K2 Edge Protocol Manual

Document version: 1.0 - 8-3-2012

Table of Contents

1. Grass Valley Product Support

Contact information: http://www.grassvalley.com/support/contact
U.S Technical Support: +1 800-547-4989 or +1 530 478 4148 or E-mail: Please use our online form
All other countries Technical Support: +800 80 80 20 20 or +33 1 48 25 20 20 or E-mail: callcentre@grassvalley.com
FAQ: http://grassvalley.novosolutions.net/
Training: https://grassvalley.csod.com/LMS/catalog/Main.aspx?tab_page_id=-67&tab_id=6

2. Introduction

This document describes third party remote control over nexos, the integrated playout 'firmware' running on the K2 Edge hardware frames. This document focuses on launching templates that were created with the Channel Composer design application. The intended reader is a programmer familiar with the C programming language and with some knowledge of networking.
Following topics are included:

- **Nexos Socket Server Communication Setup**
  - How to create a socket link with the nexos socket server.
  - How to send messages to the socket server.
  - Socket server command layout.
  - Short introduction to K2 Edge API commands.

- **K2 Edge API NRE Command**
  - The NRE command is one of the K2 Edge API commands supported by nexos, and this particular one is entirely focused on the Nexos Render Engine (NRE), the render unit built into nexos responsible for producing the graphics created in the Channel Composer application.
  - Describes the Nexos Render Engine commands, including commands for preload and play of templates, and preload and selection of scene graphs.

- **K2 Edge API NRE Feedback**
  - Feedback messages are messages generated by nexos (and in this case by the NRE unit in nexos) to deliver status reports to the initiator (you). Feedback is optional, and must be explicitly enabled.
  - Describes options for feedback generated by K2 Edge API NRE command.

- **K2 Edge API Source String Format**
  - Describes the vstream and astream fields.

- **Channel Pack Management**
  - Introduction to Channel Packs, a zipped folder structure containing the scene graph and templates created in Channel Composer.
  - A description of the Channel Pack workflow; how to bring a Channel Pack on-air.

- **Complex String Format**
  - For reference only, describes the complex string format used by the K2 Edge API commands.

3. Nexos Socket Server Communication Setup

3.1. Introduction

Nexos is the real-time playout application running on the K2 Edge frame, responsible for decoding, rendering and encoding. Nexos can be remotely controlled over a network socket link. This chapter explains how to set up remote control for third parties.

3.2. The nexos socket server

The nexos socket server listens on port #5001, waiting for clients to connect and send socket server commands. The server uses the streaming (TCP) socket protocol. The nexos server is part of the nexos application. A remote client sending only an occasional command will typically make a connection with the nexos server, send the command, then close the link.

Clients regularly sending commands will connect and maintain the connection. The nexos socket server does not close the client connection, the client has full control.
Clients expecting feedback from nexos will have to keep the connection open and wait for the feedback replies. These are returned by nexos over the same link.

3.3. Code examples

This section provides a number of coding examples.

Note that the examples below describe a situation in which a POSIX socket API is used (this API is used on most UNIX systems, including Linux and OSX). Windows systems use a different, but very similar API.

3.4. Connecting with the nexos socket server

To connect as a client with the nexos socket server, perform the following steps:

1. Create a socket with:
   ```
   int fd = socket( AF_INET, SOCK_STREAM, 0 );
   ```

2. Fill a `struct sockaddr_in` object with the address info of the remote nexos socket server:
   ```
   struct sockaddr_in sockAddress;
   sockAddress.sin_family = AF_INET;
   sockAddress.sin_addr.s_addr = inet_addr( remote-host-dot-quad-string );
   sockAddress.sin_port = htons( 5001 );
   int addressLn = sizeof( sockAddress );
   ```

3. Connect the socket with the nexos socket server with:
   ```
   result = connect( fd, &sockAddr, addressLn );
   ```

Make sure to check result values of both the `socket()` and `connect()` calls. `fd` should represent a properly connected socket link with the nexos socket server.

3.5. Writing messages to the nexos socket server

The socket file descriptor can be used to `write(2)` messages to the socket server, or alternatively `send(2)` can be used.

3.6. Waiting for feedback

If feedback is wanted, feedback messages sent by nexos will be returned over the same socket link. To receive these messages, the client can use `read(2)` or `recv(2)` on the socket `fd` to wait and read the feedback message data.

To test for available feedback data without blocking the current thread, consider non-blocking read, `ioctl(2)` or `select(2)`. Or for a more advanced approach, consider using a dedicated thread that does the reading part and simply blocks on `read( fd )`.

3.7. Closing the link when no longer needed

Remember to close the link with the socket server when communication has finished, using `close(2)` on the socket `fd`. Nexos will not close the link; this is the client's responsibility.

Several different approaches exist regarding the lifetime of the client-server link:

1) Open the link, send a command, wait for optional feedback, then close the link. This approach does not support overlapping commands during communication.
2) Open the link (and keep it open until the client application closes), and perform write() and read() calls for different commands. This allows for overlapping commands, suitable for clients designed to send more than a single command.

3.8. Socket server command layout

The nexos socket server only accepts incoming commands that meet specific requirements. The socket server supports a number of different services, but all of them follow the command layout described next.

An incoming command is only valid when it has the following fields, in given order:

- First the lowercase characters '`nex:`' (without the quotes).
- A space.
- A decimal number (in printable ASCII) representing the nexos channel number. First channel is '0', second is '1', etc.
- A space.
- A decimal number (in ASCII) representing the nexos render layer, lowest layer is '0'. If not relevant for the command use '0'.
- A space.
- The socket server *service* name (for example `complex`).
- A space.
- Optional additional arguments supported by the given service in ASCII, separated by spaces, no newlines.
- And finally a single newline character (`\n`, ASCII LF, `0x0A`) to terminate the command.

In pseudo grammar format:

```
<nex:> <chan> <layer> <service> [arguments] <newline>
```

A real-world example:

```
nex: 0 4 complex {cmd=ping;}\n
```

The terminating newline indicates the "end-of-message". Note that newlines embedded in the command are not allowed. Also note that it is possible for a remote client to send several commands in one go, each of them terminated by a newline character.

3.9. The API protocol

The nexos socket server supports a number of different services, the most important being the *complex* service. The complex service implements the K2 Edge API protocol. This protocol supports a group of commands that allow control over many of the nexos playout features. For example, using K2 Edge API commands, clips and animations can be preloaded and started at a given time, GPIO ports can be triggered and Channel Composer templates can be preloaded and launched.

The K2 Edge API protocol uses the complex string format as the 'language' to communicate messages, instead of binary data. The simple ASCII format allows one or more field-value pairs to be placed in a string between curly braces. Each pair is terminated with a semicolon.

API commands sent to the nexos socket server using the complex service share the following features:

1. They all follow the Complex String Format syntax.
2. They all have a field called 'cmd' where the value defines the command name. (Most commands use more than just one field-value pair though.)

For example, for the *ping* command, it looks like this:

```
{cmd=ping;}
```

This is a simple command with just one field-value pair, in this case the mandatory cmd field with value ping. Note the terminating semi-colon after the value bit, and the enclosing curly braces. To send this command to nexos using the complex service, to channel #0 and layer #4 and following the layout described above:

```
nex: 0 4 complex {cmd=ping;}\n
```

The Complex String Format is described in more detail in the appendix.

4. API NRE Command

4.1. Introduction

The K2 Edge API NRE command allows remote control over the K2 Edge Render Engine (NRE), the part of the nexos playout application that can render templates previously created in Channel Composer.

4.2. Supported sub commands

The NRE command supports a number of sub commands listed in the table below:

sub command	description
preload-play	Both preload and play a template.
preload	Preload a template, preparing it for playback.
play	Start playback of a previously preloaded template.
sg-preload	Preload a new scene graph from a given channel pack.
sg-select	Select a previously preloaded scene graph.
clear	Stops and clears all templates running in the active (currently rendering) scene graph. Not implemented yet.
object-info	Returns info on given template via feedback.
sp-print	Print scene parameter values.
control	Allows for various levels of control over NRE.

These sub commands will be discussed in detail in the next chapters.

Like all K2 Edge API commands, the NRE command follows the Complex String Format syntax. The value of the mandatory cmd field is nre

An example NRE command:

```
{cmd=nre; subcmd=play; template=ShowLogo; inst=10;}
```

4.3. The exid field

The `exid` field in the command string is a mandatory field. This field is reserved for defining a unique external ID, numeric or alphanumeric, for each command. (An easy way to implement this is to use a 32-bit or 64-bit integer counter and increment the counter value for each new command.) The `exid` field and value are guaranteed to be left untouched and are returned in the feedback. Using the `exid` field value found in feedback messages from nexos, it is easy to associate the feedback message with the original command, which is especially useful when nexos is working on several commands simultaneously.

Keep following rules in mind:

- The `exid` field is mandatory for all sub commands.
- The `exid` value format is under control of the initiator (numeric, alpha or mixed).
- The `exid` value must be unique for each new command sent to nexos.
- The `exid` field and value are guaranteed to be left untouched and returned through feedback messages.
- The `exid` field value is the only way to link feedback messages generated by nexos back to the original command.

4.4. The inst field

When nexos receives a command to preload a template for playback, a temporary instance of this template is created. The instance lifetime covers the preload- and playback phase, the instance will automatically be removed when playback is finally stopped. Each template instance needs a unique identifier so that commands can address it.

