cx**	**CA**	The session that shall be closed can be identified by the SessionID in the Header
	remote	**Cx**	**CA**	

Command	Addressing by …	Request	Response	Remark
Remote Scan ("Hello")		**Hx(Uint InterfaceNumber)**	**HA** **HR(Uint InterfaceNumber) (Address)**(Scandata)**	The hub first answers with a conformation that the scan was started. After this the hub answers with (multiple) responses from the local scan it performs. (Address) is a network address. If the found device is not directly accessible (e.g. network parameters do not match) the address is set to zero. **=see Address
Remote SetNetworkAddress	—	**NE(Uint InterfaceNumber) (USInt[6] MacAddress) (UDInt IpAddress) (UDInt NetMask) (UDInt DefaultGateway) (Bool UseDHCP)**	**NA** **NR(SetIpParameter_ReturnValue)**	The hub first answers with a confirmation that the parameter matches to the interface. After this the hub tries to send the UDP command. In case that the send fails (e.g. no Ethernet cable plugged) the hub answers with an FA. When an UDP answer is received it will be forwarded to the client. If no answer arrives within a timeout the hub send a FA to the client.

Command	Addressing by ...	Request	Response	Remark
Remote Beep/ Blink	—	**BE (Uint InterfaceNumber) (USInt[6] MacAddress) (UInt Duration)**	**BA** **BR(Uint InterfaceNumber)**	The hub first answers with a confirmation that the parameter matches to the interface. After this the hub tries to send the UDP command. In case that the send fails (e.g. no Ethernet cable plugged) the hub answers with an FA. When an UDP answer is received it will be forwarded to the client. If no answer arrives within a timeout the hub send a FA to the client.
Join Network		**Jx(Uint InterfaceNumber) (Address)****	**JA(UDInt SockId)**	**=see Address
Device Info		**Dx**	**DA (Scandata)**	The SOPAS 'D' Command is used for connections where no UDP Scan is possible (e.g. serial or USB). It retrieves the same data an UDP scan does.
Device Info		**DP(Uint InterfaceNumber)**	**DA (Scandata)**	The SOPAS 'D' with mode 'P' can be used to read the Device Info from another interface (not the one the client is connected to)

Read Variable

A variable is read out of the variable server of the device by using the SOPAS command **'R'**.

Definition:

Addressing by ...	Request	Response
By index	**RI(Uint SVIdx)**	**RA(Uint SVIdx) (Data)**
By name	**RN_(Name)_**	**RA_(Name)_(Data)**

If the command failed, an error answer is sent instead of the above shown response.

Examples:

The Variable named Temperature has a SOPAS index of 32 (0x20) and has a value of 256 (0x100):

Byte order	Addressing by ...	Request	Response
BigEndian	By index	**RI<0x00><0x20>**	**RA<0x00><0x20><0x01><0x00>**
	By name	**RN_Temperature_**	**RA_Temperature_ <0x01><0x00>**
LittleEndian	By index	**RI<0x20><0x00>**	**RA<0x20><0x00><0x00><0x01>**
	By name	**RN_Temperature_**	**RA_Temperature_ <0x00><0x01>**

Write Variable

A variable is written to the variable server of the device by using the SOPAS command **"W"**.

Definition:

Addressing by ...	Request	Response
By index	**WI(UInt SVIdx)(New Value)**	**WA(UInt SVIdx)**
By name	**WN_(Name)_ (New Value)**	**WA_(Name)_**

If the command failed, an error answer is sent instead of the above shown response.

Examples:

The Variable named Angle, has a SOPAS index of 35 (0x23) and shall be set to 456 (0x1C8):

Byte order	Addressing by ...	Request	Response
BigEndian	By index	**WI<0x00><0x23><0x01><0xC8>**	**WA<0x00><0x23>**
	By name	**WN_Angle_<0x01><0xC8>**	**WA_Angle_**
LittleEndian	By index	**WI<0x23><0x00><0xC8><0x01>**	**WA<0x23><0x00>**
	By name	**WN_Angle_<0xC8><0x01>**	**WA_Angle_**

Method Invocation

A method is invoked on the SOPAS device using the SOPAS command **'M'**.

Definition:

Addressing by ...	**Request**	**[Optional Acknowledge] Response**
By index	**MI(UInt SMIdx) [(Pars)]**	**[MA(UInt SMIdx)]** **AI(UInt SMIdx) [(RetVal)]**
By name	**MN_(Name)_[(Pars)]**	**[MA_(Name)_]** **AN_(Name)_[(RetVal)]**

If the SOPAS command failed, an error answer is sent instead of the above shown response. If the implementation of the method inside fails it should signal that using the return value(s) RetVal. It will nevertheless send an **AI** / **AN** instead of a **FA**.

Timing Constraints:

Synchronous Method without an ExecutionTimeout in CID

Including the transmission of the request and the transmission of the response, the method invocation command (**MN** → **AN**) must not exceed the time specified in chapter «9 Timing Constraints» for the given CoLa Transport Layer.

Synchronous Method with an ExecutionTimeout in CID

Including the transmission of the request and the transmission of the response, the method invocation command (**MN** → **AN**) must not exceed the time specified in chapter «9 Timing Constraints» for the given CoLa Transport Layer.

Asynchronous Method without an ExecutionTimeout in CID

Including the transmission of the request and the transmission of the acknowledge message, the method invocation command (MN → MA) must not exceed the time specified in chapter «9 Timing Constraints» for the given CoLa Transport Layer. Furthermore the definition of an **ExecutionTimeout** can limit the time until the AN should have been received.

Examples:

The Method for this example is synchronously working and is named Int iCalcTax(Int divalue).It has a SOPAS index of 12 (0x0C) and shall be called to calculate the tax (19%) of 429,12€ (42912 cent → 0xA7A0):

Byte order	Addressing by ...	Request	Response
BigEndian	By index	**MI<0x00><0x0C><0xA7><0xA0>**	**AI<0x00><0x0C><0x1F><0xD9>**
	By name	**MN_iCalcTax_<0xA7><0xA0>**	**AN_iCalcTax_<0x1F><0xD9>**
LittleEndian	By index	**MI<0x0C><0x00><0xA0><0xA7>**	**AI<0x0C><0x00><0xD9><0x1F>**
	By name	**MN_iCalcTax_<0xA0><0xA7>**	**AN_iCalcTax_<0xD9><0x1F>**

The Method for the next example is asynchronously working and is named Bool bTestRam (Uint uiStartAddress, Uint uiEndAddress).It has a SOPAS index of 13 (0x0D) and returns True if the Rom is ok:

Byte order	Addressing by ...	Request	Response
BigEndian	By index	**MI<0x00><0x0D><0x40><0x00><0x80><0x00>**	**MA<0x00><0x0D>** **AI<0x00><0x0D><0x01>**
	By name	**MN_bTestRam_<0x40><0x00><0x80><0x00>**	**MA_bTestRam_** **AN_bTestRam_<0x01>**
LittleEndian	By index	**MI<0x0D><0x00><0x00><0x40><0x00><0x80>**	**MA<0x0D><0x00>** **AI<0x0D><0x00><0x01>**
	By name	**MN_bTestRam_<0x00><0x40><0x00><0x80>**	**MA_bTestRam_** **AN_bTestRam_<0x01>**

General remarks to the asynchronous Methods
SRT does support asynchronous Methods. BUT: If a Method is still running, it is not possible to start another Method. Hence if a Method requires calling of SetAccessMode() and Run(), this will fail in SOPAS-ET. SRTOS can support multiple parallel working Methods if implemented correctly in the SU_OSAI module.

Event registration

A client (PC) can register to an Event of the SOPAS device using the SOPAS command **'E'**.

Definition:

Addressing by ...	**Request**	**Response**
By index	**EI(UInt Sidx)[(New State)]**	**EA(UInt Sidx) (New State)**
By name	**EN_(Name)_ [(New State)]**	**EA_(Name)_ (New State)**

The request EI / AN optionally carries the (new) state of the event (registered:1 or not registered:0). Hence omitting this state in the request can be used to query the current state.

If the command failed, an error answer is sent instead of the above shown response.

Examples:

The Event is named gotMessage and has String[5] as data (parameter). It has a SOPAS index of 2:

Registration

Byte order	Addressing by ...	Request	Response
BigEndian	By index	**EI<0x00><0x02><0x01>**	**EA<0x00><0x02><0x01>**
	By name	**EN_gotMessage_<0x01>**	**EA_gotMessage_<0x01>**
LittleEndian	By index	**EI<0x02><0x00><0x01>**	**EA<0x02><0x00><0x01>**
	By name	**EN_gotMessage_<0x01>**	**EA_gotMessage_<0x01>**

Deregistration:

Byte order	Addressing by ...	Request	Response
BigEndian	By index	**EI<0x00><0x02><0x00>**	**EA<0x00><0x02><0x00>**
	By name	**EN_gotMessage_<0x00>**	**EA_gotMessage_<0x00>**
LittleEndian	By index	**EI<0x02><0x00><0x00>**	**EA<0x02><0x00><0x00>**
	By name	**EN_gotMessage_<0x00>**	**EA_gotMessage_<0x00>**

Query of state:

Byte order	Addressing by ...	Request	Response
BigEndian	By index	**EI<0x00><0x02>**	**EA<0x00><0x02><0x00>**
	By name	**EN_gotMessage_**	**EA_gotMessage_<0x00>**
LittleEndian	By index	**EI<0x02><0x00>**	**EA<0x02><0x00><0x00>**
	By name	**EN_gotMessage_**	**EA_gotMessage_<0x00>**

Event Transmission

The device (server) sends an event using the SOPAS command **'S'**.

Definition:

Addressing by ...	**"Response" (actually not a response but this direction)**
By index	**SI(UInt Sidx) [(data)]**
By name	**SN_(Name)_ [(data)]**

The data may be omitted corresponding to the CID definition.

Examples:

The Event is named gotMessage and has String[5] as data (parameter). It has a SOPAS index of 2:

Byte order	Addressing by …	Request
BigEndian	By index	**SI<0x00><0x02><'H'><'e'><'l'><'l'><'o'>**
	By name	**SN_gotMessage_<'H'><'e'><'l'><'l'><'o'>**
LittleEndian	By index	**SI<0x02><0x00><'H'><'e'><'l'><'l'><'o'>**
	By name	**SN_gotMessage_<'H'><'e'><'l'><'l'><'o'>**

Open Session

A client opens a session, using the command **'O'**.

