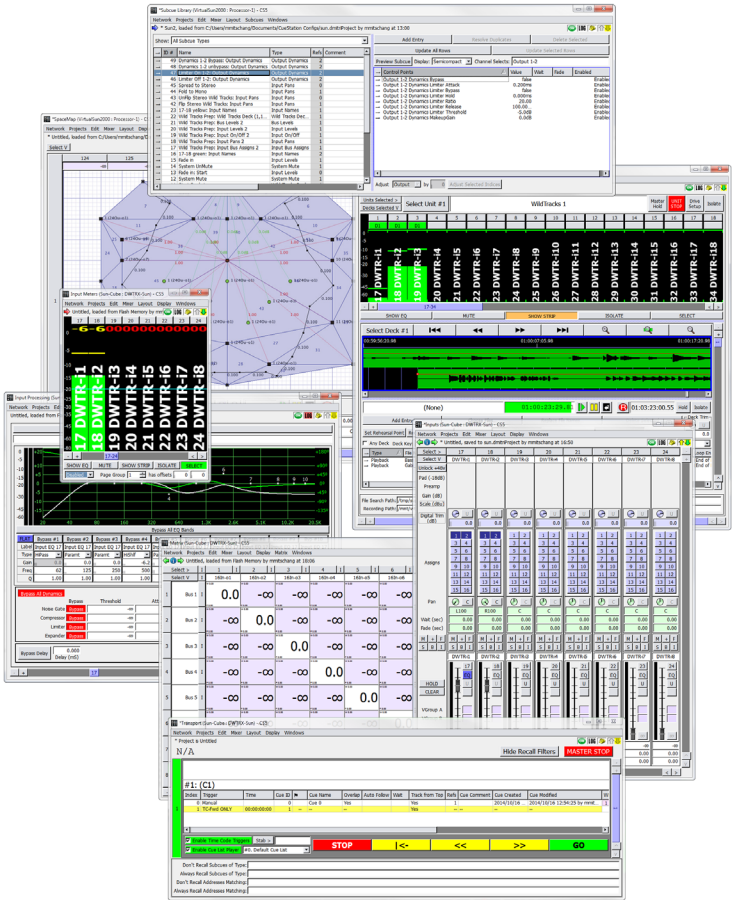
CueStation OSC

thinking sound Meyer® Sound



OSC

open sound control

Keep these important operating instructions.
Check www.meyersound.com for updates.

CueStation OSC User Guide, PN 05.176.130.01 A

CONTENTS

CHAPTER 1: INTRODUCTION

HOW TO USE THIS MANUAL

As you read these instructions, you will encounter the following icons for notes, tips, and cautions:

NOTE: A note identifies an important or useful piece of information relating to the topic under discussion.

TIP: A tip offers a helpful tip relevant to the topic at hand.

CAUTION: A caution gives notice that an action may have serious consequences and could cause harm to equipment or personnel, or could cause delays or other problems.

Information and specifications are subject to change. Updates and supplementary information are available at www.meyersound.com.

Meyer Sound Technical Support is available at:

- **Tel:** +1 510 486.1166
- **Tel:** +1 510 486.0657 (after hours support)
- **Web:** www.meyersound.com/support
- **Email:** techsupport@meyersound.com

CUESTATION OSC

The purpose of this guide is to allow users and programmers to have a reference document regarding the OSC server in Meyer Sound's CueStation 5 software. It reviews basic network implementations, shows the differences between the UDP and TCP transmission protocols, introduces Open Sound Control in general, explains the server on the CueStation side, and provides an as exhaustive as possible reference for each control point available in the system.

This document assumes the reader has some familiarity with the OpenSoundControl protocol. If not, please read about OpenSoundControl at their web page:

http://www.cnmat.berkeley.edu/OpenSoundControl/

Disclaimer

This work has been compiled from many sources. It is not meant to replace those, as well as proper documentation and education regarding networking. The D-Mitri audio control system can provide very loud sound pressure levels to the audience, therefore proper network security, risk assessment, and discipline are strongly recommended before using remote control over a network using third-party software. The authors of this work shall not hold any responsibility for the misuse of this document, for improper implementation, or carelessness.

CHAPTER 2: NETWORKING

GENERAL

Networking is a crucial part of communication between machines. Almost all machines use network rather than point-to-point communication. Managing networks and network security are tasks that carry responsibility because of the actions that can be triggered remotely. It is beyond the scope of this user guide to be exhaustive about the use of networks, but it is worth noting that proper network management is essential. D-Mitri uses an IPv6 address range by default, and can be configured to use an IPv4 address range. It is controlled by Meyer Sound's CueStation control software, using a proprietary network control protocol over TCP/IP. The server is multi-client; it allows multiple clients to be connected at the same time.

TRANSMISSION PROTOCOLS

The OSC server on CueStation listens to two different networking protocols, each with different advantages and constraints: TCP/IP and UDP.

TCP/IP

The Transmission Control Protocol is a connection-oriented protocol, which means that it requires handshaking to set up end-to-end communication. Once a connection is set up, user data may be sent bi-directionally over the connection.

- Reliable – TCP manages message acknowledgment, retransmission and timeout. Multiple attempts to deliver the message are made. If some part of the data is lost along the way, the server will re-request the lost part. In TCP, there's either no missing data, or, in case of multiple timeouts, the connection is dropped.
- Ordered – If two messages are sent over a connection in sequence, the first message will reach the receiving application first. When data segments arrive in the wrong order, TCP buffers delay the out-of-order data until all data can be properly re-ordered and delivered to the application.
- Heavyweight – TCP requires three packets to set up a socket connection before any user data can be sent. TCP handles reliability and congestion control.

