D-Cerno_1.6 Communication TCCP

Copyright

All information in this document is subject to change without notice. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Televic.

© 2013 Televic NV. All rights reserved.

Document history

Version	Author	Date	Description
1.0	PT	13/03/2014	Initial version : started from UniCOS API
1.1	PT	03/04/2014	Fix set loudspeaker/headphone volume
1.2	PT	16/7/2014	Draft extended microphone status
1.3	PT	14/10/2014	Draft extended microphone status
1.4	DIV	27/10/2015	Add tcp port
1.5	CLY	08/09/2016	Correct syntax of TCCP examples Added "tim" field in connect command
1.6	DIV	28/10/2020	Correction in 3.2.4.24 (O and C swapped in example string)

1 Introduction & Scope

1.1 Introduction

1.2 Scope

1.2.1 In scope

1.2.2 Out of scope

Table of contents

Table of Contents

List of figures

No table of figures entries found.

List of tables

No table of figures entries found.

Terminology

Name	Meaning

References

ID	Reference	Version	Name and meaning

2 TCCP Header

2.1 Introduction

Most of the TCS systems expose all, or a subset of their functionalities. Be it to support remote configuration by a technician, real-time control over microphones by means of a PC application by an operator, starting/stopping of a recording by external parties, etc. In order to provide a high level common interface to these systems, the **T**elevic **C**ommon **C**ommunication **P**rotocol (TCCP) has been designed.

As this common protocol is used for several products within the Televic range and will be used for future developments, it will bundle development efforts in a wide range of areas: SW components, tools, testing, documentation, ...

This document describes only the communication protocol applied for Televic systems. In order to develop specific applications, the command description document of the involved system is required. (e.g. WCAP+ API)

2.2 Requirements

In order to support different kinds of systems, different mediums (TCP, RS232, memory sharing, …), provide enough flexibility to support future needs, etc … the following set of requirements needs to be fulfilled:

Protocol requirements	Why ?
Format - Field separator	Ensure that field name and field value lengths are flexible.
Format - Field identification	Ensure that fields - can be interpreted without having to know the position of the field in the data - fields can be added or removed without breaking down compatibility
Unique command identification	Ensure that a sender knows to what outgoing request an incoming reply belongs.
Independent of the medium	The protocol has to be flexible enough to be used with TCP/IP, RS232, ...
Low and high end systems	Support full functional high end servers as well as low(er) end embedded systems. Allow subsets of the protocol.
Textual data and binary data streams	Textual protocol is easy to develop, debug, test, trace, ... Binary data is needed to send files, complex data structures, ...

Depending on the restrictions of the sender/receiver the data can support all, or just a subset of the protocol.

2.2.1 Command sets

We need a ***basic command set***, to be able to set up a basic communication with all of the systems, this basic set for instance includes "Identify", "Help", "Message", ...

Next to this basic set of commands, we also need to provide a flexible and extensible format, so it is possible to add functionality in the future, without having to upgrade the protocol.

System specific functionality has to be encapsulated in a ***system specific command set***, for instance "SwitchMicrophoneON", "StartVoting", "GetVoteResults", ...

By splitting up the command set into a basic command set and a system specific command set, a sender/receiver always can understand as much as possible without having to know the details of the system. If for instance, a sender notifies an event to a receiver, and the receiver doesn't know how to deal with the event, at least the receiver knows it's an event and the receiver can make the user aware of this event.

2.2.2 States

A sender and a receiver are either ***disconnected*** or ***connected***.

State	Description
Disconnected	Only a subset of packet types are allowed (idy,hlp,con).
Connected	All of the packet types are allowed.

2.2.3 Packet

A ***packet*** is sent from a sender to a receiver.
A packet starts with a STX (Start of TeXt, and ends with an ETX (End of TeXt):

STX	Protocol ID [2]	**':'**	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	**':'**	Body	**ETX**

TCCP does not include a transport layer. If data integrity for instance has to be preserved by means of a CRC, chunk scrambling detection or any other kind, this has to be done in a transport layer.

The length of the packet header between the two colons is not fixed! In the future, fields might be inserted right in front of the second colon. In order to find the beginning of the body, search for the second colon.

2.2.4 STX & ETX

The ***STX*** (0x02) and ***ETX*** (0x03) are not meant to support a transport layer in the protocol. These are introduced to be able to split up sequential packets easily.