For example: sub command `preload` is used to preload a template instance with instance ID value X; the following play sub command must specify the same identifier value X, so nexos can locate the instance. The `inst` field defines a unique template instance identifier value.

Keep following rules in mind:

1. The `inst` field is mandatory for the following sub commands: preload-play, preload and play.
2. The `inst` field value format is numeric.
3. For each preload or preload-play command, a unique `inst` field value must be defined, since each time these commands are issued a new template instance is created.
4. Note that when working with the preload and play commands (instead of preload-play), the field values for the play and preload commands must be identical, since both commands will refer to the same template instance.

4.5. The role field

The NRE command supports the concept of initiator *roles*, where different roles have different responsibilities and privileges. At the moment the following roles are recognized:

role	description
schedule (default)	The *schedule* role represents the single initiator generating NRE commands on behalf of the programmed (planned) schedule.
realtime	The *realtime* role represents an optional initiator producing real-time (non-planned, ad hoc) NRE commands.

4.5.1. The schedule role

These are the characteristics of the schedule role:

- No more than one initiator can assume the schedule role.
- The single initiator acting in the schedule role is responsible for implementing the planned playout schedule. Only the schedule role is allowed to preload- and select scene graphs.
- The schedule role initiator has the responsibility (and knowledge) to carefully plan a scene graph switch, and the nexos will perform the switch exactly as instructed. Note that the switch will forcedly stop every running template in the scene graph that is about to be replaced, including templates triggered by the realtime role initiator.

4.5.2. The realtime role

These are the characteristics of the realtime role:

- Zero or more initiators can assume the realtime role.
- An initiator acting in the realtime role will typically trigger additional templates that normally should not interfere with the scheduled templates.
- The templates made available for realtime initiators are explicitly marked as safe for realtime use by the channel designer.
- The realtime role is not allowed to control scene graph switching, and all relevant sub commands and fields will be ignored.
- Commands issued by a realtime initiator will always be executed as soon as possible and the stm and stm-play fields will be ignored.
- Commands issued by a realtime role will always be rendered in the currently active scene graph, even if the scene graph is due for a switch in the near future.
- Commands issued by a realtime role may be aborted while running, or not even be executed at all as a result of scene graph switching dictated by schedule role commands.

Example:
To specify a role, a command initiator must define the `role` field in the NRE command. When the `role` field is not found, the `schedule` role is assumed.

```
{cmd=nre; subcmd=preload; template=ShowLogo; role=realtime; }
```

4.6. Scene parameters

Some sub commands accept *scene parameters*, plug-in values similar to arguments for a function call. Scene parameters come in two parts: the parameter *name* and parameter *value*. Sub commands that accept scene parameters, allow (re-)definition of one or more of these parameters.

In the example below, two scene parameters named 'id' and 'count' are given a value, along with sub command preload:

```
{cmd=nre; subcmd=preload; template=ShowLogo; spname0=id; spvalue0=a0001;
 spname1=count; spvalue1=42;}
```

Note how each of the scene parameters is defined over two numbered fields; a matching `spnameN` and `spvalueN` field.

These are the rules for specifying scene parameters in the command string:

- The parameter name is defined as the value of a spnameN field, while the parameter value is defined as the value of a spvalueN field with identical value for N.
- N is '0' (and not '00' or '000') for the first scene parameter, and increments in steps of 1.
- Breaking the sequence is interpreted as end-of-parameters.
  For example, a sequence of `spname0 spname1 spname3` is seen as just two parameters because the sequence is broken after spname1.
- Any number of scene parameters can be defined (as long as the sequence is kept intact).
- Order of definition of scene parameters is not important.
- Scene parameters are global variables shared by all NRE instances.
- The scene parameter values used by a template instance are fetched and stored as local copies at the start of the preload phase. Changing scene parameter values during the preload or playback phase of an instance will have no effect.

4.7. Selected and active scene graph

It is important here to understand the following two concepts:

1. The *selected* scene graph is the one receiving NRE commands.
2. The *active* scene graph is the one being used for actual rendering. Or in other words, the scene graph that produces output.

In the normally simple situation there is only one scene graph in use, and is both the selected- and active scene graph.

Selecting a scene graph using sub command `sg-select` (or using the `sg-select` field) makes it the new target for all following NRE commands. Note however that the selected scene graph will not immediately be made the *active* one. Instead, this will only happen the moment the first template running on the selected scene graph has finished preloading and starts the playback phase.

Switching active scene graphs is performed with a *hard cut* from the old- to the new scene graph. The switch will abruptly stop all templates still running on the older scene graph.

4.8. Sub command preload-play

Function: preloads a template instance and automatically starts playback, where both the start time of the preload- and playback phase can be optionally controlled.

field name	field type	Value
cmd	mandatory	nre
subcmd	mandatory	preload-play
template	mandatory	Name of the template to be preloaded, as defined in Channel Composer. For example: 'PlayMovie' (without the quotes).
exid	mandatory	External ID. Free format. See above.
inst	mandatory	Unique template instance ID. Numeric. See above.
stm	optional	Start time for the preload phase of the command in hh:mm:ss:ff timecode format. When not specified, the command starts as soon as possible.
stm-play	optional	Start time for the playback phase of the command in hh:mm:ss:ff timecode format. Note that `stm-play` cannot be earlier in time than ready point of preload phase. When not specified, the playback phase starts as soon as the preload phase is ready.
spname0 **spvalue0**	optional	Optional scene parameters. See above.
sg-select	optional	Embedded alternative to separate `sg-select` sub command, where the value for this field must be set to what the `rootfolder` field expects in the `sg-select` command. See section on sub command `sg-select` below for details.
role	optional	Role field. See above.
fb-sw	optional	Feedback switch. Set value to `true` to request feedback.
fb-lev	optional	Feedback detail level. Supported values are `basic`, `most` and `full`. Default is `basic`.

Example:

```
{cmd=nre; subcmd=preload-play; template=PlayMovie; inst=0; exid=0;
 stm=14:13:00:00; stm-play=14:15:30:00; spname0=clipname;
 spvalue0=a0000123.mpg;}
```

Note that newlines do not exist in K2 Edge API commands and that the example above was wrapped for layout purposes.

4.9. Sub command preload

Function: prepares a template for playback.

field name	field type	value
cmd	mandatory	nre
subcmd	mandatory	preload
template	mandatory	Name of the template to be preloaded, as defined in Channel Composer.
exid	mandatory	External ID. Free format. See above.
inst	mandatory	Unique template instance ID. Numeric. See above.
stm	optional	Start time for the preload command in hh:mm:ss:ff timecode format. When not specified, the command starts as soon as possible.
spname0 ... **spvalue0** ...	optional	Optional scene parameters. See above.
sg-select	optional	Embedded alternative to separate `sg-select` sub command, where the value for this field must be set to what the `rootfolder` field expects in the `sg-select` command. See section on sub command `sg-select` below for details.
sg-inst	optional	Scene graph instance ID needed alongside the sg-select field described above.
ttl	optional	Time-to-live value in seconds for all needed assets (clips, etc) claimed during this preload command. The asset resources will be freed after the given time. Make sure `ttl` is long enough to cover the distance in time between this `preload`- and the follow up `play` sub command. Numeric, in seconds. Value 0 indicates no ttl. Value -1 indicates default ttl value as defined in the nexos init params channel section.
role	optional	Role field. See above.
fb-sw	optional	Feedback switch. Set value to `true` to request feedback. See dedicated chapter on feedback.
fb-lev	optional	Feedback detail level. Supported values are `basic`, `most` and `full`. Default is `basic`.


The combination of the template and inst field values uniquely identifies the template instance. The follow up play command must define the same template and inst values.

Example:

```
{cmd=nre; subcmd=preload; template=PlayMovie; inst=0; exid=1;
 stm=14:13:00:00; spname0=clipname; spvalue0=a0000123.mpg;}
```


Newlines do not exist in K2 Edge API commands. Examples are wrapped for layout purposes only.

4.10. Sub command play

Function: starts playback of a template.

field name	field type	value
cmd	mandatory	nre
subcmd	mandatory	play
template	mandatory	Name of the template to be played, as defined in Channel Composer.
exid	mandatory	External ID. Free format. See above.
inst	mandatory	Template instance ID as previously defined in the `preload` sub command. Numeric. See above.
stm	optional	Start time for the play command in hh:mm:ss:ff timecode format. When not specified, the command starts as soon as possible.
role	optional	Role field. See above.
fb-sw	optional	Feedback switch. Set value to `true` to request feedback. See dedicated chapter on feedback.
fb-lev	optional	Feedback detail level. Supported values are `basic`, `most` and `full`. Default is `basic`.

Example:

```
{cmd=nre; subcmd=play; template=PlayMovie; inst=0; exid=2; stm=14:13:10:00;}
```

4.11. Sub command sg-preload

Function: prepares a scene graph for selection by loading it into memory to push it into the scene graph preload queue.

field name	field type	value
cmd	mandatory	nre
subcmd	mandatory	sg-preload
exid	mandatory	External ID. Free format. See above.
rootfolder	mandatory	Absolute path leading to root folder of channel pack holding the targeted scene graph.
sg-inst	mandatory	Unique scene graph instance ID. Numeric.
stm	optional	Start time for the command in hh:mm:ss:ff timecode format. When not specified, the command starts as soon as possible.
role	optional	Role field. See above. Note that only `schedule` roles are allowed to use this sub command.