- Streaming – Data is read as a byte stream; no distinguishing indications are transmitted to signal message (segment) boundaries.

UDP

The Universal Data Protocol is a simpler message-based connectionless protocol. Connectionless protocols do not set up a dedicated end-to-end connection. Communication is achieved by transmitting information in one direction from source to destination without verifying the readiness or state of the receiver.

However, one primary benefit of UDP over TCP in this case is the application to meters where latency and jitter are the primary concerns.

- Unreliable – When a message is sent, it cannot be known if it will reach its destination; it could get lost along the way. There is no concept of acknowledgment, retransmission, or timeout.
- Not ordered – If two messages are sent to the same recipient, the order in which they arrive cannot be predicted.
- Lightweight – There is no ordering of messages, no tracking connections, etc. It is a small transport layer designed on top of IP.
- Datagrams – Packets are sent individually and are checked for integrity only if they arrive. Packets have definite boundaries which are honored upon receipt, meaning a read operation at the receiver socket will yield an entire message as it was originally sent.
- No congestion control – UDP itself does not avoid congestion, unless congestion control measures are implemented at the application level.

Confirmation messages can also be implemented at the application level.

The more isolated the network, the more reliable UDP will be. On a totally isolated network, statistics show high reliability. Reliability drops when the network is shared with bandwidth consuming clients.

CHAPTER 3: OPEN SOUND CONTROL

Open Sound Control (OSC) is an open, transport-independent, message-based protocol developed for communication among computers, sound synthesizers, and other multimedia devices.

OSC is often used as an alternative to the 1983 MIDI standard, where higher resolution and a richer musical parameter space is desired. OSC messages are commonly transported over Ethernet protocols. It gives users more flexibility and facility in addressing and interacting with remote machines.

OSC features an open-ended symbolic naming scheme, high resolution argument data, high resolution time tags, and more. It can pack messages into bundles whose effects must occur simultaneously.

OSC MESSAGES

An OSC message consists of an address, a Type Tag string, Arguments, and optional tags. The address uses a hierarchical name space, like an URL. The Type tags are one-letter strings that tell the server how to interpret the argument data (values) that follow a Type tag.

Addressing

The addressing structure within OSC can be fairly complex. In the most common use, the address is a string-based part that can be hierarchically organized with different layers; for example, the input 1 level in CueStation will translate into “/Input/1/Level.” The address always starts with “/” and most software proposes a default addressing pattern. Please refer to the OSC documentation for more variations.

Type Tags

The Type Tag comes after the address and defines what the value of the argument translates into.

- Int32: the basic 32-bit value with no decimal, is represented by “i.”
- Float32: a 32-bit floating point number. is represented by the “f” tag.

- OSC-String: a sequence of non-null ASCII characters, or text message, is represented by “s.”
- Boolean values: binary states can be expressed with Boolean type tags: “T” means True, “F” means False. They are not followed by an argument.
- Int64: also called “double,” is a 64-bit value, allowing a larger number in a single argument. It is represented by the “h” tag.

CHAPTER 4: THE OSC SERVER IN CUESTATION

STRUCTURE

The server “dmixerd” listens for incoming OSC commands for both TCP connections or UDP packets on port 18033.

If a TCP connection is received, the expected protocol is the standard one for OSC-over-TCP: a 4-byte big-endian length field, followed by an OSC packet of that length, and repeat as necessary.

By default, the CueStation Server replies to a UDP packet on the port that sent the command or request. The port can be specified using a command.

MESSAGE METHODS

The Meyer OSC suite currently consists of the following methods:

Method	Description
/go	Equivalent to pressing the “go” button in the Transport window.
/stop	Equivalent to pressing a “stop” button in the Transport window.
/moveby	Move the “on deck cue” in the cue list up or down by N steps.
/recall	Recall a cue or subcue.
/update	Update a cue or subcue.
/set	Set one or more control points to one or more values.
/setwf	Similar to “/set,” but allows wait and fade times to be specified also.
/setblock	Similar to “/set,” but sets a block of control points instead of a list.
/setblockwf	Similar to “/setblock,” but allows wait and fade times to be specified also.
/inc	Increase one or more control points by the specified delta.
/incwf	Similar to “/inc,” but allows wait and fade times to be specified also.
/incblock	Similar to “/inc,” but increases a block of control points instead of a list.
/incblockwf	Similar to “/incblock,” but allows wait and fade times to be specified also.
/dec	Decrease one or more control points by the specified delta.

Method	Description
/decwf	Similar to “/dec,” but allows wait and fade times to be specified also.
/decblock	Similar to “/dec,” but decreases a block of control points instead of a list.
/decblockwf	Similar to “/decblock,” but allows wait and fade times to be specified also.
/step	Set control point(s) to the next in a sequence of supplied values.
/stepwf	Similar to “/step,” but allows wait and fade times to be specified also.
/stepblock	Similar to “/step,” but decreases a block of control points instead of a list.
/stepblockwf	Similar to “/stepblock,” but allows wait and fade times to be specified also.
/get	Retrieve the current values of one or more control points.
/getblock	Similar to “/get,” but with addresses specified as a block instead of a list.
/ping	Causes a “/pong” packet with the same data to be sent back to the client.
/heypython	Send a string to a running Python script.
/runpython	Launch a Python script in a new process.
/projectping	Causes a “/projectpong” packet with the same data to be sent back to the client.
/setidletimeout	Specifies the number of milliseconds of 'radio silence' before dmixerd gives up on a UDP client.
/setreplyport	Specifies an explicit UDP port number to send a client's UDP reply packets to.
/setstatusupdates	Enable or disable the reception of regular status-updates to this client.
/subscribe	Subscribe to a set of control point addresses (for “live updates”).
/subscribeblock	Similar to “/subscribe,” but with addresses specified as a block.
/subscribeevent	Subscribe to a particular pattern of event strings.
/unsubscribeevent	Unsubscribe from a particular pattern of event strings.
/unsubscribeallevents	Unsubscribe from all patterns of event strings.
/trigger	Fires any triggers matching the included event-name.
/unsubscribe	Unsubscribe from a set of control point addresses.
/unsubscribeblock	Similar to “/unsubscribe,” but with addresses specified as a block.
/unsubscribeall	Cancel all current subscriptions for this client.
/log	Sends a text message to the system log.
/print	Retrieve information about a specified set of project database nodes.