2.2.5 Protocol ID

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The ***Protocol ID*** identifies the protocol. This identification has been added in order to support future protocols. Currently the only supported protocol is the one as described in this document.

Protocol ID	Information
02	The protocol described in this document

2.2.6 Packet types

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

This following is the complete list of packet types as defined within the TCCP:

Type	Name	Sender of the packet	Reply	State	Receiver of the packet
idy	Identify packet	The sender asks the receiver to identify itself.	Y	Disconnected Connected	The receiver sends a reply packet with identification data in its body.
hlp	Help packet	The sender asks the receiver for help.	Y	Disconnected Connected	The receiver sends a reply packet with any kind of information that can help the user in setting up proper communication with the receiver. Ideally, the receiver sends the complete list with all of the supported features (calls, functions, processes, events) and their parameters, so as to provide the sender with all of the information needed to control the receiver.
con	Connect packet	The sender asks the receivers to open a connection.	Y	Disconnected	The receiver opens a connection and sends a reply packet to indicate if the connection was opened properly. Optionally the receiver can first perform a version check before opening the communication.
dis	Disconnect packet	The sender closes the connection on the receiver.	N	Connected	The receiver disconnects from the sender and removes the connection.
cal	Call packet	The sender calls functionality on the receiver.	N	Connected	The receiver will execute the call, but will *not* send any reply packet with status information. However, the call might trigger events on the receiver, which will be notified by the receiver through event packets.
fnc	Function packet	The sender calls a function on the receiver and waits for it to return.	Y	Connected	The receiver executes the function, and sends a single reply with the return status of the function.
pro	Process packet	The sender starts a process on the receiver.	Y	Connected	The receiver starts the process, and sends reply packets with status information for as long as the process is ongoing.
set	Setter packet	The sender sets the value(s) of one or more object properties on the receiver.	N	Connected	The receiver sets the property values on the object.
get	Getter packet	The sender gets the value(s) of all of an object's properties on the receiver.	Y	Connected	The receiver sends a reply with the values of all of the object's properties.
evt	Event packet	The sender notifies an event to the receiver.	N	Connected	The receiver might be interested in something which happened at the sender side.
msg	Message packet	The sender sends a message to the receiver.	N	Connected	A message is *not* meant to have a receiver *doing* something. It's a way of telling the *user of the receiving system* something.

dat	Data packet	The sender sends data to the receiver.	Y	Connected	Data packets have to be acknowledged by the receiver by means of reply packets, in order to avoid overflows at the receiver.
lfc	Life check	The sender sends a life check to verify if the receiver is still alive.	Y	Connected	The receiver sends a reply packet to notify that he is still alive.
rep	Positive reply packet	The receiver sends a positive reply to the sender.	N	Connected	Whenever a receiver receives a packet from a sender, it might send a positive reply packet to notify the sender that the packet is handled successfully.
ren	Negative reply packet	The receiver sends a negative reply to the sender.	N	Connected	Whenever a receiver receives a packet from a sender, it might send a negative reply packet to notify the sender that the packet can't be handled properly.
reo	Ongoing reply packet	The receiver sends an ongoing reply to the sender.	N	Connected	Whenever a receiver receives a packet from a sender, it might send a reply packet to notify the sender that the packet was received properly, that the packet handling was started properly, but that the packet handling is still ongoing.

2.2.7 Packet ID

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

Each packet is unique and therefore identified by an ID.

2.2.8 Reply packet

When the receiver receives a packet from the sender, the receiver can reply to the sender with a ***reply packet.***

TCCP defines 3 types of reply packets:

- ***Positive*** reply packet
  When the receiver successfully handled the packet.
- ***Negative*** reply packet
  When the receiver wasn't able to successfully handle the packet.
- ***Ongoing*** reply packet
  When the receiver successfully started handling the packet, but handling isn't finished yet. The receiver can send subsequent *ongoing reply packets* to the sender to indicate the ongoing state of the packet.
  -> If at one point in time, the receiver successfully finished handling the packet, the receiver will send a *positive reply packet* to the sender to mark the end of the packet handling.
  -> If at one point in time, the receiver can't finish handling the packet successfully, the receiver will send a *negative reply packet* to the sender to mark the end of the packet handling.