Notes:

- A preloaded scene graph has no effect on playout unless it is selected via sub command `sg-select` (described below) or via the dedicated `sg-select` field.
- A nexos channel supports a preload queue (a FIFO) for at least two scene graphs. This queue is initially empty.
- This `sg-preload` sub command pushes the scene graph to the end of the queue.
- If the queue was already full when a new scene graph is added, the scene graph waiting at the front of the queue (the oldest of the group) is popped and removed to make place.
- Once preloaded and waiting in the queue, it takes a follow-up `sg-select` sub command to select a scene graph (described below).
- Nexos will automatically preload and select the last used scene graph after a restart.

Example:

```
{cmd=nre; subcmd=sg-preload;
 rootfolder=/publitronic/objects/channelpack/music;
 sg-inst=15; exid=3; stm=14:08:10:00;}
```

Newlines do not exist in K2 Edge API commands. Examples are wrapped for layout purposes only.

4.12. Sub command sg-select

Function: selects a previously preloaded scene graph waiting in the preload queue as the target for incoming NRE commands.

field name	field type	value
cmd	mandatory	nre
subcmd	mandatory	sg-select
exid	mandatory	External ID. Free format. See above.
rootfolder	mandatory	Path to channel pack as used in `sg-preload`, **OR** the reserved keyword `next`. If a path is specified, the scene graph preload queue is scanned for a scene graph preloaded from exactly that path, and if found the scene graph is (a) selected as the current target for incoming NRE commands, and (b) removed from the preload queue. If the keyword `next` is specified however, the scene graph waiting at the front of the preload queue (if any) is (a) selected as the current target for incoming NRE commands, and (b) removed from the preload queue.
sg-inst	mandatory when path is specified for `rootfolder` field.	Unique scene graph instance ID as was specified with the corresponding `sg-preload` sub command.
stm	optional	Start time for the command in hh:mm:ss:ff timecode format. When not specified, the command starts as soon as possible.
role	optional	Role field. See above. Note that only `schedule` roles are allowed to use this sub command.

Notes:

- Please paragraph 4.7 for an explanation of selected and active scene graphs.
- Only a scene graph previously preloaded with `sg-preload` and still waiting in the preload queue can be selected as the new target for incoming NRE commands.
- Once selected, the scene graph is removed from the preload queue, and thus cannot be selected again without another `sg-preload`.
- Selecting a scene graph that cannot be found in the preload queue has no side effects.

Examples:

```
{cmd=nre; subcmd=sg-select;
 rootfolder=/publitronic/objects/channelpack/music;
 sg-inst=15; exid=4; stm=14:08:10:00;}

{cmd=nre; subcmd=sg-select; rootfolder=next; exid=5; stm=14:08:10:00;}
```

4.13. Sub command object-info

Function: returns information via feedback on an NRE object (e.g. a template) with given name as found in either the active- or selected scene graph.

field name	field type	value
cmd	mandatory	nre
subcmd	mandatory	object-info
exid	mandatory	External ID. Free format. See above.
object	mandatory	Name of object as defined in the scene graph. For templates, this is the template name as defined in Channel Composer.
active-sw	mandatory	If `true` the *active* scene graph will be tested for given object, and else the *selected* scene graph will be tested. See section *Selected and active scene graph*, above.
stm	optional	Start time for the command in hh:mm:ss:ff timecode format. When not specified, the command starts as soon as possible.
role	optional	Role field. See above.
fb-sw	optional	Make sure to enable feedback to get the results of the object-info sub command returned.

The requested info will be returned via the standard feedback mechanism, so it is important to enable feedback with `fb-sw=true` with this command.

Two feedback messages will be returned: the standard `ack` or `nak` message followed by a final message holding the requested information. In this final message field `fb-stat` will reflect the presence of the given object in the selected scene graph. Value `ok` indicates the object exists, and value `error` indicates the object could not be found.

When the object was found, field `fb-info` holds the object information. Note that at the moment information will only be returned on template objects, and this information is currently restricted to the set of scene parameters defined for the template.

4.14. Sub command sp-print

Function: prints the values of all scene parameters to the standard out.

field name	field type	value
cmd	mandatory	nre
subcmd	mandatory	sp-print
exid	mandatory	External ID. See above.
stm	optional	Start time for the command in hh:mm:ss:ff timecode format. When not specified, the command starts as soon as possible.
role	optional	Role field. See above.

Example:

```
{cmd=nre; subcmd=sp-print; exid=6; stm=14:00:10:00;}
```

4.15. Sub command control

Function: allows for various levels of control over NRE.

field name	field type	Value
cmd	mandatory	nre
subcmd	mandatory	control
exid	mandatory	External ID. See above.
nosignal-sw	optional	*true* to enable the visibility of no-signal content, i.e. the content that is shown when the original content failed to load.
stats-sw	optional	*true* to display playback statistics.
perfstats-sw	optional	*true* to display playback performance statistics. Requires *stats-sw* to be set to *true* as well.
stm	optional	Start time for the command in hh:mm:ss:ff timecode format. When not specified, the command starts as soon as possible.
role	optional	Role field. See above.

Example:

```
{cmd=nre; subcmd=control; nosignal-sw=true; stats-sw=true;
 exid=7; stm=14:00:10:00;}
```

5. K2 Edge API NRE Feedback

5.1. Introduction

The K2 Edge API NRE commands support a dedicated feedback protocol that has following distinctive features:

- NRE commands allow the preloading and playing of templates. A single template can depend on the presence of several assets.
- The NRE feedback protocol provides feedback for each of these assets in error situations.
- The NRE feedback protocol is based on the life cycle of a template. Many templates, once started, never end unless they are stopped by other templates. The protocol provides feedback on both the moment a template enters its endless running state, as well as on the point it has finished.
- The NRE feedback protocol allows for various levels of feedback detail.

5.2. Supported commands

The following NRE sub commands support the feedback protocol described here:

- **preload-play**
- **preload**
- **play**
- **object-info**

5.3. Asynchronous nature

The K2 Edge API protocol - used for both commands and feedback messages - is asynchronous in nature. This means that:

- Incoming commands are not guaranteed to be processed in order of reception.
- Outgoing feedback is not guaranteed to be sent in the order commands came in.
- Several commands can be sent in succession, before the first feedback from nexos has been received.
- The parallel nature of nexos will allow it to process several commands simultaneously, and as a result return feedback on all of these commands simultaneously.

5.4. The exid field

The `exid` field in the NRE sub commands is a mandatory field, reserved to define an external ID. This ID enables linking feedback to commands. A unique `exid` value must be defined (for example an incrementing serial number) for each command sent to nexos. The `exid` field value makes it is easy to link feedback to commands. This is especially useful when nexos is working on several commands simultaneously. The `exid` field and value will always be returned in the feedback, unmodified.

5.5. Life cycle points and events

When nexos is instructed to preload and/or play an instance of a template (through one of the supported commands), the instance is going through a life cycle. This cycle begins when the command is received and ends when the template instance is finally stopped.

During this life cycle the template instance passes a number of well-defined *points* such as *start* or *play*, and reaching these points is what we call an *event*.

The feedback messages described here report on these events via the `event` field in the feedback message. Possible field values are shown in the right most column in the table below.

The recognized points in the lifecycle of NRE commands are:

point name	phase/type	description	fb-event field value
reception		Reception of a command in nexos: acknowledged or not acknowledged.	ack or nak
start	preload	Start of the preload phase. Preloading is needed to prepare assets for immediate start when a template play is received.	start
error	preload	Optional errors encountered during preloading, for example assets that could not be found or have invalid content. Feedback will only be generated when errors are encountered.	error
ready	preload	End of the preload phase. The template is now ready for playback. If errors were encountered during preload, playing the template will result in limited functionality.	ready
play	playback	Start of the template playback phase.	play
error	playback	Errors encountered during playback, for example problems when accessing new assets needed later down the time line. Feedback will only be generated when errors are encountered.	error
safe	playback	The point in time during playback where a template no longer depends on new media data and is guaranteed not to encounter any errors. From this point on, a template can run forever until stopped.	safe
stop	playback	The point in time where a template instance is explicitly or implicitly stopped. At this point the template instance is removed from memory and can no longer be accessed.	stop
ready	one-shot commands	One-shot commands not associated with a template instance (e.g. sub command `object-info`) only have two points: the standard reception point and a ready point that indicates the finish of the short time the command actually ran.	ready

5.6. Feedback levels

Three feedback levels can be defined:

level	description
basic	The default feedback level. Only the reception point and the first following point are covered. Any following events, including error events will not be reported.
most	This level covers most of the events, including error events. The only event excluded is the *stop* event, which can take a long time to occur. This level informs about all possible problems, without having to wait for the template instance to be actually stopped.
full	This level covers all events, including the *stop* event.

5.7. Enabling feedback

To enable feedback for one of the supported NRE sub commands, follow these steps:

1. Add the `fb-sw=true` field to the NRE command.
2. Optionally add the `fb-lev=xxx` field/value pair to the NRE command, where `xxx` defines the actual feedback level as described above. When not added, nexos will default to `basic` feedback level.
3. Add the `exid=yyy` field/value pair to the NRE command, where `yyy` represents a unique ID that allows the initiator to link feedback messages and commands.
4. Communicate with nexos over a socket link and stay connected, waiting for feedback messages from nexos.

