TP-7 Control Protocol Specification

CONTENTS

[CONTENTS 1](#_Toc523402631)

[1. GENERAL DESCRIPTION 2](#_Toc523402632)

[2. MESSAGE CODING AND SYNTAX 3](#_Toc523402633)

[3. THE MESSAGES: COMMON PART 4](#_Toc523402634)

[3.1. Message: msg = "getdevicedesc”: Application -> Mixing console: Query console model/type 4](#_Toc523402635)

[3.2. Message: msg = "devicedesc": Mixing console -> Application: Reply to console model/type query. 4](#_Toc523402636)

[3.3. Message: msg = "idle": Any direction: Heartbeat/Keep-alive. 4](#_Toc523402637)

[4. THE MESSAGES: MIXER LINES 5](#_Toc523402638)

[4.1. Message: msg = "getlinelist”: Application -> Mixing console: Request of a list of mixer channels 5](#_Toc523402639)

[4.2. Message: msg = "linelist": Mixing console -> Application: Reply to a request of a list of mixer channels. 5](#_Toc523402640)

[4.3. Message: msg = "getlineinfo”: Application -> Mixing console: Channel (line) status request. 6](#_Toc523402641)

[4.4. Message: msg = "lineinfo": Mixing console -> Application: Reply to request about channel (line) status/ Notification about changing of status of channel (line). 6](#_Toc523402642)

[4.5. msg = "setlineinfo”: Application -> Mixing console: Change channel status. 7](#_Toc523402643)

[5. THE MESSAGES: COMMON MIXING CONTROL 8](#_Toc523402644)

[5.1. Message: msg = "setcue": Application -> Mixing console: Set cue channel status 8](#_Toc523402645)

[6. THE MESSAGES: BUTTONS AND OTHER GENERIC PARAMETERS 9](#_Toc523402646)

[6.1. Message: msg = "getparlist": Application -> Mixing console: Request parameter list 9](#_Toc523402647)

[6.2. Message: msg = "parlist": Mixing console -> Application: Reply to a request of a list of parameters. 9](#_Toc523402648)

[6.3. Message: msg = "getpar”: Application -> Mixing console: Parameter state/value request. 10](#_Toc523402649)

[6.4. Message: msg = "par": Mixing console -> Application: Reply to request about parameter state/value/ Notification about changing of parameter state/value. 11](#_Toc523402650)

[6.5. msg = "setpar”: Application -> Mixing console: Set parameter state/value. 12](#_Toc523402651)

[6.6. The “cue\_button” parameter example 12](#_Toc523402652)

[7. VENDOR SPECIFIC PROTOCOL PARTS 13](#_Toc523402653)

[7.1. Clyde Broadcast Mixing Consoles 13](#_Toc523402654)

[7.2. Clyde Broadcast Mixing Consoles 15](#_Toc523402655)

[8. MESSAGES FORMAL GRAMMAR (IN EBNF) 16](#_Toc523402656)

[8.1. Grammar rules for parsing TCP byte stream and JSON messages. 16](#_Toc523402657)

[8.2. Remarks on grammar rules: 17](#_Toc523402658)

1. GENERAL DESCRIPTION

Protocol is intended for communication between mixing console and external application or software for remote control of faders and other mixer controls, for receiving status information from mixer channels and modules.

For such external applications the mixing console implements a TCP server, to which the external applications – clients can connect for message exchange.

From the protocol point of view both the server (mixing console) and the clients (broadcasting or control applications) may send and receive messages. There is no response message on protocol level (Note: Response logic and timeouts, if any, have to be handled on application level). The TCP server on mixing console side may support several connections from client applications in parallel (Note: Mixing console logic is not defined by protocol when it is under control of several applications).

The protocol is real-time oriented. A protocol implementation on transmitting side must provide immediate sending of the message over TCP connection to the partner; a buffering and delays are not allowed if they are not caused by IP network performance. A protocol implementation on receiving side must provide immediate message handling, deserialization and delivery to application after receiving of zero byte terminating next incoming JSON message object from incoming TCP stream.

1. MESSAGE CODING AND SYNTAX

A single protocol message is a text string in UTF-8 coding with terminating zero byte. The messages follow one after another in TCP byte stream. The null-terminated text string is encoded in JSON syntax (Note: JSON text representation syntax is defined in RFC-7159; see also syntax diagrams here in last chapter). Every null-terminated string contains either single message or a group of messages. The second variant provides a method of simultaneous sending of the messages in group, which is handled by console or application atomically as one “macromessage”. From message syntax point of view, one message corresponds to one JSON object, and group of messages is an array of JSON objects. Therefore, after separating a string from TCP byte stream via null byte terminators, that string must be either single JSON object or array of JSON objects; other variants are wrong in syntax (remark: in general JSON syntax allows separate strings, numbers, literals, or arrays of arrays, etcetera – all this stuff is not correct for this protocol).