Method	Description
/watch	Start watching a specified set of project database nodes.
/unwatch	Stop watching a specified set of project database nodes.
/unwatchall	Cancel all watches for this client.
/docommand	Execute the contained subcue-style OSC command.
/enablemodule	Enable the specified D-Mitri module(s).
/disablemodule	Disable the specified D-Mitri module(s).
/togglemodule	Toggle the enabled/disabled state of the specified D-Mitri module(s).
/setaliases	Updates the alias table in the active mixer configuration
/wait	Wait a specified number of milliseconds before executing any further OSC commands in this command stream.
/cancel	Cancel any waits-in-progress (and any queued-up deferred commands that follow them).
/cancelall	Same as /cancel, except cancels events for all clients, not just the current one.

OSC Message Packet Methods (Detail)

Below are descriptions of the method commands that clients can send to mixerd in their OSC packets.

/go

Method	Arguments	Description
/go	“” “,i”	Equivalent to pressing the “go” button in the Transport window.

This command takes zero or one arguments. It causes the current cue-on-deck in the current Cue List to be recalled, and the cue-on-deck pointer to be advanced.

If an integer argument is specified, it is taken to be the Cue List Player index (0-127). If it isn't specified, player #0 is assumed (for backwards compatibility)

/stop

Method	Arguments	Description
/stop	“” “,i”	Equivalent to pressing a “stop” button in the Transport window.

This command takes zero or one arguments. If zero arguments are provided, /stop causes all current automation activity (including cue list autofollows, SpaceMap trajectory playback, Wild Tracks audio playback, fader fades, etc) to be immediately stopped.

If an integer argument is specified, it is taken to be a Cue List Player index (0-126), and only that Cue List Player will be stopped. If the index is 127, this is interpreted as a special case meaning “all Cue List Players in the current config.”

/moveby

Method	Arguments	Description
/moveby	“,i” “,ii”	Move the “on deck cue” in the cue list up or down by N steps.

This command takes a single integer argument that indicates the number of entries that the “on deck cue pointer” should move forward in the current cue list.

For example, +1 is equivalent to pressing the “next” button in the Transport window once, and -1 is equivalent to pressing the “previous” button in the Transport window once. If you wish to move immediately to the top or bottom of the cue list, you can do so by passing a very large positive or negative value as the argument here. (this works because the cue-on-deck pointer's position will be constrained to keep it from moving “off the ends” of the cue list). If a second integer argument is specified, it is taken to be the Cue List Player index (0-126).

If it isn't specified, player #0 is assumed (for backwards compatibility). If 127 is specified, all cue list players will be affected.

/recall

Method	Arguments	Description
/recall	“,ii” “,iii”	Recall a cue or subcue.

This command takes two or three integer arguments. The first argument should either be zero (to indicate a cue recall) or one (to indicate a subcue recall). The second argument is the ID of the cue (or subcue) to be recalled.

If a third argument is specified, it will be parsed as the index of the Cue List Player that should do the cue/subcue recall. If the third argument is not specified, Cue List Player #0 (aka the first one) will be used by default.

/update

Method	Arguments	Description
/update	“,ii”	Update a cue or subcue.

This command takes two integer arguments. The first argument should either be zero (to indicate a cue update) or one (to indicate a subcue update). The second argument is the ID of the cue (or subcue) to be updated.

/set

Method	Arguments	Description
/set	“,s*” “,A**”	Set one or more control points to one or more values.

This command is quite flexible in that its arguments can be sent in several forms. In any form, however, the arguments indicate a set of one or more ControlPointAddresses, followed by a list of one or more ControlPointValues. The specified ControlPointValues will be assigned to the specified ControlPointAddresses, in the order they are listed.

The easy way to specify the set of ControlPointAddresses is as a string. In this case, the string takes uses the exact same human-readable format as is seen in the Subcue Library window. Here are a few example address strings that you could use:

```
“Input 1 Mute”
“Bus 1,4,9 Invert”
“Output 1-8 Level”
“Bus 1-3 Output 5,7,15-20 Level”
```

You can even specify several sets in a single string if you prefer, by separating the sets with semicolons, like this:

```
“Input 1-4 Mute; Output 1-8 Level; System Level”
```

A less user-friendly but more server-CPU-efficient way to specify the set of ControlPointAddresses is as a set of one or more “address arguments.” An address argument represents a ControlPointAddress using a custom data type ('A'). The data of this type is always 16 bytes long, and always consists of 8 2-byte big-endian integers. Negative numbers have special meanings as tokens, as documented in cp_indices.py. For example, -32758 is the token for “Input,” and -32757 is the token for “Output,” and so on. Therefore, if you wanted to express “Input 1 Mute” as an Address, the array of int16s would look like this:

```
[-32758, -32741, 0, 0, 0, 0, 0, 0] // i.e. (ciInput, ciMute, 0, ...)
```

NOTE: “0” indicates input 1 (indices are always numbered starting from zero in this format) and that the numeric indices should always appear after the negative token values.

