zenoh-pico

Release 0.6.0-beta.1

ZettaScale Zenoh team

Sep 30, 2022

CONTENTS

The *libzenoh-pico* library provides a C client API for the zenoh protocol.

CHAPTER

ONE

API REFERENCE

1.1 Zenoh Types

1.1.1 Enums

enum **z_whatami_t**

Whatami values, defined as a bitmask.

enumerator **Z_WHATAMI_ROUTER**

Bitmask to filter Zenoh routers.

enumerator **Z_WHATAMI_PEER**

Bitmask to filter for Zenoh peers.

enumerator **Z_WHATAMI_CLIENT**

Bitmask to filter for Zenoh clients.

enum **zp_keyexpr_canon_status_t**

Status values for keyexpr canonization operation. Used as return value of canonization-related functions, like *z_keyexpr_is_canon()* or *z_keyexpr_canonize()*.

enumerator **Z_KEYEXPR_CANON_SUCCESS**

The key expression is canon.

enumerator **Z_KEYEXPR_CANON_LONE_DOLLAR_STAR**

The key contains a $* chunk, which must be replaced by *.

enumerator **Z_KEYEXPR_CANON_SINGLE_STAR_AFTER_DOUBLE_STAR**

The key contains ** / *, which must be replaced by * / **.

enumerator **Z_KEYEXPR_CANON_DOUBLE_STAR_AFTER_DOUBLE_STAR**

The key contains ** / **, which must be replaced by **.

enumerator **Z_KEYEXPR_CANON_EMPTY_CHUNK**

The key contains empty chunks.

enumerator **Z_KEYEXPR_CANON_STARS_IN_CHUNK**

The key contains a * in a chunk without being escaped by a DSL, which is forbidden.

enumerator **Z_KEYEXPR_CANON_DOLLAR_AFTER_DOLLAR_OR_STAR**

The key contains $*$ or $$, which is forbidden.

enumerator **Z_KEYEXPR_CANON_CONTAINS_SHARP_OR_QMARK**

The key contains # or ?, which is forbidden.

enumerator **Z_KEYEXPR_CANON_CONTAINS_UNBOUND_DOLLAR**

The key contains a $ which is not bound to a DSL.

enum **z_sample_kind_t**

Sample kind values.

enumerator **Z_SAMPLE_KIND_PUT**

The Sample was issued by a `put` operation.

enumerator **Z_SAMPLE_KIND_DELETE**

The Sample was issued by a `delete` operation.

enum **z_encoding_prefix_t**

Zenoh encoding values. These values are based on already existing HTTP MIME types and extended with other relevant encodings.

enumerator **Z_ENCODING_PREFIX_EMPTY**

Encoding not defined.

enumerator **Z_ENCODING_PREFIX_APP_OCTET_STREAM**

`application/octet-stream`. Default value for all other cases. An unknown file type should use this type. Z_ENCODING_PREFIX_APP_CUSTOM: Custom application type. Non IANA standard.

enumerator **Z_ENCODING_PREFIX_TEXT_PLAIN**

`text/plain`. Default value for textual files. A textual file should be human-readable and must not contain binary data. Z_ENCODING_PREFIX_APP_PROPERTIES: Application properties type. Non IANA standard. Z_ENCODING_PREFIX_APP_JSON: `application/json`. JSON format.

enumerator **Z_ENCODING_PREFIX_APP_SQL**

Application sql type. Non IANA standard. Z_ENCODING_PREFIX_APP_INTEGER: Application integer type. Non IANA standard. Z_ENCODING_PREFIX_APP_FLOAT: Application float type. Non IANA standard.

enumerator **Z_ENCODING_PREFIX_APP_XML**

`application/xml`. XML.

enumerator **Z_ENCODING_PREFIX_APP_XHTML_XML**

`application/xhtml+xml`. XHTML.

enumerator **Z_ENCODING_PREFIX_APP_X_WWW_FORM_URLENCODED**

application/x-www-form-urlencoded. The keys and values are encoded in key-value tuples separated by '&', with a '=' between the key and the value.

Z_ENCODING_PREFIX_TEXT_JSON: Text JSON. Non IANA standard. Z_ENCODING_PREFIX_TEXT_HTML Markup Language (HTML). Z_ENCODING_PREFIX_TEXT_XML: `text/xml`. *Application/xml* is recommended as of RFC 7303 (section 4.1), but *text/xml* is still used sometimes. Z_ENCODING_PREFIX_TEXT_CSS: `text/css`. Cascading Style Sheets (CSS). Z_ENCODING_PREFIX_TEXT_CSV: `text/csv`. Comma-separated values (CSV).

enumerator **Z_ENCODING_PREFIX_TEXT_JAVASCRIPT**

`text/javascript`. JavaScript.

enumerator **Z_ENCODING_PREFIX_IMAGE_JPEG**

`image/jpeg`. JPEG images.

enumerator **Z_ENCODING_PREFIX_IMAGE_PNG**

`image/png`. Portable Network Graphics.

enumerator **Z_ENCODING_PREFIX_IMAGE_GIF**

`image/gif`. Graphics Interchange Format (GIF).

enum **z_consolidation_mode_t**

Consolidation mode values.

enumerator **Z_CONSOLIDATION_MODE_AUTO**

Let Zenoh decide the best consolidation mode depending on the query selector.

enumerator **Z_CONSOLIDATION_MODE_NONE**

No consolidation is applied. Replies may come in any order and any number.

enumerator **Z_CONSOLIDATION_MODE_MONOTONIC**

It guarantees that any reply for a given key expression will be monotonic in time w.r.t. the previous received replies for the same key expression. I.e., for the same key expression multiple replies may be received. It is guaranteed that two replies received at t1 and t2 will have timestamp ts2 > ts1. It optimizes latency.

enumerator **Z_CONSOLIDATION_MODE_LATEST**

It guarantees unicity of replies for the same key expression. It optimizes bandwidth.

enum **z_reliability_t**

Reliability values.

enumerator **Z_RELIABILITY_BEST_EFFORT**

Defines reliability as `BEST_EFFORT`

enumerator **Z_RELIABILITY_RELIABLE**

Defines reliability as `RELIABLE`

enum **z_reply_tag_t**

Reply tag values.

enumerator **Z_REPLY_TAG_DATA**

Tag identifying that the reply contains some data.

enumerator **Z_REPLY_TAG_FINAL**

Tag identifying that the reply does not contain any data and that there will be no more replies for this query.

enum **z_congestion_control_t**

Congestion control values.

enumerator **Z_CONGESTION_CONTROL_BLOCK**

Defines congestion control as `BLOCK`. Messages are not dropped in case of congestion control.

enumerator **Z_CONGESTION_CONTROL_DROP**

Defines congestion control as `DROP`. Messages are dropped in case of congestion control.

enum **z_priority_t**

Priority of Zenoh messages values.

enumerator **_Z_PRIORITY_CONTROL**

Priority for `Control` messages.

enumerator **Z_PRIORITY_REAL_TIME**

Priority for `RealTime` messages.

enumerator **Z_PRIORITY_INTERACTIVE_HIGH**

Highest priority for `Interactive` messages.

enumerator **Z_PRIORITY_INTERACTIVE_LOW**

Lowest priority for `Interactive` messages.

enumerator **Z_PRIORITY_DATA_HIGH**

Highest priority for `Data` messages.

enumerator **Z_PRIORITY_DATA**

Default priority for `Data` messages.

enumerator **Z_PRIORITY_DATA_LOW**

Lowest priority for `Data` messages.

enumerator **Z_PRIORITY_BACKGROUND**

Priority for `Background traffic` messages.

enum **z_submode_t**

Subscription mode values.

enumerator **Z_SUBMODE_PUSH**

Defines the subscription with a push paradigm.

enumerator **Z_SUBMODE_PULL**

Defines the subscription with a pull paradigm.

enum **z_query_target_t**

Query target values.

enumerator **Z_QUERY_TARGET_BEST_MATCHING**

The nearest complete queryable if any else all matching queryables.

enumerator **Z_QUERY_TARGET_ALL**

All matching queryables.

enumerator **Z_QUERY_TARGET_ALL_COMPLETE**

A set of complete queryables.

1.1.2 Data Structures

typedef int **z_zint_t**

Represents a variable-length encoding unsigned integer.

It is equivalent to the size of a `size_t`.

typedef int **z_bytes_t**

Represents an array of bytes.

size_t **len**

The length of the bytes array.

uint8_t ***start**

A pointer to the bytes array.

type **z_id_t**

Represents a Zenoh ID.

In general, valid Zenoh IDs are LSB-first 128bit unsigned and non-zero integers.

uint8_t **id**[16]

The array containing the 16 octets of a Zenoh ID.

typedef int **z_string_t**

Represents a string without null-terminator.

size_t **len**

The length of the string.

const char ***val**

A pointer to the string.

typedef int **z_keyexpr_t**

Represents a key expression in Zenoh.

Operations over *z_keyexpr_t* must be done using the provided functions:

- *z_keyexpr()*
- *z_keyexpr_is_initialized()*
- *z_keyexpr_to_string()*
- *zp_keyexpr_resolve()*

type **z_config_t**

Represents a Zenoh configuration.

Configurations are usually used to set the parameters of a Zenoh session upon its opening.

Operations over *z_config_t* must be done using the provided functions:

- *z_config_new()*
- *z_config_default()*
- *zp_config_get()*
- *zp_config_insert()*

type **z_session_t**

Represents a Zenoh session.

type **z_subscriber_t**

Represents a Zenoh (push) Subscriber entity.

Operations over *z_subscriber_t* must be done using the provided functions:

- *z_declare_subscriber()*
- *z_undeclare_subscriber()*

type **z_pull_subscriber_t**

Represents a Zenoh Pull Subscriber entity.

Operations over *z_pull_subscriber_t* must be done using the provided functions:

- *z_declare_pull_subscriber()*
- *z_undeclare_pull_subscriber()*

- *z_subscriber_pull()*

type **z_publisher_t**

Represents a Zenoh Publisher entity.

Operations over *z_publisher_t* must be done using the provided functions:

- *z_declare_publisher()*
- *z_undeclare_publisher()*
- *z_publisher_put()*
- *z_publisher_delete()*

type **z_queryable_t**

Represents a Zenoh Queryable entity.

Operations over *z_queryable_t* must be done using the provided functions:

- *z_declare_queryable()*
- *z_undeclare_queryable()*

typedef int **z_encoding_t**

Represents the encoding of a payload, in a MIME-like format.

z_encoding_prefix_t **prefix**

The integer prefix of this encoding.

z_bytes_t **suffix**

The suffix of this encoding. It MUST be a valid UTF-8 string.

type **z_value_t**

Represents a Zenoh value.

z_bytes_t **payload**

The payload of this zenoh value.

z_encoding_t **encoding**

The encoding of the *payload*.

type **z_subscriber_options_t**

Represents the set of options that can be applied to a (push) subscriber, upon its declaration via *z_declare_subscriber()*.

z_reliability_t **reliability**

The subscription reliability.

type **z_pull_subscriber_options_t**

Represents the set of options that can be applied to a pull subscriber, upon its declaration via *z_declare_pull_subscriber()*.

z_reliability_t **reliability**

The subscription reliability.

type **z_query_consolidation_t**

Represents the replies consolidation to apply on replies to a *z_get()*.

z_consolidation_mode_t **mode**

Defines the consolidation mode to apply to the replies.

type **z_publisher_options_t**

Represents the set of options that can be applied to a publisher, upon its declaration via *z_declare_publisher()*.

z_congestion_control_t **congestion_control**

The congestion control to apply when routing messages from this

publisher. z_priority_t priority: The priority of messages issued by this publisher.

type **z_queryable_options_t**

Represents the set of options that can be applied to a queryable, upon its declaration via *z_declare_queryable()*.

bool **complete**

The completeness of the queryable.

type **z_query_reply_options_t**

Represents the set of options that can be applied to a query reply, sent via *z_query_reply()*.

z_encoding_t **encoding**

The encoding of the payload.

type **z_put_options_t**

Represents the set of options that can be applied to the put operation, whenever issued via *z_put()*.

z_encoding_t **encoding**

The encoding of the payload.

z_congestion_control_t **congestion_control**

The congestion control to apply when routing this message.

z_priority_t **priority**

The priority of this message when routed.

type **z_delete_options_t**

Represents the set of options that can be applied to the delete operation, whenever issued via *z_delete()*.

z_congestion_control_t **congestion_control**

The congestion control to apply when routing this message.

z_priority_t **priority**

The priority of this message when router.

type **z_publisher_put_options_t**

Represents the set of options that can be applied to the put operation by a previously declared publisher, whenever issued via *z_publisher_put()*.

z_encoding_t **encoding**

The encoding of the payload.

type **z_publisher_delete_options_t**

Represents the set of options that can be applied to the delete operation by a previously declared publisher, whenever issued via *z_publisher_delete()*.

type **z_get_options_t**

Represents the set of options that can be applied to the get operation, whenever issued via *z_get()*.

z_query_target_t **target**

The queryables that should be targeted by this get.

z_query_consolidation_t **consolidation**

The replies consolidation strategy to apply on replies.

typedef int **z_sample_t**

Represents a data sample.

A sample is the value associated to a given *z_keyexpr_t* at a given point in time.

z_keyexpr_t **keyexpr**

The keyexpr of this data sample.

z_bytes_t **payload**

The value of this data sample.

z_encoding_t **encoding**

The encoding of the value of this data sample.

z_sample_kind_t **kind**

The kind of this data sample (PUT or DELETE).

z_timestamp_t **timestamp**

The timestamp of this data sample.

typedef int **z_hello_t**

Represents the content of a *hello* message returned by a zenoh entity as a reply to a *scout* message.

unsigned int **whatami**

The kind of zenoh entity.

z_bytes_t **pid**

The Zenoh ID of the scouted entity (empty if absent).

z_str_array_t **locators**

The locators of the scouted entity.

typedef int **z_reply_t**

Represents the reply to a query.

z_reply_data_t **data**

the content of the reply.

typedef int **z_reply_data_t**

Represents the content of a reply to a query.

z_sample_t **sample**

The `_z_sample_t` containing the key and value of the reply.

z_bytes_t **replier_id**

The id of the replier that sent this reply.

type **zp_task_read_options_t**

Represents the set of options that can be applied to the read task, whenever issued via *zp_start_read_task()*.

type **zp_task_lease_options_t**

Represents the set of options that can be applied to the lease task, whenever issued via *zp_start_lease_task()*.

type **zp_read_options_t**

Represents the set of options that can be applied to the read operation, whenever issued via *zp_read()*.

type **zp_send_keep_alive_options_t**

Represents the set of options that can be applied to the keep alive send, whenever issued via *zp_send_keep_alive()*.

1.1.3 Arrays

type **z_str_array_t**

Represents an array of `char *`.

Operations over *z_str_array_t* must be done using the provided functions:

- `char *z_str_array_get(z_str_array_t *a, size_t k);`
- `size_t z_str_array_len(z_str_array_t *a);`
- `uint8_t z_str_array_array_is_empty(z_str_array_t *a);`

1.1.4 Owned Types

Like most `z_owned_X_t` types, you may obtain an instance of `z_X_t` by loaning it using `z_X_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_X_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a move, `val` will still exist, but will no longer be valid. The destructors are double-free-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_X_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid.

type **z_owned_bytes_t**

A zenoh-allocated *z_bytes_t*.

type **z_owned_string_t**

A zenoh-allocated *z_string_t*.

type **z_owned_keyexpr_t**

A zenoh-allocated *z_keyexpr_t*.

type **z_owned_config_t**

A zenoh-allocated *z_config_t*.

type **z_owned_session_t**

A zenoh-allocated *z_session_t*.

type **z_owned_subscriber_t**

A zenoh-allocated *z_subscriber_t*.

type **z_owned_pull_subscriber_t**

A zenoh-allocated *z_pull_subscriber_t*.

type **z_owned_publisher_t**

A zenoh-allocated *z_publisher_t*.

type **z_owned_queryable_t**

A zenoh-allocated *z_queryable_t*.

type **z_owned_reply_t**

A zenoh-allocated *z_reply_t*.

type **z_owned_str_array_t**

A zenoh-allocated *z_str_array_t*.

1.1.5 Closures

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks:

- context: a pointer to an arbitrary state.
- call: the typical callback function. `context` will be passed as its last argument.
- drop: allows the callback's state to be freed. `context` will be passed as its last argument.

Closures are not guaranteed not to be called concurrently.

It is guaranteed that:

- `call` will never be called once `drop` has started.
- `drop` will only be called **once**, and **after every** `call` has ended.
- The two previous guarantees imply that `call` and `drop` are never called concurrently.

type **z_owned_closure_sample_t**

Represents the sample closure.

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks.

void ***context**

a pointer to an arbitrary state.

_z_data_handler_t **call**

*void *call(const struct z_sample_t*, const void *context)* is the callback function.

_z_dropper_handler_t **drop**

*void *drop(void*)* allows the callback's state to be freed.

type **z_owned_closure_query_t**

Represents the query callback closure.

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks.

void ***context**

a pointer to an arbitrary state.

_z_questionable_handler_t **call**

*void (*_z_questionable_handler_t)(z_query_t *query, void *arg)* is the callback

function. _z_dropper_handler_t drop: *void *drop(void*)* allows the callback's state to be freed.

type **z_owned_closure_reply_t**

Represents the query reply callback closure.

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks.

void *context

a pointer to an arbitrary state.

z_owned_reply_handler_t call

*void (*z_owned_reply_handler_t)(z_owned_reply_t reply, void *arg)* is the callback

function. _z_dropper_handler_t drop: *void *drop(void*)* allows the callback's state to be freed.

type z_owned_closure_hello_t

Represents the Zenoh ID callback closure.

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks.

void *context

a pointer to an arbitrary state.

z_owned_hello_handler_t call

*void (*z_owned_hello_handler_t)(const z_owned_hello_t *hello, void *arg)* is the

callback function. _z_dropper_handler_t drop: *void *drop(void*)* allows the callback's state to be freed.

type z_owned_closure_zid_t

Represents the Zenoh ID callback closure.

A closure is a structure that contains all the elements for stateful, memory-leak-free callbacks.

void *context

a pointer to an arbitrary state.

z_id_handler_t call

*void (*z_id_handler_t)(const z_id_t *id, void *arg)* is the callback function.

_z_dropper_handler_t drop

*void *drop(void*)* allows the callback's state to be freed.

1.2 Zenoh Functions

1.2.1 Macros

z_loan(x)

Defines a generic function for loaning any of the z_owned_X_t types.

Parameters

- **x** – The instance to loan.

Returns Returns the loaned type associated with *x*.

z_move(x)

Defines a generic function for moving any of the z_owned_X_t types.

Parameters

- **x** – The instance to move.

Returns Returns the instance associated with *x*.

z_check(x)

Defines a generic function for checking the validity of any of the `z_owned_X_t` types.

Parameters

- **x** – The instance to check.

Returns Returns `true` if valid, or `false` otherwise.

z_clone(x)

Defines a generic function for cloning any of the `z_owned_X_t` types.

Parameters

- **x** – The instance to clone.

Returns Returns the cloned instance of *x*.

z_drop(x)

Defines a generic function for droping any of the `z_owned_X_t` types.

Parameters

- **x** – The instance to drop.

z_closure()

Defines a variadic macro to ease the definition of callback closures.

Parameters

- **callback** – the typical `callback` function. `context` will be passed as its last argument.
- **droper** – allows the callback's state to be freed. `context` will be passed as its last argument.
- **context** – a pointer to an arbitrary state.

Returns Returns the new closure.

1.2.2 Primitives

z_keyexpr_t **z_keyexpr**(const char *name)

Data Types Handlers

Constructs a *z_keyexpr_t* departing from a string. It is a loaned key expression that aliases `name`.

Parameters

- **name** – Pointer to string representation of the keyexpr as a null terminated string.

Returns The *z_keyexpr_t* corresponding to the given string.

char ***z_keyexpr_to_string**(*z_keyexpr_t* keyexpr)

Get null-terminated string departing from a *z_keyexpr_t*.

If given keyexpr contains a declared keyexpr, the resulting value will be `NULL`. In that case, the user must use *zp_keyexpr_resolve()* to resolve the nesting declarations and get its full expanded representation.

Parameters

- **name** – Pointer to string representation of the keyexpr as a null terminated string.

Returns The *z_keyexpr_t* corresponding to the given string.

char *zp_keyexpr_resolve(*z_session_t* zs, *z_keyexpr_t* keyexpr)

Constructs a null-terminated string departing from a *z_keyexpr_t* for a given *z_session_t*. The user is responsible of droping the returned string using z_free.

Parameters

- **zs** – A loaned instance of the the *z_session_t* to resolve the keyexpr.
- **keyexpr** – A loaned instance of *z_keyexpr_t* to be resolved.

Returns The string representation of a keyexpr for a given session.

_Bool z_keyexpr_is_initialized(*z_keyexpr_t* *keyexpr)

Checks if a given keyexpr is valid.

Parameters

- **keyexpr** – A loaned instance of *z_keyexpr_t* to be checked.

Returns Returns true if the keyexpr is valid, or false otherwise.

int8_t z_keyexpr_is_canon(const char *start, size_t len)

Check if a given keyexpr is valid and in its canonical form.

Parameters

- **start** – Pointer to the keyexpr in its string representation as a non-null terminated string.
- **len** – Number of characters in start.

Returns Returns 0 if the passed string is a valid (and canon) key expression, or a negative value otherwise. Error codes are defined in *zp_keyexpr_canon_status_t*.

int8_t zp_keyexpr_is_canon_null_terminated(const char *start)

Check if a given keyexpr is valid and in its canonical form.

Parameters

- **start** – Pointer to the keyexpr in its string representation as a null terminated string.
- **len** – Number of characters in start.

Returns Returns 0 if the passed string is a valid (and canon) key expression, or a negative value otherwise. Error codes are defined in *zp_keyexpr_canon_status_t*.

int8_t z_keyexpr_canonize(char *start, size_t *len)

Canonization of a given keyexpr in its its string representation. The canonization is performed over the passed string, possibly shortening it by modifying len.

Parameters

- **start** – Pointer to the keyexpr in its string representation as a non-null terminated string.
- **len** – Number of characters in start.

Returns Returns 0 if the canonization is successful, or a negative value otherwise. Error codes are defined in *zp_keyexpr_canon_status_t*.

int8_t zp_keyexpr_canonize_null_terminated(char *start)

Canonization of a given keyexpr in its its string representation. The canonization is performed over the passed string, possibly shortening it by modifying len.

Parameters

- **start** – Pointer to the keyexpr in its string representation as a null terminated string.

- **len** – Number of characters in `start`.

Returns Returns `0` if the canonization is successful, or a `negative value` otherwise. Error codes are defined in *zp_keyexpr_canon_status_t*.

int8_t **z_keyexpr_includes**(*z_keyexpr_t* l, *z_keyexpr_t* r)

Check if a given keyexpr contains another keyexpr in its set.

Parameters

- **l** – The first keyexpr.
- **r** – The second keyexpr.

Returns Returns `0` if `l` includes `r`, i.e. the set defined by `l` contains every key belonging to the set defined by `r`. Otherwise, it returns a `negative value`.

_Bool **zp_keyexpr_includes_null_terminated**(const char *l, const char *r)

Check if a given keyexpr contains another keyexpr in its set.

Parameters

- **l** – Pointer to the keyexpr in its string representation as a null terminated string.
- **llen** – Number of characters in `l`.
- **r** – Pointer to the keyexpr in its string representation as a null terminated string.
- **rlen** – Number of characters in `r`.

Returns Returns true if `l` includes `r`, i.e. the set defined by `l` contains every key belonging to the set defined by `r`.

int8_t **z_keyexpr_intersects**(*z_keyexpr_t* l, *z_keyexpr_t* r)

Check if a given keyexpr intersects with another keyexpr.

Parameters

- **l** – The first keyexpr.
- **r** – The second keyexpr.

Returns Returns `0` if the keyexprs intersect, i.e. there exists at least one key which is contained in both of the sets defined by `l` and `r`. Otherwise, it returns `negative value`.

_Bool **zp_keyexpr_intersect_null_terminated**(const char *l, const char *r)

Check if a given keyexpr intersects with another keyexpr.

Parameters

- **l** – Pointer to the keyexpr in its string representation as a null terminated string.
- **llen** – Number of characters in `l`.
- **r** – Pointer to the keyexpr in its string representation as a null terminated string.
- **rlen** – Number of characters in `r`.

Returns Returns `true` if the keyexprs intersect, i.e. there exists at least one key which is contained in both of the sets defined by `l` and `r`. Otherwise, it returns `false`.

int8_t **z_keyexpr_equals**(*z_keyexpr_t* l, *z_keyexpr_t* r)

Check if a two keyexprs are equal.

Parameters

- **l** – The first keyexpr.
- **r** – The second keyexpr.

Returns Returns `0` if both `l` and `r` are equal, or `negative value` otherwise.

_Bool **zp_keyexpr_equals_null_terminated**(const char *l, const char *r)

Check if a two keyexprs are equal.

Parameters

- **l** – Pointer to the keyexpr in its string representation as a null terminated string.
- **llen** – Number of characters in `l`.
- **r** – Pointer to the keyexpr in its string representation as a null terminated string.
- **rlen** – Number of characters in `r`.

Returns Returns `true` if both `l` and `r` are equal, or `false` otherwise.

z_owned_config_t **z_config_new**(void)

Return a new, zenoh-allocated, empty configuration. It consists in an empty set of properties for zenoh session configuration.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_config_t* by loaning it using `z_config_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_config_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_config_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Returns Returns a new, zenoh-allocated, empty configuration.

z_owned_config_t **z_config_default**(void)

Return a new, zenoh-allocated, default configuration. It consists in a default set of properties for zenoh session configuration.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_config_t* by loaning it using `z_config_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_config_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_config_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Returns Returns a new, zenoh-allocated, default configuration.

const char *zp_config_get(*z_config_t* config, unsigned int key)

Gets the property with the given integer key from the configuration.

Parameters

- **config** – A loaned instance of *z_owned_config_t*.
- **key** – Integer key for the requested property.

Returns Returns the property with the given integer key from the configuration.

int8_t zp_config_insert(*z_config_t* config, unsigned int key, *z_string_t* value)

Inserts or replaces the property with the given integer key in the configuration.

Parameters

- **config** – A loaned instance of *z_owned_config_t*.
- **key** – Integer key for the property to be inserted.
- **value** – Property value to be inserted.

Returns Returns `0` if the insertion is successful, or a `negative value` otherwise.

z_owned_scouting_config_t z_scouting_config_default(void)

Return a new, zenoh-allocated, default scouting configuration. It consists in a default set of properties for scouting configuration.

Like most `z_owned_X_t` types, you may obtain an instance of `z_owned_scouting_config_t` by loaning it using `z_scouting_config_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_config_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_scouting_config_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Returns Returns a new, zenoh-allocated, default scouting configuration.

z_owned_scouting_config_t z_scouting_config_from(*z_config_t* config)

Return a new, zenoh-allocated, scouting configuration extracted from a *z_owned_config_t*. It consists in a default set of properties for scouting configuration.

Like most `z_owned_X_t` types, you may obtain an instance of `z_owned_scouting_config_t` by loaning it using `z_scouting_config_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_config_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_scouting_config_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **config** – A loaned instance of *z_owned_config_t*.

Returns Returns a new, zenoh-allocated, default scouting configuration.

const char ***zp_scouting_config_get**(z_scouting_config_t config, unsigned int key)

Gets the property with the given integer key from the configuration.

Parameters

- **config** – A loaned instance of `z_owned_scouting_config_t`.
- **key** – Integer key for the requested property.

Returns Returns the property with the given integer key from the configuration.

int8_t **zp_scouting_config_insert**(z_scouting_config_t config, unsigned int key, *z_string_t* value)

Inserts or replaces the property with the given integer key in the configuration.

Parameters

- **config** – A loaned instance of `z_owned_scouting_config_t`.
- **key** – Integer key for the property to be inserted.
- **value** – Property value to be inserted.

Returns Returns `0` if the insertion is successful, or a `negative value` otherwise.

z_encoding_t **z_encoding_default**(void)

Constructs a default encoding.

Returns Returns the constructed *z_encoding_t*.

z_query_target_t **z_query_target_default**(void)

Constructs a default query target.

Returns Returns the constructed *z_query_target_t*.

z_query_consolidation_t **z_query_consolidation_auto**(void)

Automatic query consolidation strategy selection.

A query consolidation strategy will automatically be selected depending the query selector. If the selector contains time range properties, no consolidation is performed. Otherwise the *z_query_consolidation_latest()* strategy is used.

Returns Returns the constructed *z_query_consolidation_t*.

z_query_consolidation_t **z_query_consolidation_default**(void)

Constructs a default *z_query_consolidation_t*.

Returns Returns the constructed *z_query_consolidation_t*.

z_query_consolidation_t **z_query_consolidation_latest**(void)

Latest consolidation.

This strategy optimizes bandwidth on all links in the system but will provide a very poor latency.

Returns Returns the constructed *z_query_consolidation_t*.

z_query_consolidation_t **z_query_consolidation_monotonic**(void)

Monotonic consolidation.

This strategy offers the best latency. Replies are directly transmitted to the application when received without needing to wait for all replies. This mode does not garantee that there will be no duplicates.

Returns Returns the constructed *z_query_consolidation_t*.

z_query_consolidation_t **z_query_consolidation_none**(void)

No consolidation.

This strategy is usefull when querying timeseries data bases or when using quorums.

Returns Returns the constructed *z_query_consolidation_t*.

z_bytes_t **z_query_parameters**(const z_query_t *query)

Get a query's value selector by aliasing it.

Parameters

- **query** – Pointer to the query to get the value selector from.

Returns Returns the value selector wrapped as a *z_bytes_t*, since value selector is a user-defined representation.

z_keyexpr_t **z_query_keyexpr**(const z_query_t *query)

Get a query's key by aliasing it.

Parameters

- **query** – Pointer to the query to get keyexpr from.

Returns Returns the *z_keyexpr_t* associated to the query.

z_owned_closure_sample_t **z_closure_sample**(_z_data_handler_t call, _z_dropper_handler_t drop, void *context)

Return a new sample closure. It consists on a structure that contains all the elements for stateful, memory-leak-free callbacks.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_closure_sample_t* by loaning it using `z_closure_sample_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_closure_sample_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_closure_sample_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **call** – the typical callback function. `context` will be passed as its last argument.
- **drop** – allows the callback's state to be freed. `context` will be passed as its last argument.
- **context** – a pointer to an arbitrary state.

Returns Returns a new sample closure.

z_owned_closure_query_t **z_closure_query**(_z_questionable_handler_t call, _z_dropper_handler_t drop, void *context)

Return a new query closure. It consists on a structure that contains all the elements for stateful, memory-leak-free callbacks.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_closure_query_t* by loaning it using `z_closure_query_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_closure_query_loan(&val)`.

Like all z_owned_X_t, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using z_move(val) instead of &val as the argument. After a z_move, val will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your val is valid.

To check if val is still valid, you may use z_closure_query_check(&val) or z_check(val) if your compiler supports _Generic, which will return true if val is valid, or false otherwise.

Parameters

- **call** – the typical callback function. context will be passed as its last argument.
- **drop** – allows the callback's state to be freed. context will be passed as its last argument.
- **context** – a pointer to an arbitrary state.

Returns Returns a new query closure.

z_owned_closure_reply_t **z_closure_reply**(z_owned_reply_handler_t call, _z_dropper_handler_t drop, void *context)

Return a new reply closure. It consists on a structure that contains all the elements for stateful, memory-leak-free callbacks.

Like most z_owned_X_t types, you may obtain an instance of *z_owned_closure_reply_t* by loaning it using z_closure_reply_loan(&val). The z_loan(val) macro, available if your compiler supports C11's _Generic, is equivalent to writing z_closure_reply_loan(&val).

Like all z_owned_X_t, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using z_move(val) instead of &val as the argument. After a z_move, val will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your val is valid.

To check if val is still valid, you may use z_closure_reply_check(&val) or z_check(val) if your compiler supports _Generic, which will return true if val is valid, or false otherwise.

Parameters

- **call** – the typical callback function. context will be passed as its last argument.
- **drop** – allows the callback's state to be freed. context will be passed as its last argument.
- **context** – a pointer to an arbitrary state.

Returns Returns a new reply closure.

z_owned_closure_hello_t **z_closure_hello**(z_owned_hello_handler_t call, _z_dropper_handler_t drop, void *context)

Return a new hello closure. It consists on a structure that contains all the elements for stateful, memory-leak-free callbacks.

Like most z_owned_X_t types, you may obtain an instance of *z_owned_closure_hello_t* by loaning it using z_closure_hello_loan(&val). The z_loan(val) macro, available if your compiler supports C11's _Generic, is equivalent to writing z_closure_hello_loan(&val).

Like all z_owned_X_t, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using z_move(val) instead of &val as the argument. After a z_move, val will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your val is valid.

To check if `val` is still valid, you may use `z_closure_hello_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **call** – the typical callback function. `context` will be passed as its last argument.
- **drop** – allows the callback's state to be freed. `context` will be passed as its last argument.
- **context** – a pointer to an arbitrary state.

Returns Returns a new hello closure.

z_owned_closure_zid_t **z_closure_zid**(z_id_handler_t call, _z_dropper_handler_t drop, void *context)

Return a new zid closure. It consists on a structure that contains all the elements for stateful, memory-leak-free callbacks.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_closure_zid_t* by loaning it using `z_closure_zid_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_closure_zid_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_closure_zid_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **call** – the typical callback function. `context` will be passed as its last argument.
- **drop** – allows the callback's state to be freed. `context` will be passed as its last argument.
- **context** – a pointer to an arbitrary state.

Returns Returns a new zid closure.

z_owned_hello_t **z_hello_null**(void)

Constructs a gravestone value for hello, useful to steal one from a callback. This is useful when you wish to take ownership of a value from a callback to *z_scout()*:

- copy the value of the callback's argument's pointee,
- overwrite the pointee with this function's return value,
- you are now responsible for dropping your copy of the hello.

int8_t **z_scout**(z_owned_scouting_config_t *config, *z_owned_closure_hello_t* *callback)

Primitives

Looks for other Zenoh-enabled entities like routers and/or peers.

Parameters

- **config** – A moved instance of `z_owned_scouting_config_t` containing the set properties to configure the

scouting. callback: A moved instance of *z_owned_closure_hello_t* containg the callbacks to be called.

Returns Returns `0` if the scouting is successful triggered, or a `negative value` otherwise.

z_owned_session_t **z_open**(*z_owned_config_t* *config)

Opens a Zenoh session.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_session_t* by loaning it using `z_session_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_session_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_session_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **config** – A moved instance of *z_owned_config_t* containing the set properties to configure the session.

Returns A *z_owned_session_t* with either a valid open session or a failing session. Should the session opening fail, `z_check(val)` ing the returned value will return `false`.

int8_t **z_close**(*z_owned_session_t* *zs)

Closes a Zenoh session.

Parameters

- **zs** – A moved instance of the the *z_owned_session_t* to close.

Returns Returns `0` if the session is successful closed, or a `negative value` otherwise.

int8_t **z_info_peers_zid**(const *z_session_t* zs, *z_owned_closure_zid_t* *callback)

Fetches the Zenoh IDs of all connected peers.

callback will be called once for each ID. It is guaranteed to never be called concurrently, and to be dropped before this function exits.

Parameters

- **zs** – A loaned instance of the the *z_session_t* to inquiry.
- **callback** – A moved instance of *z_owned_closure_zid_t* containg the callbacks to be called.

Returns Returns `0` if the info is successful triggered, or a `negative value` otherwise.

int8_t **z_info_routers_zid**(const *z_session_t* zs, *z_owned_closure_zid_t* *callback)

Fetches the Zenoh IDs of all connected routers.

callback will be called once for each ID. It is guaranteed to never be called concurrently, and to be dropped before this function exits.

Parameters

- **zs** – A loaned instance of the the *z_session_t* to inquiry.
- **callback** – A moved instance of *z_owned_closure_zid_t* containg the callbacks to be called.

Returns Returns `0` if the info is successful triggered, or a `negative value` otherwise.

z_id_t **z_info_zid**(const *z_session_t* zs)

Get the local Zenoh ID associated to a given Zenoh session.

Unless the *z_session_t* is invalid, that ID is guaranteed to be non-zero. In other words, this function returning an array of 16 zeros means you failed to pass it a valid session.

Parameters

- **zs** – A loaned instance of the the *z_session_t* to inquiry.

Returns Returns the local Zenoh ID of the given *z_session_t*.

z_put_options_t **z_put_options_default**(void)

Constructs the default values for the put operation.

Returns Returns the constructed *z_put_options_t*.

z_delete_options_t **z_delete_options_default**(void)

Constructs the default values for the delete operation.

Returns Returns the constructed *z_delete_options_t*.

int8_t **z_put**(*z_session_t* zs, *z_keyexpr_t* keyexpr, const uint8_t *payload, *z_zint_t* payload_len, const *z_put_options_t* *options)

Puts data for a given keyexpr.

Parameters

- **zs** – A loaned instance of the the *z_session_t* through where data will be put.
- **keyexpr** – A loaned instance of *z_keyexpr_t* to put.
- **payload** – Pointer to the data to put.
- **payload_len** – The length of the `payload`.
- **options** – The put options to be applied in the put operation.

Returns Returns `0` if the put operation is successful, or a `negative value` otherwise.

int8_t **z_delete**(*z_session_t* zs, *z_keyexpr_t* keyexpr, const *z_delete_options_t* *options)

Deletes data from a given keyexpr.

Parameters

- **zs** – A loaned instance of the the *z_session_t* through where data will be put.
- **keyexpr** – A loaned instance of *z_keyexpr_t* to put.
- **options** – The delete options to be applied in the delete operation.

Returns Returns `0` if the delete operation is successful, or a `negative value` otherwise.

z_get_options_t **z_get_options_default**(void)

Constructs the default values for the get operation.

Returns Returns the constructed *z_get_options_t*.

int8_t **z_get**(*z_session_t* zs, *z_keyexpr_t* keyexpr, const char *parameters, *z_owned_closure_reply_t* *callback, const *z_get_options_t* *options)

Issues a distributed query for a given keyexpr.

Parameters

- **zs** – A loaned instance of the the *z_session_t* through where data will be put.

- **keyexpr** – A loaned instance of *z_keyexpr_t* to put.
- **parameters** – Pointer to the parameters as a null-terminated string.
- **callback** – A moved instance of *z_owned_closure_reply_t* containg the callbacks to be called.
- **options** – The get options to be aplied in the distributed query.

Returns Returns `0` if the put operation is successful, or a `negative value` otherwise.

z_owned_keyexpr_t **z_declare_keyexpr**(*z_session_t* zs, *z_keyexpr_t* keyexpr)

Declares a keyexpr, so that it is internally mapped into into a numerical id.

This numerical id is used on the network to save bandwidth and ease the retrieval of the concerned resource in the routing tables.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_keyexpr_t* by loaning it using `z_keyexpr_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_keyexpr_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_keyexpr_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where to declare the keyexpr.
- **keyexpr** – A loaned instance of *z_keyexpr_t* to declare.

Returns A *z_owned_keyexpr_t* with either a valid or invalid keyexpr. Should the keyexpr be invalid, `z_check(val)` ing the returned value will return `false`.

int8_t **z_undeclare_keyexpr**(*z_session_t* zs, *z_owned_keyexpr_t* *keyexpr)

Undeclares the keyexpr generated by a call to *z_declare_keyexpr()*.

Parameters

- **zs** – A loaned instance of the the *z_session_t* through where data will be put.
- **keyexpr** – A moved instance of *z_owned_keyexpr_t* to undeclare.

Returns Returns `0` if the undeclare keyexpr operation is successful, or a `negative value` otherwise.

z_publisher_options_t **z_publisher_options_default**(void)

Constructs the default values for the publisher entity.

Returns Returns the constructed *z_publisher_options_t*.

z_owned_publisher_t **z_declare_publisher**(*z_session_t* zs, *z_keyexpr_t* keyexpr, *z_publisher_options_t* *options)

Declares a publisher for the given keyexpr.

Data can be put and deleted with this publisher with the help of the *z_publisher_put()* and *z_publisher_delete()* functions.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_publisher_t* by loaning it using `z_publisher_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_publisher_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_publisher_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where to declare the publisher.
- **keyexpr** – A loaned instance of *z_keyexpr_t* to associate with the publisher.
- **options** – The options to apply to the publisher. If `NULL` is passed, the default options will be applied.

Returns A *z_owned_publisher_t* with either a valid publisher or a failing publisher. Should the publisher be invalid, `z_check(val)` ing the returned value will return `false`.

int8_t **z_undeclare_publisher**(*z_owned_publisher_t* *pub)

Undeclares the publisher generated by a call to *z_declare_publisher()*.

Parameters

- **pub** – A moved instance of *z_owned_publisher_t* to undeclare.

Returns Returns `0` if the undeclare publisher operation is successful, or a `negative value` otherwise.

z_publisher_put_options_t **z_publisher_put_options_default**(void)

Constructs the default values for the put operation via a publisher entity.

Returns Returns the constructed *z_publisher_put_options_t*.

z_publisher_delete_options_t **z_publisher_delete_options_default**(void)

Constructs the default values for the delete operation via a publisher entity.

Returns Returns the constructed *z_publisher_delete_options_t*.

int8_t **z_publisher_put**(const *z_publisher_t* pub, const uint8_t *payload, size_t len, const *z_publisher_put_options_t* *options)

Puts data for the keyexpr associated to the given publisher.

Parameters

- **pub** – A loaned instance of *z_publisher_t* from where to put the data.
- **options** – The options to apply to the put operation. If `NULL` is passed, the default options will be applied.

Returns Returns `0` if the put operation is successful, or a `negative value` otherwise.

int8_t **z_publisher_delete**(const *z_publisher_t* pub, const *z_publisher_delete_options_t* *options)

Deletes data from the keyexpr associated to the given publisher.

Parameters

- **pub** – A loaned instance of *z_publisher_t* from where to delete the data.
- **options** – The options to apply to the delete operation. If `NULL` is passed, the default options will be applied.

Returns Returns `0` if the delete operation is successful, or a `negative value` otherwise.

z_subscriber_options_t **z_subscriber_options_default**(void)

Constructs the default values for the subscriber entity.

Returns Returns the constructed *z_subscriber_options_t*.

z_owned_subscriber_t **z_declare_subscriber**(*z_session_t* zs, *z_keyexpr_t* keyexpr, *z_owned_closure_sample_t* *callback, const *z_subscriber_options_t* *options)

Declares a (push) subscriber for the given keyexpr.

Received data is processed by means of callbacks.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_subscriber_t* by loaning it using `z_subscriber_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_subscriber_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_subscriber_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where to declare the subscriber.
- **keyexpr** – A loaned instance of *z_keyexpr_t* to associate with the subscriber.
- **callback** – A moved instance of *z_owned_closure_sample_t* containg the callbacks to be called and the

context to pass to them. options: The options to apply to the subscriber. If `NULL` is passed, the default options will be applied.

Returns A *z_owned_subscriber_t* with either a valid subscriber or a failing subscriber. Should the subscriber be invalid, `z_check(val)` ing the returned value will return `false`.

int8_t **z_undeclare_subscriber**(*z_owned_subscriber_t* *sub)

Undeclares the (push) subscriber generated by a call to *z_declare_subscriber()*.

Parameters

- **sub** – A moved instance of *z_owned_subscriber_t* to undeclare.

Returns Returns `0` if the undeclare (push) subscriber operation is successful, or a `negative value` otherwise.

z_pull_subscriber_options_t **z_pull_subscriber_options_default**(void)

Constructs the default values for the pull subscriber entity.

Returns Returns the constructed *z_pull_subscriber_options_t*.

z_owned_pull_subscriber_t **z_declare_pull_subscriber**(*z_session_t* zs, *z_keyexpr_t* keyexpr, *z_owned_closure_sample_t* *callback, const *z_pull_subscriber_options_t* *options)

Declares a pull subscriber for the given keyexpr.

Data can be pulled with this subscriber with the help of the `z_pull()` function. Received data is processed by means of callbacks.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_pull_subscriber_t* by loaning it using `z_pull_subscriber_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_pull_subscriber_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_pull_subscriber_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where to declare the subscriber.
- **keyexpr** – A loaned instance of *z_keyexpr_t* to associate with the subscriber.
- **callback** – A moved instance of *z_owned_closure_sample_t* containg the callbacks to be called and the

context to pass to them. options: The options to apply to the pull subscriber. If `NULL` is passed, the default options will be applied.

Returns A *z_owned_pull_subscriber_t* with either a valid subscriber or a failing subscriber. Should the pull subscriber be invalid, `z_check(val)` ing the returned value will return `false`.

int8_t **z_undeclare_pull_subscriber**(*z_owned_pull_subscriber_t* *sub)

Undeclares the pull subscriber generated by a call to *z_declare_pull_subscriber()*.

Parameters

- **sub** – A moved instance of *z_owned_pull_subscriber_t* to undeclare.

Returns Returns `0` if the undeclare pull subscriber operation is successful, or a `negative value` otherwise.

int8_t **z_subscriber_pull**(const *z_pull_subscriber_t* sub)

Pulls data for *z_owned_pull_subscriber_t*. The pulled data will be provided by calling the **callback** function provided to the *z_declare_pull_subscriber()* function.

Parameters

- **sub** – A loaned instance of *z_pull_subscriber_t* from where to pull the data.

Returns Returns `0` if the pull operation is successful, or a `negative value` otherwise.

z_queryable_options_t **z_queryable_options_default**(void)

Constructs the default values for the queryable entity.

Returns Returns the constructed *z_queryable_options_t*.

z_owned_queryable_t **z_declare_queryable**(*z_session_t* zs, *z_keyexpr_t* keyexpr, *z_owned_closure_query_t* *callback, const *z_queryable_options_t* *options)

Declares a queryable for the given keyexpr.

Received queries are processed by means of callbacks.

Like most `z_owned_X_t` types, you may obtain an instance of *z_owned_queryable_t* by loaning it using `z_queryable_loan(&val)`. The `z_loan(val)` macro, available if your compiler supports C11's `_Generic`, is equivalent to writing `z_queryable_loan(&val)`.

Like all `z_owned_X_t`, an instance will be destroyed by any function which takes a mutable pointer to said instance, as this implies the instance's inners were moved. To make this fact more obvious when reading your code, consider using `z_move(val)` instead of `&val` as the argument. After a `z_move`, `val` will still exist, but will no longer be valid. The destructors are double-drop-safe, but other functions will still trust that your `val` is valid.

To check if `val` is still valid, you may use `z_queryable_check(&val)` or `z_check(val)` if your compiler supports `_Generic`, which will return `true` if `val` is valid, or `false` otherwise.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where to declare the subscriber.
- **keyexpr** – A loaned instance of *z_keyexpr_t* to associate with the subscriber.
- **callback** – A moved instance of *z_owned_closure_query_t* containg the callbacks to be called and the context

to pass to them. options: The options to apply to the queryable. If `NULL` is passed, the default options will be applied.

Returns A *z_owned_queryable_t* with either a valid queryable or a failing queryable. Should the queryable be invalid, `z_check(val)` ing the returned value will return `false`.

int8_t **z_undeclare_queryable**(*z_owned_queryable_t* *queryable)

Undeclares the queryable generated by a call to *z_declare_queryable()*.

Parameters

- **queryable** – A moved instance of *z_owned_queryable_t* to undeclare.

Returns Returns `0` if the undeclare queryable operation is successful, or a `negative value` otherwise.

int8_t **z_query_reply**(const z_query_t *query, const *z_keyexpr_t* keyexpr, const uint8_t *payload, size_t payload_len, const *z_query_reply_options_t* *options)

Sends a reply to a query.

This function must be called inside of a *z_owned_closure_query_t* callback associated to the *z_owned_queryable_t*, passing the received query as parameters of the callback function. This function can be called multiple times to send multiple replies to a query. The reply will be considered complete when the callback returns.

Parameters

- **query** – Pointer to the received query.
- **keyexpr** – A loaned instance of *z_keyexpr_t* to associate with the subscriber.
- **payload** – Pointer to the data to put.
- **payload_len** – The length of the `payload`.
- **options** – The options to apply to the send query reply operation. If `NULL` is passed, the default options will be

applied.

Returns Returns `0` if the send query reply operation is successful, or a `negative value` otherwise.

z_owned_reply_t **z_reply_null**(void)

Returns an invalidated *z_owned_reply_t*.

This is useful when you wish to take ownership of a value from a callback to *z_get()*:

- copy the value of the callback's argument's pointee,
- overwrite the pointee with this function's return value,
- you are now responsible for dropping your copy of the reply.

_Bool **z_reply_is_ok**(const *z_owned_reply_t* *reply)

Checks if the queryable answered with an OK, which allows this value to be treated as a sample.

If this returns `false`, you should use `z_check` before trying to use *z_reply_err()* if you want to process the error that may be here.

Parameters

- **reply** – Pointer to the received query reply.

Returns Returns `true` if the queryable answered with an OK, which allows this value to be treated as a sample, or

`false` otherwise.

z_sample_t **z_reply_ok**(*z_owned_reply_t* *reply)

Yields the contents of the reply by asserting it indicates a success.

You should always make sure that *z_reply_is_ok()* returns `true` before calling this function.

Parameters

- **reply** – Pointer to the received query reply.

Returns Returns the *z_sample_t* wrapped in the query reply.

z_value_t **z_reply_err**(const *z_owned_reply_t* *reply)

Yields the contents of the reply by asserting it indicates a failure.

You should always make sure that *z_reply_is_ok()* returns `false` before calling this function.

Parameters

- **reply** – Pointer to the received query reply.

Returns Returns the *z_value_t* wrapped in the query reply.

zp_task_read_options_t **zp_task_read_options_default**(void)

Multi Thread Taks helpers

Constructs the default values for the session read task.

Returns Returns the constructed *zp_task_read_options_t*.

int8_t **zp_start_read_task**(*z_session_t* zs, const *zp_task_read_options_t* *options)

Start a separate task to read from the network and process the messages as soon as they are received.

Note that the task can be implemented in form of thread, process, etc. and its implementation is platform-dependent.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where to start the read task.
- **options** – The options to apply when starting the read task. If `NULL` is passed, the default options will be

applied.

Returns Returns `0` if the read task started successfully, or a `negative value` otherwise.

int8_t **zp_stop_read_task**(*z_session_t* zs)

Stop the read task.

This may result in stopping a thread or a process depending on the target platform.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where to stop the read task.

Returns Returns `0` if the read task stopped successfully, or a `negative value` otherwise.

zp_task_lease_options_t **zp_task_lease_options_default**(void)

Constructs the default values for the session lease task.

Returns Returns the constructed *zp_task_lease_options_t*.

int8_t **zp_start_lease_task**(*z_session_t* zs, const *zp_task_lease_options_t* *options)

Start a separate task to handle the session lease.

This task will send `KeepAlive` messages when needed and will close the session when the lease is expired. When operating over a multicast transport, it also periodically sends the `Join` messages. Note that the task can be implemented in form of thread, process, etc. and its implementation is platform-dependent.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where to start the lease task.
- **options** – The options to apply when starting the lease task. If `NULL` is passed, the default options will be

applied.

Returns Returns `0` if the lease task started successfully, or a `negative value` otherwise.

int8_t **zp_stop_lease_task**(*z_session_t* zs)

Stop the lease task.

This may result in stopping a thread or a process depending on the target platform.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where to stop the lease task.

Returns Returns `0` if the lease task stopped successfully, or a `negative value` otherwise.

zp_read_options_t **zp_read_options_default**(void)

Single Thread helpers

Constructs the default values for the reading procedure.

Returns Returns the constructed *zp_read_options_t*.

int8_t **zp_read**(*z_session_t* zs, const *zp_read_options_t* *options)

Triggers a single execution of reading procedure from the network and processes of any received the message.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where trigger the reading procedure.
- **options** – The options to apply to the read. If `NULL` is passed, the default options will be

applied.

Returns Returns `0` if the reading procedure was executed successfully, or a `negative value` otherwise.

zp_send_keep_alive_options_t **zp_send_keep_alive_options_default**(void)

Constructs the default values for sending the keep alive.

Returns Returns the constructed *zp_send_keep_alive_options_t*.

int8_t **zp_send_keep_alive**(*z_session_t* zs, const *zp_send_keep_alive_options_t* *options)

Triggers a single execution of keep alive procedure.

It will send `KeepAlive` messages when needed and will close the session when the lease is expired.

Parameters

- **zs** – A loaned instance of the the *z_session_t* where trigger the leasing procedure.
- **options** – The options to apply to the send of a `KeepAlive` messages. If `NULL` is passed, the default options

will be applied.

Returns Returns `0` if the leasing procedure was executed successfully, or a `negative value` otherwise.

INDEX

Z