Definition:

Addressing by …	Request	Response
Local/ Remote	**Ox(Data)***	**OA**

*=see Session

**=see Address

Examples:

The client wants to open a session to a device that is in the same network as the client. The session timeout should be 120 seconds and the client ID is “Dev1”. The socked ID for the session is 0x00000005:

Local/ Remote Session:

Byte order	Request	Response
Big Endian	**0x<0x78><0x00><0x04><'D'><'e'><'v'><'1'>**	**OA**
Little Endian	**0x<0x78><0x04><0x00><'D'><'e'><'v'><'1'>**	**OA**

Close Session

A client closes a session, using the command **‘C’**. While opening the session distinguishes between local and remote session, the session is always closed by addressing the server directly. Hence the command makes no difference for remote sessions.

Definition:

Session type	**Request**	**Response**
Local	**Cx**	**CA**
Remote		

Remote Scan (Hello)

Scan remote Subnet (“Hello”)

A client scans one or several network, using the command **‘H’**.

Definition:

Addressing by ...	**Request**	**Response**
Remote	**Hx(Uint InterfaceNumber)**	**HA** **HR(Uint InterfaceNumber) (FlexString32 Device1Address) (DIStruct Device1Info)** **HR(Uint InterfaceNumber) (FlexString32**

		Device2Address) (DIStruct Device2Info) **HR(Uint InterfaceNumber) (FlexString32 Device3Address) (DIStruct Device3Info)** **....**

The hub first answers with a conformation that the scan was started. After this the hub answers with (multiple) replies from the local scan it performs. Each reply contains the Name and the Scandata from the device connected to the hub device.

(InterfaceNumber) is the interface scaned by the hub. 0xFFFF means scan all interfaces except the interface the remote scan came from. All differing entries describe the interface number of the interface like it is published in the device info. The Scan answer contains the interface number of the interface the answer was received.

(DeviceXAddress) specifies the address the hub will need to open a remote session. The address is a visible string which can be represented to users. The content of this string is not specified in detail. The content is generated and interpreted by the hub device only. If a found device is not directly accessible (e.g. network parameters do not match) the address is set to zero.

(DeviceXInfo) contains the complex structure with all information relevant for a first identification of the device (names, versions, state and communication properties). The structure is describes in chapter 8.2.2.

The **(InterfaceNumber)** and the **(DeviceXAddress)** are always transmitted in Big Endian.

In case that an interface is busy following message flow will take place:

Definition:

Addressing by ...	Request	Response
Remote	**Hx(Uint InterfaceNumber)**	**FA(ScanAlreadyActive)**

Examples:

A hub device receives a remote scan command containing the instruction to execute a scan on interface 1. Three other devices were found at this interface: Emma, Gillian and Leon.

Byte order	Request	Response
Big Endian	**Hx<0x00> <0x01>**	**HA** **HR<0x00><0x01><0x00><0x04><'E'><'m'><'m'><'a'>(Emmas Device Info)** **HR<0x00><0x01><0x00><0x07><'G'><'i'><'l'><'l'><'i'><'a'><'n'> (Gillians Device Info)** **HR<0x00><0x01><0x00><0x04><'L'><'e'><'o'><'n'>(Leons Device Info)**
Little Endian	**Hx<0x01> <0x00>**	

Remote SetNetworkAddress

A client changes the network parameter of a remote device, using the command **'N'**.

Definition:

Addressing by ...	Request	Response
Remote SetNetworkAddress	NE**(Uint InterfaceNumber)** **(USInt[6] MacAddress)** **(UDInt IpAddress)** **(UDInt NetMask)** **(UDInt DefaultGateway)** **(Bool UseDHCP)**	NA NR **(SetIpParameter_ReturnValue)**

The hub first answers with a confirmation that the parameter matches to the interface. The parameters of the requests are serialized in the Endianess of the hub device. The response is fix in Big Endian format.

After this the hub tries to send the UDP command. In case that the send fails (e.g. no Ethernet cable plugged) the hub answers with an FA.

The UDP commands are sent as UDP broadcast telegrams.

When an UDP answer is received it will be forwarded to the client. If no answer arrives within a timeout the hub send a FA to the client.

(InterfaceNumber) is the interface the remote device is connected to.

(USInt[6] MacAddress) is used to address the remote device the network parameter should be changed.

(UDInt IpAddress) contains the new IP address for the remote device.

(UDInt NetMask) contains the new subnet mask for the remote device.

(UDInt DefaultGateway) contains the new default gateway for the remote device.

(Bool UseDHCP) is true when DHCP should be enabled.

(SetIpParameter_ReturnValue) is the return value like defined in chapter 8.3.1

Examples:

A hub device receives a command to change the IP address of the remote device with the Mac Address 00:06:77:82:1f:5a connected at interface 0x0001. The device answers per UDP with the parameter **(SetIpParameter_ReturnValue)**

Byte order	Request	Response
Big Endian	NE**<0x00><0x01>** **<0x00><0x06><0x77><0x82><0x1f><0x5a>** **<0xc0><0xa8><0x01><0x69>** **<0xff><0xff><0xff><0x00>**	NA NR **(SetIpParameter_ReturnValue)**

	<0xc0><0xa8><0x01><0x01> <0x00>	
Little Endian	NE<0x01><0x00> <0x00><0x06><0x77><0x82><0x1f><0x5a> <0x69><0x01><0xa8><0xc0> <0x00><0xff><0xff><0xff> <0x01><0x01><0xa8><0xc0> <0x00>	

Remote Beep/ Blink

A client requests a remote device to start blinking/ beeping, using the command **'B'**.

Definition:

Addressing by ...	Request	Response
Remote Beep/ Blink	**BE(Uint InterfaceNumber) (USInt[6] MacAddress) (UInt Duration)**	BA BR

The hub first answers with a confirmation that the parameter matches to the interface. The parameters of the requests are serialized in the Endianess of the hub device. The response is fix in Big Endian format.

After this the hub tries to send the UDP command. In case that the send fails (e.g. no Ethernet cable plugged) the hub answers with an FA.

The UDP commands are sent as UDP broadcast telegrams.

When an UDP answer is received it will be forwarded to the client. If no answer arrives within a timeout the hub send a FA to the client.

(InterfaceNumber) is the interface the remote device is connected to.

(USInt[6] MacAddress) is used to address the remote device the network parameter should be changed.

(UInt Duration) contains the duration in seconds after that the device should stop blinking/ beeping.

Examples:

A hub device receives a command to let the remote device with the Mac Address 00:06:77:82:1f:5a connected at interface 0x0001 blink. First answer (BA) is sent to the client when the parameter set is ok, second answer (BR) is sent to the client after the UPD response was received.

Byte order	Request	Response

Big Endian	BE<0x00><0x01> <0x00><0x06><0x77><0x82><0x1f><0x5a> <0x00><0x1e>	BA BR
Little Endian	BE<0x01><0x00> <0x00><0x06><0x77><0x82><0x1f><0x5a> <0x1e><0x00>	

Join Network

Addressing by ...	Request	Response
Remote	Jx(Uint InterfaceNumber) (Address)	JA(UDInt SockId)

(InterfaceNumber) specifies the interface where the device is connected to. The Scan answer contains the interface number of the interface the answer was received.

(Address) specifies the address the hub will need to open a remote session. The address is a visible string which can be represented to users. The content of this string is not specified in detail. The content is generated and interpreted by the hub device only.

In case of error the Join network returns an Error Notification with error code 0x0023 (**CANNOT_CONNECT**)

Examples:

The client wants to open a remote session to Device2 that is not in the same network than the client. Device 2 was found by a Remote Scan from Device 1 at the interface 0x0300 and the address of the Device 2 is "192.0168.0.10". The socked ID for the session is 0x03000007:

Remote Session:

Byte order	Request	Response
Big Endian	Jx<0x03><0x00><0x00><0x0c>"192.168.1.10"	JA<0x03><0x00><0x00><0x07>
Little Endian	Jx<0x00><0x03><0x0c><0x00>"192.168.1.10"	JA<0x07><0x00><0x00><0x03>

Error Notification

The device can respond every request with an error answer instead of the above defined response using the SOPAS command **'F'**. The (ErrorNumber) is a Uint.

Definition:

Alternative Response
FA(Uint: ErrorNumber)

Examples:

Dialect	Alternative Response
CoLa 2.0	**FA<0x00><0x0E>**
CoLa 2.0 Intel	**FA<0x0E><0x00>**

Known Error Numbers of SRT:

Number	Text	Remark
0x0001	**METHODIN_ACCESSDENIED**	Wrong userlevel, access to method not allowed.
0x0002	**METHODIN_UNKNOWNINDEX**	Trying to access a method with an unknown SOPAS index.
0x0003	**VARIABLE_UNKNOWNINDEX**	Trying to access a variable with an unknown SOPAS index
0x0004	**LOCALCONDITIONFAILED**	Local condition violated, e.g. giving a value that exceeds the minimum or maximum allowed value for this variable
0x0005	**INVALID_DATA**	Invalid data given for variable.
0x0006	**UNKNOWN_ERROR**	An error with unknown reason occurred.
0x0007	**BUFFER_OVERFLOW**	The communication buffer was too small for the amount of data that should be serialized
0x0008	**BUFFER_UNDERFLOW**	More data was expected, the allocated buffer could not be filled.
0x0009	**ERROR_UNKNOWN_TYPE**	The variable that shall be serialized has an unknown type. This can only happen when there are variables in the firmware of the device that do not exist in the released description of the device.
0x000A	**VARIABLE_WRITE_ACCESSDENIED**	It is not allowed to write values to this variable. Possibly the variable is defined as read-only
0x000B	**UNKNOWN_CMD_FOR_NAMESERVER**	When using names instead of indices, a command was issued that the nameserver does not understand

0x000C	**UNKNOWN_COLA_COMMAND**	The CoLa protocol specification does not define the given command, command is unknown
0x000D	**METHODIN_SERVER_BUSY**	It is not possible to issue more than one method at a time to an SRT device.
0x000E	**FLEX_OUT_OF_BOUNDS**	An array was accessed exceeding its maximum length
0x000F	**EVENTREG_UNKNOWNINDEX**	The index of an Event is unknown.
0x0010	**COLA_A_VALUE_OVERFLOW**	The value does not fit into the value field, it is too large
0x0011	**COLA_A_INVALID_CHARACTER**	Character is unknown, probably not alphanumeric
0x0012	**OSAI_NO_MESSAGE**	Only when using SRTOS in the firmware and distributed variables this error can occur. It is an indication that no operating system message could be created. This happens when trying to GET a variable.
0x0013	**OSAI_NO_ANSWER_MESSAGE**	This is the same as SOPAS_Error_OSAI_NO_MESSAGE with the difference that it is thrown when trying to PUT a variable.
0x0014	**INTERNAL**	Internal error in the firmware.
0x0015	**HubAddressCorrupted**	The SOPAS Hubaddress is either too short or too long.
0x0016	**HubAddressDecoding**	The SOPAS Hubaddress is invalid, it cannot be decoded (Syntax)
0x0017	**HubAddressAddressExceeded**	Too many hubs in the address
0x0018	**HubAddressBlankExpected**	When parsing a HubAddress an expected blank was not found. The HubAddress is not valid
0x0019	**AsyncMethodsAreSuppressed**	An asynchronous method call was made although the device was built with "AsyncMethodsSuppressed".
0x001A	**Reserved**	
…		
0x0020		
0x0020	**ComplexArraysNotSupported**	Device was built with "ComplexArraysSuppressed" because the compiler does not allow recursions. But now a complex array was found.
0x021	**SESSION_NORESOURCES**	A Session cannot be created because the server ran out of resources (too many clients connected)

0x022	**SESSION_UNKNOWNID**	Unknown Session ID in telegram header
0x023	**CANNOT_CONNECT**	Remote Connection ("J" command) cannot be created.
0x024	**InvalidPortId**	The given PortId is unknown to the server.
0x025	**ScanAlreadyActive**	An additional scan command was issued although a scan is already running.
0x026	**OutOfTimers**	The server cannot create a timer for SopasScan.
0x0027	**Reserved**	
...		
0x003F		

7.3. Serialization of Data Types

7.3.1. Integer Types

Integer types are USInt, Uint, UDInt, ULInt, Sint, Int, Dint and Lint.