If these conditions are met, nexos will send feedback replies over the socket link used for sending commands. Feedback messages are returned as an updated 'echo' of the original NRE command, enhanced with one or more fields generated by nexos, as described in the following section.

5.8. What is returned

Feedback messages are a copy of the original NRE command, enriched with a number of extra field-value pairs. The `exid` field stays untouched and links the feedback message to the original command.

The *order* of fields in the feedback message is *not* guaranteed to be identical to the one found in the original command.

These are the fields that nexos adds to the feedback message:

field	value	presence	description
fb-event	point name	always	The `fb-event` field shows the name of the point (event) just reached.
fb-stat	ok or error	always	The `fb-stat` field reports on the status of the template instance; `ok` if everything went well up to this point, or `error` from the moment an error is encountered. The `stat` field will stay in this state for the rest of the template's life cycle.
fb-info	Additional info.	optional	The optional `fb-info` field brings additional information when needed. With error events, the `fb-info` field will hold the asset name that caused the trouble. For sub commands that return information (like `object-info`), the `fb-info` field holds the requested information. For most other events the field value is empty.
fb-term-sw	true	optional	The feedback terminator switch field. This field is only present for the terminating feedback message (i.e. the final message in a command's feedback), and when present always has value `true`.
chan, **layer**, **sfd**		always	For nexos internal use only, should be ignored.

5.9. Sub command preload-play

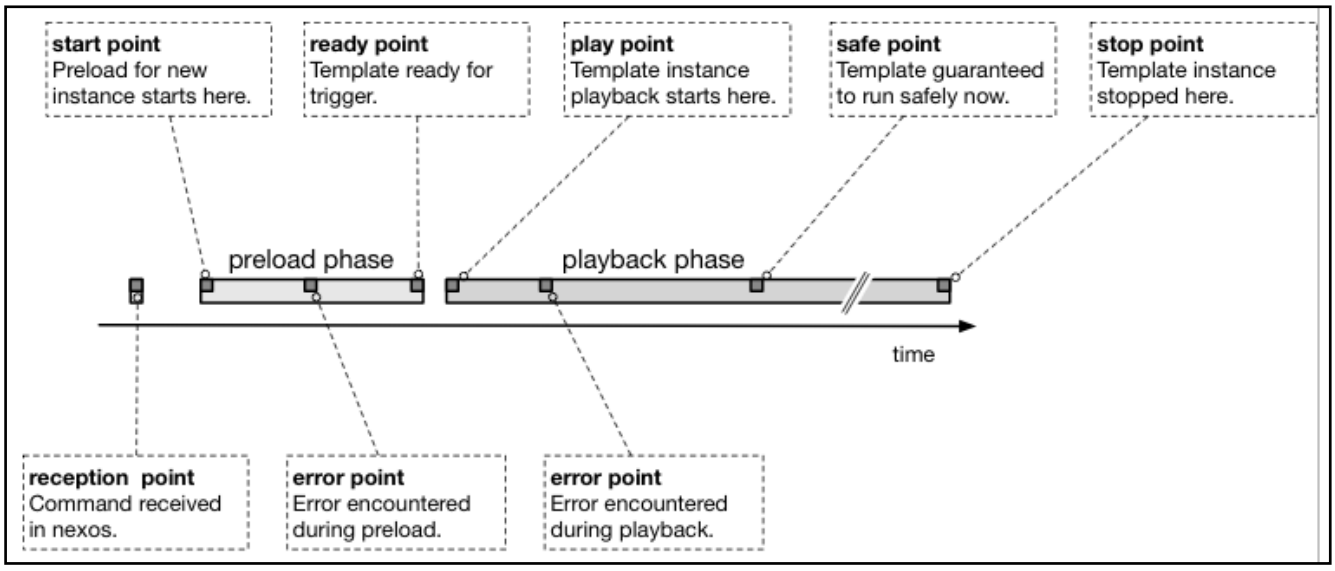
The `preload-play` command combines the individual `preload` and `play` commands described below. Its feedback is almost an exact combination of the feedback produced by these individual commands.

5.9.1. Life cycle diagram

The diagram below shows the life cycle of the `preload-play` command.

The *start* point can be scheduled with the `stm` (start time) field, and if not defined the preload phase starts ASAP.

The *play* point can be scheduled with the `stm-play` field, and if not defined the playback phase starts as soon as the preload phase has completed.



5.9.2. Feedback level 'basic'

Coverage from the *reception* point up to and including the *start* point, spanning only part of the first preload phase. Any event beyond the *start* point is suppressed in the feedback. The only information the initiator will receive is whether or not the (preload part of) the command was successfully started.

The following tables show feedback messages for all possible paths through the life cycle of the command. There is only one success path, and there are always several failure paths. Select a path and read from left to right to see which messages can be expected for which events. The bold text indicates the terminating feedback message.

path	note	field	reception	start	error	ready	play	error	safe	stop
success	(1)	fb-event fb-stat	ack ok	**start** **ok**						
failure	(2)	fb-event fb-stat	**nak** **error**							
failure	(3)	fb-event fb-stat	ack ok	**start** **error**						

Notes:

1) The command is considered valid at the reception point and the preload phase started successfully. No feedback is provided on any of the following events, including possible errors. The template will run until it reaches the *stop* point, possibly with limited functionality due to errors encountered during the preload and/or playback phase of the command.
2) The command is considered invalid at the reception point. The command is not executed.
3) The command is considered valid at the reception point, but failed to start (for example due to a bad template name). The command is not executed.

Example:
Here is an example of a `preload-play` command sent to nexos, and the two feedback messages received from nexos in the default `basic` feedback level. The command instructs nexos to preload and play a template called ShowLogo, instance number 42.

```
{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=42; exid=1;
 fb-sw=true; fb-lev=basic;}
```

The feedback messages from nexos, assuming the success path, are these:

```
{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=42; exid=1;
 fb-sw=true; fb-lev=basic; fb-event=ack; fb-stat=ok;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=42; exid=1;
 fb-sw=true; fb-lev=basic; fb-event=start; fb-stat=ok; fb-term-sw=true;}
```

The messages above were wrapped for layout purposes. Newlines do not exist in K2 Edge API commands.

Note the presence and value of `fb-term-sw` in the last message, indicating feedback ends for this command. Also note that the `fb-info` field was not added to the feedback, since there was nothing to mention.

5.9.3. Feedback level 'most'

Coverage from the *reception* point up to and including the *safe* point, spanning all of the preload phase and most of the playback phase. The only event not covered in the feedback is reaching the *stop* point, which theoretically can be hours away from the *safe* point.

path	note	field	reception	start	error	ready	play	error	safe	stop
success	(4)	fb-event fb-stat	ack ok	start ok		ready ok	play ok		**safe** **ok**	
failure	(2)	fb-event fb-stat	**nak** **error**							
failure	(3)	fb-event fb-stat	ack ok	**start** **error**						
failure	(5)	fb-event fb-stat fb-info	ack ok	start ok	error error asset	ready error	play error		**safe** **error**	
failure	(6)	fb-event fb-stat fb-info	ack ok	start ok		ready ok	play ok	error error asset	**safe** **error**	

Notes:

4) The command is considered valid at the *reception* point. The preload phase completed successfully. The playback phase started and reached the *safe* point without errors. The template will continue running until the *stop* point, but this event is not reported in the feedback.

5) The command is considered valid at the *reception* point. The preload phase started successfully but preload errors were encountered before the *ready* point was reached. The template is played though (with limited functionality). The last event reported in the feedback is reaching the *safe* point. The template will continue to play until it reaches the *stop* point.

6) The command is considered valid at the reception point. The preload phase completed successfully. The playback phase started successfully but errors were encountered before reaching the safe point. The template will continue playing until the stop point is reached, but this event is not reported in the feedback.

Example:

This example of a `preload-play` command sent to nexos uses feedback level 'most'. The command is identical to the one used in the previous example, and again we assume the success path. Note how we incremented both the `inst` and `exid` value; every new command should get new unique values for both fields.

```
{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=43; exid=2;
 fb-sw=true; fb-lev=most;}
```

Feedback messages from nexos, assuming the success path, are listed below. Check the fb-event field values to see how each of the events is reported. Also note that only the last, terminating feedback message has the fb-term-sw=true field.

```
{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=43; exid=2;
 fb-sw=true; fb-lev=most; fb-event=ack; fb-stat=ok;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=43; exid=2;
 fb-sw=true; fb-lev=most; fb-event=start; fb-stat=ok;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=43; exid=2;
 fb-sw=true; fb-lev=most; fb-event=ready; fb-stat=ok;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=43; exid=2;
 fb-sw=true; fb-lev=most; fb-event=play; fb-stat=ok;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=43; exid=2;
 fb-sw=true; fb-lev=most; fb-event=safe; fb-stat=ok; fb-term-sw=true;}
```

5.9.4. Feedback level 'full'

Coverage from the *reception* point up to and including the final *stop* point. Note that the *stop* point can theoretically be hours away from the *safe* point.

path	note	field	reception	start	error	ready	play	error	safe	stop
success	(7)	fb-event fb-stat	ack ok	start ok		ready ok	play ok		safe ok	**stop** **ok**
failure	(2)	fb-event fb-stat	**nak** **error**							
failure	(3)	fb-event fb-stat	ack ok	**start** **error**						
failure	(8)	fb-event fb-stat fb-info	ack ok	start ok	error error asset	ready error	play error		safe error	**stop** **error**
failure	(9)	fb-event fb-stat fb-info	ack ok	start ok		ready ok	play ok	error error asset	safe error	**stop** **error**

Notes:

7) The command is considered valid at the *reception* point. The preload phase completed successfully. The playback phase started and reached the *safe* point without errors. The template will continue running until the *stop* point, and this event will be reported in the feedback.

