osc4py3

User & Developer Documentation

User Documentation

Introduction

What is osc4py3 ?

It is a fresh implementation of the Open Sound Control (OSC)[1] protocol for Python3. It tries to have a "pythonic" programming interface at top layers, and to support different scheduling scheme in lower layers to have efficient processing.

What Python ?

Target Python3.2 or greater, see other OSC implementations for older Python versions (OSC.py, simpleosc, TxOSC).

Why a new implementation ?

It was initially developed for use in Blender Game Engine as a background system communicating via OSC in separate threads, spending less possible time in main game engine thread : sending OSC is just building messages/bundles objects and identifying targets, receiving OSC is just calling identified functions with OSC messages. All other operations, encoding and decoding, writing and reading, monitoring for communication channels availability, pattern matching messages addresses, etc, can be realized in other threads. On a multi-core CPU with multi-threading, these optimizations allow the blender C code to run in parallel with communication operations, we win milliseconds which are welcome in our context of human real-time experiments with sound or images.

What are the differences with other implementations ?

The whole package is really bigger and the underlying system is complex to understand. But, two base modules, `oscbuildparse` and `oscmethod` (see the modules documentation), are autonomous and can be used as is to provide : encoding and decoding of OSC messages and bundles, pattern matching and functions calling.

As we start from scratch, we use latest informations about OSC with support for advanced data types (adding new types is easy), and plan support for OSC extensions like message pattern compression.

Is is complicated to use ?

No. See the "Simple use" chapter below. It may be even easier to use than other Python OSC implementations.

Don't mistake, the underlying processing is complex, but the high level API hide it.

Quick OSC

See http://opensoundcontrol.org/ for complete OSC documentation.

1 See http://opensoundcontrol.org/ web site for complete informations.

OSC is a simple way to do remote procedure calls between applications, trigged on string pattern matching, transported by any way inside binary packets. It mainly defines packets encoding format to transmit data and pattern matching to select functions to call.

Messages

In OSC messages are basic structures with a string identifying an address pattern, a string describing associated data, and data. Address pattern are matched using regular expressions.

Simple use

You can take a look at the `rundemo.py` script in `demos/` directory. It is a common demo for the different scheduling options, where main osc4py3 functions are highlighted by surrounding comments. You can also look at `speedudpsrv.py` and `speedudpcli.py` in the `demos/` directory.

All things are lightly coupled via names, where name resolution is only done when some event occur - so creation order of the different parts is not important.

Setup the environment

From `osc4py3` package, you must import functions of one of the `as_eventloop`, `as_comthreads`, `as_allthreads` modules. And you must also import `oscbuildparse` module (or directly some of its content) to build new OSC messages, bundles & Co.

```
from osc4py3 import oscbuildparse,
from osc4py3.as_eventloop import *
```

Star import is safe with `as_…` modules, they only export this way the functions described here, which are all prefixed with `osc_`.

You must begin by initializing the system with a call to `osc_startup` function, with an optional Python `logging.logger` parameter (as named `logger`= parameter) if you want to be able to trace operations[2].

```
osc_startup()
```

Then you can install some OSC server channels, to get incoming OSC packets and process them. Here a server to listen on a specific network, and another listening on all IPV4 networks.

```
osc_udp_server("192.168.0.0", 3721, "aservername")
osc_udp_server("0.0.0.0", 3724, "anotherserver")
```

You can also create servers via `osc_broadcast_server` or `osc_multicast_server`.

To send outgoing OSC packets to other systems, you must install some OSC client channels. The name will be used for identification of communication way to use when sending data from other places in your program.

```
osc_udp_client("192.168.0.4", 2781, "aclientname")
```

You can also create clients via `osc_broadcast_client` or `osc_multicast_client` (providing ad-hoc broadcast/multicast addresses).

2 This is highly recommended in case of multi-threads usage - see module `demoslogger.py` in `demos` directory for a basic example.

For incoming packets, you can install several OSC methods to handle specific address pattern messages. You can also create servers via `osc_broadcast_server` or `osc_multicast_server`.

```
osc_method("/test", handlerfunction)
```

The handler will be used each time an OSC message with pattern beginning by `/test` come in the system. The handler is a Python callable (function or bounded method, or partial), which will simply be called with the OSC message *arguments* as parameters. You can also require to receive the message *address* as first parameter using a needaddress=True parameter)

```
osc_method("/test", handlerfunction2, needaddress=True)
```

Now, all is ready to work.