The packet ID of the reply packet has to be the same as the ID in the original packet. This way a sender can identify to what packet the incoming packet reply relates to.

A reply packet only can include the following fields in the body:

Field	Short description
`sta`	The status of the packet handling.
`inf`	Information on the packet state.
`[type]`	An element of the same type as the type of the original packet. This element may contain sub-elements.

2.2.9 Body format type & body

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The body supports different formatting types. The body format type is specified in the packet. A receiver supports one or more body format types. However, we want to reduce the number of body format types. Only in very specific situations, one should consider to introduce a new body format type. We want to standardize as much as possible on as little formats as possible. This being said, we need at least one format. Because of the flexibility, the hierarchical setup and the huge number of tools and libraries available to support the format, we've chosen to promote an ASCII XML formatting as default formatting type. For MultiCos we've chosen to use json as formatting type.

The following is a list of supported body format types:

Body format type	Short description
00	No body or unknown body
01	ASCII XML format
02	ASCII json format

2.2.10 QOS

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The QOS byte is a priority setting for packet handling at the receiver side. By giving different priorities to the packets, the receiver will re-order the packets in its packet-queue and will give priority to the packets with the highest priority.

Lowest priority = '0'
Highest priority = '9'

2.2.11 Tx type

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The Tx type specifies the transmitter.

The following is a list of supported Tx types:

Tx types	Short description
C	Central Unit
D	Conference Desk
I	Interpreter Desk
O	CoCon
N	NIOS

2.2.12 Tx id

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The transmitter is identified by a Tx id.

2.2.13 Rx type

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The Rx type specifies the receiver.

The following is a list of supported Rx groups:

Rx types	Short description
C	Central Unit
D	Conference Desk
I	Interpreter Desk
O	Cocon
N	NIOS
8	All except CU
9	All

2.2.14 Rx id

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The receiver is identified by a Rx id. "99999" is used to address multiple receivers.

2.2.15 Tx Property

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The Tx property indicates if a frame is a single frame, part of a stream or the last frame of a stream.

Tx Property	Short description
0	Single frame; Tx session = 0
1	Frame(s) will follow; Tx session is used
9	End of stream; Tx session is used

2.2.16 Tx Session

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The Tx session is used to group a number of frames to a logical sequence (e.g. to transmit large json lists). In this way a single frame(s) may be transmitted during the transmission of a stream. Also, multiple streams can be transmitted at the same time.

The Tx Session ranges from 1 uptil 9. Zero is reserved for single frames.

The Tx property is set to 1 or 9 (last frame from the stream)

2.2.17 Room-ID

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The ***Room-ID*** identifies the room you want to address the packet to. For some type of packets, this field will not be taken into account. (e.g con)

2.2.18 Packet length

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

The ***packet length*** specifies the total length of the packet expressed in bytes (from STX uptil ETX) , this field will not be taken into account.

2.2.19 Body (ASCII format)

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

Only ASCII characters are allowed. For XML element names, XML element attribute names, this is no problem, because they are part of the protocol.
On top of that, all of the attributes content also needs to be ASCII.
For instance, the parameter names of functions and processes have to be in ASCII.

On the other hand, the system might use other encoding for element content.
If text content uses other encoding than ASCII, the encoding needs to be mentioned in the attribute "encoding" of the element, and the text itself has to be HEX-encoded in order not to interfere with any control characters in the protocol !

Example of a packet

01:idy0001010C00001D00001000001400:	Protocol ID
01:idy0001010C00001D00001000001400:	Packet type =
01:idy0001010C00001D00001000001400:	Packet ID
01:idy0001010C00001D00001000001400:	Body format type = 01 (ASCII XML)
01:idy0001010C00001D00001000001400:	QOS =
01:idy0001010C00001D00001000001400:	Tx type : C =
01:idy0001010C00001D00001000001400:	Tx id =
01:idy0001010C00001D00001000001400:	Rx type : D = Delegate Unit
01:idy0001010C00001D00001000001400:	Rx id =
01:idy0001010C00001D00001000001400:	TxProperty =
01:idy0001010C00001D00001000001400:	TxSession = 0
01:idy0001010C00001D00001000001400:	Room-id = 0
01:idy0001010C00001D00001000001400:	Packet length = 1400

2.2.20 D-Cerno header section usage

STX	Protocol ID [2]	':'	Type [3]	ID [4]	Body format type [2]	QOS[1]	Tx type [1]	Tx id[5]	Rx type [1]	Rx id[5]	Tx prop [1]	Tx session [1]	Room-ID [3]	Packet len[4]	':'	Body	ETX

For D-Cerno only following header section are used:

- Protocol ID
- Type
- ID
- Body format type
- Body

All the other section may be filled up with ‘0’.