As another example, here is “Bus 5 Output 7 Level” expressed as an Address:

```
[-32754, -32757, -32744, 4, 6, 0, 0, 0] // i.e. (ciBus, ciOutput, ciLevel, 4, 6, ...)
```

You can specify as many Addresses as you like in this command; each Address specifies one ControlPointAddress to set to the values that are specified next.

Next, you should specify one or more value arguments. These arguments may be any of the following OpenSoundControl argument types:

Argument	Description
i	32-bit integer.
f	32-bit floating point.
s	OSC string.
F	Boolean false.
T	Boolean true.
h	64-bit integer.
P	Point (a non-standard Meyer type which consists of 2 32-bit big-endian floats).
A	ControlPointAddress The 16-byte non-standard Meyer type described above.
M	Message (a non-standard Meyer type which has the same format as an OSC-blob but indicates that the data is a flattened MUSCLE Message argument).

Argument	Description
C	Config (a non-standard Meyer type which has the same format as ‘M’, but indicates that the data represents an Meyer System Config argument) The number of value arguments included in your /set command should be less than or equal to the number of addresses specified in the address arguments — if there are more values than addresses, the surplus values will be ignored.

If the number of ControlPointValues specified is less than the number of ControlPointAddresses specified, the last ControlPointValue in the values list will be re-used for the remaining ControlPointAddresses. This behaviour is useful when you wish to set a large number of control points to the same value.

For example, to clear an entire 64x64 Matrix, you might send this packet:

```
"/set"                    // method name
",sf" // argument types string
(string arg) "Bus 1-64 Output 1-64 Level"// control point addresses string
(float arg) -90.0f        // a single value for all (-90.0 == -inf)
```

/setblock

Method	Arguments	Description
/setblock	",s*" ",A**"	similar to "/set," but sets a block of control points instead of a list.

When used with a string address, this command behaves identically to "/set." When used with "Address-addressing," however, the Addresses are interpreted differently. Each pair of specified Addresses is presumed to indicate a "block" of ControlPointAddresses. The server will iterate through all addresses in the block in order. So, for example, if you specified two "Address" arguments, say (Input 1 Level) and (Input 8 Level), they would be interpreted as indicating inputs levels 1 through 8 inclusive.

```
[-32758, -32744, 0, 0, 0, 0, 0, 0]  (i.e. first address = ciInput, ciLevel, 0)
[-32758, -32744, 7, 0, 0, 0, 0, 0]  (i.e. last address  = ciInput, ciLevel, 7)
```

A block can be multi-dimensional. For example, if you wanted to specify Outputs 1-8 of Buses 2-4 in the matrix, you could add these two Addresses:

```
[-32754, -32757, -32744, 0, 0, 0, 0, 0]
```

(i.e. first address = ciBus, ciOutput, ciLevel, 1, 0)

```
[-32758, -32744, 7, 0, 0, 0, 0, 0]
```

(i.e. last address = ciBus, ciOutput, ciLevel, 3, 7)

You can add more than one pair of Addresses, if you wish to specify more than one block of addresses. For example, if you wanted to specify Input Mutes 3-8 _and_ Input Levels 10-20, you could add these:

```
[-32758, -32741, 2,  0, 0, 0, 0, 0]  (i.e. first address = ciInput, ciMute, 3)
[-32758, -32741, 7,  0, 0, 0, 0, 0]  (i.e. last address  = ciInput, ciMute, 8)
[-32758, -32744, 9,  0, 0, 0, 0, 0]  (i.e. first address = ciInput, ciLevel, 10)
[-32758, -32744, 19, 0, 0, 0, 0, 0]  (i.e. last address  = ciInput, ciLevel, 20)
```

If there are an odd number of Addresses, the last Address is interpreted as a single address.

After the Addresses, the control point value field(s) should then be specified as described in "/set" on page 16.

/setwf

Method	Arguments	Description
/setwf	“,s*ff*” “,A*ff*”	set one or more control points to one or more values with wait and fade times

This command is similar to /set, except that after the address argument(s) (but before the value argument(s)) you must specify two floating point arguments: a wait time and then a fade time. Both arguments should be expressed in seconds, and should range between 0.0 and 1000.0.

NOTE: The D-Mitri firmware only supports doing waits and fades on certain control points (e.g. Matrix Levels). If you specify control points for which wait and fade times are not supported, the wait and fade times will be ignored and those control points will be set to the specified value immediately.

Due to the more complex semantics of this command, feedback from the server is not suppressed. So for example if you send a /setwf command to set Input 1 Level to unity, and you are subscribed to Input 1 Level, you will soon receive a /got OSC message indicating that Input 1 Level has been set to unity.

For example, to wait 3 seconds and then fade up the first row of a 64x64 Matrix over the next 5 seconds, you might send this packet:

```
"/setwf"                // method name
",sfff"                 // argument types string
(string arg) "Bus 1 Output 1-64 Level"// control point addresses string
(float arg) 3.0f        // wait time (3 seconds)
(float arg) 5.0f        // fade time (5 seconds)
(float arg) 0.0f        // a single value for all (0.0 == unity)
```

/setblockwf

Method	Arguments	Description
/setblockwf	“,sff*” “,A*ff*”	similar to “/setblock,” but sets a block of control points instead of a list

This command is similar to /setblock, except that after the address argument(s) (but before the value argument(s)) you must specify two floating point arguments: a wait time and then a fade time. Both arguments should be expressed in seconds, and should range between 0.0 and 1000.0.