Every message – that is, either separate JSON object or element of JSON-array in case when message group is encoded in protocol string – must contain mandatory “msg” field. Type of message is defined by that “msg” field, which is the only mandatory field in the message. Type of message (i.e. content of the “msg” field) defines a set of other fields that must be (or may be) included into JSON object of the message. Implementation of protocol performs serialization or deserialization of all known fields; at that unknown fields of JSON-object are ignored at receiving side.

All the keywords – parameter names, value lists, etcetera – must be written in lowercase. The protocol is sensitive to character case and keywords must not be recognized by implementation if they are written in uppercase, capitalized, and in any other ways. Parameters carrying custom text values (for example, mixer fader names, preset names, and so on) may use any letter case and contain any characters if this is not forbidden by description of the specific message and/or parameter.

1. THE MESSAGES: COMMON PART
   1. Message: msg = "getdevicedesc”:  
      Application -> Mixing console:  
      Query console model/type

This message is sent by application to a mixing console for getting back device model, type, manufacturer and other information about connected device.

{

"msg":"getdevicedesc"

}

* 1. Message: msg = "devicedesc":  
     Mixing console -> Application:  
     Reply to console model/type query.

This message is sent by a mixer as a reply for "getdevicedesc" message

{

"msg":"devicedesc",

"model":"xxx1",

"manufacturer":"xxx2",

"version":"xxx3",

"protocol\_level":nnn

}

Parameters "model", "manufacturer" and "version" are just voluntary information strings.

Parameter "protocol\_level" in future may affect some interaction details between application and console. It is planned that its value at the moment is fixed to integer constant 1 and matches to protocol specification generation (first digit in protocol spec version).

* 1. Message: msg = "idle":  
     Any direction:  
     Heartbeat/Keep-alive.

This message can be sent by application and by mixing console. It does not have any effect except informing partner about partner and channel “alive” status.

{

"msg":"idle"

}

1. THE MESSAGES: MIXER LINES
   1. Message: msg = "getlinelist”:  
      Application -> Mixing console:  
      Request of a list of mixer channels

This message is sent to a mixer for getting back number and names of channels (lines)

{

"msg":"getlinelist"

}

* 1. Message: msg = "linelist":  
     Mixing console -> Application:  
     Reply to a request of a list of mixer channels.

This message is sent by a mixer as a reply for "getline" Message

{

"msg":"linelist",

"lines":[

"xxx1",

"xxx2",

. . . . . . . ,

"xxxN"

]

}

"xxxN" - string – name of a channel (line)

Channel numbers in massive "lines" correspond to values of index of channel element in massive plus 1.

Element with zero index corresponds to first channel, element with index 1 corresponds to 2nd channel etc.

* 1. Message: msg = "getlineinfo”:  
     Application -> Mixing console:  
     Channel (line) status request.

This message is sent to a mixer for getting back channel (channels) status.

{

"msg":"getlineinfo",

"num":nnn

}

nnn - integer – channel number

nnn parameter should identify the channel being requested.

If parameter is not specified, mixer should return a number of messages specifying status of all lines or channels

* 1. Message: msg = "lineinfo":  
     Mixing console -> Application:  
     Reply to request about channel (line) status/  
     Notification about changing of status of channel (line).

This message is sent by a mixer console in two cases:

* as a reply for message "getline" received from client application, or
* as unsolicited message by console initiative in case if any parameter of this line has been changed (for example, console has to send this message in case if operator has moved fader on the line, or has pushed button so that state has been changed).

{

"msg":"lineinfo",

"num":nnn,

"name":"xxx",

"state":"sss1",

"pfl":"sss2",

"gain":ggg

}

nnn - integer – channel number

"sss1" - string - status: one of the following strings (note: values are in lowercase!):

"off" – channel is in OFF state

"on" – channel is in ON state

"sss2" – status of prelistening PFL

"off" -- PFL is OFF at a channel

"on" -- PFL is ON at a channel

ggg - float – channel transfer ratio in dB (depends on fader position)

* 1. msg = "setlineinfo”:  
     Application -> Mixing console:  
     Change channel status.