Packet at a glance for json:

Packet type	Packet body	Reply packet body

con	{
	"typ":"",
	"nam":"",
	"ver":"",
	"inf":"",
	"svr":0,
	"tim":""
	}
		{
		"sta":"",
		"inf":"",
		"con":{
		"typ":"",
		"nam":"",
		"ver":"",
		"svr":0
		}
		}

3 D-Cerno Commands

3.1 Introduction

This document describes the commands needed to develop custom applications on the D-Cerno system.

The Televic Common Communication Protocol (TCCP) description document will be needed in order to communicate with the involved system.

3.2 Connection

Connection is established via tcp port 5011

3.2.1 Connect

The sender asks the receiver to open a connection. Based on the version data included in the connection packet, the receiver can check the sender's version. The sender on his turn can check the receiver's version based on the version data in the reply packet.
Sender:

```
<STX> 02:con<id(4)>020O<tx id(5)>C<rx id(5)>000000000:
        {
        "typ":"Application",
        "nam":"DU",
         "ver":"1.01",
        "inf":"",
        "svr":0,
        "tim":"<time according to ISO 8601>"
        }
<ETX>
```

Note: the time parameter "tim" can also be left empty: ""

Receiver: connection accepted

```
<STX> 02:rep<id(4)>020C<tx id(5)>O<rx id(5)>000000000:
        {
                "sta":0,
                "inf":"We're connected … Welcome",
                "con": {
                        "typ":"D-Cerno",
                        "nam":"CU",
                        "ver":"0.09.01",
                         "svr":0
                        }
        }
<ETX>
```

Receiver: connection not accepted

```
<STX> 02:rep<id(4)>020C<tx id(5)>O<rx id(5)>00:
        {
                "sta":-1,
                "inf":"Not allowed, because …",
                "con": {
                        "typ":"D-Cerno",
                        "nam":"CU",
                         "ver":"0.05.01",
                         "svr":0
                        }
        }
<ETX>
```

3.2.2 Disconnect

The sender tells the receiver to close the established connection.
The id gives the reason of the disconnection. The text field is the description of the id.

Sender:
<STX> 02:dis<id(4)>020C<tx id(5)>O<rx id(5)>000000000:
{
"id":"01",
"inf":"System shutting down"
"svr":0
}
<ETX>

Disconnect Id's	
ID	Description
00	Normal disconnect
01	System shutting down
02	Invalid version
03	To many connections

3.2.3 Life check

The sender sends a life check packet to the receiver to verify if it's still operational.
Sender:
<STX> 02:lfc<id(4)>020O<tx id(5)>C<rx id(5)>000000000: <ETX>

Receiver:
<STX> 02:rep<id(4)>020C<tx id(5)>O<rx id(5)>000000000:
{
"sta":0,
"inf":"I'm still alive",
"lfc":""
"svr":0
}
<ETX>

3.2.4 Operational commands

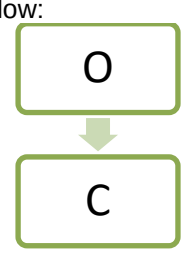
3.2.4.1 Toggle Microphone status

This command is used to toggle the microphone status.

Type: set
Name: micstat: micStatus
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type
uid	serial of the unit	String
stat	0 = toggle	String

Example: O 00000 → C
<stx>02:set0000029O00000C00000000000000:{"nam":"micstat","uid":"101008d2","stat":"0"}<etx>
Reply: evt: microphone status.
Flow:



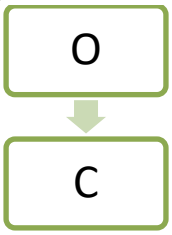
3.2.4.2 Set Microphone status

This command is used to toggle the microphone status.