Run the system, process incoming osc

If you choose to run the system with **as_eventloop** or with **as_comthreads**, you must periodically call the function `osc_process` sometime in your event loop.

```
osc_process()
```

This call all code needed in the context of the event loop (all transmissions and dispatching code for as_eventloop, and only OSC methods dispatching for as_comthreads).

If you choose to run the system with **as_allthreads**, you don't need to call `osc_process` (this function is defined in this module too, but does nothing). But you should have checked that your methods will not have problems with concurrent access to resources (if needed there is a way to simply install a method to be called with an automatic Lock).

Send your messages

Simply create an OSC message, and send it using client channel names[3] you choose.

```
msg = oscbuildparse.OSCMessage("/test/me/", ",sif", ["text", 672, 8.871])
osc_send(msg, "aclientname")
```

For basic data types, you can let osc4py3 detect the type tags.

```
msg = oscbuildparse.OSCMessage("/test/me/", None, ["text", 672, 8.871])
osc_send(msg, "aclientname")
```

You can also send a bundle of messages, with a delayed execution time.

```
exectime = time.time() + 10   # execute in 10 seconds
msg1 = oscbuildparse.OSCMessage("/sound/levels", None, [1, 5, 3])
msg2 = oscbuildparse.OSCMessage("/sound/bits", None, [32])
msg3 = oscbuildparse.OSCMessage("/sound/freq", None, [42000])
bun = oscbuildparse.OSCBundle(oscbuildparse.unixtime2timetag(exectime),
                    [msg1, msg2, msg3])
osc_send(bun, "aclientname")
```

It is possible to send other OSC data types, to request an immediate execution time with time tag set to `oscbuildparse.OSC_IMMEDIATELY`, etc. See following section and **oscbuildparse** module help for details about all these possibilities.

Note that when running the system with **as_eventloop**, several calls to `osc_process` may be needed to achieve complete processing of message sending[4].

3 You can directly specify a collection of channel names to use to communicate, via a list or tuple or set, eventually nested - this allow to easily define groups of channels as target for sending.

4 The message itself is generally sent / received in one `osc_process` call if possible, but some monitoring status on sockets may need other calls to be stopped, and several sent messages may take time to be processed on sockets.

At the end

When you have finished, to properly release resources, you should simply call `osc_terminate`.

```
osc_terminate()
```

This correctly close communication resources and terminate background threads if any.

OSC Messages and Bundles

> OSC structures are defined in module **oscbuildparse**. This module provides all low level tools to manipulate these structures, build them, encode them to raw OSC packets, and decode them back. Here is a partial[5] formated output of the module documentation string. You may specially be interested by message construction in the examples (encoding and decoding is normally done automatically for you by osc4py3).
> If you find a difference with module's doc string, the doc string is the reference.

This module is only here to translate OSC packets from/to Python values. It can be reused anywhere as is (only depend on Python3 standard modules).

Examples

```
>>> from osc4py3.oscbuildparse import *
>>> dir()
['OSCBundle', 'OSCCorruptedRawError', 'OSCError', 'OSCInternalBugError',
'OSCInvalidDataError', 'OSCInvalidRawError', 'OSCInvalidSignatureError',
'OSCMessage', 'OSCUnknownTypetagError', 'OSC_BANG', 'OSC_IMMEDIATELY',
'OSC_IMPULSE', 'OSC_INFINITUM', 'OSCbang', 'OSCmidi', 'OSCrgba',
'OSCtimetag',
'__builtins__', '__doc__', '__name__', '__package__', 'decode_packet',
'dumphex_buffer', 'encode_packet', 'float2timetag', 'timetag2float',
'timetag2unixtime', 'unixtime2timetag']

>>> msg = OSCMessage('/my/pattern',',iisf',[1,3,"a string",11.3])
>>> raw = encode_packet(msg)
>>> dumphex_buffer(raw)
000:2f6d792f 70617474 65726e00 2c696973     /my/ patt ern. ,iis
016:66000000 00000001 00000003 61207374     f... .... .... a st
032:72696e67 00000000 4134cccd              ring .... A4..
>>> decode_packet(raw)
OSCMessage(addrpattern='/my/pattern', typetags=',iisf', arguments=(1, 3,
'a string', 11.300000190734863))

>>> import time
>>> bun = OSCBundle(unixtime2timetag(time.time()+1),
        [OSCMessage("/first/message",",ii",[1,2]),
         OSCMessage("/second/message",",fT",[4.5,True])])
>>> raw = encode_packet(bun)
>>> dumphex_buffer(raw)
000:2362756e 646c6500 d2c3e04f 455a9000     #bun dle. ...O EZ..
016:0000001c 2f666972 73742f6d 65737361     .... /fir st/m essa
032:67650000 2c696900 00000001 00000002     ge.. ,ii. .... ....
048:00000018 2f736563 6f6e642f 6d657373     .... /sec ond/ mess
064:61676500 2c665400 40900000              age. ,fT. @...
>>> decode_packet(raw)
```