This message is sent to a mixer for changing of a channel status

{

"msg":"setlineinfo",

"num":nnn,

"state":"sss1",

"pfl":"sss2",

"gain":ggg

}

nnn - integer - channel number

"sss1" - string - status: one of the following strings (note: values are in lowercase!):

"off" – channel is in OFF state

"on" – channel is in ON state

"sss2" – status of prelistening PFL

"off" -- PFL is OFF at a channel

"on" -- PFL is ON at a channel

ggg - float – channel transfer ratio in dB

nnn parameter is and ID for a channel being requested.

1. THE MESSAGES: COMMON MIXING CONTROL
   1. Message: msg = "setcue":  
      Application -> Mixing console:  
      Set cue channel status

This message is sent to a mixer for changing of common cue channel status

{

"msg":"setcue",

"state":"sss",

}

"sss" - string - status: one of the following strings:

"off" – Cue channel is in OFF state

"on" – Cue channel is in ON state

1. THE MESSAGES: BUTTONS AND OTHER GENERIC PARAMETERS
   1. Message: msg = "getparlist":  
      Application -> Mixing console:  
      Request parameter list

This message can be sent to mixer console to request list of console (generic) parameters so that application knows their names and quantity.

{

"msg":"getparlist"

}

* 1. Message: msg = "parlist":  
     Mixing console -> Application:  
     Reply to a request of a list of parameters.

This message is sent by a mixer as a reply for "getparlist" message

{

"msg":"parlist",

"pars":[

"id-xxx1",

"id-xxx2",

. . .,

"id-xxxN"

]

}

"id-xxx..." - string – parameter name

An order of parameter names within "pars" array is not defined and does not have any significance

* 1. Message: msg = "getpar”:  
     Application -> Mixing console:  
     Parameter state/value request.

This message is sent to a mixer for getting back parameter(s) value.

{

"msg":"getpar",

"id":"id-xxx"

}

"id-xxx" - string – identification of requested parameter (should be chosen from "parlist" array)

If field "id" is not specified (absent in the message), then mixer console must send a number of messages to provide state/value for every existing (and listed by “parlist”) parameter.

The protocol allows to use either group of messages (i.e. JSON array) or a sequence of separate messages.

* 1. Message: msg = "par":  
     Mixing console -> Application:  
     Reply to request about parameter state/value/  
     Notification about changing of parameter state/value.

This message is sent by a mixer console in two cases:

* as a response to "getpar" message received from client application, or
* as unsolicited message by console initiative in case if any parameter value/state has been changed (for example, console has to send this message in case if operator has pushed/released parameter button so that state has been changed; or rotated knob of parameter control on console, or whatever else changed).

{

"msg":"par",

"id":"id-xxx",

"val":"val-xxx"

}

"id-xxx" - string – identification of parameter (should be chosen from "parlist" array)

"val-xxx" - string –parameter state/value.

The value may have string or numeric type. The type and set of possible parameter states/values is specific for certain parameter and is not defined by protocol. For values that have two states like enabled/disabled, pressed/released, lighten/dark, etcetera, the protocol recommends the following value set:

* "off" – parameter is switched off or disabled or similar (e.g. button is released, or brought to released state in case of event notification);
* "on" – parameter is switched on or enabled or similar (e.g. button is pressed, or brought to pressed state in case of event notification)

Within event messages (that carry notifications about parameter state change) a new (changed) parameter state/value must be sent.

* 1. msg = "setpar”:  
     Application -> Mixing console:  
     Set parameter state/value.

This message is sent by application to a mixer console to change parameter value/state.

{

"msg":"setpar",

"id":"id-xxx",

"val":"val-xxx"

}

"id-xxx" - string – identification of the parameter to be changed (should be chosen from "parlist" array)

"val-xxx" - string –new parameter state/value.

* 1. The “cue\_button” parameter example

When the button on console is pressed, the console sends notification message:

{

"msg":"par",

"id":"cue\_button",

"val":"on"

}

When the button on console is released, the console sends notification message:

{

"msg":"par",

"id":"cue\_button",

"val":"off"

}

1. VENDOR SPECIFIC PROTOCOL PARTS
   1. Clyde Broadcast Mixing Consoles

This chapter defines generic parameters for Clyde Broadcast mixing consoles. A supported subset of these parameters must be listed by console in "parlist" message in response to "getparlist" message from application. The parameters are summarized in the table below with mark whether specific parameter can be used in messages "getpar" and "setpar" command messages from application to console.