Serialization with CoLa 2.0

Integer types are USInt, Uint, UDInt, Sint, Int, Dint, ULInt and Lint.

The serialization in CoLa 2.0 corresponds to the big Endian byte order for storing integers (Motorola format, big endianess, high byte first).

The serialization in CoLa 2.0 Intel corresponds to the little Endian byte order for storing integers (Intel format, little endianess, low byte first).

Definition and examples

Definition:

Byte Order	Serialization
Big Endian	[<most sign.byte>, ...] <least sign.byte>
Little Endian	[<least sign.byte>, ...] <most sign.byte>

Examples:

Byte Order	Type and Value of example	Serialization
Big Endian	USInt 24	<0x18>
	Sint -11	<0xF5>
	Uint 291	<0x1><0x23>
	UDInt 159357	<0x0><0x2><0x6E><0x7D>
	Lint -786	<0xFF><0xFF><0xFF><0xFF><0xFF><0xFF><0xFC><0xEE>
Big Endian	USInt 24	<0x18>
	Sint -11	<0xF5>
	Uint 291	<0x23><0x1>
	UDInt 159357	<0x7D><0x6E><0x2><0x0>
	Lint -786	<0xEE><0xFC><0xFF><0xFF><0xFF><0xFF><0xFF><0xFF>

7.3.2. Floating Point Types

Floating point types are Real and Lreal.

The serialization in CoLa 2.0 corresponds to the big Endian byte order for storing floating point types (Motorola format, big endianess, high byte first).

The serialization in CoLa 2.0 Intel corresponds to the little Endian byte order for storing floating point types (Intel format, little endianess, low byte first).

7.3.3. Boolean

The SOPAS Basic Type Bool is serialized as a USInt. The following values are applied:

Meaning	**Value**	**Representation**
True	0x01	0x01
False	0x00	0x00

7.3.4. Structures

Structures are serialized by concatenating the serialization of all members in order of definition beginning with the very top one.

Definition:

Serialization

(serialization of 1st member)(serialization of 2nd member) ... (serialization of last member)

Example:

Value of example	Serialization
Struct { USInt Mode ; Sint X ; USInt Y ; } MyVal = { 3 , 0xA5 , 0x53 }	<0x03><0xA5><0x53>

7.3.5. Arrays

FixArrays

Arrays are serialized by concatenating the serialization of all members in order of the index beginning with the index 0.

Definition:

Serialization
(serialization of data[0])(serialization of data[1]) ... (serialization of data[n-1])
In which n is the length of the array

Example:

Value of example	Serialization
Sint MyVal[3] = { 3 , 0xA5 , 0x53 }	<3><0xA5><0x53>

FlexArrays

Flexible arrays ("FlexArray" / attribute FixedLength=False) are serialized as fixed arrays (see above) after the serialization of the current length as Uint.

There is also the possibility for FlexArrays to have the length as a 32-Bit UDInt (Attribute LengthWidth="32"). Through the course of this document we will call these longer FlexArrays **DflexArrays.**

Definition:

Serialization
(serialization of current length as Uint or UDInt)(serialization of data[0])(serialization of data[1]) ... (serialization of data[m-1])
In which m is the current length of the array

Example:

Byte Order	Value of example	Serialization
Big Endian	Sint MyVal[] = { 5 , 0xA5 , 0x53 }	<0><3><5><0xA5><0x53>
Little Endian		<3><0><5><0xA5><0x53>

7.3.6. Xbyte (deprecated!)

Xbytes are not used (by the developer) anymore. But there are still legacy variables (like StandardInfo) that contain Xbytes. Therefore Xbytes are also described.

Bitset types (Byte, Word, Dword, Lword and Xbyte) are serialized as an array of USInt. The bit order of Bitset members is described in the separate document "Xbyte Serialisierung".

Example:

Dialect	Value of example	Serialization
	Struct { USInt Mode **: 4** ; USInt X **: 8** ; USInt Y **: 8** ; } MyVal = { 3 , 0xA5 , 0x53 }	
CoLa 2.0		<0x53><0x3A><0x05>
CoLa 2.0 Intel		<0x53><0x3A><0x05>

General remarks to the Xbyte types
Due to the incompatibility of Xbytes with the IO-Link specification, these types are deprecated. See new definition of UxCnt (bit definition containers) in the extension CoLa 2.0
These types are also not supported by LWSRT and SRT++

7.3.7. Cont Types

The types Scont, Cont, Dcont and Lcont replace the former Intel format oriented Xbytes (Byte, Word, Dword, Lword).

These new types are serialized and treated equal to their corresponding unsigned integer type (see table below). The only difference is, that the meaning of the several bits in the integer type are described in detail in the CID and that macros are generated to set and extract the member bits out of the integer type.

Type	**Width**	**Treated and serialized like**
Scont	**8 bit**	**USInt**
Cont	**16 bit**	**Uint**
DCnt	**32 bit**	**UDInt**
LCont	**64 bit**	**ULInt**

7.3.8. Strings

Strings are serialized as they are.

Since CoLa 2.x doesn't use the STx/ETx framing anymore, all characters are allowed.

With the definition of the String in CID, the attribute Coding specifies whether 8 bit ASCII (ISO-8859-15) characters are used or UTF-8.

When ISO-8859-15 coding is used, the former restriction is still given: Only characters between 0x04…0xFF are allowed.

FixString

A FixString (attribute FixedLength=True or not specified) is always serialized in the whole length (fixed number of characters).

Definition:

Serialization
(String «as is»)

Example:

Value of example	Serialization
Char s[] = "Hello"	<'H'><'e'><'l'><'l'><'o'>

Note: Char is the SOPAS data type for a 1 Byte Character.

FlexString

As FlexArrays, the FlexString is serialized with a leading serialization of its length as UInt.

Example:

The String "Hello" shall be serialized

Definition:

CoLa 2.0	(serialization of current length as UInt) (String «as is»)
CoLa 2.0 Intel	(serialization of current length as UInt) (String «as is»)

Example:

Byte order	Value of example (pseudocode)	Serialization

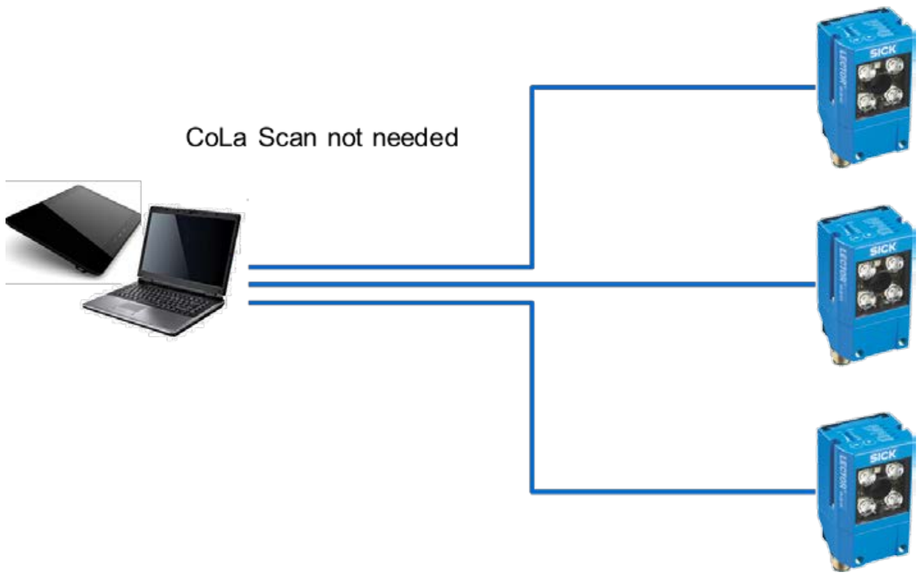
Big endian	Struct {
	  UInt uiLength = 5;
	  aCharData s[] = "Hello";
	}FlexString;
		<0x00><0x05><'H'><'e'><'l'><'l'><'o'>

Little endian		<0x05><0x00><'H'><'e'><'l'><'l'><'o'>

8. CoLa Scan

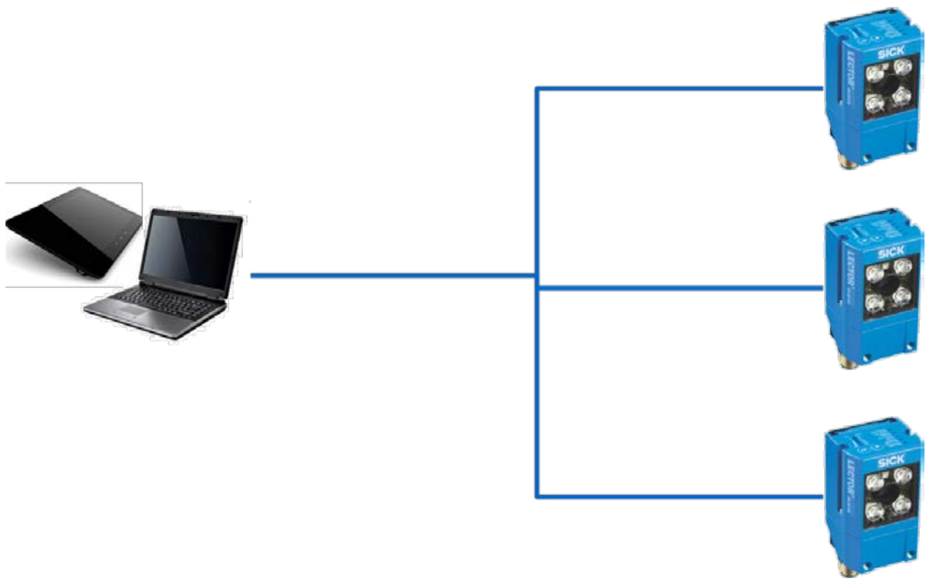
The CoLa Scan defines how to find CoLa devices in a network infrastructure within a few seconds.

