pyczmq Documentation

Release 0.1.1

Michel Pelletier

October 15, 2013

CONTENTS

pyczmq is a CFFI wrapper around CZMQ.

Contents:

CHAPTER

ONE

PYCZMQ PACKAGE

1.1 Submodules

1.2 pyczmq.types module

class pyczmq.types.**Beacon**
Bases: object

class pyczmq.types.**Context**(*iothreads=1*)
Bases: object

Object wrapper around a zctx

set_iothreads(*thread_count*)

set_linger(*msecs*)

set_pipehwm(*hwm*)

set_rcvhwm(*hwm*)

set_sndhwm(*hwm*)

socket(*typ*)

class pyczmq.types.**Frame**(*data=None*, *frame=None*)
Bases: object

Object wrapper around a zframe

DONTWAIT = 4

MORE = 1

REUSE = 2

bytes()

dup()

frame

more()

reset(*data*)

set_more(*val=0*)

strhex()

class pyczmq.types.**Loop**
Bases: object

Object wrapper around a zloop

poller(*poll_item*, *handler*, *arg=None*)

poller_end(*poll_item*)

set_tolerant(*item*)

set_verbose(*verbose*)

start()

timer(*delay*, *times*, *handler*, *arg=None*)

timer_end(*arg*)

class pyczmq.types.**Message**(*msg=None*)
Bases: object

Object wrapper around a zmsg

append(*frame*)

content_size()

dup()

first()

last()

load(*filename*)

msg

next()

pop()

popstr()

push(*frame*)

save(*filename*)

class pyczmq.types.**Socket**(*ctx*, *typ*)
Bases: object

Wrapper class around zsocket/zsockopt

bind(*endpoint*)

connect(*endpoint*)

ctx

disconnect(*endpoint*)

poll(*timeout=0*)

recv()

recv_frame()

recv_frame_nowait()

recv_msg()

recv_nowait()

send(*msg*)

send_frame(*frame*, *flags=0*)

send_msg(*msg*)

sock

type()

unbind(*endpoint*)

1.3 pyczmq.zauth module

pyczmq.zauth.**allow**(*auth*, *addr*)
C: `void zauth_allow (zauth_t *self, char *address);`

Allow (whitelist) a single IP address. For NULL, all clients from this address will be accepted. For PLAIN and CURVE, they will be allowed to continue with authentication. You can call this method multiple times to whitelist multiple IP addresses. If you whitelist a single address, any non-whitelisted addresses are treated as blacklisted.

pyczmq.zauth.**configure_curve**(*auth*, *domain*, *location*)
C: `void zauth_configure_curve (zauth_t *self, char *domain, char *location, ...);`

Configure CURVE authentication for a given domain. CURVE authentication uses a directory that holds all public client certificates, i.e. their public keys. The certificates must be in zcert_save () format. The location is treated as a printf format. To cover all domains, use "*". You can add and remove certificates in that directory at any time. To allow all client keys without checking, specify CURVE_ALLOW_ANY for the location.

pyczmq.zauth.**configure_plain**(*auth*, *domain*, *filename*)
C: `void zauth_configure_plain (zauth_t *self, char *domain, char *filename, ...);`

Configure PLAIN authentication for a given domain. PLAIN authentication uses a plain-text password file. The filename is treated as a printf format. To cover all domains, use "*". You can modify the password file at any time; it is reloaded automatically.

pyczmq.zauth.**deny**(*auth*, *addr*)
C: `void zauth_deny (zauth_t *self, char *address);`

Deny (blacklist) a single IP address. For all security mechanisms, this rejects the connection without any further authentication. Use either a whitelist, or a blacklist, not not both. If you define both a whitelist and a blacklist, only the whitelist takes effect.

pyczmq.zauth.**destroy**(*auth*)
C: `void zauth_destroy (zauth_t **self_p);`

Destructor

pyczmq.zauth.**new**(*ctx*)
C: `zauth_t * zauth_new (zctx_t *ctx);`

Install authentication for the specified context. Returns a new zauth object that you can use to configure authentication. Note that until you add policies, all incoming NULL connections are allowed (classic ZeroMQ behaviour), and all PLAIN and CURVE connections are denied. If there was an error during initialization, returns NULL.

pyczmq.zauth.**set_verbose**(*auth*, *verbose*)
C: `void zauth_set_verbose (zauth_t *self, bool verbose);`

Enable verbose tracing of commands and activity

pyczmq.zauth.**test**(*verbose*)
C: `` int zauth_test (bool verbose);``

Selftest

1.4 pyczmq.zbeacon module

The zbeacon class implements a peer-to-peer discovery service for local networks. A beacon can broadcast and/or capture service announcements using UDP messages on the local area network. This implementation uses IPv4 UDP broadcasts. You can define the format of your outgoing beacons, and set a filter that validates incoming beacons. Beacons are sent and received asynchronously in the background. The zbeacon API provides a incoming beacons on a ZeroMQ socket (the pipe) that you can configure, poll on, and receive messages on. Incoming beacons are always delivered as two frames: the ipaddress of the sender (a string), and the beacon data itself (binary, as published).

pyczmq.zbeacon.**destroy**(*beacon*)
C: `void zbeacon_destroy (zbeacon_t **self_p);`

Destroy a beacon

pyczmq.zbeacon.**hostname**(*beacon*)
C: `char * zbeacon_hostname (zbeacon_t *self);`

Returns the hostname string for this beacon's interface

pyczmq.zbeacon.**new**(*port*)
C: `zbeacon_t * zbeacon_new (int port_nbr);`

Create a new beacon on a certain UDP port

pyczmq.zbeacon.**noecho**(*beacon*)
C: `void zbeacon_noecho (zbeacon_t *self);`

Filter out any beacon that looks exactly like ours

pyczmq.zbeacon.**publish**(*beacon*, *string*)
C: `void zbeacon_publish (zbeacon_t *self, void *transmit, size_t size);`

Start broadcasting beacon to peers at the specified interval

pyczmq.zbeacon.**set_interval**(*beacon*, *interval*)
C: `void zbeacon_set_interval (zbeacon_t *self, int interval);`

Set broadcast interval in milliseconds (default is 1000 msec)

pyczmq.zbeacon.**silence**(*beacon*)
C: `void zbeacon_silence (zbeacon_t *self);`

Stop broadcasting beacons

pyczmq.zbeacon.**socket**(*beacon*)
C: `void * zbeacon_socket (zbeacon_t *self);`

Get beacon ZeroMQ socket, for polling or receiving messages

pyczmq.zbeacon.**subscribe**(*beacon*, *string*)
C: `void zbeacon_subscribe (zbeacon_t *self, char *filter, size_t size);`

Start listening to other peers; zero-sized filter means get everything

pyczmq.zbeacon.**unsubscribe**(*beacon*)
C: `void zbeacon_unsubscribe (zbeacon_t *self);`

Stop listening to other peers

1.5 pyczmq.zcert module

pyczmq.zcert.**apply**(*cert*, *zocket*)
C: `void zcert_apply (zcert_t *self, void *zocket);`

Apply certificate to socket, i.e. use for CURVE security on socket. If certificate was loaded from public file, the secret key will be undefined, and this certificate will not work successfully.

pyczmq.zcert.**destroy**(*cert*)
C: `void zcert_destroy (zcert_t **self_p);`

Destroy a certificate in memory

pyczmq.zcert.**dup**(*cert*)
C: `zcert_t * zcert_dup (zcert_t *self);`

Return copy of certificate

pyczmq.zcert.**eq**(*cert*, *compare*)
C: `bool zcert_eq (zcert_t *self, zcert_t *compare);`

Return true if two certificates have the same keys

pyczmq.zcert.**load**(*filename*)
C: `zcert_t * zcert_load (char *fmt, ...);`

Load certificate from file (constructor) The filename is treated as a printf format specifier.

pyczmq.zcert.**meta**(*cert*, *name*)
C: `char * zcert_meta (zcert_t *self, char *name);`

Get metadata value from certificate; if the metadata value doesn exist, returns NULL.

pyczmq.zcert.**new**()
C: `zcert_t * zcert_new (void);`

Create and initialize a new certificate in memory

pyczmq.zcert.**new_from**(*public_key*, *secret_key*)
C: `zcert_t * zcert_new_from (char *public_key, char *secret_key);`

Constructor, accepts public/secret key pair from caller

pyczmq.zcert.**public_key**(*cert*)
C: `char * zcert_public_key (zcert_t *self);`

Return public part of key pair as 32-byte binary string

pyczmq.zcert.**public_txt**(*cert*)
C: `char * zcert_public_txt (zcert_t *self);`

Return public part of key pair as Z85 armored string

pyczmq.zcert.**save**(*cert*, *filename*)
C: `int zcert_save (zcert_t *self, char *fmt, ...);`

Save full certificate (public + secret) to file for persistent storage This creates one public file and one secret file (filename + "_secret"). The filename is treated as a printf format specifier.

pyczmq.zcert.**save_public**(*cert*, *filename*)
C: `int zcert_save_public (zcert_t *self, char *filename, ...);`

Save public certificate only to file for persistent storage The filename is treated as a printf format specifier.

pyczmq.zcert.**secret_key**(*cert*)
C: `char * zcert_secret_key (zcert_t *self);`

Return secret part of key pair as 32-byte binary string

pyczmq.zcert.**secret_txt**(*cert*)
C: `char * zcert_secret_txt (zcert_t *self);`

Return secret part of key pair as Z85 armored string

pyczmq.zcert.**set_meta**(*cert*, *name*, *value*)
C: `void zcert_set_meta (zcert_t *self, char *name, char *format, ...);`

Set certificate metadata from formatted string.

1.6 pyczmq.zcertstore module

pyczmq.zcertstore.**destroy**(*store*)
C: `void zcertstore_destroy (zcertstore_t **self_p);`

Destroy a certificate store object in memory. Does not affect anything stored on disk.

pyczmq.zcertstore.**dump**(*store*)
C: `void zcertstore_dump (zcertstore_t *self);`

Print out list of certificates in store to stdout, for debugging purposes.

pyczmq.zcertstore.**insert**(*store*, *cert*)
C: `void zcertstore_insert (zcertstore_t *self, zcert_t **cert_p);`

Insert certificate into certificate store in memory. Note that this does not save the certificate to disk. To do that, use zcert_save() directly on the certificate. Takes ownership of zcert_t object.

pyczmq.zcertstore.**lookup**(*store*, *key*)
C: `zcert_t * zcertstore_lookup (zcertstore_t *self, char *public_key);`

Look up certificate by public key, returns zcert_t object if found, else returns NULL. The public key is provided in Z85 text format.

pyczmq.zcertstore.**new**(*location*)
C: `zcertstore_t * zcertstore_new (char *location, ...);`

Create a new certificate store from a disk directory, loading and indexing all certificates in that location. The directory itself may be absent, and created later, or modified at any time. The certificate store is automatically refreshed on any zcertstore_lookup() call. If the location is specified as NULL, creates a pure-memory store, which you can work with by inserting certificates at runtime. The location is treated as a printf format.

1.7 pyczmq.zctx module

The zctx class wraps 0MQ contexts. It manages open sockets in the context and automatically closes these before terminating the context. It provides a simple way to set the linger timeout on sockets, and configure contexts for number of I/O threads. Sets-up signal (interrupt) handling for the process.

The zctx class has these main features:

- Tracks all open sockets and automatically closes them before calling zmq_term(). This avoids an infinite wait on open sockets.
- Automatically configures sockets with a ZMQ_LINGER timeout you can define, and which defaults to zero. The default behavior of zctx is therefore like 0MQ/2.0, immediate termination with loss of any pending messages. You can set any linger timeout you like by calling the zctx_set_linger() method.
- Moves the iothreads configuration to a separate method, so that default usage is 1 I/O thread. Lets you configure this value. Sets up signal (SIGINT and SIGTERM) handling so that blocking calls such as zmq_recv() and zmq_poll() will return when the user presses Ctrl-C.

pyczmq.zctx.**destroy**(*ctx*)
C: `void zctx_destroy (zctx_t **self_p);`

Destroy context and all sockets in it

pyczmq.zctx.**new**()
C: `zctx_t * zctx_new (void);`

Create new context.

pyczmq.zctx.**set_iothreads**(*ctx*, *iothreads*)
C: `void zctx_set_iothreads (zctx_t *self, int iothreads);`

Raise default I/O threads from 1, for crazy heavy applications The rule of thumb is one I/O thread per gigabyte of traffic in or out. Call this method before creating any sockets on the context or the setting will have no effect.

pyczmq.zctx.**set_linger**(*ctx*, *linger*)
C: `void zctx_set_linger (zctx_t *self, int linger);`

Set msecs to flush sockets when closing them, see the ZMQ_LINGER man page section for more details. By default, set to zero, so any in-transit messages are discarded when you destroy a socket or a context.

pyczmq.zctx.**set_pipehwm**(*ctx*, *pipehwm*)
C: `void zctx_set_pipehwm (zctx_t *self, int pipehwm);`

Set initial high-water mark for inter-thread pipe sockets. Note that this setting is separate from the default for normal sockets. You should change the default for pipe sockets *with care*. Too low values will cause blocked threads, and an infinite setting can cause memory exhaustion. The default, no matter the underlying ZeroMQ version, is 1,000.

pyczmq.zctx.**set_rcvhwm**(*ctx*, *rcvhwm*)
C: `void zctx_set_rcvhwm (zctx_t *self, int rcvhwm);`

Set initial receive HWM for all new normal sockets created in context. You can set this per-socket after the socket is created. The default, no matter the underlying ZeroMQ version, is 1,000.

pyczmq.zctx.**set_sndhwm**(*ctx*, *sndhwm*)
C: `void zctx_set_sndhwm (zctx_t *self, int sndhwm);`

Set initial send HWM for all new normal sockets created in context. You can set this per-socket after the socket is created. The default, no matter the underlying ZeroMQ version, is 1,000.

pyczmq.zctx.**underlying**(*ctx*)
C: `void * zctx_underlying (zctx_t *self);`

Return low-level 0MQ context object, will be NULL before first socket is created. Use with care.

1.8 pyczmq.zframe module

The zframe class provides methods to send and receive single message frames across 0MQ sockets. A frame corresponds to one zmq_msg_t. When you read a frame from a socket, the zframe_more() method indicates if the frame is part of an unfinished multipart message. The zframe_send method normally destroys the frame, but with the ZFRAME_REUSE flag, you can send the same frame many times. Frames are binary, and this class has no special support for text data.

pyczmq.zframe.**data**(*frame*)
C: `void * zframe_data (zframe_t *self);`

Return frame data

pyczmq.zframe.**destroy**(*frame*)
C: `void zframe_destroy (zframe_t **self_p);`

Destroy a frame

pyczmq.zframe.**dup**(*frame*)
C: `zframe_t * zframe_dup (zframe_t *self);`

Create a new frame that duplicates an existing frame

pyczmq.zframe.**eq**(*frame*, *other*)
C: `bool zframe_eq (zframe_t *self, zframe_t *other);`

Return TRUE if two frames have identical size and data If either frame is NULL, equality is always false.

pyczmq.zframe.**more**(*frame*)
C: `int zframe_more (zframe_t *self);`

Return frame MORE indicator (1 or 0), set when reading frame from socket or by the zframe_set_more() method

pyczmq.zframe.**new**(*data*)
C: `zframe_t * zframe_new (const void *data, size_t size);`

Create a new frame with optional size, and optional data

Note, no gc wrapper, frames self-destruct by send. If you don't send a frame, you DO have to destroy() it.

pyczmq.zframe.**new_empty**()
C: `zframe_t * zframe_new_empty (void);`

Create an empty (zero-sized) frame.

Note, no gc wrapper, frames self-destruct by send. If you don't send a frame, you DO have to destroy() it.

pyczmq.zframe.**recv**(*sock*)
C: `zframe_t * zframe_recv (void *socket);`

Receive frame from socket, returns zframe_t object or NULL if the recv was interrupted. Does a blocking recv, if you want to not block then use zframe_recv_nowait().

pyczmq.zframe.**recv_nowait**(*sock*)
C: `zframe_t * zframe_recv_nowait (void *socket);`

Receive a new frame off the socket. Returns newly allocated frame, or NULL if there was no input waiting, or if the read was interrupted.

pyczmq.zframe.**reset**(*frame*, *string*)
C: `void zframe_reset (zframe_t *self, const void *data, size_t size);`

Set new contents for frame

pyczmq.zframe.**send**(*frame*, *socket*, *flags*)
C: `int zframe_send (zframe_t **self_p, void *socket, int flags);`

Send a frame to a socket, destroy frame after sending. Return -1 on error, 0 on success.

pyczmq.zframe.**set_more**(*frame*, *more*)
C: `void zframe_set_more (zframe_t *self, int more);`

Set frame MORE indicator (1 or 0). Note this is NOT used when sending frame to socket, you have to specify flag explicitly.

pyczmq.zframe.**size**(*frame*)
C: `size_t zframe_size (zframe_t *self);`

Return number of bytes in frame data

pyczmq.zframe.**strdup**(*frame*)
C: `char * zframe_strdup (zframe_t *self);`

Return frame data copied into freshly allocated string

pyczmq.zframe.**streq**(*frame*, *string*)
C: `bool zframe_streq (zframe_t *self, const char *string);`

Return TRUE if frame body is equal to string, excluding terminator

pyczmq.zframe.**strhex**(*frame*)
C: `char * zframe_strhex (zframe_t *self);`

Return frame data encoded as printable hex string

1.9 pyczmq.zloop module

The zloop class provides an event-driven reactor pattern. The reactor handles zmq_pollitem_t items (pollers or writers, sockets or fds), and once-off or repeated timers. Its resolution is 1 msec. It uses a tickless timer to reduce CPU interrupts in inactive processes.

pyczmq.zloop.**destroy**(*loop*)
C: `void zloop_destroy (zloop_t **self_p);`

Destroy a reactor, this is not necessary if you create it with new.

pyczmq.zloop.**new**()
C: `` zloop_t * zloop_new (void);``

Create a new zloop reactor

pyczmq.zloop.**poller**(*p*, *item*, *handler*, *arg=None*)
C: `int zloop_poller (zloop_t *self, zmq_pollitem_t *item, zloop_fn handler, void *arg);`

Register pollitem with the reactor. When the pollitem is ready, will call the handler, passing the arg. Returns 0 if OK, -1 if there was an error. If you register the pollitem more than once, each instance will invoke its corresponding handler.

pyczmq.zloop.**poller_end**(*loop*, *item*)
C: `void zloop_poller_end (zloop_t *self, zmq_pollitem_t *item);`

Cancel a pollitem from the reactor, specified by socket or FD. If both are specified, uses only socket. If multiple poll items exist for same socket/FD, cancels ALL of them.

pyczmq.zloop.**set_tolerant**(*loop*, *item*)
C: `void zloop_set_tolerant (zloop_t *self, zmq_pollitem_t *item);`

Configure a registered pollitem to ignore errors. If you do not set this, then pollitems that have errors are removed from the reactor silently.

pyczmq.zloop.**set_verbose**(*loop*, *verbose*)
C: `void zloop_set_verbose (zloop_t *self, bool verbose);`

Set verbose tracing of reactor on/off

pyczmq.zloop.**start**(*loop*)
C: `int zloop_start (zloop_t *self);`

Start the reactor. Takes control of the thread and returns when the 0MQ context is terminated or the process is interrupted, or any event handler returns -1. Event handlers may register new sockets and timers, and cancel sockets. Returns 0 if interrupted, -1 if cancelled by a handler.

pyczmq.zloop.**timer**(*loop*, *delay*, *times*, *handler*, *arg*)
C: `int zloop_timer (zloop_t *self, size_t delay, size_t times, zloop_fn handler, void *arg);`

Register a timer that expires after some delay and repeats some number of times. At each expiry, will call the handler, passing the arg. To run a timer forever, use 0 times. Returns 0 if OK, -1 if there was an error.

pyczmq.zloop.**timer_end**(*loop*, *arg*)
C: `int zloop_timer_end (zloop_t *self, void *arg);`

Cancel all timers for a specific argument (as provided in zloop_timer)

1.10 pyczmq.zmq module

pyczmq.zmq.**bind**(*sock*, *addr*)
C: `int zmq_bind (void *s, const char *addr);`

None

pyczmq.zmq.**close**(*sock*)
C: `int zmq_close (void *sock);`

None

pyczmq.zmq.**connect**(*sock*, *addr*)
C: `int zmq_connect (void *s, const char *addr);`

None

pyczmq.zmq.**ctx_get**(*ctx*, *opt*)
C: `int zmq_ctx_get (void *context, int option);`

None

pyczmq.zmq.**ctx_new**()
C: `void *zmq_ctx_new (void);`

None

pyczmq.zmq.**ctx_set**(*ctx*, *opt*, *val*)
C: `int zmq_ctx_set (void *context, int option, int optval);`

None

pyczmq.zmq.**ctx_shutdown**(*ctx*)
C: `int zmq_ctx_shutdown (void *ctx_);`

None

pyczmq.zmq.**ctx_term**(*ctx*)
C: `int zmq_ctx_term (void *context);`

None

pyczmq.zmq.**disconnect**(*sock*, *addr*)
C: `int zmq_disconnect (void *s, const char *addr);`

None

pyczmq.zmq.**errno**()
C: `int zmq_errno (void);`

This function retrieves the errno as it is known to 0MQ library. The goal of this function is to make the code 100% portable, including where 0MQ compiled with certain CRT library (on Windows) is linked to an application that uses different CRT library.

pyczmq.zmq.**getsockopt**(*sock*, *opt*, *val*, *len*)
C: `int zmq_getsockopt (void *s, int option, void *optval, size_t *optvallen);`

None

pyczmq.zmq.**msg_close**(*msg*)
C: `int zmq_msg_close (zmq_msg_t *msg);`

None

pyczmq.zmq.**msg_copy**(*dest*, *src*)
C: `int zmq_msg_copy (zmq_msg_t *dest, zmq_msg_t *src);`

None

pyczmq.zmq.**msg_data**(*msg*)
C: `void *zmq_msg_data (zmq_msg_t *msg);`

None

pyczmq.zmq.**msg_get**(*msg*, *opt*)
C: `int zmq_msg_get (zmq_msg_t *msg, int option);`

None

pyczmq.zmq.**msg_init**(**args*)
C: `` typedef struct zmq_msg_t {unsigned char _ [32];} zmq_msg_t;

typedef void (zmq_free_fn) (void *data, void *hint); ``

msg_init(msg)

C: `int zmq_msg_init (zmq_msg_t *msg);`

None

pyczmq.zmq.**msg_init_data**(*msg*, *data*, *size*, *ffn*, *hint*)
C: `int zmq_msg_init_data (zmq_msg_t *msg, void *data, size_t size, zmq_free_fn *ffn, void *hint);`

None

pyczmq.zmq.**msg_init_size**(*msg*, *size*)
C: `int zmq_msg_init_size (zmq_msg_t *msg, size_t size);`

None

pyczmq.zmq.**msg_more**(*msg*)
C: `int zmq_msg_more (zmq_msg_t *msg);`

None

pyczmq.zmq.**msg_move**(*dest*, *src*)
C: `int zmq_msg_move (zmq_msg_t *dest, zmq_msg_t *src);`

None

pyczmq.zmq.**msg_recv**(*msg*, *s*, *flags*)
C: `int zmq_msg_recv (zmq_msg_t *msg, void *s, int flags);`

None

pyczmq.zmq.**msg_send**(*msg*, *s*, *flags*)
C: `int zmq_msg_send (zmq_msg_t *msg, void *s, int flags);`

None

pyczmq.zmq.**msg_set**(*msg*, *opt*, *val*)
C: `int zmq_msg_set (zmq_msg_t *msg, int option, int optval);`

None

pyczmq.zmq.**msg_size**(*msg*)
C: `size_t zmq_msg_size (zmq_msg_t *msg);`

None

pyczmq.zmq.**poll**(*items*, *nitem*, *timeout*)
C: `int zmq_poll (zmq_pollitem_t *items, int nitems, long timeout);`

None

pyczmq.zmq.**pollitem**(*socket=None*, *fd=0*, *events=0*, *revents=0*)
Helper to create a pollitem object for zmq.poll/zpoller.poller.

Must set either socket or fd, if socket is set, fd is ignored.

pyczmq.zmq.**proxy**(*frontend*, *backend*, *capture*)
C: `int zmq_proxy (void *frontend, void *backend, void *capture);`

None

pyczmq.zmq.**recv**(**args*)
C: `int zmq_send_const (void *s, const void *buf, size_t len, int flags);`

recv(sock, buf, len, flags)

C: `int zmq_recv (void *s, void *buf, size_t len, int flags);`

None

pyczmq.zmq.**recvmsg**(*sock*, *msg*, *flags*)
C: `int zmq_recvmsg (void *s, zmq_msg_t *msg, int flags);`

None

pyczmq.zmq.**send**(*sock*, *buf*, *len*, *flags*)
C: `int zmq_send (void *s, const void *buf, size_t len, int flags);`

None

pyczmq.zmq.**sendmsg**(*sock*, *msg*, *flags*)
C: `int zmq_sendmsg (void *s, zmq_msg_t *msg, int flags);`

None

pyczmq.zmq.**setsockopt**(*sock*, *opt*, *val*, *len*)
C: `int zmq_setsockopt (void *s, int option, const void *optval, size_t optvallen);`

None

pyczmq.zmq.**socket**(*ctx*, *typ*)
C: `void *zmq_socket (void *ctx, int type);`

None

pyczmq.zmq.**socket_monitor**(*sock*, *addr*, *events*)
C: `int zmq_socket_monitor (void *s, const char *addr, int events);`

None

pyczmq.zmq.**strerror**(*num*)
C: `const char *zmq_strerror (int errnum);`

Resolves system errors and 0MQ errors to human-readable string.

pyczmq.zmq.**unbind**(*sock*, *addr*)
C: `int zmq_unbind (void *s, const char *addr);`

None

pyczmq.zmq.**version**()
C: `void zmq_version (int *major, int *minor, int *patch);`

Returns the tuple (major, minor, patch) of the current zmq version.

pyczmq.zmq.**z85_encode**(*dest*, *data*, *size*)
C: `char *zmq_z85_encode (char *dest, uint8_t *data, size_t size);`

None

pyczmq.zmq.**z86_decode**(*dest*, *string*)
C: `uint8_t *zmq_z85_decode (uint8_t *dest, char *string);`

None

1.11 pyczmq.zmsg module

The zmsg class provides methods to send and receive multipart messages across 0MQ sockets. This class provides a list-like container interface, with methods to work with the overall container. zmsg_t messages are composed of zero or more zframe_t frames.

pyczmq.zmsg.**addstr**(*msg*, *fmt*)
C: `int zmsg_addstr (zmsg_t *self, const char *format, ...);`

Push string as new frame to end of message. Returns 0 on success, -1 on error.

pyczmq.zmsg.**append**(*m*, *f*)
C: `int zmsg_append (zmsg_t *self, zframe_t **frame_p);`

Add frame to the end of the message, i.e. after all other frames. Message takes ownership of frame, will destroy it when message is sent. Returns 0 on success. Deprecates zmsg_add, which did not nullify the caller's frame reference.

pyczmq.zmsg.**content_size**(*msg*)
C: `size_t zmsg_content_size (zmsg_t *self);`

Return total size of all frames in message.

pyczmq.zmsg.**destroy**(*m*)
C: `void zmsg_destroy (zmsg_t **self_p);`

Destroy a message object and all frames it contains

pyczmq.zmsg.**dup**(*msg*)
C: `zmsg_t * zmsg_dup (zmsg_t *self);`

Create copy of message, as new message object. Returns a fresh zmsg_t object, or NULL if there was not enough heap memory.

pyczmq.zmsg.**first**(*msg*)
C: `zframe_t * zmsg_first (zmsg_t *self);`

Set cursor to first frame in message. Returns frame, or NULL, if the message is empty. Use this to navigate the frames as a list.

pyczmq.zmsg.**last**(*msg*)
C: `zframe_t * zmsg_last (zmsg_t *self);`

Return the last frame. If there are no frames, returns NULL.

pyczmq.zmsg.**load**(*msg*, *filename*)
C: `zmsg_t * zmsg_load (zmsg_t *self, FILE *file);`

Load/append an open file into message, create new message if null message provided. Returns NULL if the message could not be loaded.

pyczmq.zmsg.**new**()
C: `zmsg_t * zmsg_new (void);`

Create a new empty message object,

Note, no gc wrapper, messages self-destruct by send. If you don't send a message, you DO have to destroy() it.

pyczmq.zmsg.**next**(*msg*)
C: `zframe_t * zmsg_next (zmsg_t *self);`

Return the next frame. If there are no more frames, returns NULL. To move to the first frame call zmsg_first(). Advances the cursor.

pyczmq.zmsg.**pop**(*msg*)
C: `zframe_t * zmsg_pop (zmsg_t *self);`

Remove first frame from message, if any. Returns frame, or NULL. Caller now owns frame and must destroy it when finished with it.

pyczmq.zmsg.**popstr**(*msg*)
C: `char * zmsg_popstr (zmsg_t *self);`

Pop frame off front of message, return as fresh string. If there were no more frames in the message, returns NULL.

pyczmq.zmsg.**push**(*msg*, *frame*)
C: `int zmsg_push (zmsg_t *self, zframe_t *frame);`

Push frame to the front of the message, i.e. before all other frames. Message takes ownership of frame, will destroy it when message is sent. Returns 0 on success, -1 on error.

pyczmq.zmsg.**pushstr**(*msg*, *fmt*)
C: `int zmsg_pushstr (zmsg_t *self, const char *format, ...);`

Push string as new frame to front of message. Returns 0 on success, -1 on error.

pyczmq.zmsg.**recv**(*socket*)
C: `zmsg_t * zmsg_recv (void *socket);`

Receive message from socket, returns zmsg_t object or NULL if the recv was interrupted. Does a blocking recv, if you want to not block then use the zloop class or zmq_poll to check for socket input before receiving.

pyczmq.zmsg.**remove**(*msg*, *frame*)
C: `void zmsg_remove (zmsg_t *self, zframe_t *frame);`

Remove specified frame from list, if present. Does not destroy frame.

pyczmq.zmsg.**save**(*msg*, *filename*)
C: `int zmsg_save (zmsg_t *self, FILE *file);`

Save message to an open file, return 0 if OK, else -1. The message is saved as a series of frames, each with length and data. Note that the file is NOT guaranteed to be portable between operating systems, not versions of CZMQ. The file format is at present undocumented and liable to arbitrary change.

pyczmq.zmsg.**send**(*m*, *socket*)
C: `int zmsg_send (zmsg_t **self_p, void *socket);`

Send message to socket, destroy after sending. If the message has no frames, sends nothing but destroys the message anyhow. Safe to call if zmsg is null.

pyczmq.zmsg.**size**(*msg*)
C: `size_t zmsg_size (zmsg_t *self);`

Return size of message, i.e. number of frames (0 or more).

pyczmq.zmsg.**unwrap**(*msg*)
C: `zframe_t * zmsg_unwrap (zmsg_t *self);`

Pop frame off front of message, caller now owns frame If next frame is empty, pops and destroys that empty frame.

pyczmq.zmsg.**wrap**(*msg*, *frame*)
C: `void zmsg_wrap (zmsg_t *self, zframe_t *frame);`

Push frame plus empty frame to front of message, before first frame. Message takes ownership of frame, will destroy it when message is sent.

1.12 pyczmq.zpoller module

The zpoller class provides a minimalist interface to ZeroMQ's zmq_poll API, for the very common case of reading from a number of sockets. It does not provide polling for output, nor polling on file handles. If you need either of these, use the zmq_poll API directly.

pyczmq.zpoller.**destroy**(*poller*)
C: `void zpoller_destroy (zpoller_t **self_p);`

Destroy a poller

pyczmq.zpoller.**expired**(*poller*)
C: `bool zpoller_expired (zpoller_t *self);`

Return true if the last zpoller_wait () call ended because the timeout expired, without any error.

pyczmq.zpoller.**new**(**readers*, *reader*)
C: `zpoller_t * zpoller_new (void *reader, ...);`

Create new poller

pyczmq.zpoller.**terminated**(*poller*)
C: `bool zpoller_terminated (zpoller_t *self);`

Return true if the last zpoller_wait () call ended because the process was interrupted, or the parent context was destroyed.

pyczmq.zpoller.**wait**(*poller*, *timeout*)
C: `` void * zpoller_wait (zpoller_t *self, int timeout);``

Poll the registered readers for I/O, return first socket that has input. This means the order that sockets are defined in the poll list affects their priority. If you need a balanced poll, use the low level zmq_poll method directly.

1.13 pyczmq.zsocket module

The zsocket class provides helper functions for 0MQ sockets. It doesn't wrap the 0MQ socket type, to avoid breaking all libzmq socket-related calls.

pyczmq.zsocket.**bind**(*sock*, *fmt*)
C: `int zsocket_bind (void *socket, const char *format, ...);`

Bind a socket to a formatted endpoint. If the port is specified as '*', binds to any free port from ZSOCKET_DYNFROM to ZSOCKET_DYNTO and returns the actual port number used. Otherwise asserts that the bind succeeded with the specified port number. Always returns the port number if successful.

pyczmq.zsocket.**connect**(*sock*, *fmt*)
C: `int zsocket_connect (void *socket, const char *format, ...);`

Connect a socket to a formatted endpoint Returns 0 if OK, -1 if the endpoint was invalid.

pyczmq.zsocket.**destroy**(*ctx*, *socket*)
C: `void zsocket_destroy (zctx_t *self, void *socket);`

Destroy a socket within our CZMQ context.

'pyczmq.zsocket.new' automatically registers this destructor with the garbage collector, so this is normally not necessary to use, unless you need to destroy sockets created by some other means (like a call directly to 'pyczmq.C.zsocket_new')

pyczmq.zsocket.**disconnect**(*sock*, *fmt*)
C: `int zsocket_disconnect (void *socket, const char *format, ...);`

Disonnect a socket from a formatted endpoint Returns 0 if OK, -1 if the endpoint was invalid or the function isn't supported.

pyczmq.zsocket.**new**(*ctx*, *typ*)
C: `void * zsocket_new (zctx_t *self, int type);`

Create a new socket within our CZMQ context, replaces zmq_socket. Use this to get automatic management of the socket at shutdown. Note: SUB sockets do not automatically subscribe to everything; you must set filters explicitly.

pyczmq.zsocket.**poll**(*sock*, *msecs*)
C: `bool zsocket_poll (void *socket, int msecs);`

Poll for input events on the socket. Returns TRUE if there is input ready on the socket, else FALSE.

pyczmq.zsocket.**type_str**(*sock*)
C: `char * zsocket_type_str (void *socket);`

Returns socket type as printable constant string

pyczmq.zsocket.**unbind**(*sock*, *fmt*)
C: `int zsocket_unbind (void *socket, const char *format, ...);`

Unbind a socket from a formatted endpoint. Returns 0 if OK, -1 if the endpoint was invalid or the function isn't supported.

1.14 pyczmq.zsockopt module

1.15 pyczmq.zstr module

The zstr class provides utility functions for sending and receiving strings across 0MQ sockets. It sends strings without a terminating null, and appends a null byte on received strings. This class is for simple message sending.

pyczmq.zstr.**recv**(*sock*)
C: `char * zstr_recv (void *socket);`

Receive a string off a socket, caller must free it

pyczmq.zstr.**recv_nowait**(*sock*)
C: `char * zstr_recv_nowait (void *socket);`

Receive a string off a socket if socket had input waiting

pyczmq.zstr.**recvx**(*sock*, *string_p*)
C: `int zstr_recvx (void *socket, char **string_p, ...);`

Receive a series of strings (until NULL) from multipart data Each string is allocated and filled with string data; if there are not enough frames, unallocated strings are set to NULL. Returns -1 if the message could not be read, else returns the number of strings filled, zero or more.

pyczmq.zstr.**send**(*sock*, *string*)
C: `int zstr_send (void *socket, const char *format, ...);`

Send a formatted string to a socket

pyczmq.zstr.**sendm**(*sock*, *string*)
C: `int zstr_sendm (void *socket, const char *format, ...);`

Send a formatted string to a socket, with MORE flag

pyczmq.zstr.**sendx**(**strings*, *sock*)

C: `int zstr_sendx (void *socket, const char *string, ...);`

Send a series of strings (until NULL) as multipart data Returns 0 if the strings could be sent OK, or -1 on error.

1.16 Module contents

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

p