8) The command is considered valid at the reception point. The preload phase started successfully but preload errors were encountered before the ready point was reached. The template is played though (with limited functionality). The template will continue to play and the events of reaching the safe and stop point will both be reported in the feedback.

9) The command is considered valid at the reception point. The preload phase completed successfully. The playback phase started successfully but errors were encountered before reaching the safe point. The template will continue playing until the stop point is reached, but this event is not reported in the feedback.

Example:
This third example of a `preload-play` command sent to nexos uses feedback level 'full'. The command is again identical to the one used in the previous example (and again with incremented `inst` and `exid` values).

```
{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=44; exid=3;
 fb-sw=true; fb-lev=full;}
```

The feedback messages from nexos are listed. This time we show a failure path, note (8) in the table above. Note the two `fb-event=error` messages reporting on assets a000012.mpg and a000129.tga via the `fb-info` fields. Also note that starting from the first reported error, the `fb-stat` field now shows the error state, and will continue to do so up to the terminating feedback message.

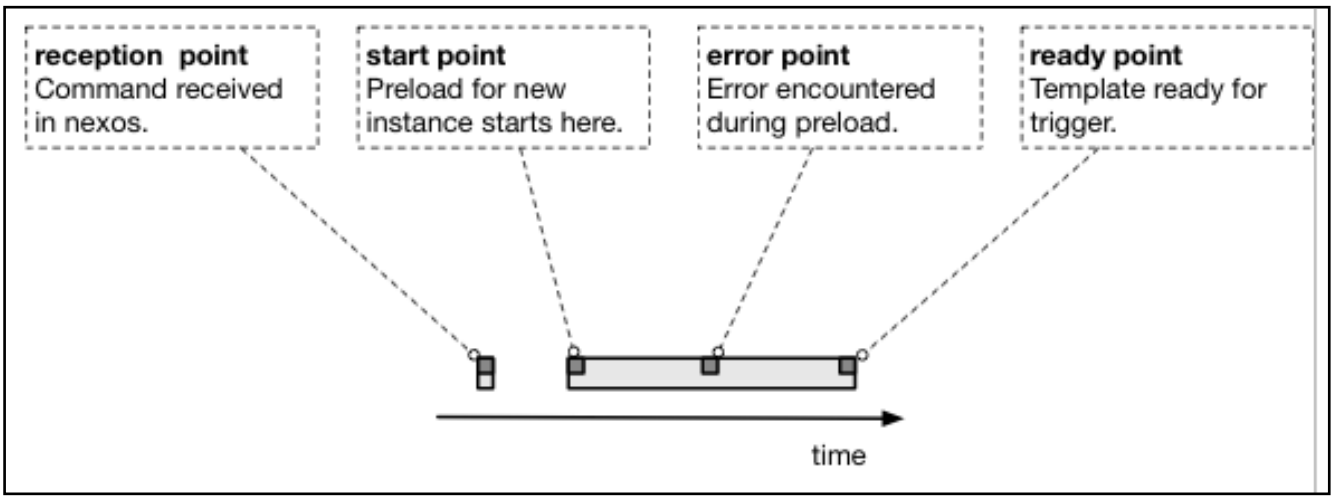
```
{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=44; exid=3;
 fb-sw=true; fb-lev=full; fb-event=ack; fb-stat=ok;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=44; exid=3;
 fb-sw=true; fb-lev=full; fb-event=start; fb-stat=ok;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=44; exid=3;
 fb-sw=true; fb-lev=full; fb-event=error; fb-stat=error;
 fb-info=a000012.mpg;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=44; exid=3;
 fb-sw=true; fb-lev=full; fb-event=error; fb-stat=error;
 fb-info=a000129.tga;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=44; exid=3;
 fb-sw=true; fb-lev=full; fb-event=ready; fb-stat=error;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=44; exid=3;
 fb-sw=true; fb-lev=full; fb-event=play; fb-stat=error;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=44; exid=3;
 fb-sw=true; fb-lev=full; fb-event=safe; fb-stat=error;}

{cmd=nre; subcmd=preload-play; template=ShowLogo; inst=44; exid=3;
 fb-sw=true; fb-lev=full; fb-event=stop; fb-stat=error; fb-term-sw=true;}
```

5.10. Sub command preload

Sub command `preload` prepares a template instance for a later `play` command.

5.10.1. Life cycle diagram

The following diagram shows the life cycle of the `preload` sub command. The *start* point can be scheduled with the `stm` (start time) field, and if not defined the preload phase starts ASAP.



5.10.2. Feedback level 'basic'

Coverage from the *reception* point up to and including the *start* point. Any event beyond the *start* point is suppressed in the feedback. The only information the initiator will receive is whether or not the command was successfully started.

path	note	field	reception	start	error	ready
success	(1)	fb-event fb-stat	ack ok	**start** **ok**		
failure	(2)	fb-event fb-stat	**nak** **error**			
failure	(3)	fb-event fb-stat	ack ok	**start** **error**		

Notes:

1) The command is considered valid at the reception point and the preload command has successfully started. No feedback is provided on any of the following events, including possible errors. The command will run until it reaches the *ready* point, possibly with limited functionality due to errors encountered during the preload process.

2) The command is considered invalid at the reception point. The command is not executed.

3) The command is considered valid at the reception point but failed to start (for example due to a bad template name). The command is not executed.

5.10.3. Feedback levels 'most' and 'full'

Coverage from the *reception* point up to and including the *ready* point, that is, coverage is complete for both feedback levels.

path	note	field	reception	start	error	ready
success	(4)	fb-event fb-stat	ack ok	start ok		**ready** **ok**
failure	(2)	fb-event fb-stat	**nak** **error**			
failure	(3)	fb-event fb-stat	ack ok	**start** **error**		
failure	(5)	fb-event fb-stat fb-info	ack ok	start ok	error error asset	**ready** **error**

Notes:

4) The command is considered valid at the reception point. The preload command has successfully completed.

5) The command is considered valid at the reception point. The preload command has successfully started but preload errors were encountered before the ready point was reached.

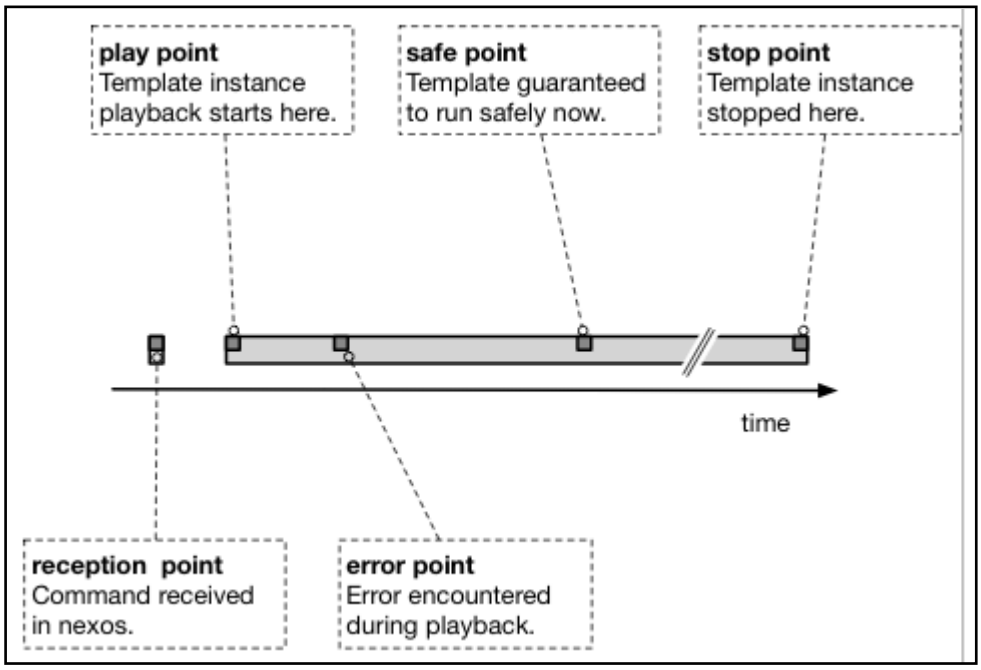
5.11. Sub command play

Sub command `play` starts a template instance at the start time, provided it has been previously preloaded with the `preload` command.

5.11.1. Life cycle diagram

The following diagram shows the life cycle of the `play` sub command.

The *start* point can be scheduled with the `stm` (start time) field, and if not defined the preload phase starts ASAP.



5.11.2. Feedback level 'basic'

Coverage from the *reception* point up to and including the *play* point. Any event beyond the *play* point is suppressed in the feedback. The only information the initiator will receive is whether or not the command was successfully started.

path	note	field	reception	play	error	safe	stop
success	(1)	fb-event fb-stat	ack ok	**play** **ok**			
failure	(2)	fb-event fb-stat	**nak** **error**				
failure	(3)	fb-event fb-stat	ack ok	**play** **error**			

Notes:

1) The command is considered valid at the reception point and the command started successfully. No feedback is provided on any of the following events, including possible errors. The template will run until it reaches the *stop* point, possibly with limited functionality due to errors encountered during playback.