The following figure shows a scenario where devices are connected to a notebook/tablet using different ports (e.g. USB ports).



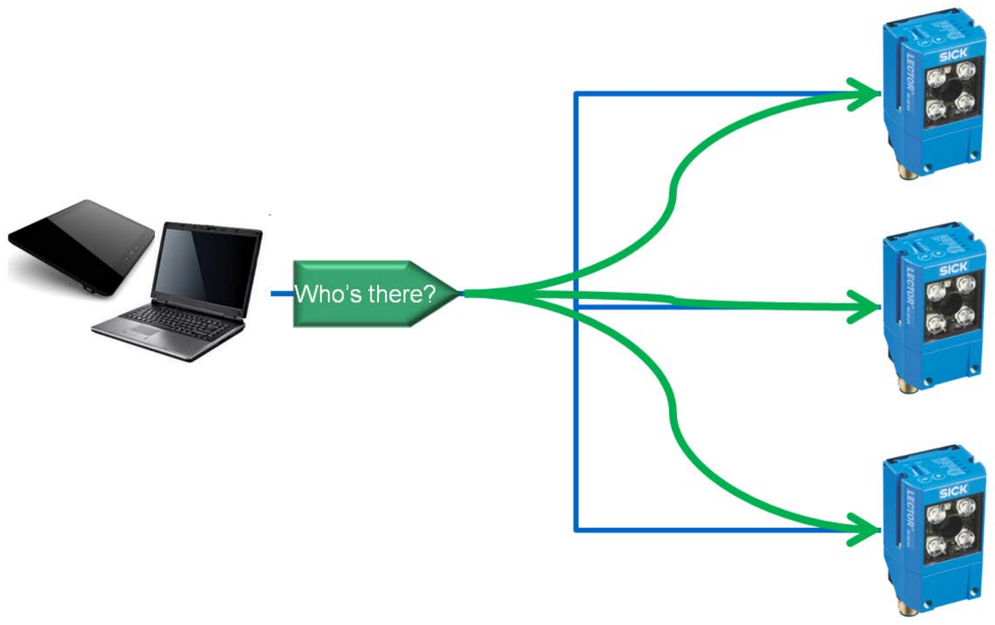
In this case the functionality of a CoLa Scan is not helpful.

The next figure shows a scenario where devices are connected to a notebook/tablet using one network (e.g. WiFi or standard Ethernet).



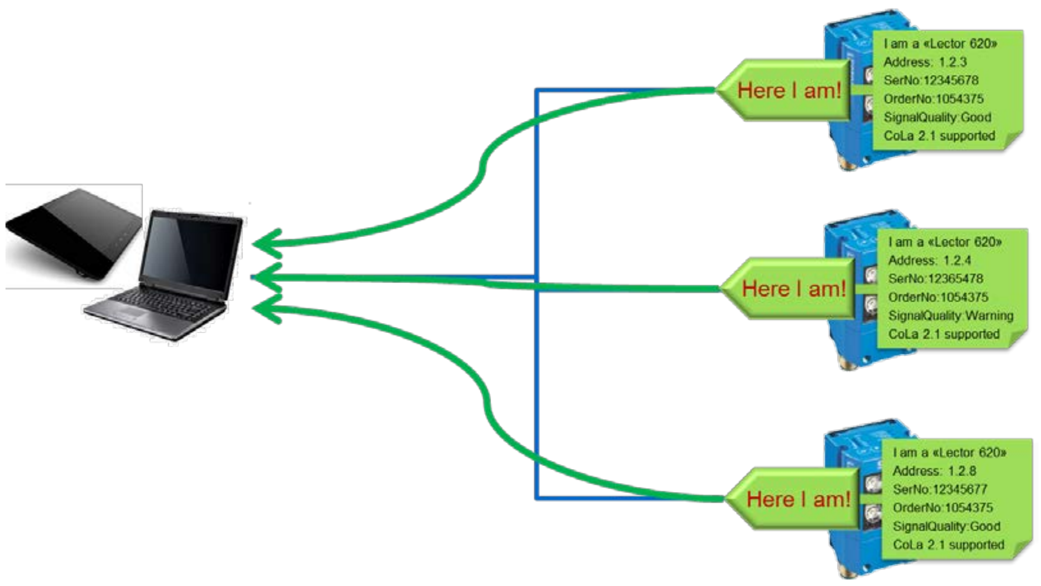
In this case the CoLa Scan defines a broadcast command which lets all devices answer with all necessary information which is needed to identify and connect the device (e.g. communication settings).

The searching client sends a broadcast message (“Who’s there”) to all devices:



This causes the devices to answer with standardized information containing

- Identification
- Basic Diagnosis
- Infrastructural Information (other communication ports)
- Communication settings (regarding the used port)



8.1. UDP Telegram

UDP broadcast and unicast telegrams are only used for CoLa Scan functionality. A Reason therefore is that the devices might not be configured well to be able to communicate in the Ethernet network.

The server (SICK device) is listening to UDP Port 30718 for incoming commands. It needs to listen to unicast telegrams, limited broadcasts as well as directed broadcasts, even if they are addressed to a different subnet.

Listening UDP port for searching devices
Port 30718

The default port for sending UDP requests is 30719. If this port is blocked by another application the next port should be taken. This mechanism is repeated until port 30738 is reached.

Send UDP port range for searching devices
Port 30719 – Port 30738

Note:

The server should bind to port 30718 and IP address 0.0.0.0 to receive limited and directed broadcasts on that port. The reason therefore is that Linux systems are not able to send limited broadcasts to all Ethernet interfaces.

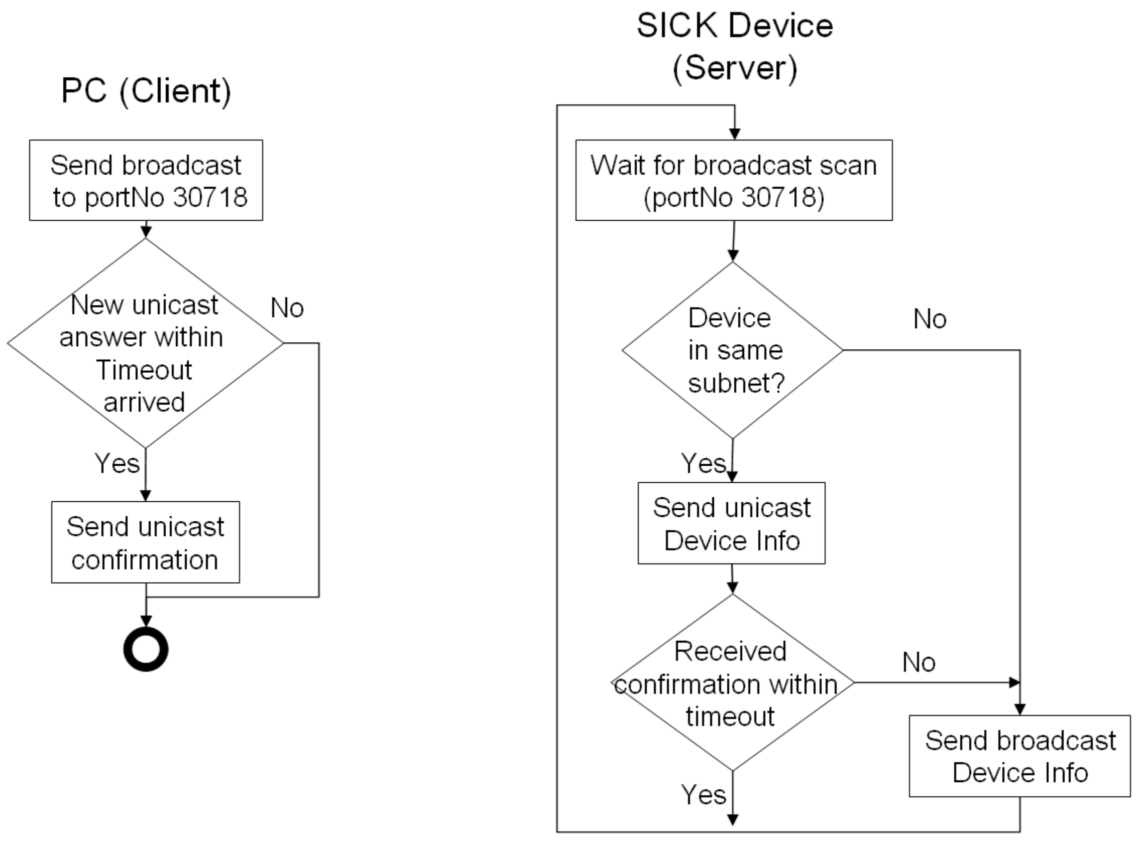
When client and server are in different subnets (IP address/ network mask does not match) the only way to communicate is a UDP broadcast. In this case all CoLa Scan commands use broadcast messages.

When client and server are in the same subnet already (checked by the server), the scan reply is a unicast message. The Client confirms this message within the timeout of 500 milliseconds. After that all CoLa Scan commands use unicast messages. A flow diagram describing the behaviour of client and server is shown in the following illustration.

In incorrect configured networks (e.g. multiple devices with the same IP address because the network was built with out of the box devices) it can happen that unicast communication does not work probably. Due to the fact that only the client is able to notice that it is the job of the client to switch to broadcast communication when devices found at a scan do not answer to ongoing UDP communication (CMD_IPCONFIG or CMD_FINDME).

Note:

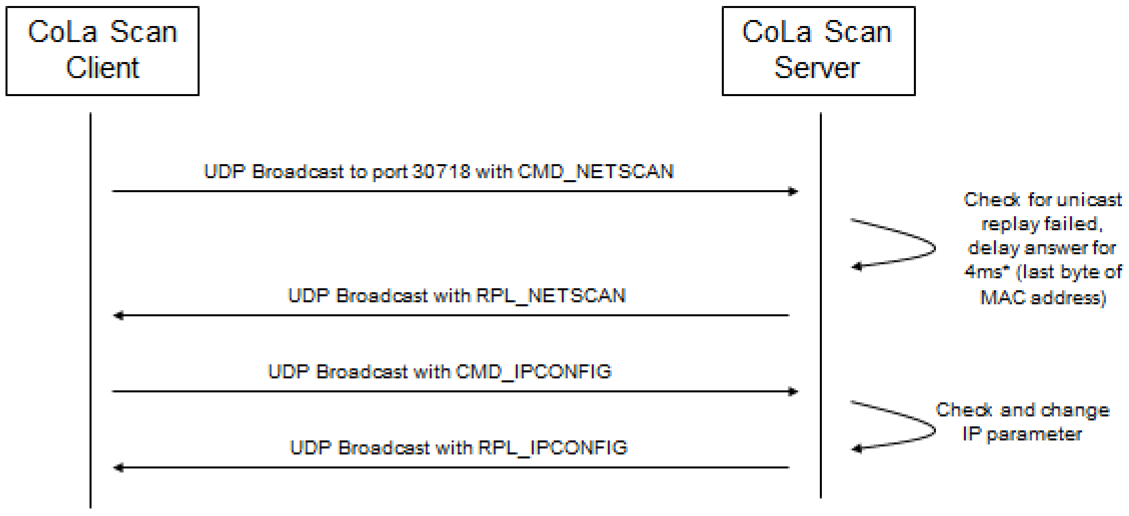
This procedure helps to reduce broadcast messages in the system.