5 Special advanced options are dismissed.

```
OSCBundle(timetag=OSCtimetag(sec=3536052435, frac=3018637312),
elements=(OSCMessage(addrpattern='/first/message', typetags=',ii',
arguments=(1, 2)), OSCMessage(addrpattern='/second/message', typetags=',fT',
arguments=(4.5, True))))

>>> msg = OSCMessage("/shortcut/with/typedetection", None,
[True, OSC_BANG, 12, 11.3])
>>> raw = encode_packet(msg)
>>> dumphex_buffer(raw)
000:2f73686f 72746375 742f7769 74682f74     /sho rtcu t/wi th/t
016:79706564 65746563 74696f6e 00000000     yped etec tion ....
032:2c544969 66000000 0000000c 4134cccd     ,TIi f... .... A4..
>>> decode_packet(raw)
OSCMessage(addrpattern='/shortcut/with/typedetection', typetags=',TIif',
arguments=(True, OSCbang(), 12, 11.300000190734863))
```

Note : you can found other examples in `osc4py3/tests/buildparse.py` module.

Programming interface

There are two main functions for advanced users:

- `encode_packet` build the binary representation for OSC data
- `decode_packet` retrieve OSC data from binary representation

An OSC packet can either be an OSC message, or an OSC bundle (which contains a collection of messages and bundles - recursively if needed).

For developer point of view, there is an **`OSCMessage`** named tuple which is used as container to encode and decode messages. It contains fields accordingly to OSC1.1 protocol :

- `addrpattern` : a string beginning by `/` and used by OSC dispatching protocol
- `typetags` : a string beginning by `,` and describing how to encode values
- `arguments` : a list or tuple of values to encode

The `typetag` must start by a comma (`','`) and use a set of chars to describe OSC defined data types, as listed in the "Supported atomic data types" table below. It may optionally be fixed to **`None`**, to have an automatic detection of type tags from values. See "Automatic type tagging" below.

And to add a time tag or group several messages in a packet, there is an **`OSCBundle`** named tuple which is used to encode and decode bundles. It contains fields accordingly to OSC1.1 protocol :

- `timetag` : a time representation using two int values, sec:frac
- `elements` : a list or tuple of mixed **`OSCMessage`** / **`OSCBundle`** values

Supported atomic data types

In addition to the required OSC1.1 `ifsbTFNIt` type tag chars, we support optional types of OSC1.0 protocol `hdScrm[]` (support for new types is easy to add if necessary) :

Tag	Data	Python	Notes
`i`	int32	`int`	signed integer (care that Python int has a range