Type: set
Name: smicstat: set microphone status
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type
uid	serial of the unit Or (0=All delegates, no chairmen)	String
stat	microphone status (0=OFF or 1=ON or 2=REQUEST or 3=TOGGLE). When uid=0 only stat=0 is accepted.	String

Example: O 00000 → C
<stx>02:set0000029O00000C00000000000000:{"nam":"smicstat","uid":"101008d2","stat":"0"}<etx>
Reply: evt: microphone status.
Flow:



3.2.4.3 Get Microphone status

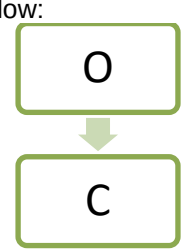
This command is used to get the status of one or all microphones.

Type: get
Name: gmicstat: set microphone status
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type
uid	serial of the unit Or (0=all)	String

Example: O 00000 → C
<stx>02:get0000029O00000C00000000000000:{"nam":"gmicstat","uid":"0"}<etx>
Reply: evt: microphone status for uid=serial.
evt: all microphones status for uid=0.

Flow:



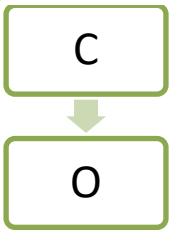
3.2.4.4 Microphone Status event

This event is sent by the Central Unit indicating that the microphone with given number is on/off/in request.

Type: evt
Name: micstat: micStatus
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
uid	Id of the unit (0=all)	String
stat	microphone status (0=OFF or 1=ON or 2=REQUEST)	String

Example: C -> O 00000
<stx>02:evt0000029C00000O00000000000000:{"nam":"micstat","uid":"101008d2","stat":"1"}<etx>
Flow:



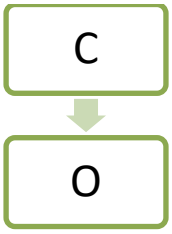
3.2.4.5 Microphone error event

This event is sent by the Central Unit indicating that an error has occurred during set microphone status.

Type: evt
Name: err: error
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
id	0xC = Prior pushed 0xD = MaxMic allowed reached 0xE = Wrong mic mode 0xF = Wrong serial number (0x00000000 or 0xFFFFFFFF)	String

Example: C -> O 00000
<stx>02:evt0000029C00000O00000000000000:{"nam":"err","id":"1"}<etx>
Flow:



3.2.4.6 Set Loudspeaker volume

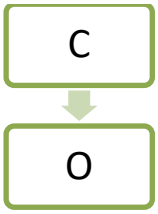
Set the loudspeaker volume

Type: set
Name: slsvol: setLsVolume
Sender: Application (O)
Receiver: Central Unit (C)
Response: evt lsvol
Par:

Name	Description	Type
vol	Volume level (0..24)	number

Example: O 00000 → C (00000)
<stx>02:set0000020O00000C00000000000000:{"nam":"slsvol","vol":5}<etx>
Reply: <stx>02:evt0000020C00000O00000000000000:{"nam":"lsvol","vol":5}<etx>

Flow**:**



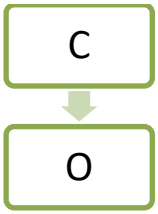
3.2.4.7 Loudspeaker volume event

The loudspeaker volume has changed

Type: evt
Name: lsvol: lsVolume
Sender: Central Unit (C)
Receiver: Application (O)
Response: /
Par:

Name	Description	Type
vol	Volume level (0..24)	Number

Example: C 00000 → O (00000)
<stx>02:evt0000020C00000O00000000000000:{"nam":"lsvol","vol":5}<etx>
Flow:



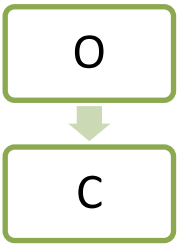
3.2.4.8 Get Loudspeaker volume

Get the loudspeaker volume

Type: get
Name: glsvol: getLsVolume
Sender: Application (O)
Receiver: Central Unit (C)
Response: reply lsvol
Par:-

Example: O 00000 → C (00000)
<stx>02:get0000020O00000C00000000000000:{"nam":"glsvol"}<etx>
Reply: <stx>02:rep0000020C00000O00000000000000:{"nam":"lsvol","vol":5}<etx>

Flow:



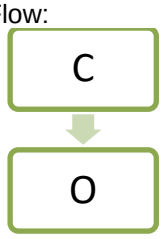
3.2.4.9 Loudspeaker volume reply

The loudspeaker volume.