Note: every supported parameter may appear in "par" message from console to application as a notification of state change, or as a result of "getpar" request.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter name | Usable in "getpar" | Usable in "setpar" | Description |
| mic\_on | yes | no | Notify/Query: Any microphone line is “on air” values: “on”, “off” |
| preset | yes | yes | Notify/Query/Set mixer preset:  values: “rec”, “auto”, “live” |
| phone\_line1 phone\_line2 … | yes | Yes | Notify/Query/Set: phone line state:  values: “ring”, “talk”, “idle” |
| monitor1 monitor2 monitor3 … | yes | yes | Notify/Query/Set: connect group/bus to monitor/headphones output line: values: “pg1”, “pg2”, “rec”, “aux”, “ext” |
| settings commutation | ??? | yes | Set: open/close corresponding console control window (settings/commutator): values: “open”, “close” |
| intro\_button outro\_button link\_button forward\_button backward\_button cue\_button play\_button | yes | yes | Notify/Query/Set:  State of buttons located on PREFADE/CUE mixer console panel |

Example:

<-- {"msg":"par","id":"phone\_line1","val":"ring"} // mixer: incoming call on line1

--> {"msg":"setpar","id":"phone\_line1","val":"talk"} // app command: answer the call

<-- {"msg":"par","id":"phone\_line1","val":"talk"} // mixer: line1 in talk state

--> {"msg":"setpar","id":"phone\_line1","val":"idle"} // app command: hangup the call

<-- {"msg":"par","id":"phone\_line1","val":"talk"} // mixer: line1 in idle state

* 1. Tract Broadcast Mixing Consoles

This chapter defines generic parameters for broadcast mixing consoles designed and manufactured by Tract, which also make use of this JSON-based protocol.

* + 1. TR-7 Broadcast Mixing Console

A supported subset of these parameters must be listed by console in "parlist" message in response to "getparlist" message from application. The parameters are summarized in the table below with mark whether specific parameter can be used in messages "getpar" and "setpar" command messages from application to console.

Note: Most of supported parameters may appear in "par" message from console to application as a notification of state change, or as a result of "getpar" request. Exceptions are parameters from “.State” group, which are notifications only.

|  |  |  |  |
| --- | --- | --- | --- |
| Parameter name | Usable in "getpar" | Usable in "setpar" | Description |
| F1.Color F2.Color F3.Color F4.Color F5.Color | yes | yes | Notify/Query/Set for color of a customizable “user button”. The value is integer (represented within JSON string) combined from R, G and B values in range 0:255 each: ( R )|( G<<8 )|( B<<16 ) |
| F1.Text F2.Text F3.Text F4.Text F5.Text | yes | yes | Notify/Query/Set for text of a customizable “user button”. The value is button title within JSON string. |
| F1.State F2.State F3.State F4.State F5.State | no | no | Notify only. Values are “on” and “off”. A sequence of on/off messages generated by console when user clicks a button on console’s touch screen. |

1. MESSAGES FORMAL GRAMMAR (IN EBNF)
   1. Grammar rules for parsing TCP byte stream and JSON messages.

TCP-byte\_stream = { protocol-item "\0" }

protocol-item = { ws } ( JSON-message | JSON\_message-group ) { ws }

JSON-message = object

JSON-message-group = "[" object [ { "," object } ] "]"

object = "{" [ member { "," member } ] "}"

member = string ":" value

value = ( literal | object | array | number | string )

array = "[" [ value { "," value } ] "]"

literal = ( "false" | "null" | "true" )

number = [ "-" ] int [ frac ] [ exp ]

int = ( "0" | digit19 { digit } )

frac = "." digit { digit }

exp = e [ sign ] digit { digit }

digit19 = ( "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" )

digit = ( "0" | digit19 )

e = ( "e" | "E" )

sign = ( "-" | "+" )

string = "\"" { ( char | escape ) } "\""

char = .... any UNICODE char in range 0x20:0x10FFFF except backslash (0x5C) and quotation mark (0x22) ....

escape = ( "\"" | "\\" | "\u" hex hex hex hex | "\/" | "\b" | "\f" | "\n" | "\r" | "\t" )

hex = ( digit | "A" | "B" | "C" | "D" | "E" | "F" )

ws = ( " " | "\t" | "\r" | "\n" )

* 1. Remarks on grammar rules:

1. WS sequences allowed to be inserted everywhere except tokens 'string' and 'number'.
2. The numbers generally assumed to be 8-byte IEEEE double float points. The true numbers allowed only, not NaNs and INFs.
3. If 'JSON-text' character stream may contain surrogate pairs but they must be coded correctly (no single values from 16-bit surrogate halves).
4. The literal names MUST be lowercase. No other literal names are allowed.