Tag	Data	Python	Notes
			with "no limit")
`t`	timetag	**`OSCtimetag`**	a named tuple with two integer parts : `sec`, `frac` where sec is number of seconds since 1/1/1900 and frac is fractional part of seconds (NTP time format)
`f`	float32	`float`	
`s`	string	`str`	ASCII string
`b`	blob	`memoryview`	mapping to part in received data
`h`	int64	`int`	to transmit larger integers
`d`	float64	`float`	like C double, to transmit larger and more accurate floats
`S`	alt-string	`str`	ASCII strings to distinguish with 's' strings
`c`	ascii-char	`str`	one ASCII byte
`r`	rgba-color	**`OSCrgba`**	four fields named tuple : `red`, `green`, `blue`, `alpha` as int in 0..255 range
`m`	midi-msg	**`OSCmidi`**	four fields named tuple : `portid`, `status`, `data1`, `data2` as int in 0..255 range
`T`	(none)	**`True`**	direct True value only with type tag
`F`	(none)	**`False`**	direct False value only with type tag
`N`	(none)	**`None`**	'nil' in OSC only with type tag
`I`	(none)	**`OSCbang`**	named tuple with no field
`[`	(none)	`tuple`	beginning of an array, followed by type tags of array data, which are grouped in a tuple
`]`	(none)		end of the array (to terminate tuple)

String and char

They are normally transmitted as ASCII, default processing ensure this encoding (replacing non-ASCII chars by `?`). An `oob` option (see “Out-Of-Band options” below) allow specifying an encoding. The value for a string/char is normally a Python `str`. You can give a `bytes` or `bytearray` or `memoryview`, but they must not contain a zero byte (except at the end - and it will be used a string termination when decoding).

For a char, the string/bytes/... must contain only one element.

Blob

They allow transmitting any binary data of your own. The value for a blob can be a `bytes` or `bytearray` or `memoryview`.

When getting blob values on packet reception, they are returned as `memoryview` objects to avoid extra data copying. You may cast them to `bytes` if you want to extract the value from the OSC packet.

Infinitum / Impulse / Bang

The Infinitum data (`'I'`), renamed as Impulse 'bang' in OSC 1.1, is returned in Python as a **`OSCbang`** named tuple (with no value in the tuple). Constant *`OSC_INFINITUM`* is defined as an **`OSCbang`** value, and aliases constants *`OSC_IMPULSE`* and *`OSC_BANG`* are also defined.

You should test on object class with `isinstance(x,OSCbang)`.

Time Tag

As time tag, composed by a tuple with a NTP time representation (a `sec` field and a `frac` field, representing a number of seconds and its fractional part from 1/1/1900). As it is not really usable with usual Python time, four conversion functions have been defined :

- `timetag2float` convert an **`OSCtimetag`** tuple into a float value in seconds from 1/1/1900
- `timetag2unixtime` convert an **`OSCtimetag`** tuple into a Unix float time in seconds from 1/1/1970 (Python time)
- `float2timetag` convert a float value of seconds from 1/1/1900 into an **`OSCtimetag`** tuple
- `unixtime2timetag` convert a Unix float value of seconds from 1/1/1970 (Python time) into an **`OSCtimetag`** tuple

The special value used in OSC to indicate an "immediate" time, with a time tag having 0 in seconds field and 1 in factional part field (represented as 0x00 000 001 value), is available for comparison in a global constant *`OSC_IMMEDIATELY`*.

Array

An array is a way to group some data in the OSC message arguments. On the Python side an array is simply a list or a tuple of values. By example, to create a message with two int followed by four grouped int, you will have :

- Type tags string: `',ii[iiii]'`
- Arguments list: `[3, 1, [4, 2, 8, 9]]`

Note : When decoding a message, array arguments are returned as tuple, not list. In this example: `(3, 1, (4, 2, 8, 9))`

Automatic type tagging

When creating an **`OSCMessage`**, you can give a **`None`** value as `typetags`. Then, message arguments are automatically parsed to identify their types and build the type tags string for you.

The following mapping is used :

What	Type tag and corresponding data
value **`None`**	`N` without data
value **`True`**	`T` without data

What	Type tag and corresponding data
value **`False`**	`F` without data
type `int`	`i` with int32
type `float`	`f` with float32
type `str`	`s` with string
type `bytes`	`b` with raw binary
type `bytearray`	`b` with raw binary
type `memoryview`	`b` with raw binary
type **`OSCrgba`**	`r` with four byte values
type **`OSCmidi`**	`m` with four byte values
type **`OSCbang`**	`I` without data
type **`OSCtimetag`**	`t` with two int32

Errors

All errors explicitly raised by the module use specify hierarchy of exceptions::

```
Exception
 OSCError
  OSCCorruptedRawError
  OSCInternalBugError
  OSCInvalidDataError
  OSCInvalidRawError
  OSCInvalidSignatureError
  OSCUnknownTypetagError
```

OSCError

This is the parent class for OSC errors, usable as a catchall for all errors related to this module.