PC (Client)
Send broadcast to portNo 30718
New unicast answer within Timeout arrived
No
Yes
Send unicast confirmation
SICK Device (Server)
Wait for broadcast scan (portNo 30718)
Device in same subnet?
No
Yes
Send unicast Device Info
Received confirmation within timeout
No
Yes
Send broadcast Device Info

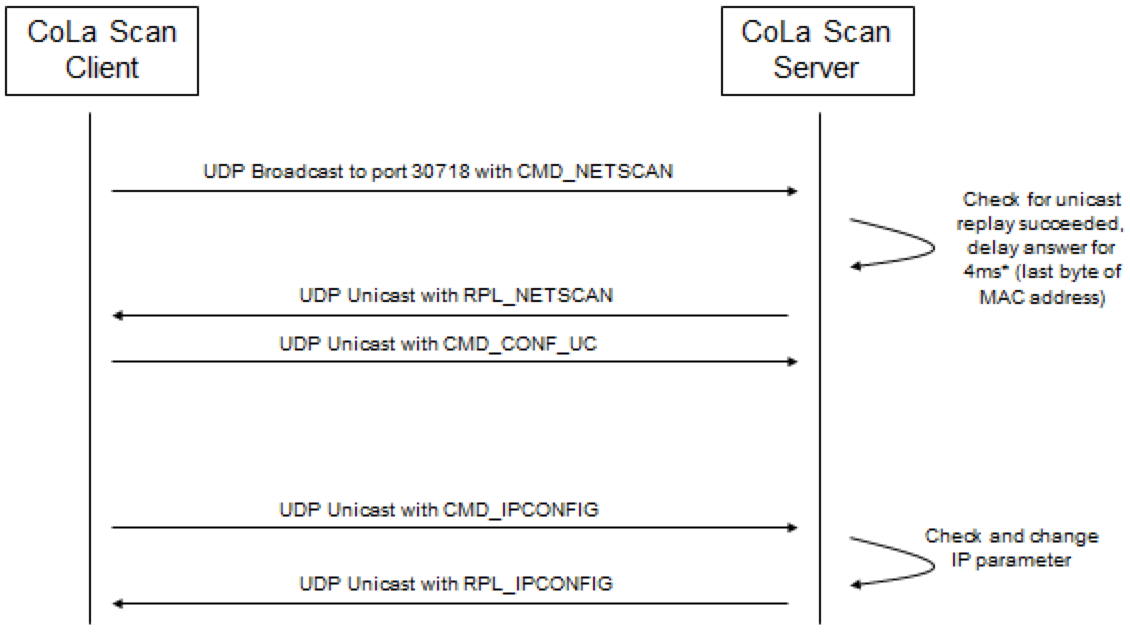
Flow diagram: How to handle broadcast and unicast telegrams by client and server

Note:

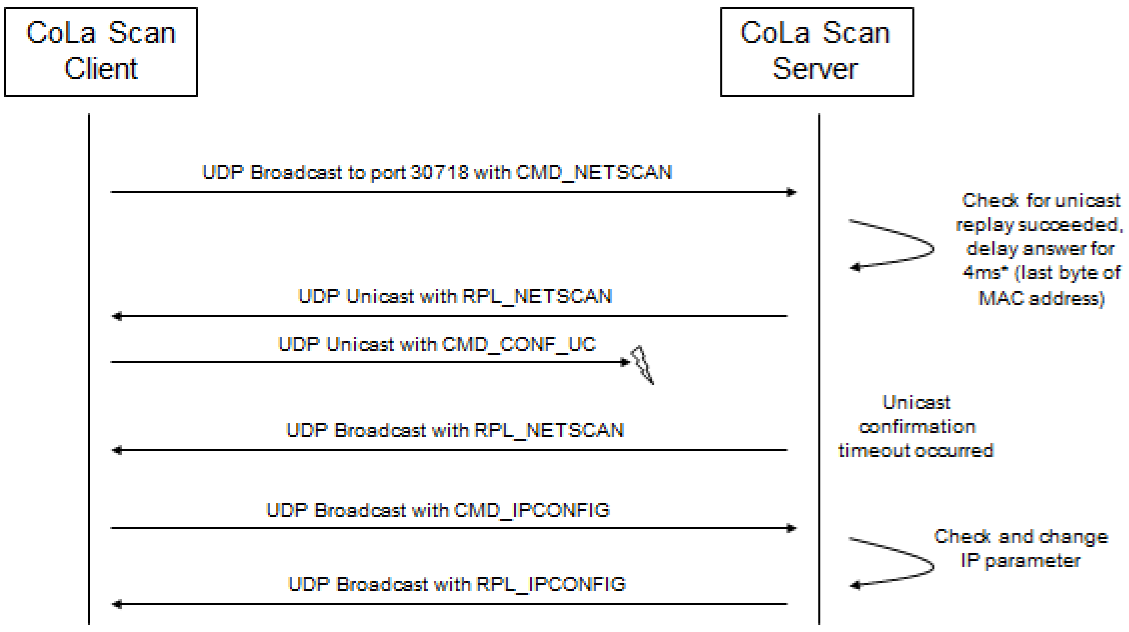
When the client confirmed the unicast scan reply all following UDP communication uses unicast telegrams. To simplify the mechanism in the server all broadcast requests (excepting the scan command) will be answered with broadcast and all unicast requests will be answered with unicast.


CoLa Scan Client
CoLa Scan Server
UDP Broadcast to port 30718 with CMD_NETSCAN
Check for unicast replay failed, delay answer for 4ms* (last byte of MAC address)
UDP Broadcast with RPL_NETSCAN
UDP Broadcast with CMD_IPCONFIG
Check and change IP parameter
UDP Broadcast with RPL_IPCONFIG

Sequence diagram: Scan to server in different subnet followed by CoLa request


CoLa Scan Client
CoLa Scan Server
UDP Broadcast to port 30718 with CMD_NETSCAN
Check for unicast replay succeeded, delay answer for 4ms* (last byte of MAC address)
UDP Unicast with RPL_NETSCAN
UDP Unicast with CMD_CONF_UC
UDP Unicast with CMD_IPCONFIG
Check and change IP parameter
UDP Unicast with RPL_IPCONFIG

Sequence diagram: Scan to server in same subnet followed by CoLa request


CoLa Scan Client
CoLa Scan Server
UDP Broadcast to port 30718 with CMD_NETSCAN
Check for unicast replay succeeded, delay answer for 4ms* (last byte of MAC address)
UDP Unicast with RPL_NETSCAN
UDP Unicast with CMD_CONF_UC
Unicast confirmation timeout occurred
UDP Broadcast with RPL_NETSCAN
UDP Broadcast with CMD_IPCONFIG
Check and change IP parameter
UDP Broadcast with RPL_IPCONFIG

Sequence diagram: Scan to server in same subnet followed by CoLa request. Unicast confirmation gets lost

The network load would be high (e.g. for the limitation of the broadcast bandwidth at the switches) by every SICK device try to answer at the same time. Therefore every device answers with a unique delay. Every device calculates a random reply delay time between 0 – 1020 milliseconds. The random reply is calculated using the last byte of the MAC address, e.g. a MAC address of xx:xx:xx:xx:xx:7F will result in a delay of 127*4 ms = 508 ms (max reply delay is 1020 ms.

8.2. Scan

This chapter describes the mechanism, how a TCP/IP host is detected and configured via CoLa Scan, independent of its actual IP configuration. This mechanism is used by the client (e.g. SOPAS Engineering Tool) to detect SICK devices in the network and change their IP configuration. This Scan mechanism reliably works even if the IP address of the device is unknown or the device is not reachable by an ordinary TCP connection, as it is configured for a different subnet.

CoLa Scan has two steps: first the PC scans the connected network and then, if needed, changes the IP configuration. CoLa Scan works with UDP broadcasts (scan) and unicasts (scan result, IP assignment and FindME, only if device is in same subnet) using port 30718 (server listen always on that port, on client side this port can vary, the devices must be able to answer to the port the request came from) to read and set the IP configuration. Therefore extensions in the application layer of the SICK device communication stack are necessary. IP configuration in a TCP/IP network requires at least three basic elements at the host: a unique IP address in its network, a subnet mask and a default gateway address. Before assigning a new IP address to a SICK sensor it has to be ensured that the IP address is not yet in use.

Needed SICK device features for CoLa Scan:

- Timer functionality, with 1 ms accuracy for RPL_NETSCAN in time replay
- UDP listener on port 30718 to receive and send messages to the sender port 30719
- A valid MAC address for each device

Note:
All telegrams need to fit into the MTU.

8.2.1. Addressing with UDP Telegrams

For addressing a single device with a broadcast message uses the following telegram structure within the UDP telegram:

Note:

Due to the fact that the client does not have any information about the devices it finds in the network to this point of time all communication related to the scan mechanism will take place in Big Endian format.