NOTE: The D-Mitri firmware only supports doing waits and fades on certain control points (e.g. Matrix Levels). If you specify control points for which wait and fade times are not supported, the wait and fade times will be ignored and those control points will be set to the specified value immediately.

Due to the more complex semantics of this command, feedback from the server is not suppressed. So for example if you send a /setblockwf command to set Input 1 Level to unity, and you are subscribed to Input 1 Level, you will soon receive a /got OSC message indicating that Input 1 Level has been set to unity.

For example, to wait 3 seconds and then fade up the first three rows of a 64x64 Matrix over the next 5 seconds, you might send this packet:

```
"/setblockwf"            // method name
",AAfff"                 // argument types string
[ciBus, ciOutput, ciLevel, 0, 0,  0, 0, 0]// block indicating start-corner of block
[ciBus, ciOutput, ciLevel, 2, 63, 0, 0, 0]// block indicating end-corner of block
(float arg) 3.0f         // wait time (3 seconds)
(float arg) 5.0f         // fade time (5 seconds)
(float arg) 0.0f         // a single value for all (0.0 == unity)
```

/get

Method	Arguments	Description
/get	“,s” “,ss” “,A*” “,A*s”	retrieve the current values of one or more control points

This command lets you query the server for the current state of one or more control points. The control point addresses should be specified either via a human-readable string or via a series of one or more Addresses, exactly as described in the “/set” documentation (see “/set” on page 16).

Values are not specified in this command, of course, since you are requesting values, not sending them.

You may, however, optionally specify a string argument after the address argument(s). This string may be any string you like. It will be used as a tag for your request, and will be sent back to you in the reply packet(s). This is useful to help determine which reply packets are associated with which requests.

If you do not include a tag string, a unique tag string will be chosen for you by the server.

The server will send back the requested data as a series of one or more “/got” OSC packets. If the “/get” command was received on from a TCP stream, the “/got” packets will be sent back to that TCP stream; if the “/get” command was received in the form of UDP packets, the “/got” packets will be sent back to the IP address and port that the “/get” packet was sent from.

For info on the “/got” packet's format, see “/got” on page 36.

/getblock

Method	Arguments	Description
/getblock	“,s” “,ss” “,A*” “,A*s”	similar to “/get,” but with addresses specified as a block instead of a list.

This command is the same as /get, except that any included Addresses will be interpreted in pairs instead of as individual addresses. For information on how this is done, see “/setblock” on page 19.

/ping

Method	Arguments	Description
/ping	Any	causes a “/pong” packet with the same data to be sent back to the client.

This command's packet will be immediately sent back to the client, with the only difference being that the method name string will be changed from “/ping” to “/pong.”

/projectping

Method	Arguments	Description
/projectping	Any	causes a “/projectpong” packet with the same data to be sent back to the client.

This command is similar to /ping, except that message is forwarded to the local Project Database daemon (dcued) and back before the corresponding /projectpong message is sent back to the OSC client. This is useful in some cases, such as when the client needs to know when a preceeding /watch or /print command has been fully processed.

/heypython

Method	Arguments	Description
/heypython	“,is”	Send a command string to a currently running Python script.

This command lets you specify a string that will be sent to a currently executing Python script (as seen in CueStation's Script Execution window). Assuming that the Python script is based on the BasicClient.py base class, the string will be passed to the UserCommandReceived() callback method. The default implementation of that method executes the string as Python source code; customized scripts are free to override it to handle the string differently, of course.

The first argument is the index of the script execution slot that the command should be sent to. This integer should be a value between 0 (the first execution slot, at the top of the Script Execution window) and 31 (the last execution slot).

There are also Some “special” values that you can pass for the index argument:

Value	Description
-1	Send to all Python scripts (whether shown in the Script Execution window or not).
-2	Send to all “foreground” Python scripts (i.e. all scripts shown in the Script Execution window).
-3	Send to all “background” Python scripts (i.e. all scripts not shown in the Script Execution window).

The string argument may be any string you like, although it does need to be short enough to fit into an OSC packet, of course. This string is the text that will be passed to the targeted Python process(es). This string may optionally contain one or more special directives of the form #include “some_support_file.py”

These tokens will be expanded into a copy of the contents of the specified file, if that file can be found in the Support Files window or Python path.

/runpython

Method	Arguments	Description
/runpython	“,is” “,iss”	Launch a Python script in a new process.

This command lets you launch a Python script in a new process.

The first argument is the index of the script-execution-slot that the new process should run in. This value can range between 0 (the first execution slot, at the top of the Script Execution window) and 31 (the last execution slot, at the bottom of the window). Or you can pass in -1 as this argument if you want the script to run in the background.

The second argument is a string containing some Python source code to execute. This string may optionally contain one or more special directives of the form #include “some_support_-file.py”

These tokens will be expanded into a copy of the contents of the specified file, if that file can be found in the Support Files window or Python path.

The third argument, if specified, is a string containing command line arguments to pass to the Python script at launch.

/subscribe

Method	Arguments	Description
/subscribe	“,s” “,A*”	Subscribe to a set of control point addresses.

This command has syntax similar to /get, except that the specified control point addresses are remembered by the server, which will continue to send “/got” updates to your client whenever any of the addresses change. This allowed your client to keep track of the current state of the specified addresses at all times, without having to constantly poll their state via /get packets.

Unlike “/get,” no request-tag can be specified in a /subscribe command. The /gotpackets returned by subscriptions will always have an empty string (“”) as their tag value.

NOTE: Subscriptions are handled per control point address, so it is possible to “build up your subscription set” via multiple /subscribe commands, and likewise you can unsubscribe from any arbitrary sub-portion of your subscription set at any time.