2) The command is considered invalid at the reception point. The command is not executed.

3) The command is considered valid at the reception point but failed to start. The command is not executed.

5.11.3. Feedback level 'most'

Coverage from the *reception* point up to and including the *safe* point. The only event not covered in the feedback is reaching the *stop* point, which theoretically can be hours away from the *safe* point.

path	note	field	reception	play	error	safe	stop
success	(4)	fb-event fb-stat	ack ok	play ok		**safe** **ok**	
failure	(2)	fb-event fb-stat	**nak** **error**				
failure	(3)	fb-event fb-stat	ack ok	**play** **error**			
failure	(5)	fb-event fb-stat fb-info	ack ok	play ok	error error asset	**safe** **error**	

Notes:

4) The command is considered valid at the reception point. The play command started and reached the safe point without errors. The template will continue running until the stop point, but this event is not reported in the feedback.

5) The command is considered valid at the reception point. The play command started successfully but errors were encountered before reaching the safe point. The template will continue playing until the stop point is reached, but this event is not reported in the feedback.

5.11.4. Feedback level 'full'

Coverage is complete from *reception* point up to and including final *stop* point. Note that the *stop* point theoretically can be hours away from the *safe* point.

path	note	field	reception	play	error	safe	stop
success	(6)	fb-event fb-stat	ack ok	play ok		safe ok	**stop** **ok**
failure	(2)	fb-event fb-stat	**nak** **error**				
failure	(3)	fb-event fb-stat	ack ok	**play** **error**			
failure	(7)	fb-event fb-stat fb-info	ack ok	play ok	error error asset	safe error	**stop** **error**

Notes:

6) The command is considered valid at the *reception* point. The play command started and reached the *safe* point without errors. The template will continue running until the *stop* point, and this final event will be reported in the feedback.

7) The command is considered valid at the reception point. The play command successfully started but errors were encountered before reaching the *safe* point. The template will continue playing until the *stop* point is reached, and this event will be reported in the feedback.

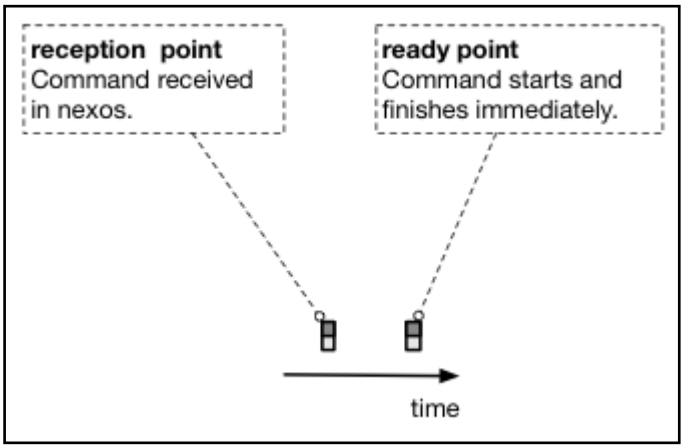
5.12. Sub command object-info

Sub command `object-info` returns info on a given NRE object via the feedback protocol.

5.12.1. Life cycle diagram

The following diagram shows the life cycle of the `object-info` sub command. Note that the command has no length, once started it stops immediately.

The execution of the sub command at the *ready* point can be scheduled with the `stm` (start time) field, and if not defined the command starts
ASAP.



5.12.2. Feedback level 'basic', 'most' and 'full'

Coverage is from the *reception* point up to and including the (only) *ready* point for all feedback levels. The requested object information is only available in the feedback for the ready point, and only if the object was actually found, indicated by the `fb-stat` field.

path	note	field	reception	ready
success	(1)	fb-event fb-stat fb-info	ack ok	**ready** **ok** object info
failure	(2)	fb-event fb-stat	**nak** **error**	
failure	(3)	fb-event fb-stat fb-info	ack ok	**ready** **error** error info

Notes:

1) 1) The command is considered valid at the reception point and the command completed successfully. The requested object was found, and the associated information is returned through the fb-info field.

2) The command is considered invalid at the reception point. The command is not executed.

3) The command is considered valid at the reception point but failed to execute because either the requested object was not found in the indicated scene graph, or some other problem occurred. The fb-info field will describe the actual problem.

At the moment object-info will only work for template objects. The information returned on template objects is the list of scene parameters.

NRE object type	fb-info value	comment
template	{spname0=xxx; spname1=yyy;}	List of scene parameter names defined for the given template. Note that the return value is a complex string. The list will be empty for templates that have no scene parameters defined.

Example:
Here is an example of an object-info command sent to nexos, and the two feedback messages received from nexos in the default basic feedback level. (The other feedback levels would have produced the exact same result for this command.)

The command instructs nexos to return information on the NRE template object named ShowLogo.

```
{cmd=nre; subcmd=object-info; object=ShowLogo; exid=4;
 fb-sw=true; fb-lev=basic;}
```

The feedback messages from nexos, assuming the success path, are these:

```
{cmd=nre; subcmd=object-info; object=ShowLogo; exid=4;
 fb-sw=true; fb-lev=basic; fb-event=ack; fb-stat=ok;}

{cmd=nre; subcmd=object-info; object=ShowLogo; exid=4;
 fb-sw=true; fb-lev=basic; fb-event=ready; fb-stat=ok; fb-term-sw=true;
 fb-info={spname0=logoname; spname1=widthpx; spname2=heightpx;};}
```


The messages above were wrapped for layout purposes. Newlines do not exist in K2 Edge API commands.

Note again the presence and value of `fb-term-sw` in the last message, indicating feedback ends for this command. Also note that the `fb-info` field value is a *nested complex string*, that is by itself terminated with a semicolon, as all field values are.

6. K2 Edge API Source String Format

6.1. Introduction

The source string format defines an asset source, using several field-value pairs. When nexos is told to preload and play an asset, the asset' source file must be defined. The simplest way to do this is by specifying a file name, as in *"A00001.mpg"*. However, when only a subsection of the asset must be preloaded and played, or when a specific selection of the available AV streams must be specified, a simple file name will not work. The complex source string format allows describing a source as a group of field-value pairs.

6.2. The vstream and astream fields

The vstream and astream fields (described below) allow for the selection and mapping of input video- and audio streams to the video decoder's video- and audio outputs. For example, if the sum of the specified input files provides more than one video elementary stream to play for a clip, or if only a subset is needed from the sum of available audio streams (and possibly in a different order), the vstream and astream fields are needed to define a *routing table.* This table connects a selected set of input streams to decoder outputs ports. The decode output ports are connected to the nexos render engine.

6.2.1. vstream fields

The vstream0 and vstream1 fields represent the decoder's two video output ports. The vstream0 and vstream1 fields instruct the decoder which of the available video elementary streams are to be used for the main (vstream0) and secondary video decoder outputs. When playing a clip, only the main video output is needed. When playing an animation, both video outputs are needed (the second to provide the alpha channel).

The value of vstream0 and vstream1 represents an input video elementary stream, indicated by a small integer index number. This number identifies a video elementary stream, among all the video streams provided by all input files. The first video stream found receives index number 0. The stream index is then incremented for all the video streams found in the same and following input files, where the files are scanned in the order specified. Note that non-video streams are simply ignored in the count.

For example, assume we have two input files. File0 provides 2 video streams and 2 audio streams, and File1 provides 1 video stream. In this case, the first input video stream (in File0) has index #0, while the single video input stream in File1 (the third video stream) has index #2.

6.2.2. astream fields

The astream0, astream1... up to and including astream15 fields represent the decoder's 16 mono audio outputs. Internally, nexos only works with mono audio streams, and the master audio mixer used in the render engine supports up to 16 mono channels.

The value of the astream fields represents an input mono audio channel. Note that this is not an *audio elementary stream*. The reason is that audio elementary streams often are multichannel streams, delivering a number of mono audio channels in a single stream. For the decoder's audio routing table to make sense, the astream fields must specify which mono audio input channel must be connected to which of the decoder's audio mono output ports.

An input audio mono channel is indicated with a small integer index number, which identifies a given input mono audio channel among all the mono channels provided by all the audio elementary streams, provided by all the input files. The first audio mono channel found in the first audio elementary stream receives index number 0. The channel index is then incremented for all the mono channels found in the same- and following audio elementary streams, and the counting is continued for the following input files,

where the files are scanned in the order specified. Note that non-audio streams are ignored.

For example, assume we have three input files. File0 provides a single video stream and two stereo audio streams. File1 holds a single 4-channel audio stream. File2 holds a video stream and an 8-channel audio stream. In total 4 + 4 + 8 mono audio input channels are available. The mono audio channel index numbers then look like this:

- File0: index 0, 1 and 2, 3 for the four mono channels found in the two stereo audio streams.
- File1: index 4, 5, 6 and 7 for the four mono channels found in the 4-chan audio stream.
- File2: index 8, 9, ... 15 for the eight mono channels found in the 8-chan audio stream.

Now to connect the first two mono audio channels from the 8-chan audio elementary stream from File2 to the first two audio output port of the decoder:

```
astream0=8; astream1=9;
```

Keep in mind that the astream0 and astream1 fields represent the decoder's mono audio output ports #0 and #1.