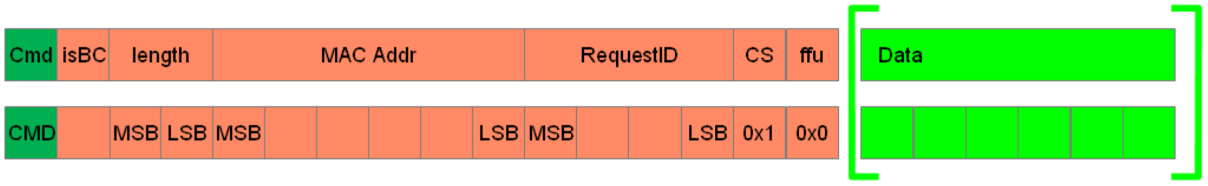


Illustration: Structure of the UDP Scan telegram

Field	Description
CMD/ RPL	Telegram token (PC sends only commands, the hosts only replies)
isBC	Indicates weather telegram was sent as UDP broadcast or UDP unicast telegram (0x00 means Unicast, 0x01 means Broadcast)
length	Length of data area
MAC Addr	MAC Address of the host. Because telegrams sent by the client might be UDP broadcast and received by any host, there must be a unique identification of the host. At an application broadcast (net scan) the MAC Address is set to 0xFFFFFFFFFFFF
RequestID	Request identification. The client sends out the command of each connected interface, so the server (SICK device) could receive the same command several times and creates to each received telegram an answer. To avoid this, the client creates a 32 bit random number for the RequestID for the outgoing telegram. The server compares the RequestID of the recent telegram with the RequestID of the previous telegram, if these both IDs are equal the server discard the recent telegram, otherwise he executes the telegram and replies with the corresponding RequestID. Each new command must contain a new RequestID, either the old incremented by one or a new random number
CS	Indicates that telegram is a CoLa Scan telegram
ffu	for future use

Data	Command specific data area which keeps the detailed main information

8.2.2. Commands

Message	Identifier	Description
CMD_NETSCAN	0x10	sent by client; to initiate any SICK host in the network to answer
CMD_CONF_UC	0x13	sent by client; to confirm unicast communication
reserved	0x90	Reserved for compatibility to AutoIP Scan
CMD_IPCONFIG	0x11	sent by client; request to change IP configuration
RPL_IPCONFIG	0x91	replied by sensor; confirmation to IP change
CMD_FINDME	0x12	Sent by client; request to device for sending out an visual or acoustic signal
RPL_FINDME	0x92	replied by sensor; confirmation to CMD_FINDME
RPL_NETSCAN	0x95	replied by sensors; with information like device name, serial number, IP conf.,

CMD_NETSCAN

The client sends an UDP broadcast with the command identifier 0x10. The following illustration shows the structure of the netscan command:

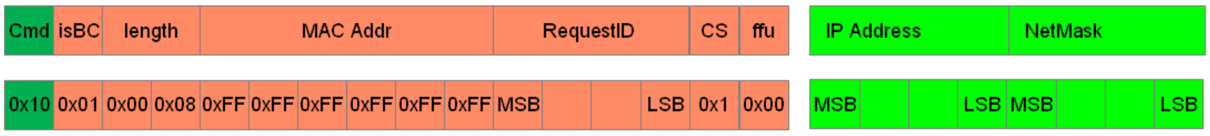
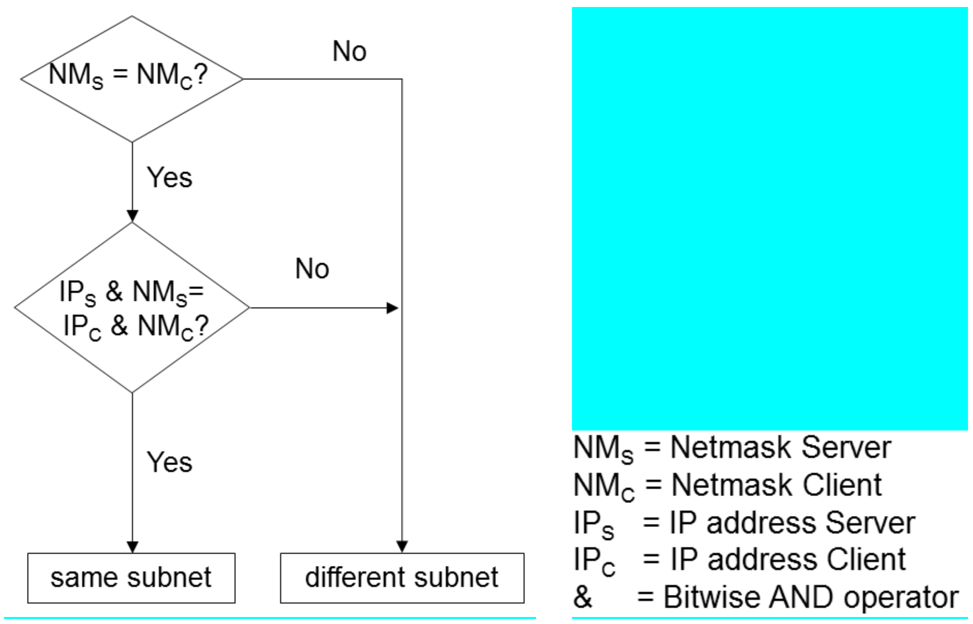


Illustration: Structure of the CMD_NETSCAN telegram

With this CMD_NETSCAN the server receives the IP address and the network mask of the client. This information is needed to check weather client and server are in the same subnet and communication via unicast messages is possible

This check is done by following rule:



Flow diagram: Server checks whether client is in the same subnet or not

RPL_NETSCAN

Any server in the network replies as follows:

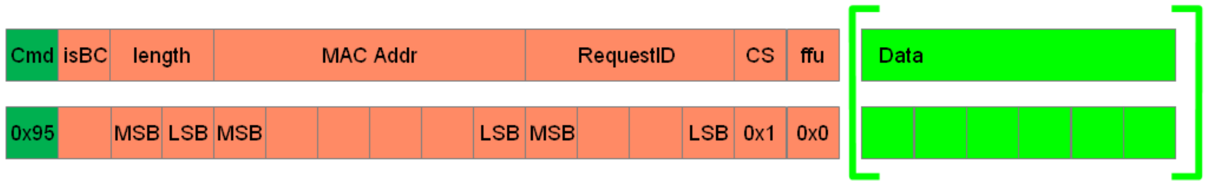


Illustration: Structure of the RPL_NETSCAN reply

The network load would be high (e.g. for the limitation of the broadcast bandwidth at the switches) by every SICK device try to answer at the same time. Therefore every device answers with a unique delay. Every device calculates a random reply delay time between 0 – 1020 milliseconds. The random reply is calculated using the last byte of the MAC address, e.g. a MAC address of xx:xx:xx:xx:xx:7F will result in a delay of 127 * 4 ms = 508 ms (max reply delay is 1024 ms). At multihomed hosts the max reply delay time has to be multiplied by the number of Ethernet interfaces (n), because it's possible that the device receives the NetScan command n times and wants to answer each with the time delay.

The 'Data' field of the reply keeps important device information. The data is arranged in the following structure:

```
Struct DeviceInfo[
        FlexString32 CidName;
        Struct CidVersion[
                UInt MajVersion;
                UInt MinVersion;
                UInt MicroVersion;]
        Enum8 DeviceState;
        FlexString32 ApplicationSpecificName;
        FlexString12 SerialNumber;
        FlexString24 TypeCode;
        FlexString24 FirmwareVersion;
        UDInt OrderNumber;
        SCont Flags;
        FlexArray128 auxEntries[
                FixString4 Key;
                FlexArray32 Value[USInt];];
        FlexArray32 ScanIF[
                UInt InterfaceNumber;
                FlexString16];
        FlexArray64 generalComSettings[
                FixString4 Key;
                FlexArray32 Value[USInt];];
        FlexArray8 EndpointSettings[
                Enum8 Protocol;
                FlexArray64 generalComSettings[
                        FixString4 Key;
                        FlexArray32 Value[USInt];];];
]
```

An example is given in the following illustration. All values and the corresponding attributes are listed in the table afterwards.

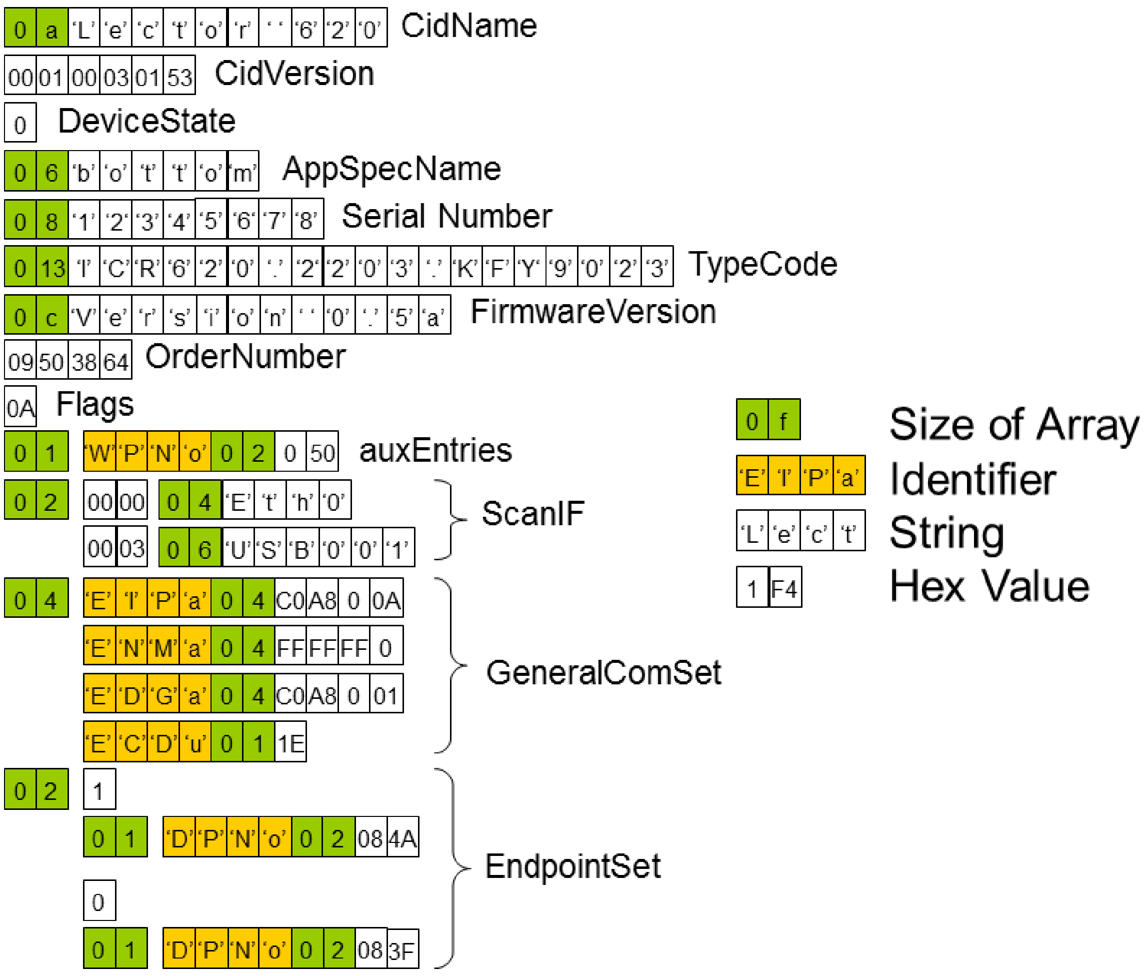

0 a 'L' 'e' 'c' 't' 'o' 'r' ' ' '6' '2' '0' CidName
00 01 00 03 01 53 CidVersion
0 DeviceState
0 6 'b' 'o' 't' 't' 'o' 'm' AppSpecName
0 8 '1' '2' '3' '4' '5' '6' '7' '8' Serial Number
0 13 'I' 'C' 'R' '6' '2' '0' '.' '2' '2' '0' '3' '.' 'K' 'F' 'Y' '9' '0' '2' '3' TypeCode
0 c 'V' 'e' 'r' 's' 'i' 'o' 'n' ' ' '0' '.' '5' 'a' FirmwareVersion
09 50 38 64 OrderNumber
0A Flags
0 1 'W' 'P' 'N' 'o' 0 2 0 50 auxEntries
0 2 00 00 0 4 'E' 't' 'h' '0'
00 03 0 6 'U' 'S' 'B' '0' '0' '1' ScanIF
0 4 'E' 'I' 'P' 'a' 0 4 C0 A8 0 0A
'E' 'N' 'M' 'a' 0 4 FF FF FF 0
'E' 'D' 'G' 'a' 0 4 C0 A8 0 01
'E' 'C' 'D' 'u' 0 1 1E GeneralComSet
0 2 1
0 1 'D' 'P' 'N' 'o' 0 2 08 4A
0
0 1 'D' 'P' 'N' 'o' 0 2 08 3F EndpointSet
0 f Size of Array
'E' 'I' 'P' 'a' Identifier
'L' 'e' 'c' 't' String
1 F4 Hex Value