Trying to subscribe again to control points that you are already subscribed to will cause the server to immediately re-send you those control points' current values, but otherwise will have no effect (i.e. a client can't have two simultaneous subscriptions to the same control point).

For TCP clients, the subscriptions will remain active until they are countermanded by an /unsubscribe, /unsubscribeblock or /unsubscribeall command, or until the client's TCP connection is broken.

For UDP clients, the subscriptions will remain active until they are countermanded by an /unsubscribe, /unsubscribeblock or /unsubscribeall command, or until at least 30 seconds have passed without the server receiving any UDP packets from your client.

For this reason, it is important to be sure that your UDP client sends an OSC packet to the server at least once every 30 seconds, to avoid losing your subscriptions. An empty “/ping” packet will be sufficient.

/subscribeblock

Method	Arguments	Description
/subscribeblock	“,A*”	Similar to “/subscribe,” but with addresses specified as a block

/subscribeblock works the same as “/subscribe,” except that any Address pairs will be interpreted as blocks of addresses, rather than as separate single addresses. See “/setblock” on page 19 for syntax details.

/subscribeevent

Method	Arguments	Description
/subscribeevent	“,s”	Subscribe to a particular pattern of event strings

This command tells the server that this client wants to be notified (via the “/event” OSC message) about events whose event strings match the specified pattern. The argument may include wildcards (e.g. “*” would match all event strings).

/trigger

Method	Arguments	Description
/trigger	“,s”	Sends the named event(s) to the system to trigger the matching triggers.

The matching triggers can be set up via the “External Control / Setup Triggers” external.

/unsubscribe

Method	Arguments	Description
/unsubscribe	“,s” “,A*”	Unsubscribe from a set of control point addresses.

This command has the same syntax as “/subscribe,” but the opposite effect. Any control point addresses specified by this command that are currently subscribed to by this client will have their subscription-records removed from the server, so that in the future when those control point addresses change, your client will no longer be notified. For control points that your client is not currently subscribed to, this command will have no effect.

/unsubscribeblock

Method	Arguments	Description
/unsubscribe-block	“,A*”	Similar to “/unsubscribe,” but with addresses specified as a block.

/subscribeblock works the same as “/unsubscribe,” except that any Address pairs will be interpreted as blocks of addresses, rather than as separate single addresses. See “/setblock” on page 19 for syntax details.

/unsubscribeevent

Method	Arguments	Description
/unsub-scribeevent	“,s”	Unsubscribe from a particular pattern of event strings.

This command tells the server that this client no longer wants to be notified about events whose event strings match the specified pattern. The argument should be a string that was previously specified as part of a /subscribeevent command.

/unsubscribeall

Method	Arguments	Description
/unsubscribeall	None	Cancel all current subscriptions for this client.

This command takes no arguments. It causes any and all current subscriptions for this client to be canceled. This command is useful for returning your client the default “completely-unsubscribed” state without having to specify all the control points your client is currently subscribed to.

It's recommended that UDP clients always send this packet on startup, just so that they can be guaranteed a known state on the server (otherwise the server might still have subscriptions in your client's name, from a previous session).

/unsubscribeallevents

Method	Arguments	Description
/unsubscribeall	None	Unsubscribe from all patterns of event strings

This command takes no arguments. It causes any an all current event subscriptions for this client to be canceled.

/log

Method	Arguments	Description
/log	“,s” “,is”	Sends the specified log message to the system log.

This command sends the log message you specify to the Meyer log (as shown in the CueStation Log window). The first argument is an integer indicating the “severity level” of the message, and it should be one of the following values:

Value	Description
1	Critical error: indicates a critical system failure.
2	Error: indicates an error condition.
3	Warning: indicates a warning.
4	Info: A normal informational message.
5	Debug: Contains debugging information only.
6	Trace: Used to trace program execution.

If you skip the first argument (i.e. include only a string argument) then the message will default to Level 4 (Info). The second argument is the string that you want to have added to the log.

/watch

Method	Arguments	Description
/watch	“,s” “,sT”	Start watching a specified set of project database nodes.

This command tells the server that you would like to be notified about the specified set of Project Database nodes. For example, a /watch with argument “log/*” would result in the client receiving /watched messages describing all of the entries currently in the log, and being notified whenever the contents of the log changes.

If a T (true) argument is specified as the second argument, the watch will be a “quiet watch” in that the client will receive updates about future changes to the subscribed database nodes, but it will not get the initial set of /watched messages describing the current contents of the log.

/unwatch

Method	Arguments	Description
/unwatch	“,s”	Stop watching a specified set of project database nodes.

This command cancels a previously set watch for this client. The argument is a node path string that was previously sent in a /watch messages.

/unwatchall

Method	Arguments	Description
/unwatchall	“,”	Cancel all watches for this client.

This command cancels all previously set watches for this client.

/print

Method	Arguments	Description
/print	“,s” “,si”	Retrieve information about a specified set of project database nodes.

This command is similar to /watch, except that it is a one-shot query rather than a continuing watch. Information about any project database nodes that match the specified node path (and optionally, descendants of those nodes in the database hierarchy) will be sent back to the client in the form of /watched replies.

For example, a /print of “default/subcues” would result in the client receiving a listing of all subcues currently in the project.

By default, all descendents of matching nodes are sent as well (this is different from the /watch command, which only sends the matching nodes and never their descendants). If you want to limit that behavior, you can do so by specifying a second argument that is an int32 representing the maximum depth underneath the matching nodes that should be reported. For example, specifying a second argument of zero will result in no descendants being returned (i.e. the /watch behavior). Specifying 1 will result in matching nodes and their direct children being returned; 2 will include their grandchildren also, and so on.