OSCInvalidDataError

There is a problem in some OSC data provided for encoding to raw OSC representation.

OSCInvalidRawError

There is a problem in a raw OSC buffer when decoding it.

OSCInternalBugError

Hey, we detected a bug in OSC module. Please, signal it with description of the context, data processed, options used.

OSCUnknownTypetagError

Found an invalid (unknown) type tag when encoding or decoding. This include type tags not in a subset with `restrict_typetags` option.

OSCInvalidSignatureError

Check of raw data with signature failed due bad source or modified data. This can only occur with advanced packet control enabled and signature functions installed in out-of-band.

OSCCorruptedRawError

Check of raw data with checksum failed. This can only occur with advanced packet control enabled and checksum functions installed in out-of-band.

Details about simple use

Logging OSC operations

You can enable OSC logging by giving a `logging.Logger` object as `logger=` parameter when calling `osc_start()` function. Example :

```
import logging
logging.basicConfig(format='%(asctime)s - %(threadName)s ø %(name)s - '
    '%(levelname)s - %(message)s')
logger = logging.getLogger("osc")
logger.setLevel(logging.DEBUG)
osc_startup(logger=logger)
```

Multi-threading

Multi-threading add some overhead in the whole processing (due to synchronization between threads). By example, on my computer, transmitting 1000 messages each executing a simple method call, `speedudpsrv.py` goes from 0.2 sec with *eventloop* scheduling to 0.245 sec with *comthreads* scheduling and 0.334 sec with *allthreads* scheduling.
But multi-threading can remain interesting if your main thread does C computing while osc4py3 process messages in background (which is the first use case of this development).

Using **`as_allthreads`** module, threads are used in several places, including message handlers calls - by default a pool of ten working threads are allocated for message handlers calls. To control the final execution of methods, you can add an `execthreadscount` named parameter to `osc_startup()`. Setting it to 0 disable executions of methods in working threads, and all methods are called in the context of the raw packets processing thread, or if necessary in the context of the delayed bundle processing thread. Setting it to 1 build only one working thread to process methods calls, which are then all called in sequence in the context of that thread. If you want to protect your code against race conditions between your main thread and message handlers calls, you should better use **`as_comthreads`**.

Using **`as_comthreads`** module, threads are used for communication operations (sending and receiving, intermediate processing), and messages are stored in queues between threads. Message handler methods are called only when you call yourself `osc_process()`.

This is optimal if you want to achieve maximum background processing of messages transmissions, but final messages processing must occure within an event loop

Using **`as_eventloop`** module, there is no multi-threading, no background processing. All operations (communications and message handling) take place only when you call `osc_process()`.

Advanced use

Message handlers

If some handler fuctions need more informations from received messages than simply its parameters and address, you can directly build an `oscmethod.MethodFilter` object, and register it via `oscdispatching.register_method()` — see `osc_method()` functions in **`as_…`** modules.

This allows you to specify a working queue, a lock to acquire when calling the function, a specific logger to use, some extra parameters to add after message arguments, or eventually require the context of the message (make available the whole message but also informations about its source).

Developer Documentation

First, you should keep the osc4py-`bigpicture.svg` graphic on hand. It shows the general organization of the packages, classes, functions, how they interact, how data are transmitted between the different layers.

The osc4py3 package is cut among two OSC protocol implementation modules : **`oscbuildparse`** and **`oscmethod`** ; four core modules to run processing : **`oscchannel`** with **`oscscheduling`**, **`oscdistributing`**, **`oscdispatching`** ; helped by specialized tool modules, by transport protocol specific modules and by a set of user helper modules providing different scheduling policies.

Module **`oscchannel`** manage transport (emission, reception) of raw OSC packets. It uses monitoring tools from **`oscscheduling`** module to get information of incoming data or connexion availability for outgoing data.

Module **`oscdistributing`** transform OSC message and bundles into/from OSC raw packets and provide them to ad-hoc objects/functions for processing.

Module **`oscdispaching`** identify methods to call from messages address pattern matching and control delayed processing of bundle with future time tag.