Illustration: Simplified example for a Device Info

Content	Remark Example	Type	
Fix DeviceInfo Structure			
DeviceInfoVersion	Version of the Device Info	UInt16	m
CidName	Name of device (corresponding to DeviceIdent.Name) e.g. "Lector 62x"	FlexString32, ISO 8859-15	m

CidVersion	Version of the CID
	e.g. {1,4,0}
	CoLaB variable DeviceIdent.Version is generated out of this entry if available
		Struct {
		UInt MajorVersion,
		UInt MinorVersion,
		UInt PatchVersion,
		UDInt BuildNumber,
		Enum8 VersionClassifier
		}
			m

Content	Remark Example	Type	
DeviceState	Choice: "*DS_UnknownState*", "*DS_Startup*", "*DS_ServiceMode*", "*DS_NormalOperation*", "*DS_SuspendedOperation*", "*DS_ServiceRecommended*", "*DS_ServiceRequired*", "*DS_RecoverableError*" ,"*DS_FatalError*"	Enum8	m

RequiredUserAction	Bit field where required actions to get the device running in operational mode are shown	Cont {
		BoolX ConfigureDevice
		BoolX CheckConfiguration
		BoolXCheckEnvironment
		BoolX CheckApplicationInterfaces
		BoolX CheckDevice
		}
			m

DeviceName	corresponding to Device Name (set by user)	FlexString32 ISO 8859-15	m
ApplicationSpecificName	corresponding to LocationName (set by user)	FlexString32 ISO 8859-15	m
ProjectName	corresponding to ProjectName (set by user)	FlexString32 ISO 8859-15	m
SerialNumber	Serial number of device	FlexString32 ISO 8859-15	m
TypeCode	Type code of the device	FlexString32 ISO 8859-15	m
FirmwareVersion	Firmware Version of the device (corresponding to FirmwareVersion)	FlexString32 ISO 8859-15	m
OrderNumber	corresponding to OrderNumber	FlexString32 ISO 8859-15	m
Flags	Flag field containing information about Endianess, Communication and Sdd-Upload Bit0: IsLittleEndian Bit1: Communication by Index available Bit2: Communication by Name available Bit3: Sdd Upload supported Bit 4: Supports Challenge Response for login Bit 5: Supports TLS encryption Bit 6-7: reserved	SCont	m

Content	Remark Example	Type	
Extensions of the structure (generic key value pairs)			

Aux Entries	All possible entries are listed in the table Aux Entries.
	The inner FlexArray has a maximum length of 32 entries. The data type of the entries is USInt
		FlexArray128[
		Char key[4];
		FlexArray32;
		]
			m

Communication properties			
Scan IF	List of interfaces which can be scanned. The InterfaceNumber is indicating which Interface should be scanned at a remote scan. When choosing 0xFFFF all interfaces are scanned. The Interface the device info is requested from is excluded from the list and also not considered for remote scan to all interfaces.	FlexArray32[UINT; FlexString64*;]	m

General Com Settings	All possible entries are listed in the table General Com Settings Ethernet respectively USB.
	The inner FlexArray has a maximum length of 32 entries. The data type of the entries is USInt
		FlexArray64[
		Char key[4];
		FlexArray32;
		]
			m

Endpoints	All possible entries are listed in the table Endpoints.
	The inner FlexArray has a maximum length of 32 entries. The data type of the entries is USInt
		FlexArray16[
		Enum8;
		FlexArray64[
		Char key[4];
		FlexArray32;
		]
			m

Table: Device Info structure

m: mandatory

c: conditional, only mandatory if feature is supported;

o: Optional

*: All FlexStrings are coded in ISO 8859-15

Aux Entries	Identifier	Remark Example	Type	
WebServerPortNo	"WPNo"	Port number of web server If not supported this entry is not available	UInt, Big Endian	o

Table: Aux Entries

General Com Settings - Ethernet	Identifier	Remark Example	Type	

General Com Settings - Ethernet	**Identifier**	**Remark Example**	**Type**	
IpAddress	“EIPa”	IP Address of device	UDInt, Big Endian	c
SubnetMask	“ENMa”	Subnet mask of device	UDInt, Big Endian	c
DefaultGateway	“EDGa”	Default gateway of device	UDInt, Big Endian	c
UseDHCP	“EDhc”	<0x01> DHCP activated If entry is not available DHCP is not supported	USInt	c
MacAddress	“EMAC”	MAC address of Device	USInt[6]	c
IpConfigDuration	“ECDu”	Time the device need after reconfiguration in seconds	UDInt, Big Endian The highword of the duration is reserved and must be zero	c
ZeroConf	“EZeC”	<0x01> Zeroconf supported	USInt	c

Table: General Com Settings for Ethernet

General Com Settings - USB	**Identifier**	**Default**	**Type**	
Input Pipe short packet terminate	“UISP”	“False”	Bool	c
Output Pipe short packet terminate	“UOSP”	“False”	Bool	c
Input Pipe Transfer timeout	“UIPT”	n.a.	UDInt, Big Endian	c
Output Pipe Transfer timeout	“UOPT”	10000	UDInt, Big Endian	c

Table: General Com Settings for USB

Endpoints	**Identifier**	**Remark Example**	**Type**	
Protocol		Choice: “CoLaB”, “CoLa2_0”, “CoLa2_1”	Enum8	m
PortNumber	“DPNo”		UInt, Big Endian	c

Table: Endpoints

CMD_CONF_UC

The client sends the telegram after receiving a unicast net scan reply to signalize to the server that unicast communication is accepted.

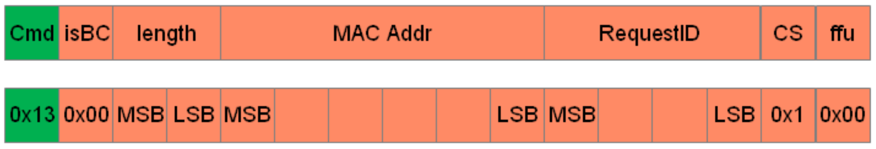


Illustration: Structure of the CMD_CONF_UC command

CMD_IPCONFIG

The client sends the telegram:

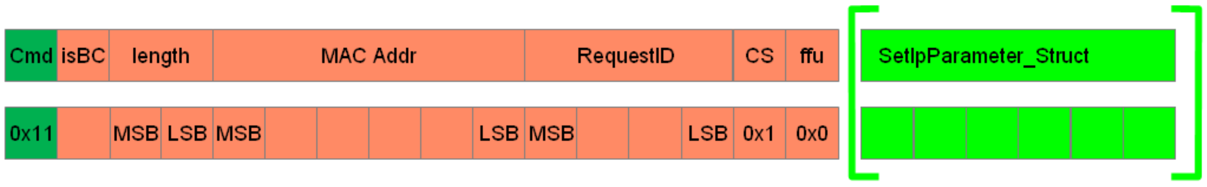


Illustration: Structure of the CMD_IPCONFIG command

The ‘SetIpParameter_Struct’ field contains the IP settings as parameter serialized in Big Endian format. The structure is described in the next chapter.

RPL_IPCONFIG

The server replies as follows:

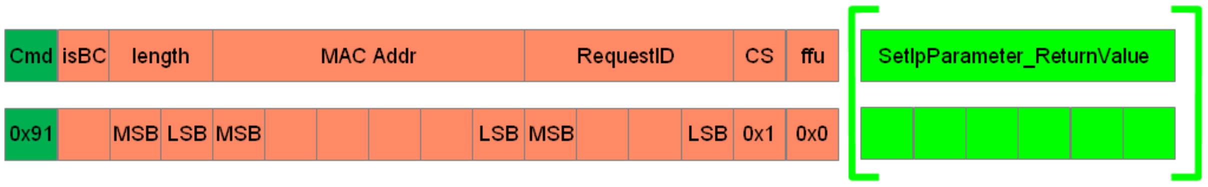


Illustration: Structure of the RPL_IPCONFIG response

The ‘SetIpParameter_ReturnValue’ field contains the return serialized in Big Endian format. The structure is described in the next chapter.

CMD_FINDME

The client sends the telegram:

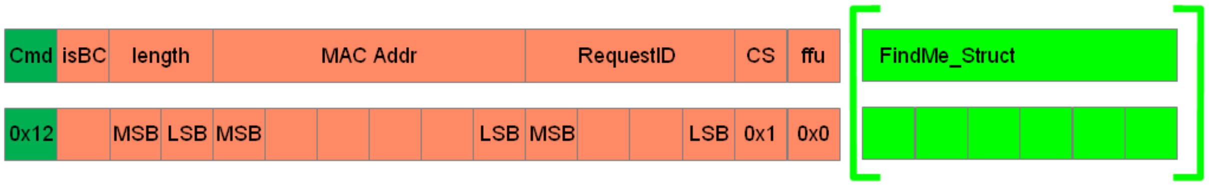
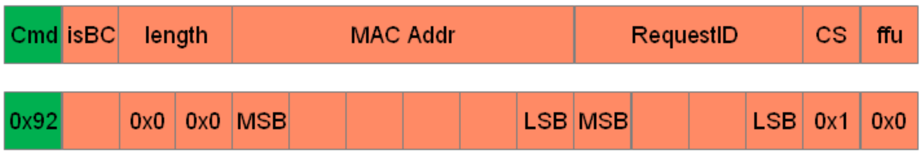


Illustration: Structure of the CMD_FINDME command

The ‘FindMe_Struct’ field contains the parameter of the FindMe method serialized in Big Endian format which are used to call the CoLa standard method FineMe described in the next chapter.