/docommand

Method	Arguments	Description
/docommand	“,s*”	Execute the supplied subcue-style OSC command.

In CueStation 5, the commands inside a Command subcue represented internally as OSC messages. The precise structure of the OSC data varies from one command type to the next, but in all cases it can be viewed in the text pane at the bottom of the Command Subcue Editor area in CueStation 5.

The “/docommand” command causes the CueStation system to execute such a command. For example, in CueStation 5, the OSC form of a “Page Groups / Change Page” command might look like this:

```
Method = [/dcasld/changepage]
Argtypes = [,iiiii]
Arg #0: Int32 = 0 (0x0)
Arg #1: Int32 = 1 (0x1)
Arg #2: Int32 = 2 (0x2)
Arg #3: Int32 = 4 (0x4)
Arg #4: Int32 = 0 (0x0)
```

If you wanted to dynamically create and execute such a command from your OSC client (instead of creating the subcue in advance and then sending a /recall OSC command to recall it), you could send this OSC command:

```
“/docommand”
“,siiiii”
“/dcasld/changepage”
0
1
2
4
0
```

The advantage to doing it this way is that you can compose the command's arguments “on the fly” inside your OSC client, rather than having to set them up in advance inside a subcue.

The Argtypes string we send has an “s” prepended to it, since we need to send the embedded OSC command's own command-string as a string argument also.

/enablemodule, /disablemodule, /togglemodule

Method	Arguments	Description
/enablemodule /disablemodule /togglemodule	“,s*”	Enable, Disable, or Toggle a specific module.

These commands allow you to tell CueStation that a specified module (or modules, using wildcarding) should be enabled/disabled/toggled. The enabled/disabled state of a module shows up in the System Status window; all modules are enabled by default. Disabling a module causes the module to stop routing audio, and the system will reassign that module's functions to another module if possible.

/setaliases

Method	Arguments	Description
/setaliases	“,iiss”	Update Mixer Configuration aliases.

This command allows you to update the Aliases table in the currently active Mixer Configuration. This can be useful if you want to adjust IP addresses, or if you want to use the aliases as “global variables” to adjust the behavior of various text commands, etc.

The first argument is an integer. Set it to one if you want to delete all aliases currently in the Aliases table before proceeding. Set it to zero otherwise.

The second argument is an action code. Legal values are:

Value	Description
0	Set alias: sets the value(s) of an alias to the specified values, creating the alias first if necessary.
1	Delete aliases: delete any aliases whose names match the specified name pattern
2	Add alias values: will add the specified values to existing aliases whose names match (namepattern), if they aren't already present. An alias will be created first, if no matching aliases exist.
3	Remove alias values: remove the matching values from existing aliases whose names match (namepattern), if they exist.

The third argument is a name (as would appear in the “Name” colum in the Mixer configuration window's Alias table).

The fourth argument is a value string (as would appear in the “Expand To” column of that same table). This value is ignored if the action code is set to 1 (aka “delete aliases”), but it must still be present in the command.

REPLY METHODS

The list of OSC reply methods that the server may send back to the client are as follows:

Reply	Description
/event	Contains the name of an subscribed event that has occurred.
/got	Contains control point value data previously requested by a “/get” or “/subscribe” packet.
/pong	Response to a /ping command
/projectpong	Response to a /projectping command
/watched	Contains database node info previously requested by a /watch or /print command.

OSC Reply Packet Methods (Detail)

OSC reply packets are sent by mixerd back to clients.

/event

Packet	Arguments	Description
/event	“,s”	Contains the name of a subscribed event that has occurred.

The /event command should not be sent by clients -- rather, it is sent by the server back to the clients, if an event has occurred that the client had previously subscribed to (via a “subscribeevent” command). This command will have a single argument, which is the name of the event that occurred.

/got

Packet	Arguments	Description
/got	",s((A*)*)"	Contains control point value data previously requested by a "/get" or "/subscribe" packet.

The /got packet should not be sent by clients -- rather, it is sent by the server back to the clients in response to a "/get" or "/subscribe" packet. The /got packet contains as its first argument the tag string that was passed in to the "/get" command. If no tag string was passed in to "/get," the server will generate a unique string for the /got packets. If the "/got" packet is being sent in response to a "/subscribe" packet, the response tag string will always be empty (i.e. "").

After the tag string, the remainder of the packet will consist of one or more (Address, value) pairs. The Address format and value formats are the same as the ones described in the /set command documentation. Each pair indicates that the current value for (that address) is (the provided value).

A single /get command may result in more than one /got reply, if there is too much data to fit into a single packet. You can use the tag string to determine whether or not two different /got packets correspond to the same "/get" command.

If you need to know whether or not you have received all the /got packets corresponding to your request yet, or whether more are coming, you can find out by sending a second "/get" command immediately after your first one. The second "/get" command can be trivial; just make sure it has a different tag-string from the first one. Once you receive a packet with the second command's tag string, you can assume that the first request has been fully handled (requests are handled in FIFO order).

NOTE: You cannot use the "/ping" command for this purpose, since "/ping" short-circuits the data path that /get uses and so you would likely receive the /pong reply before you received all of your /got replies)

/pong

Packet	Arguments	Description
/pong	Any	Response to the /ping command.

Any arguments will match those of the corresponding /ping).

When the server receives a /ping command, it will send back a /pong reply that is otherwise byte-for-byte identical to the /ping packet. This command is useful for network testing and as a keep-alive for UDP subscriptions (for more information, see “/subscribe” on page 26).