Implementation Modules

The two modules described here may be used without core modules if you require a simpler implementation of communications and distribution of OSC packets.

oscbuildparse

This module has an extensive documentation string you are invited to read.

Out-Of-Band options

These options are transmitted among building and parsing functions to activate / deactivate some processing alternatives. Options (keys, type, usage) are listed in the module documentation.

That way, you can control encoding, trace data, restrict supported type tags, activate address pattern compression, enable checksum or authentication signature or encryption, etc (warning : some code is untested).

oscmethod

This module has an extensive documentation string you are invited to read.

Pattern matching can use two syntax, OSC defined syntax or Python regular expression syntax (former is rewritten as later).

You basically create a **`MethodFilter`** object with at least an address pattern (a string) and a callable object. By default address pattern use OSC messages patterns syntax (parameter `patternkind`)

Core Modules

oscchannel

Organize communication channels to send and receive OSC packets from different transport protocols. **`TransportChannel`** provides an abstract class with ad-hoc interfaces. It gives some common services to subclasses and allow organization of sockets management to avoid multiplying threads (use of select or ad-hoc platform specific socket monitoring service upon a scheduling scheme). Specific handlers can be installed at transport channel object level, to have special operations occuring at identified processing time of in/out data (see `actions_handlers`).

oscscheduling

This module does the real job of monitoring the transport channels and transmitting data

oscdistributing

oscdispatching

Specialized Tools

oscpacketoptions

osctoolspools

oscnettools

Transport Protocol

Base transport class

Each transport protocol has its own module defining one or more **`TransportChannel`** subclasses. All these subclasses use the same construction scheme :

```
channel = ChannelClass("channelname", mode, { 'option_key': optionvalue })
```

The mode is a combination of `'r'` and `'w'` for read and write channels (ie. OSC servers to receive/read packets and OSC clients to send/write packets), and `'e'` for stream based channels waiting for connection event.

Common channel options

Most parameters of channels are set via the third parameter in the `options` dictionary. Here is a description of possible keys and values (with default value) :

- `auto_start` (**True**) – `bool` flag to immediately activate the channel and start monitoring its activity once is has been initialized (else you must call yourself `activate()` and `begin_scheduling()` methods)
- `logger` (**None**) – Python `logging.Logger` to trace activity. If left to **None**, there is almost no overhead… but you silently pass all errors. You may better setup one logger with an `logging.ERROR` filtering level.
- `actions_handlers` ({}) – map `{ str : callable / str }` of action verb and code or message to call, see "Action Handlers" below. For advanced usage with personal hacks at some processing times.
- `monitor` (**Monitoring**) – monitor used for this channel - the channel register itself among this monitor when becoming scheduled. This is the tool used to detect state modification on the channel and activate reading or writing of data.
- `monitor_terminate` (**False**) – `bool` flag to indicate that our monitor must be terminated with the channel deletion. Default upon automatic monitor, or use monitor_terminate key in options.
- `read_forceident` (**None**) – map `{ source : identification }` informations to use for peer identification on read data (dont use other identification ways).
- `read_dnsident` (**True**) – `bool` flag to use address to DNS mapping automatically built from **`oscnettools`** module.
- `read_datagram` (**False**) – `bool` flag to consider received data as entire datagrams (no data remain in the buffer, its all processed when received).
- `read_maxsources` (500) – `int` count of maximum different sources allowed to send data to this reader simultaneously, used to limit an eventuel DOS on the system by sending incomplete packets. Set it to 0 to disable the limit. Default to *`MAX_SOURCES_ALLOWED`* (500).
- `read_withslip` (**False**) – `bool` flag to enable SLIP protocol on received data.
- `read_withheader` (**False**) – `bool` flag to enable detection of packet size in the beginning of data, and use `read_headerunpack`.
- `read_headerunpack` (**None**) – `(str,int,int)` for automatic use of `struct.unpack()` or `fct(data) → packsize,headsize` to call a function. For `struct.unpack()`, data is a tuple with : the format to decode header, the fixed header size, and the index of packet length value within the header tuple returned by `unpack()`. For function, it will receive the currently accumulated data and must return a tuple with : the packet size extracted from the header, and the total header size. If there is no enough data in header to extract packet size, the function must return `(0, 0)`. Default to `("!I", 4, 0)` to detect a packet length encoded in 4 bytes unsigned int with network bytes order. If the function need to

pass some data in the current buffer (ex. remaining of an old failed communication), it can return an header size corresponding to the bytes count to ignore, and a packet size of 0 ; this will consume data with an -ignored- empty packet.

- `read_headermaxdatasize` (1 MiB) – `int` maximum count of bytes allowed in a packet size field when using headers. Set it to 0 to disable the limit. Default to *`MAX_PACKET_SIZE_WITH_HEADER`* (1 MiB).
- `write_workqueue` (**`None`**) – **`WorkQueue`** queue of write jobs to execute. This allow to manage initial writing to peers in their own threads (nice for blocking write() calls) or in an event loop if working without thread. The queue will be filled when we detect that a write can occur (ie same channel will have maximum one write operation in the workqueue, even if there are multiple pending operations in the write_pending queue).
- `write_wqterminate` (**`False`**) – `bool` flag to indicate to call work queue termination when terminating the channel.
- `write_withslip` (**`False`**) – `bool` flag to enable SLIP protocol on sent data.
- `write_slip_flagatstart` (**`True`**) – `bool` flag to insert the SLIP END code (192) at beginning of sent data (when using SLIP).

Network address options

For channels using network address and port, the **`oscnettools`** module provide a common way to retrieve these informations. This is done via a options with variable prefix :

- `<prefix>_host` (no default) – `str` representing an host, as a host name resolved via DNS, or as an IP address in IPV4 format or IPV6 format. Can use `"*"` string to specify wildcard address (ex. use with TCP to have server socket on all networks).
- `<prefix>_port` (no default) – `int` or `str` representing a network port number or service name. Can use `0` integer or `"None"` string to specify random port.
- `<prefix>_forceipv4` (**`False`**) – `bool` flag to require use of IPV4 address in case of multiple address family resolution by a DNS.
- `<prefix>_forceipv6` (**`False`**) – `bool` flag to require use of IPV6 address in case of multiple address family resolution by a DNS.

Datagram transport class

Module **`oscudpmc`** manage transport for datagram protocols over IP : UDP, multicast, broadcast, etc. All these protocols share the same **`UdpMcChannel`** transport class, multicast and broadcast simply being enabled via options. An **`UdpMcChannel`** object can only act as a server or as a client, not both.

Datagram channel options

As such transport channels can be used either as a server (reader, receiving OSC packets) or as a client (writer, sending OSC packets), some options have `read` or `write` prefix and are only considered when using channel accordingly.

Read options

- `udpread_host` – see “Network address options“, above. This is the address where the socket is bound for reading.
- `udpread_port` – see “Network address options“, above. This is the port where the socket is bound for reading.
- `udpread_forceipv4` – see “Network address options“, above.
- `udpread_forceipv6` – see “Network address options“, above.
- `udpread_dontcache` (**False**) – `bool` flag to not cache data in case of DNS resolution. By default resolved DNS addresses are cached in the application.
- `udpread_reuseaddr` (**True**) – `bool` flag to enable ioctl settings for reuse of socket address.
- `udpread_nonblocking` (**True**) – `bool` flag to enable non-blocking on the socket.
- `udpread_identusedns`: (**False**) – `bool` flag to translate address to DNS name using oscnettools DNS addresses cache.
- `udpread_identfields` (2) – `int` count of fields of remote address identification to use for source identification. Use 0 for all fields. Default to 2 for (hostname, port) even with IPV6.
- udpread_asstream (**False**) – `bool` flag to process UDP packets with stream-based methods, to manage rebuild of OSC packets from multiple UDP reads. Bad idea - but if you need it, don't miss to set up options like `read_withslip`, `read_withheader`…

Write options

- `udpwrite_host` – see “Network address options“, above. This is the address where the socket will send written packets.
- `udpwrite_port` – see “Network address options“, above. This is the port where the socket will send written packets.
- `udpwrite_outport` (0) – `int` number of port to bind the socket locally. Default to 0 for auto-select.
- `udpwrite_forceipv4` – see “Network address options“, above.
- `udpwrite_forceipv6` – see “Network address options“, above.
- `udpwrite_dontcache` (**False**) – `bool` flag to not cache data in case of DNS resolution. By default resolved DNS addresses are cached in the application.
- `udpread_reuseaddr` (**True**) – `bool` flag to enable ioctl settings for reuse of socket address.
- `udpwrite_ttl` (**None**) - `int` time to leave counter for packets, also used for multicast hops. Default leave OS default settings.

- `udpwrite_nonblocking` (**True**) – `bool` flag to enable non-blocking on the socket.

Multicast & Broadcast options

- `mcast_enabled` (**False**) – `bool` flag to enable multicast. If True, the `udpwrite_host` must be a multicast group, and its a good idea to set `udpwrite_ttl` to 1 (or more if need to reach furthest networks).
- `mcast_loop` (**None**) – `bool` flag to enable/disable looped back multicast packets to host. Normally enabled by default at the OS level. Default to None (don't modify OS settings).
- `bcast_enabled` (**False**) – `bool` flag to enable broadcast. If True, the `udpwrite_host` must be a network broadcast address, and its a good idea to set `udpwrite_ttl` to 1 (or more if need to reach furthest networks).

Stream transport class

Network stream based transport using TCP is defined in module **`osctcp`**. It uses the class **`TcpChannel`** to manage connection and to transmissions.

TCP

- `tcp_consocket` (**None**) – `socket` specified when creating a TcpChannel just after a connection, to manage communications with peer.
- `tcp_consocketspec` (**None**) – `tuple` specifying socket specs.
- `tcp_host` – see "Network address options", above. For a TCP server, you may generally use "*" here to require a server listening on all networks. For a TCP client, just specify the server host.
- `udpread_port` – see "Network address options", above.
- `udpread_forceipv4` – see "Network address options", above.
- `udpread_forceipv6` – see "Network address options", above.
- `tcp_reuseaddr` (**True**) – `bool` flag to enable ioctl settings for reuse of socket address.

User Helpers

These modules are described in the user documentation - we will not describe their interface.

as_eventloop

as_allthreads

as_comthreads

Action Handlers

These are action verbs which can be associated, at the transport channel level, to locally dispatched OSC messages or to direct callback functions to have special processing in some conditions.

Generic

These action handlers apply to all channel kind.

activating

The channel is being activated (ie. system access via open() or like). No action parameter.

activated

The channel has been activated. No action parameter.

deactivating

The channel is being deactivated (ie. stop system access via close() or like). No action parameter.

deactivated

The channel has been deactivated.

scheduling

The channel is being scheduled (ie. begin monitoring of I/O on the underlying layers). No action parameter.

scheduled

The channel has been scheduled. No action parameter.

unscheduling

The channel is being unscheduled. No action parameter.

unscheduled

The channel has been unscheduled. No action parameter.

TODO: Add handlers to get packetoption structures from the channel.

encodepacket

A packet must be encoded to send via the channel.

Two action parameters, the OSC source packet to encode and the packet options for processing. The handler must be a direct callback (as the second parameter is not valid for sending via OSC messages).

If the handler return None, the processing of the packet continue (standard encoding, then sending).

If the handler return (None, None), the processing of the packet stop - we consider that the handler manage itself the packet transmission to the channel.

If the handler return (rawoscdata, packet option), they are used to transmit the raw packet via the channel, and standard encoding is not used.

decodepacket

A packet coming from a transport channel must be decoded.

Two action parameters, the raw OSC packet data to decode and the packet options for processing. The handler must be a direct callback (as the second parameter is not valid for sending via OSC messages).

If the handler return None, the processing of the packet continue (standard decoding, then queue for dispatching).

If the handler return (None, None), the processing of the packet stop - we consider that the handler manage itself the packet transmission to the dispatcher.

If the handler return (packet, packet option), they are used to queue the packet for dispatching, and standard decoding is not used.

UDP

bound

The UDP socket has been bound to a port, waiting for writing or reading. Action parameter is the port number.

TCP

conreq

A connexion request has been received and a transport channel will be created to manage communications on the corresponding socket. A callback trigged on this handler can raise an exception to cancel the establishment of TCP communications. If a callback method

Two parameters : (address, sockfileno). The remote network address as a tuple (maybe more than two items with IPV6) and the socket file number as an integer.

connected

A connexion has been established with a remote pair.