RPL_FINDME

The server replies as follows:



Cmd	isBC	length		MAC Addr						RequestID				CS	ffu
0x92		0x0	0x0	MSB					LSB	MSB			LSB	0x1	0x0

Illustration: Structure of the RPL_COLA response

8.3. Network Configuration

The configuration of an Ethernet Network can be described in two steps: First the client uses a network scan command, to find the SICK servers (devices) in the network and get their basic information. In a second step the client is able to send a CoLa command to the server. This command is limited on two functions: The FindMe() function to identify a SICK sensor by flashing a LED or an acoustic signal. The second function is the assignment of new IP parameter to be able to establish a TCP/IP connection.

8.3.1. IP Address Assignment

Assignment of IP parameter via UDP might be necessary when a sensor is located in a different subnet or is not configured at all.

This IP parameter for this configuration are defined as follows

Item	Method
Name	SetIpParameter
Parameter	Struct

UDInt	udiIpAddress
UDInt	udiSubnetMask
UDInt	udiDefaultGateway
Bool	bDHCPClientEnabled

ReturnValue	Struct:

UInt	uiErrorCode
UInt	uiIpConfigDuration in seconds
FlexString64	sUserText

Error Code	Description
0x00	No error
0x01	Device not ready
0x02	Invalid IP address
0x03	Invalid subnet mask
0x04	Invalid default gateway

0x05	Access denied
0xff	Unspecified error

During configuration the server must check the IP parameter for validity before using them. If the IP parameters are not valid an error has to be returned.

IP Address	Reason?
x.x.x.0	Invalid IP address
x.x.x.255	Broadcast addresses
127.0.0.0 – 127.0.0.255	Local host addresses
224.0.0.0 - 239.255.255.255	Multicast addresses

The subnet mask is defined as a sequence of ones followed by a sequence of zeros. All other values are invalid.

Example:	255.255.248.0	11111111 11111111 11111000 00000000	valid
	255.255.249.0	11111111 11111111 1111100**1** 00000000	invalid

8.3.2. FindMe Functionality

To identify a SICK sensor in a machine the CoLa standard method FindMe initiates an acoustic or visual signal for a defined period of time. The method is defined as follows:

Item	Method
Name	FindMe
CommunicationName	findMe
Parameter	UInt uiDuration in seconds
ReturnValue	Ø
UserLevel	RUN
Fixed SOPAS Index	0x0016 (22)

9. Timing Constraints

In CoLa 2.0 only the timing of a direct connection between the client and the device is specified in dependence of the used CoLa Transport Layer. These timings will be tested during conformance test procedure.

CoLa 2.1 will specify the timings in remote scenarios (using CoLa Hub routing and in off-shore scenarios.)

9.1. TCP/IP Timings

The following timing specification shall be treated as a test specification. Hence certain assumptions are made.

When directly connected to the device (no switch or equivalent infrastructure in between), the time between start of transmission of a request telegram and end of transmission of the response telegram is specified as listed in the table below:

Command	Max time @ 10MBit	Max time ≥ 100 MBit
Read Variable	1500ms	1000ms
Write Variable	2000ms	1500ms
Method Invocation[i]	1500ms	1000ms
Register Event	200ms	100ms
Upload	500ms	500ms
Download	500ms	500ms
Hello (Remote Scan)	20000ms	20000ms
Connect	200ms	200ms
Remote Connect	2000ms	2000ms

9.2. USB Timings

The following timing specification shall be treated as a test specification.

The time between start of transmission of a request telegram and end of transmission of the response telegram is specified as listed in the table below:

Command	Max time
Read Variable	1500ms
Write Variable	2000ms
Method Invocation[ii]	1500ms
Register Event	200ms
Upload	500ms
Download	500ms
Hello (Remote Scan)	20000ms
Connect	200ms
Remote Connect	2000ms

10. Annex

The annex contains information which are important but not part of the protocol specification.

10.1. Standard Interfaces

The following table gives an overview of all standard interfaces of CoLa 2.0 including the deprecated interfaces from CoLa 1:

Item	Name	CommunicationName (when given)	Fixed Index	Note
Variable	DeviceIdent		0x0000	deprecated
Variable	SOPASVersion		0x0001	deprecated
Variable	ApplicationSpecificName	AppNam	0x0002	LocationName
Variable	SerialNumber	~~SerNum~~	0x0003	
Variable	FirmwareVersion	~~FrmVer~~	0x0004	
Variable	ProcIndex		0x0005	no standard
Variable	SopasInfo		0x0006	deprecated
Variable	SubDevices		0x0007	deprecated
Variable	FirmwareDownloadAlgorithm		0x0008	deprecated
Variable	CIDChecksum	~~CidChk~~	0x0009	
Variable	AdditionalTimeout		0x000A	deprecated
Variable	CidName	~~CidNam~~		
Variable	CidVersion	CidVer	0x000B	
Variable	TypeCode	TypCod	0x000D	
Variable	OrderNumber	OrdNum	0x000E	
Variable	DeviceState	DevSta	0x000F	
Variable	RequiredUserAction		0x0010	
Variable	DeviceName		0x0011	
Variable	ProjectName		0x0012	
Method	SetAccessMode()		0x0000	deprecated
Method	GetDescription()		0x0003	deprecated
Method	SetPassword()	sPaswd	0x0004	
Method	CheckPassword()		0x0005	deprecated
Method	GetPDOConfigChks()		0x0006	specific usage
Method	GetItemDescriptor()		0x0007	specific usage
Method	ObtainAccess()		0x000C	only with ExclusiveAccess in CoLa 2.1
Method	ReleaseAccess()	rA	0x000D	only with ExclusiveAccess in CoLa 2.1
Method	FindMe()	findMe	0x000E	
Method	Ø		0x0010...0x001F	Not in use

10.1.1. Identification

The following chapter defines all mandatory items (Variables, Methods, Events) of a CoLa 2.x device.

10.1.1.1. CID Association

The associated CID (SDD) of a device is associated to one CID. The link

10.1.1.1.1. Cid Name

Item	Variable
Name	CidName
CommunicationName	CidNam
Type	FlexString [32] (ISO 8859-15)
Fixed SOPAS Index	0x000B (11)
Note	This name identifies the device or the device family. Devices with the same CidName shall have a very similar set of items in the CID. The content is equal to the former DeviceIdent.Name.

10.1.1.1.2. Cid Version

Item	Variable
Name	CidVer
CommunicationName	CidV
Type	Struct:

UInt	Major Version
UInt	Minor Version
UInt	Micro Version

Fixed SOPAS Index	0x000C (12)
Note	This variable differs between different versions of a CID. At least the least significant value “MicroVersion” shall change when any content of the CID changes. The content corresponds to the former DeviceIdent.Version.

10.1.1.1.3. CID Checksum

Item	Variable
Name	CidChecksum
CommunicationName	CidChk
Type	FixArray[16] of USInt

Fixed SOPAS Index	0x0009 (9)
Note	This variable's value is generated by the SRT generator. It assures the assignment of the right CID. The generator filters all information relevant for compatibility of the CID and calculates and MD5 hash value on it.

10.1.1.2. Extended Device Information

10.1.1.2.1. Serial Number

Item	Variable
Name	SerialNumber
CommunicationName	SerNum
Type	FlexString[32] (ISO 8859-15)
Fixed SOPAS Index	0x0003 (3)
Note	This variable's value is to identify an instance of a given type. The value is written during production of the device with a unique value within the device family (CidName). Hence when the value of CidName and SerialNumber is identical it must be the same instance of the device.

10.1.1.2.2. Application Specific Name

Item	Variable
Name	ApplicationSpecificName
CommunicationName	AppNam
Type	FlexString[32] (ISO 8859-15)
Fixed SOPAS Index	0x0002 (2)
Note	This variable's value is defined by customer. It identifies an instance of a device by the point of view of the application.

10.1.1.2.3. Firmware Version

Item	Variable
Name	FirmwareVersion
CommunicationName	FrmVer
Type	FlexString[32] (ISO 8859-15)
Fixed SOPAS Index	0x0004 (4)
Note	This variable's value signals the version of the given firmware.

The content is not comparable. (no greater or smaller operation available)

10.1.1.2.4. Type Code

Item	Variable
Name	TypeCode
CommunicationName	TypCod
Type	FlexString[24] (ISO 8859-15)
Fixed SOPAS Index	0x000D (13)
Note	This variable's value matches the SICK type code as it is used in SAP (first 18 characters). Example: "ICR620H-T11503"

10.1.1.2.5. Order Number

Item	Variable
Name	OrderNumber
CommunicationName	OrdNum
Type	UDInt
Fixed SOPAS Index	0x000E (14)
Note	This variable's value matches the SICK order number ("million number") in SAP. The value may be a constant but most devices will require to write the order number during production.

10.1.1.2.1. Device State

Item	Variable
Name	DeviceState
CommunicationName	DevSta
Type	Enum8
Fixed SOPAS Index	0x000F (15)
Note	

10.1.1.2.2. Required User Action

Item	Variable
Name	RequiredUserAction
CommunicationName	ReqAct

Type	Cont
Fixed SOPAS Index	0x0010 (16)
Note	

10.1.1.2.3. Device Name

Item	Variable
Name	DeviceName
CommunicationName	DevNam
Type	FlexString[32] (ISO 8859-15)
Fixed SOPAS Index	0x0011 (17)
Note	

10.1.1.2.4. Project Name

Item	Variable
Name	ProjectName
CommunicationName	PrjNam
Type	FlexString[32] (ISO 8859-15)
Fixed SOPAS Index	0x0012 (18)
Note	

10.1.2. Diagnosis

TBD

10.1.2.1. Device State

TBD

10.1.2.2. Error Log

TBD

10.1.3. Scan Functionality

10.1.3.1. FindMe

Item	Method
Name	FindMe

CommunicationName	findMe
Parameter	UInt uiDuration in seconds
ReturnValue	Ø
UserLevel	RUN
Fixed SOPAS Index	0x000E (14)

10.2. Miscellaneous

Design Decision

The following new features are designed into the CoLa 2.0 protocol:

1) Improved sub device routing (paths through gateways to different networks) will be available.
2) No version marker in the protocol is implemented. There is only one global version, which the device supports. This version can be determined by the client by evaluating the answer to the CoLa Scan.
3) Request ID: Requests and their answers always have the same Request ID. Answers can always be correctly matched to requests.
4) Explicit session management will be available in CoLa 2.0. A Session ID is provided in a CoLa 2.0 frame.
5) Streaming of data to and from the device will be possible, but only for certain data types, like binary files and blobs.
6) Variable Timeouts can be assigned to sessions.
7) Performance is improved for CoLa 2.0 sessions routed through high latency connections.

[i] Unless an extended execution timeout is specified in CID

[ii] Unless an extended execution timeout is specified in CID