/projectpong

Packet	Arguments	Description
/projectpong	“,”	Response to the /projectping command.

This packet is the reply counterpart to a previously sent /projectping command.

/watched

Packet	Arguments	Description
/watched	“,s”	A reply to a previously sent /watch or /print command.

This packet will contain a string argument of the following format:

```
“path: ObjectType: attr1=[val1] attr2=[val2] [...]”
```

where path is a project database node path, such as log/I0 or default/subcues/5 or default/cues/62 or etc. ObjectType will be the name of the C++ class that is used to implement the data item in the server. Currently, these names include:

- QNetAccessPolicyData
- QNetAccessPolicyHeader
- QNetChannelSet
- QNetChatEntry
- QNetCue
- QNetCueEntry
- QNetCueList
- QNetDriveInfo
- QNetKeyMapping
- QNetLogEntry
- QNetParameterSet
- QNetSpaceLink
- QNetSpaceMap
- QNetSpaceNode
- QNetSpacePic
- QNetStub
- QNetSubcueEntry

- QNetSubcueHeader
- QNetSubcueTypeData
- QNetSubcueTypeHeader
- QNetSupportFileHeader
- QNetTrajectory
- QNetTrajectoryNode
- QNetTriset

The actual names of attr1, attr2, att3, etc will be things like “name,” “comment,” “when,” etc, and will vary by type.

If the string contains only the path: part and nothing else, that means that the /watched message is a notification that the object at that path has been removed from the database.

EXAMPLE OSC PACKETS

Below are hex dumps showing the contents of various UDP packets that represent valid Meyer/OSC commands and replies. These same byte sequences are valid for sending over TCP to the Meyer system, but when using TCP you must preface them with a 4-byte, big-endian byte-count field, so that the TCP OSC parse can know where the packet ends and the next packet begins.

Packet	Hex
/go	2f 67 6f 00 2c 00 00 00
/stop	2f 73 74 6f 70 00 00 00 2c 00 00 00
/moveby 1	2f 6d 6f 76 65 62 79 00 2c 69 00 00 00 00 00 01
/moveby -1	2f 6d 6f 76 65 62 79 00 2c 69 00 00 ff ff ff ff
/recall cue 5	2f 72 65 63 61 6c 6c 00 2c 69 69 00 00 00 00 00 00 00 00 05
/recall subcue 58	2f 72 65 63 61 6c 6c 00 2c 69 69 00 00 00 00 01 00 00 00 3a
/set input 1 mute = true	2f 73 65 74 00 00 00 00 2c 73 54 00 69 6e 70 75 74 20 31 20 6d 75 74 65 00 00 00 00
/set output 5 level = 5.5	2f 73 65 74 00 00 00 00 2c 73 66 00 6f 75 74 70 75 74 20 35 20 6c 65 76 65 6c 00 00 40 b0 00 00
/get input 1-8 level	2f 67 65 74 00 00 00 00 2c 73 00 00 69 6e 70 75 74 20 31 2d 38 20 6c 65 76 65 6c 20 00 00 00 00
/ping HelloSailor	2f 70 69 6e 67 00 00 00 2c 73 00 00 48 65 6c 6c 6f 53 61 69 6c 6f 72 00
/python 1 “print Hello”	2f 70 79 74 68 6f 6e 00 2c 69 73 00 00 00 00 00 00 70 72 69 6e 74 20 22 48 65 6c 6c 6f 22 00 00 00
/subscribe input 1-8 level	2f 73 75 62 73 63 72 69 62 65 00 00 2c 73 00 00 69 6e 70 75 74 20 31 2d 38 20 6c 65 76 65 6c 20 00 00 00 00
/unsubscribe input 3,5,7 mute	2f 75 6e 73 75 62 73 63 72 69 62 65 00 00 00 00 2c 73 00 00 69 6e 70 75 74 20 33 2c 35 2c 37 20 6d 75 74 65 20 00 00 00
/unsubscribeall	2f 75 6e 73 75 62 73 63 72 69 62 65 61 6c 6c 00 2c 00 00 00

SENDING OSC TO MODULES WITH UNKNOWN IP

It is possible for a client to send OpenSoundControl UDP packets to a D-Mitri system even if the client does not know the IP address of any of the modules in the D-Mitri system. To do this, the client just needs to know the D-Mitri system's system-name string. The client can then use a hashing algorithm equivalent to the one shown in the Python code below to compute the multicast IP address that corresponds to the system-name string, and send UDP packets to that IP address. The senior dmixerd peer will be listening to that multicast group and will receive and process the UDP packets.

```
# Given a D-Mitri system name as an argument, this script
# prints out the IPv6 multicast address that the senior
# dmixerd peer will listen on for incoming OpenSoundControl
# and ASCII text commands over UDP.

import sys
import socket

systemName = sys.argv[1]
print "For D-Mitri system name:   ", systemName

hash = 631741  # arbitrary magic starting number
for c in systemName:
   hash = (hash*33)+ord(c)
hash = hash % 18446744073709551616 # keep hash bounded to 64-bits

s = "ff12:0000:0001:0000"
for i in range(0, 16):
   if (i%4) == 0:
      s = s + ":"
   s = s + "%x"%((hash>>(60-(i*4)))&0x0F)
print "IPv6 multicast address is: ", socket.inet_ntop(socket.AF_INET6,
socket.inet_pton(socket.AF_INET6, s))
```

Meyer Sound Laboratories Inc.
2832 San Pablo Ave.
Berkeley, CA 94702

www.meyersound.com
+1 510 486.1166

CueStation OSC User Guide, PN 05.176.130.01 A