To combine the last mono audio channel from File0 and the second mono audio channel from File1 to form a stereo decoder output:

```
astream0=3; astream1=5;
```

6.2.3. Single mono channel to several outputs

Note that it is possible to connect an input mono audio channel to several of the decoder's outputs, for example to make a single mono channel into a stereo (dual mono) channel:

```
astream0=4; astream1=4;
```

However, it is *not* possible to mix two input mono audio channels into a single decoder mono output port:

```
astream0=4; astream0=5;      INVALID!!
```

6.3. Supported fields

A complex source string is limited to a set of well-defined field/value pairs, some of them mandatory, the others optional.

6.3.1. Input file fields

These fields specify the media file(s) to be played. It is possible to compose a clip out of several essence files, for example a video file and number of audio files. Normally, the `file0` field will be sufficient to define an input file that holds all the needed streams.

field name	field type	value	description	since
file0	mandatory	file name or path	The name of the first (and possibly only) input media file, including extension. Relative or absolute paths are allowed. Examples: a000123.avf, or clips/a000123.mpg, or ../clips/a00012.mov, or /some/absolute/path/a000123.mxf.	1.56
file			Identical to `file0` field described above. *Deprecated since nexos-v1.56.* Please use the `file0` field from now on.	
file1 ... **fileN**	optional	file name or path	The name of extra input media files, use for clips that use separate physical files for the different elementary streams. The first additional input file must be defined with field `file1`, followed by `file2`, `file3`, etc. No gaps are allowed in the field name numbering. Input media files are processed in order of numbering. See field `file0` for supported formats.	

6.3.2. Routing table fields

The following fields set up the routing table, where video- or mono audio input streams are mapped to specific decoder output ports. For more details see the vstream and astream fields section above.

field name	field type	value	description	since
vstream0	optional	video elementary stream index number [0,...]	Links an input video elementary stream to the decoder's video output port #0. Specifies the index number for the first video elementary stream from the imaginary stack of video streams supplied by the sum of the input files. For normal clips this is the video stream used for playout. For animations, this is the stream used for the fill data. See the section on vstream fields.	
vstream1	optional	video elementary stream index number [0,...]	Links an input video elementary stream to the decoder's video output port #1. Specifies the second video elementary stream, see `vstream0` for details. This field is only needed for animations, where `vstream1` specifies the key (alpha channel) data.	
astream0	optional	audio mono channel index number [0,...]	Links an input mono audio channel to the decoder's mono audio output port #0. Specifies the first input audio mono channel from the imaginary stack of input mono channels supplied by the sum of the input files. See the section on astream fields. When neither the `vstream0` nor `astream0` fields are defined, all of the video- and audio streams are used	
astreamN	optional	audio mono channel index number	Links an input mono audio channel to the decoder's mono audio output port #*N*. Specifies the next input mono audio channel to be used for output, where N is in the range [1-15]. See the section on astream fields.	

6.3.3. Timecode range fields

The following timecode range fields allow the initiator to play only a specific range of the selected input files. If only a start time is specified, the media file(s) will play from that start time until end of file. If only an end time is specified, the media file(s) will play from begin of file up to *and including* the specified end time. When several input files are involved, it is possible to define individual timecode ranges for each of the files.

field name	field type	value	description	since
tc-start	optional	hh:mm:ss:ff timecode string in timecode format of media file	The timecode of the first frame of the range to be played. This start time applies to all input files, unless overruled by a dedicated `fileN-tc-start` or `fileN-rtc-start` field. If not specified, playback will start with the first valid frame found in the file.	
tc-end	optional	hh:mm:ss:ff timecode string in timecode format of media file	The timecode of the last frame of the range to be played. That is, this time is *inclusive*. This end time applies to all input files, unless overruled by a dedicated `file`*`N`*`-tc-end` or `fileN-rtc-end` field. If not specified, playback will continue up to and including the last valid frame found in the file.	
file*N*-tc-start	optional	hh:mm:ss:ff timecode string in timecode format of media file	Similar to the `tc-start` field, but only applies to the media file defined with the associated `file`*`N`* field. N is in the range [0, ...]. For example, if a complex source string defines a clip comprised of three media files, it is possible to define an individual tc-start time for the last input file with something like `tc-start=00:11:00:00; file2-tc-start=00:10:42:00;`	1.54
file*N*-tc-end	optional	hh:mm:ss:ff timecode string in timecode format of media file	Similar to the `tc-end` field, but only applies to the media file defined with the associated `file`*`N`* field. N is in the range [0, ...]. See `fileN-tc-start` for an example.	1.54

6.3.4. Relative timecode range fields

Similar to the timecode range fields above, but this time with *relative timecode values* (i.e. relative to start of file), meaning that the first valid frame produced by the decoder will always be referred to as having time value 00:00:00:00.

field name	field type	value	description	since
rtc-start	optional	hh:mm:ss:ff timecode string in timecode format of media file	Relative timecode of the first frame of the range to be played. *Relative* meaning that the first valid video frame always has time 00:00:00:00. This start time applies to all input files, unless overruled by a dedicated `fileN-tc-start` or `fileN-rtc-start` field. If not specified, playback will start with the first valid frame found in the file.	1.56
rtc-end	optional	hh:mm:ss:ff timecode string in timecode format of media file	Relative timecode of the last frame of the range to be played. That is, this time is *inclusive*. *Relative* meaning that the first valid video frame always has time 00:00:00:00. This end time applies to all input files, unless overruled by a dedicated `file`*`N`*`-tc-end` or `fileN-rtc-end` field. If not specified, playback will continue up to and including the last valid frame found in the file.	1.56
file*N*-rtc-start	optional	hh:mm:ss:ff timecode string in timecode format of media file	Similar to the `rtc-start` field, but only applies to the media file defined with the associated `file`*`N`* field. N is in the range [0, ...]. For example, if a complex source string defines a clip comprised of three media files, it is possible to define an individual rtc-start time for the last input file with something like `rtc-start=00:11:00:00; file2-rtc-start=00:10:42:00;`	1.56
file*N*-rtc-end	optional	hh:mm:ss:ff timecode string in timecode format of media file	Similar to the `rtc-end` field, but only applies to the media file defined with the associated `file`*`N`* field. N is in the range [0, ...]. See `fileN-rtc-start` for an example.	1.56

6.3.5. Absolute timecode range fields for MXF files

Similar to the timecode range fields above, but this time with *absolute timecode values*, i.e. absolute, with the starttime timeCodeStart from the MXF header.

field name	field type	value	description	since
atc-start	optional	hh:mm:ss:ff timecode string in timecode format of media file	Absolute timecode of the first frame of the range to be played. *Absolute* meaning that the first valid video frame always has time timeCodeStart from the MXF header. This start time applies to all input files, unless overruled by a dedicated `fileN-tc-start` or `fileN-atc-start` field. If not specified, playback will start with the first valid frame found in the file.	Multiplat
atc-end	optional	hh:mm:ss:ff timecode string in timecode format of media file	Absolute timecode of the last frame of the range to be played. That is, this time is *inclusive*. *Absolute* meaning that the first valid video frame always has time startTimeCode from MXF header. Warning: Due to technical design, atc-end can only be used if atc-start is also used!!! This end time applies to all input files, unless overruled by a dedicated `fileN-tc-end` or `fileN-atc-end` field. If not specified, playback will continue up to and including the last valid frame found in the file.	Multiplat
file*N*-atc-start	optional	hh:mm:ss:ff timecode string in timecode format of media file	Similar to the `atc-start` field, but only applies to the media file defined with the associated `file`*`N`* field. N is in the range [0, ...]. For example, if a complex source string defines a clip comprised of three media files, it is possible to define an individual atc-start time for the last input file with something like `atc-start=10:11:00:00; file2-atc-start=10:10:42:00;`	Multiplat

file*N*-atc-end	optional	hh:mm:ss:ff timecode string in timecode format of media file	Similar to the `atc-end` field, but only applies to the media file defined with the associated `file*N*` field. N is in the range [0, ...]. Warning: Due too technical design, atc-end can only be used if atc-start is also used! See `fileN-atc-start` for an example.	Multiplat

6.3.6. Lip sync fields

The fields below control the lip sync feature. By default, when a clip is started, the decoder assumes that the audio and video streams are in sync. If this is not the case, the lip sync feature should be enabled.

field name	field type	value	description	since
clip-lipsync-sw	optional, semi-auto	true or false	When enabled, the source media file(s) are opened with extra care for correct sync between video and audio streams. This is normally not needed for correctly encoded media files. This field will automatically be generated by nexos when the identically named channel parameter is found enabled in the *nexos-init-params.txt* file, but only if not already explicitly defined by the initiator. Use `clip-lipsync-sw` when the media file is started without any of the tc-start or rtc-start fields, and the video and audio is not in sync.	
clip-lipsync-offset-ms	optional, semi-auto	[-10000, 10000]	Specifies an explicit offset in milliseconds between video- and audio streams. This offset applies to all audio streams from all input files unless `fileN-lipsync-offset-ms` is defined for one or more of the input files. This field will automatically be generated by nexos when the identically named channel parameter is found in the *nexos-init-params.txt* file, but only if not already explicitly defined by the initiator. Since nexos-v1.56 the specified offset works even when `clip-lipsync-sw` is disabled.	

file*N*-lipsync-offset-ms	optional	[-10000, 10000]	Specifies an explicit lipsync offset in milliseconds between video- and all audio streams provided by input file *N*. N is in the range [0, ...]. For example, `file0-lipsync-offset-ms` defines the lipsync offset for the audio streams provided by input file #0, which is defined by the `file` field, and `file2-lipsync-offset-ms` defines the offset for input file #2, defined by the `file2` field. This field overrides the global lipsync offset value defined with the `clip-lipsync-offset-ms` field for the associated input file. The specified offset works even when `clip-lipsync-sw` is disabled.	v1.53
clip-mpeg-prescan-sw	auto	true or false	This field is a copy of the identically named nexos channel parameter found in the *nexos-init-param.txt* file. It is automatically added by nexos when the parameter is found enabled. When enabled, and `tc-start` is defined as well, a corrupt mpeg video stream is assumed, and some (non-destructive) repair is done before the tc-start jump.	

7. Channel Pack Management

7.1. Introduction

When designing a channel, all elements of the channel design are created and organized in Channel Composer. These elements, such as templates, graphic stills, animations, fonts, etc, are kept in a Channel Composer project. Once the channel design is finished, it needs to be brought on-air. In Channel Composer this results in what can be considered an end product of the design process, called a *channel pack*. A channel pack is a file in which all design elements are bundled for playout. Channel pack files have the file extension ".cpk".

7.2. Channel pack contents

A channel pack contains the following items:

1. **Scene graph**
The scene graph contains all objects, templates and scene parameters that were defined during the design process.

2. **Assets**
All assets that were imported during the design process are stored in content-type specific folders within the channel pack. The following folders can be found in a channel pack:

ani	Animations
applets	Applet binaries
audio	Audio files
fonts	Font files
meshclips	Mesh clip files
metadata	Metadata definition files. These files are not used during playout (only used at design time by Channel Composer)
stills	Still graphic files
video	

3. **Channel Composer project**
This is the project file used by Channel Composer and it is not used during playout (only used at design time by Channel Composer).

4. **Contents description**
The contents description file (contents.xml) is intended for third-party implementers and describes the contents of a channel pack in detail in XML-format. This file can be consulted when you want to know exactly which templates, applets and scene parameters part of a channel pack.

7.3. Channel pack workflow

The workflow to bring a channel pack on-air contains the following steps:

1. **Build a channel design**
   Build the channel design in Channel Composer. This results in a channel pack that is created via one of the export options (detailed in the next chapter).

   Note that when working with a main and backup playout server, following steps have to be performed on both the main and backup playout servers.

2. **Transport the channel pack to a playout server**
   For a channel pack to be usable on-air, it needs to be physically present on the playout server. The channel pack thus needs to be transported to the playout server. Channel packs are transferred using the FTP-protocol.

   Channel pack files must be placed in directory /system/objects/channelpack on the playout server(s).

3. **Extract the channel pack**
   To extract the channel pack file, use the `cpkmgr` command-line tool. The cpkmgr tool comes pre-installed on all playout servers and is located in the /system/objects/code folder.
   For more information on available command-line options, please run the cpkmgr tool on the command-line using the "-h" (help) option: `./cpkmgr -h`

   The cpkmgr tool will unpack the channel pack file and preload the bundled scene graph. Preloading of the scene graph requires you to specify a start time. The channel pack is unpacked in a folder within /system/objects/channelpack with the same name as the channel pack file (without the .cpk extension).

   The following example extracts the channel pack file "MusicChannelV1.cpk" into /system/objects/channelpack/MusicChannelV1. After unpacking the channel pack file, cpkmgr will preload the bundled scene graph on channel 0 on (start time) 10:00:00:00.

   ```
   ./cpkmgr -u /system/objects/channelpack/MusicChannelV1.cpk 0 10:00:00:00
   ```

4. **Preload the scene graph**
   The scene graph that is part of the channel pack has to be preloaded. This is done by sending the corresponding `sg-preload` sub command to the playout server.

5. **Select the graph scene**
   Once preloaded, the scene graph is waiting to be selected as the new target for incoming NRE commands. One can select a scene graph using the `sg-select` sub command.

7.4. How to bring a Channel Pack on-air

The previous section generically described the steps that are required to bring a channel pack on-air in third party environments. In practice, two options are available to do this:

1. Use Channel Director to perform these steps, or
2. Bring the channel pack on-air manually.

The following sections will explain these two options in more detail.

7.4.1. Channel Director option

The easiest way to bring a channel pack on-air is via Channel Composer using the **Export to playout** option. The export to playout option can be found under the **File** menu in Channel Composer. You will be guided through the export process using a wizard-type interface. During the export you are required to specify the IP-addresses of the applicable playout servers. Additionally, you are required to specify the start time that will be used for preloading the bundled scene graph.

The **Export to Playout** option in Channel Composer performs steps 1 through 4 in the channel pack workflow. Note that you are still required to select the bundled scene graph yourself (step 5 in the workflow).

7.4.2. Manual option

When choosing this option, you are expected to perform some of the steps in the channel pack workflow yourself. The following steps are required:

1) Create a channel pack file by using the **Export to Disk** option in Channel Composer.
2) Transport the channel pack file to one or more playout servers using the FTP-protocol [see the previous chapter *Channel pack workflow* for more details].
3) Use the `cpkmgr` command-line tool to unpack the channel pack [see the previous section *Channel pack workflow* for more details]. The cpkmgr tool will preload the bundled scene graph after unpacking. The scene graph preloading requires you to specify a start time.
4) Select the preloaded scene graph.

8. Complex String Format

8.1. Introduction

A complex string is a text string defining zero or more field-value pairs in a simple format.

Example:

```
{ file=a000123.avf; block-count=42; }
```

A complex string always starts with an open curly-brace character and ends with a close curly brace character. In between the braces zero or more field-value pairs are defined, each of them terminated with a semi-colon.

The term complex refers to the complex data type. A complex string can hold several values as apposed to a simple data type, that can only hold one.

8.2. Field-value pair components

A field-value pair consists of:

1. the field-identifier;
2. an equals sign;
3. the optional field-value;
4. a semi-colon;
5. optional whitespace.

This is how the complex string format looks in pseudo grammar:

```
complex-string:
    { whitespace_opt field-value-seq whitespace_opt }

field-value-seq:
    field-value-pair_opt
    field-value-seq field-value-pair

field-value-pair:
    field-identifier=field-value_opt ; whitespace_opt

field-identifier:
    [a-z][a-z0-9-]*

field-value:
    [a-zA-Z0-9 -\;]*
    #qot(...)toq#
    #u08(...)80u#
    complex-string
```

A complex string contains zero or more field-value pairs between curly braces. Every field-value pair is terminated by a semicolon.

Whitespaces between field-value pairs are allowed and are ignored. Whitespaces in a field-value item are allowed and included as part of value.

8.3. Field identifier

The field identifiers (or field names) consist of lower case letters, digits and the hyphen '-' (not the underscore). Field identifiers always start with a lower case letter. In the example above, `file` is one of the field identifiers.

Examples of valid and invalid field identifiers are:

identifier	comment
`file`	Valid field name.
`bock-size`	Valid field name.
`channel-0`	Valid field name.
`File`	Invalid! No uppercase characters allowed.
`video_stream`	Invalid! No underscores allowed.
`0file`	Invalid! Field identifiers must start with lower case alpha character.

8.4. Field value

The field value is a string of characters. The string starts right after the equals sign (=) and is terminated by the semi-colon (;). the semi-colon ends the field-value-pair. All characters found between the = and ; delimiters are part of the value string, including any spaces found. The value part can be an empty string, clearing the associated field.

8.5. Nested complex string field values

Field values can be nested. For example:

```
{ count=42; list={a=1;b=2;c=3;}; }
```

This example shows a complex string with two fields: `count` and `list`. The value for `list` is a (nested) string that looks like this: `{a=1;b=2;c=3;}`.

Note how the value part of field `list` is terminated with the standard semicolon, as is done with all field values.

Nesting of complex strings is not restricted, but readability will suffer when nesting more than one level deep.

8.6. Data wrappers

The complex string format allows for data to be wrapped by a number of wrapper functions. An example:

```
{ count=42; text=#qot(Hello world!)toq#; }
```

This example demonstrates the `qot()` wrapper that quotes the text between parenthesis to prevent it from being misinterpreted by a complex string parser.

Wrapper functions apply to the following rules:

1. Wrapper functions always start with a `#` sign at the first character of value part, right after the = sign.
2. Next, a three-letter function name (currently either `qot` or `u08`)
3. Next, the open parenthesis, emulating a function calls like construct.
4. Next the data being wrapped, *Hello world!* in our example.
5. And finally 1, 2 and 3 mirrored, as shown in the example above.

8.6.1. The qot() wrapper

The qot() wrapper function *quotes* the wrapped text, preventing the complex string parser from misinterpreting the text for formal language.

For example:

```
text=#qot(one;two;three)toq#;
```

As a general rule, `qot()` should be used with any free-from text as it may contain characters that have a special meaning in the complex string format syntax.

8.6.2. The u08() wrapper

The `u08()` wrapper function is used to store UTF8 (8-bit) data in a field, and prevents the complex string parser from misinterpreting the wrapped text (just like `qot)`.

An example:

```
text=#u08(hello world)80u#;
```