Type: evt
Name: lsvol: lsVolume
Sender: Central Unit (C)
Receiver: Application (O)
Response: /
Par:

Name	Description	Type
vol	Volume level (0..24)	Number

Example: C 00000 → O (00000)
<stx>02:rep0000020C00000O00000000000000:{"nam":"lsvol","vol":5}<etx>
Flow:



3.2.4.10 Set Headphone volume

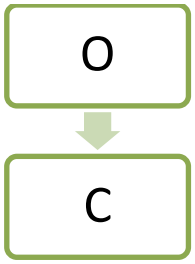
Set the headphone volume (floor)

Type: set
Name: shpvol: setHpVolume
Sender: Application (O)
Receiver: Central Unit (C)
Response: evt hpvol
Par:

Name	Description	Type
vol	Volume level (0..24)	number

Example: O 00000 → C (00000)
<stx>02:set0000020O00000C00000000000000:{"nam":"shpvol","vol":5}<etx>

Flow:



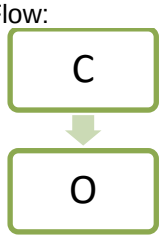
3.2.4.11 Headphone volume event

The headphone volume (floor) has changed

Type: evt
Name: hpvol: hpVolume
Sender: Central Unit (C)
Receiver: Application (O)
Response: /
Par:

Name	Description	Type
vol	Volume level (0..24)	Number

Example: C 00000 → O (00000)
<stx>02:evt0000020C00000O00000000000000:{"nam":"hpvol","vol":5}<etx>
Flow:



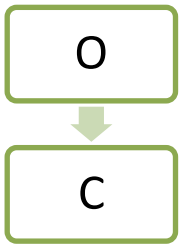
3.2.4.12 Get Headphone volume

Get the headphone volume (floor)

Type: get
Name: ghpvol: getHpVolume
Sender: Application (O)
Receiver: Central Unit (C)
Response: reply hpvol
Par:-

Example: O 00000 → C (00000)
<stx>02:get0000020O00000C00000000000000:{"nam":"ghpvol"}<etx>
Reply: <stx>02:rep0000020C00000O00000000000000:{"nam":"hpvol","vol":5}<etx>

Flow:



3.2.4.13 Headphone volume reply

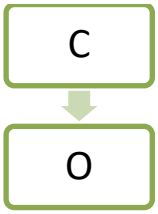
The headphone volume (floor) has changed

Type: rep
Name: hpvol: hpVolume
Sender: Central Unit (C)
Receiver: Application (O)

Response: /
Par:

Name	Description	Type
vol	Volume level (0..24)	Number

Example: C 00000 → O (00000)
<stx>02:rep0000020C00000O000000000000000:{"nam":"hpvol","vol":5}<etx>
Flow:



3.2.4.14 Set maximum active microphones

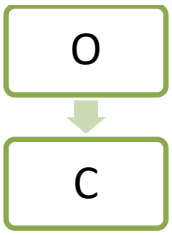
Type: set
Name: smam:setMaxActiveMicrophones
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type
mam	Number (0..8)	number

Reply: short reply

Evt: mam:maxActiveMicrophones
Par : mam = number

Example: C 00000 → C 00000
<stx>02:set0000020O00000C00000000000000:{"nam":"smam", "mam":8 }<etx>
Flow:



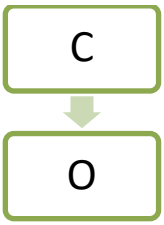
3.2.4.15 Maximum active microphones event

Type: evt
Name: mam: MaxActiveMicrophones
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
mam	Number (0..8)	number

Reply: -

Example: C 00000 → O 00000
<stx>02:evt0000020C00000O00000000000000:{"nam":"mam", "mam":8}<etx>
Flow:



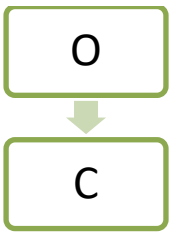
3.2.4.16 Get maximum active microphones

Type: get
Name: gmam:getMaxActiveMicrophones
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type

Reply: rep mam:maxActiveMicrophones
Par : mam = number

Example: O 00000 → C 00000
<stx>02:get0000020C00000O00000000000000:{"nam":"gmam"}<etx>
Flow:



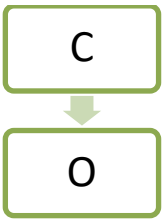
3.2.4.17 Maximum active microphones reply

Type: rep
Name: mam: MaxActiveMicrophones
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
mam	Number (0..8)	number

Reply: -

Example: C 00000 → O 00000
<stx>02:rep0000020C00000O00000000000000:{"nam":"mam", "mam":8}<etx>
Flow:



3.2.4.18 Set Microphone mode

The following table shows the different microphone modes with the following data:

- Column 1: name of the working mode
- Column 2: name of the option
- Columns 3-4: Parameters to use

Working modes	Options	parameters		
		mmo	mio	mat
Direct Access				
	Toggle	1	0	1
	Push	1	0	2
FIFO				
	Toggle	2	4	1
	Group	2	7	1
	Vox	2	4	4
Request		0	3	1

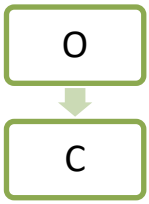
Type: set
Name: smmo:setMicrophoneMode
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type
mmo	Microphone mode: Operator Direct speak Group request	Number 0 1 2
mio	Options: none Request allowed Cancel request allowed Use override	Number 0 1 2 4
mat	Activation type: None Toggle Push Vox	Number 0 1 2 4

Note that the parameter values are bit-wise because in one situation they might be combined. This is the following :
mio = 3
Meaning that both « Request allowed » and « Cancel Request allowed » are active.

Reply: short reply
Evt: mmo:microphoneMode
Par : mmo = mode
mio : microphone options
mat : microphone activation type
Example: O 00000 → C 00000
<stx>02:set0000020O00000C00000000000000:{"nam":"smmo", "mmo":1, "mio":0, "mat":1}<etx>

Flow:



3.2.4.19 Microphone mode event

The following table shows the different microphone modes with the following data:

- Column 1: name of the working mode
- Column 2: name of the option
- Columns 3-4: Parameters to use

Working modes	Options	parameters		
		mmo	mio	mat
Direct Access				
	Toggle	1	0	1
	Push	1	0	2
FIFO				
	Toggle	2	4	1
	Group	2	7	1
	Vox	2	4	4
Request		0	3	1

Note that the parameter values are bit-wise because in one situation they might be combined. This is the following :
mio = 3
Meaning that both « Request allowed » and « Cancel Request allowed » are active.

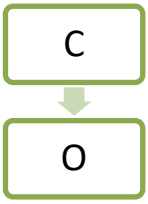
Type: evt
Name: mmo: Microphone Mode
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
mmo	Microphone mode: Operator Direct speak Group request	Number 0 1 2
mio	Options: none Request allowed Cancel request allowed Use override	Number 0 1 2 4
mat	Activation type: None Toggle Push Vox	Number 0 1 2 4

Example: C 00000 → O 00000

<stx>02:evt0000020C00000O00000000000000:{"nam":"mmo", "mmo":1, "mio":0, "mat":1}<etx>

Flow:



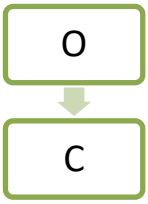
3.2.4.20 Get Microphone mode

Type: get
Name: gmmo:getMicrophoneMode
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type

Reply: short reply
rep: mmo:microphoneMode
Par : mmo = mode
mio : microphone options
mat : microphone activation type
Example: O 00000 → C 00000
<stx>02:get0000020O00000C00000000000000:{"nam":"gmmo"}<etx>

Flow:



3.2.4.21 Microphone mode reply

The following table shows the different microphone modes with the following data:

- Column 1: name of the working mode
- Column 2: name of the option
- Columns 3-4: Parameters to use

Working modes	Options	parameters		
		mmo	mio	mat
Direct Access				
	Toggle	1	0	1
	Push	1	0	2
FIFO				
	Toggle	2	4	1

	Group	2	7	1
	Vox	2	4	4
Request		0	3	1

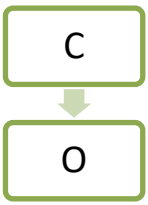
Note that the parameter values are bit-wise because in one situation they might be combined. This is the following :
mio = 3
Meaning that both « Request allowed » and « Cancel Request allowed » are active.

Type: rep
Name: mmo: Microphone Mode
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
mmo	Microphone mode: Operator Direct speak Group request	Number 0 1 2
mio	Options: none Request allowed Cancel request allowed Use override	Number 0 1 2 4
mat	Activation type: None Toggle Push Vox	Number 0 1 2 4

Example: C 00000 → O 00000
<stx>02:rep0000020C00000O00000000000000:{"nam":"mmo", "mmo":1, "mio":0, "mat":1}<etx>

Flow:



3.2.4.22 Set recording status

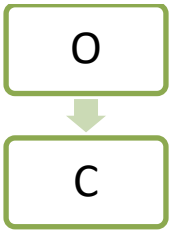
This command is used to set the recording status.

Type: set
Name: srecstat: recording status
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type
stat	1 = stopped 2 = record 3 = paused	number

Example: O 00000 → C
<stx>02:set0000029O00000C00000000000000:{"nam":"srecstat","stat":"1"}<etx>
Reply: evt: recording status

Flow:



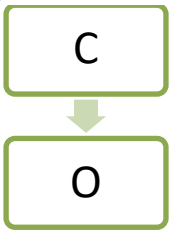
3.2.4.23 Recording status event

This event is sent by the Central Unit indicating the recording status.

Type: evt
Name: recstat: recording status
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
stat	1 = stopped 2 = record 3 = paused	number

Example: C -> O 00000
<stx>02:evt0000029C00000O00000000000000:{"nam":"recstat", "stat":"1"}<etx>
Flow:



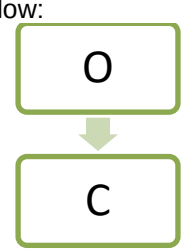
3.2.4.24 Get recording status

This command is used to get the recording status.

Type: get
Name: grecstat: recording status
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type

Example: O 00000 → C
<stx>02:get0000029O00000C00000000000000:{"nam":"grecstat"}<etx>
Reply: rep: recording status
Flow:



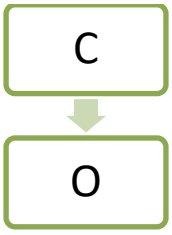
3.2.4.25 Recording status reply

This event is sent by the Central Unit indicating the recording status.

Type: rep
Name: recstat: recording status
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
stat	1 = stopped 2 = record 3 = paused 4 = playing 5 =playing paused	number

Example: C -> O 00000
<stx>02:rep0000029C00000O00000000000000:{"nam":"recstat", "stat":"1"}<etx>
Flow:



3.2.4.26 Get all units

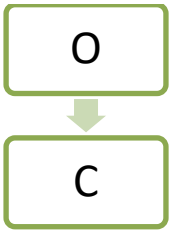
This command is used to get the serials & microphone status of all the units. It also starts the transmission of presense - & status events.

Type: get
Name: gunits: get all units
Sender: Application (O)
Receiver: Central Unit (C)
Par:

Name	Description	Type

Example: O 00000 → C
<stx>02:get0000029O00000C00000000000000:{"nam":"gunits"}<etx>
Reply: unit event

Flow:



3.2.4.27 All units reply

This reply is sent by the Central Unit an contains a list of al the units in the system.

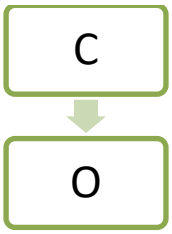
Type: rep
Name: units: all units
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
S	Data model	collection

Data model**:**

Name	Description	Type
Uid	Id of the unit	String
Stat	microphone status (0=OFF or 1=ON or 2=REQUEST)	String

Example: C -> O 00000
<stx>02:rep0000029C00000O00000000000000:{"nam":"units", "s", [{"uid":"101008d2","stat":"1"}]}<etx>
Flow:



3.2.4.28 Unit presence change event

This event is sent by the Central Unit to indicate that the presence of an unit has changed.

Type: evt
Name: unit: unit presense event
Sender: Central Unit (C)
Receiver: Application (O)
Par:

Name	Description	Type
Uid	Id of the unit (0=all)	String
Pres	0: missing unit 1: new unit	String

Example: C -> O 00000
<stx>02:evt0000029C00000O00000000000000:{"nam":"unit","uid":"101008d2","pres":"1"}<etx>
Flow: