

---

# **zenoh-pico**

***Release 0.7.2.1***

**ZettaScale Zenoh team**

**Jun 07, 2023**



CONTENTS

1 API Reference 3

1.1 Zenoh Types 3

1.1.1 Enums 3

1.1.2 Data Structures 6

1.1.3 Arrays 11

1.1.4 Owned Types 11

1.1.5 Closures 12

1.2 Zenoh Functions 13

1.2.1 Macros 13

1.2.2 Primitives 14

Index 33



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



## 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\_URL\_ENCODED**

**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  $t_1$  and  $t_2$  will have timestamp  $ts_2 > ts_1$ . 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.

typedef int **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.

typedef int **z\_value\_t**

Represents a Zenoh value.

`z_encoding_t` **encoding**

The encoding of the *payload*.

`z_bytes_t` **payload**

The payload of this zenoh value.

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.

`z_value_t` **value**

The payload to include in the query.

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.

uint8\_t **whatami**

The kind of zenoh entity.

`z_bytes_t` **zid**

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);`
- `_Bool 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\_str\_t**

A zenoh-allocated `char *`.

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.

*\_z\_data\_handler\_t* **call**

*void \*call(const struct z\_sample\_t\*, const void \*context)* is the callback function.

*\_z\_droppper\_handler\_t* **drop**

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

*void \****context**

a pointer to an arbitrary state.

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.

*\_z\_questionable\_handler\_t* **call**

*void (\*\_z\_questionable\_handler\_t)(z\_query\_t \*query, void \*arg)* is the callback

**function.** *\_z\_droppper\_handler\_t drop: void \*drop(void\*)* allows the callback's state to be freed. *void \****context:** a pointer to an arbitrary state.



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.

**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. *void \*context:* a pointer to an arbitrary state.

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.

**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. *void \*context:* a pointer to an arbitrary state.

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.

**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.

*void \*context*

a pointer to an arbitrary state.

## 1.2 Zenoh Functions

### 1.2.1 Macros

**z\_loan(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)**

**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 dropping 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.

**z\_null(x)**

Defines a generic function for making null object of any of the `z_owned_X_t` types.

**Returns** Returns the uninitialized instance of *x*.

## 1.2.2 Primitives

**z\_keyexpr\_t z\_keyexpr(const char \*name)**

Constructs a `z_keyexpr_t` departing from a string. It is a loaned key expression that aliases `name`. Unlike it's counterpart in zenoh-c, this function does not test passed expression to correctness.

**Parameters**

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

**Returns** The `z_keyexpr_t` corresponding to the given string.

**z\_owned\_str\_t 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 owned string will be uninitialized. In that case, the user must use `zp_keyexpr_resolve()` to resolve the nesting declarations and get its full expanded representation.

**Parameters**

- **keyexpr** – A loaned instance of `z_keyexpr_t`

**Returns** The `z_owned_str_t` containing key expression string representation if it's possible

**z\_owned\_str\_t 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 dropping the returned string using `z_free`.

**Parameters**

- **zs** – A loaned instance of 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(const 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 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 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 -1, or other **negative** value for errors.

`int8_t 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 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 -1, or other **negative** value for errors.

`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 a -1, or other **negative** value for errors.

`int8_t 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 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 a -1, or other **negative** value for errors.

`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. Otherwise, it returns a -1, or other negative value for

errors.

int8\_t **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 0 if both l and r are equal. Otherwise, it returns a -1, or other negative value for

errors.

*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, uint8\_t 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, uint8\_t 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, uint8\_t 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, uint8\_t 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 guarantee 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 useful 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_value_t z_query_value(const z_query_t *query)`

Get a query's payload value by aliasing it. Note: This API has been marked as unstable: it works as advertised, but we may change it in a future release.

**Parameters**

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

**Returns** Returns the payload value wrapped as a `z_value_t`, since payload value is a user-defined representation.

`_Bool z_value_is_initialized(z_value_t *value)`

Checks if a given value is valid.

**Parameters**

- **value** – A loaned instance of `z_value_t` to be checked.

**Returns** Returns true if the value is valid, or false otherwise.

`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.

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` containing 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)` in 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` containing 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` containing 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` containing the callbacks to be called.
- **options** – The get options to be applied 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, const 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 `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 `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` containing 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)` in 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 `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` containing 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 `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` containing 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.

`_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(const 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 `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 *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

- `z_bytes_t` (*C type*), 6
- `z_bytes_t.len` (*C member*), 6
- `z_bytes_t.start` (*C member*), 6
- `z_check` (*C macro*), 13
- `z_clone` (*C macro*), 13
- `z_close` (*C function*), 23
- `z_closure` (*C macro*), 14
- `z_closure_hello` (*C function*), 22
- `z_closure_query` (*C function*), 21
- `z_closure_reply` (*C function*), 21
- `z_closure_sample` (*C function*), 20
- `z_closure_zid` (*C function*), 22
- `z_config_default` (*C function*), 17
- `z_config_new` (*C function*), 17
- `z_config_t` (*C type*), 7
- `z_congestion_control_t` (*C enum*), 5
- `z_congestion_control_t.Z_CONGESTION_CONTROL_BLOCK` (*C enumerator*), 5
- `z_congestion_control_t.Z_CONGESTION_CONTROL_DROP` (*C enumerator*), 5
- `z_consolidation_mode_t` (*C enum*), 5
- `z_consolidation_mode_t.Z_CONSOLIDATION_MODE_AUTO` (*C enumerator*), 5
- `z_consolidation_mode_t.Z_CONSOLIDATION_MODE_LATEST` (*C enumerator*), 5
- `z_consolidation_mode_t.Z_CONSOLIDATION_MODE_MONOTONIC` (*C enumerator*), 5
- `z_consolidation_mode_t.Z_CONSOLIDATION_MODE_NONE` (*C enumerator*), 5
- `z_declare_keyexpr` (*C function*), 25
- `z_declare_publisher` (*C function*), 25
- `z_declare_pull_subscriber` (*C function*), 28
- `z_declare_queryable` (*C function*), 28
- `z_declare_subscriber` (*C function*), 27
- `z_delete` (*C function*), 24
- `z_delete_options_default` (*C function*), 24
- `z_delete_options_t` (*C type*), 9
- `z_delete_options_t.congestion_control` (*C member*), 9
- `z_delete_options_t.priority` (*C member*), 9
- `z_drop` (*C macro*), 14
- `z_encoding_default` (*C function*), 19
- `z_encoding_prefix_t` (*C enum*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_APP_OCTET_STREAM` (*C enumerator*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_APP_SQL` (*C enumerator*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_APP_X_WWW_FORM_URL_ENCODED` (*C enumerator*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_APP_XHTML_XML` (*C enumerator*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_APP_XML` (*C enumerator*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_EMPTY` (*C enumerator*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_IMAGE_GIF` (*C enumerator*), 5
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_IMAGE_JPEG` (*C enumerator*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_IMAGE_PNG` (*C enumerator*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_TEXT_JAVASCRIPT` (*C enumerator*), 4
- `z_encoding_prefix_t.Z_ENCODING_PREFIX_TEXT_PLAIN` (*C enumerator*), 4
- `z_encoding_t` (*C type*), 8
- `z_encoding_t.prefix` (*C member*), 8
- `z_encoding_t.suffix` (*C member*), 8
- `z_get` (*C function*), 24
- `z_get_options_default` (*C function*), 24
- `z_get_options_t` (*C type*), 9
- `z_get_options_t.consolidation` (*C member*), 10
- `z_get_options_t.target` (*C member*), 9
- `z_get_options_t.value` (*C member*), 10
- `z_hello_t` (*C type*), 10
- `z_hello_t.locators` (*C member*), 10
- `z_hello_t.whatami` (*C member*), 10
- `z_hello_t.zid` (*C member*), 10
- `z_id_t` (*C type*), 6
- `z_id_t.id` (*C member*), 7
- `z_info_peers_zid` (*C function*), 23
- `z_info_routers_zid` (*C function*), 23
- `z_info_zid` (*C function*), 24

z\_keyexpr (*C function*), 14  
 z\_keyexpr\_canonize (*C function*), 15  
 z\_keyexpr\_equals (*C function*), 16  
 z\_keyexpr\_includes (*C function*), 15  
 z\_keyexpr\_intersects (*C function*), 16  
 z\_keyexpr\_is\_canon (*C function*), 15  
 z\_keyexpr\_is\_initialized (*C function*), 15  
 z\_keyexpr\_t (*C type*), 7  
 z\_keyexpr\_to\_string (*C function*), 14  
 z\_loan (*C macro*), 13  
 z\_move (*C macro*), 13  
 z\_null (*C macro*), 14  
 z\_open (*C function*), 23  
 z\_owned\_bytes\_t (*C type*), 11  
 z\_owned\_closure\_hello\_t (*C type*), 13  
 z\_owned\_closure\_hello\_t.call (*C member*), 13  
 z\_owned\_closure\_query\_t (*C type*), 12  
 z\_owned\_closure\_query\_t.call (*C member*), 12  
 z\_owned\_closure\_reply\_t (*C type*), 12  
 z\_owned\_closure\_reply\_t.call (*C member*), 13  
 z\_owned\_closure\_sample\_t (*C type*), 12  
 z\_owned\_closure\_sample\_t.call (*C member*), 12  
 z\_owned\_closure\_sample\_t.context (*C member*), 12  
 z\_owned\_closure\_sample\_t.drop (*C member*), 12  
 z\_owned\_closure\_zid\_t (*C type*), 13  
 z\_owned\_closure\_zid\_t.call (*C member*), 13  
 z\_owned\_closure\_zid\_t.context (*C member*), 13  
 z\_owned\_closure\_zid\_t.drop (*C member*), 13  
 z\_owned\_config\_t (*C type*), 11  
 z\_owned\_keyexpr\_t (*C type*), 11  
 z\_owned\_publisher\_t (*C type*), 11  
 z\_owned\_pull\_subscriber\_t (*C type*), 11  
 z\_owned\_queryable\_t (*C type*), 12  
 z\_owned\_reply\_t (*C type*), 12  
 z\_owned\_session\_t (*C type*), 11  
 z\_owned\_str\_array\_t (*C type*), 12  
 z\_owned\_str\_t (*C type*), 11  
 z\_owned\_subscriber\_t (*C type*), 11  
 z\_priority\_t (*C enum*), 5  
 z\_priority\_t.Z\_PRIORITY\_CONTROL (*C enumerator*), 5  
 z\_priority\_t.Z\_PRIORITY\_BACKGROUND (*C enumerator*), 6  
 z\_priority\_t.Z\_PRIORITY\_DATA (*C enumerator*), 6  
 z\_priority\_t.Z\_PRIORITY\_DATA\_HIGH (*C enumerator*), 6  
 z\_priority\_t.Z\_PRIORITY\_DATA\_LOW (*C enumerator*), 6  
 z\_priority\_t.Z\_PRIORITY\_INTERACTIVE\_HIGH (*C enumerator*), 5  
 z\_priority\_t.Z\_PRIORITY\_INTERACTIVE\_LOW (*C enumerator*), 6  
 z\_priority\_t.Z\_PRIORITY\_REAL\_TIME (*C enumerator*), 5  
 z\_publisher\_delete (*C function*), 27  
 z\_publisher\_delete\_options\_default (*C function*), 26  
 z\_publisher\_delete\_options\_t (*C type*), 9  
 z\_publisher\_options\_default (*C function*), 25  
 z\_publisher\_options\_t (*C type*), 8  
 z\_publisher\_options\_t.congestion\_control (*C member*), 9  
 z\_publisher\_put (*C function*), 26  
 z\_publisher\_put\_options\_default (*C function*), 26  
 z\_publisher\_put\_options\_t (*C type*), 9  
 z\_publisher\_put\_options\_t.encoding (*C member*), 9  
 z\_publisher\_t (*C type*), 8  
 z\_pull\_subscriber\_options\_default (*C function*), 27  
 z\_pull\_subscriber\_options\_t (*C type*), 8  
 z\_pull\_subscriber\_options\_t.reliability (*C member*), 8  
 z\_pull\_subscriber\_t (*C type*), 7  
 z\_put (*C function*), 24  
 z\_put\_options\_default (*C function*), 24  
 z\_put\_options\_t (*C type*), 9  
 z\_put\_options\_t.congestion\_control (*C member*), 9  
 z\_put\_options\_t.encoding (*C member*), 9  
 z\_put\_options\_t.priority (*C member*), 9  
 z\_query\_consolidation\_auto (*C function*), 19  
 z\_query\_consolidation\_default (*C function*), 19  
 z\_query\_consolidation\_latest (*C function*), 19  
 z\_query\_consolidation\_monotonic (*C function*), 19  
 z\_query\_consolidation\_none (*C function*), 19  
 z\_query\_consolidation\_t (*C type*), 8  
 z\_query\_consolidation\_t.mode (*C member*), 8  
 z\_query\_keyexpr (*C function*), 20  
 z\_query\_parameters (*C function*), 20  
 z\_query\_reply (*C function*), 29  
 z\_query\_reply\_options\_t (*C type*), 9  
 z\_query\_reply\_options\_t.encoding (*C member*), 9  
 z\_query\_target\_default (*C function*), 19  
 z\_query\_target\_t (*C enum*), 6  
 z\_query\_target\_t.Z\_QUERY\_TARGET\_ALL (*C enumerator*), 6  
 z\_query\_target\_t.Z\_QUERY\_TARGET\_ALL\_COMPLETE (*C enumerator*), 6  
 z\_query\_target\_t.Z\_QUERY\_TARGET\_BEST\_MATCHING (*C enumerator*), 6  
 z\_query\_value (*C function*), 20  
 z\_queryable\_options\_default (*C function*), 28  
 z\_queryable\_options\_t (*C type*), 9  
 z\_queryable\_options\_t.complete (*C member*), 9  
 z\_queryable\_t (*C type*), 8

z\_reliability\_t (C enum), 5  
 z\_reliability\_t.Z\_RELIABILITY\_BEST\_EFFORT (C enumerator), 5  
 z\_reliability\_t.Z\_RELIABILITY\_RELIABLE (C enumerator), 5  
 z\_reply\_data\_t (C type), 10  
 z\_reply\_data\_t.replier\_id (C member), 10  
 z\_reply\_data\_t.sample (C member), 10  
 z\_reply\_err (C function), 30  
 z\_reply\_is\_ok (C function), 30  
 z\_reply\_ok (C function), 30  
 z\_reply\_t (C type), 10  
 z\_reply\_t.data (C member), 10  
 z\_reply\_tag\_t (C enum), 5  
 z\_reply\_tag\_t.Z\_REPLY\_TAG\_DATA (C enumerator), 5  
 z\_reply\_tag\_t.Z\_REPLY\_TAG\_FINAL (C enumerator), 5  
 z\_sample\_kind\_t (C enum), 4  
 z\_sample\_kind\_t.Z\_SAMPLE\_KIND\_DELETE (C enumerator), 4  
 z\_sample\_kind\_t.Z\_SAMPLE\_KIND\_PUT (C enumerator), 4  
 z\_sample\_t (C type), 10  
 z\_sample\_t.encoding (C member), 10  
 z\_sample\_t.keyexpr (C member), 10  
 z\_sample\_t.kind (C member), 10  
 z\_sample\_t.payload (C member), 10  
 z\_sample\_t.timestamp (C member), 10  
 z\_scout (C function), 22  
 z\_scouting\_config\_default (C function), 18  
 z\_scouting\_config\_from (C function), 18  
 z\_session\_t (C type), 7  
 z\_str\_array\_t (C type), 11  
 z\_string\_t (C type), 7  
 z\_string\_t.len (C member), 7  
 z\_string\_t.val (C member), 7  
 z\_submode\_t (C enum), 6  
 z\_submode\_t.Z\_SUBMODE\_PULL (C enumerator), 6  
 z\_submode\_t.Z\_SUBMODE\_PUSH (C enumerator), 6  
 z\_subscriber\_options\_default (C function), 27  
 z\_subscriber\_options\_t (C type), 8  
 z\_subscriber\_options\_t.reliability (C member), 8  
 z\_subscriber\_pull (C function), 28  
 z\_subscriber\_t (C type), 7  
 z\_undeclare\_keyexpr (C function), 25  
 z\_undeclare\_publisher (C function), 26  
 z\_undeclare\_pull\_subscriber (C function), 28  
 z\_undeclare\_queryable (C function), 29  
 z\_undeclare\_subscriber (C function), 27  
 z\_value\_is\_initialized (C function), 20  
 z\_value\_t (C type), 8  
 z\_value\_t.encoding (C member), 8  
 z\_value\_t.payload (C member), 8  
 z\_whatami\_t (C enum), 3  
 z\_whatami\_t.Z\_WHATAMI\_CLIENT (C enumerator), 3  
 z\_whatami\_t.Z\_WHATAMI\_PEER (C enumerator), 3  
 z\_whatami\_t.Z\_WHATAMI\_ROUTER (C enumerator), 3  
 z\_zint\_t (C type), 6  
 zp\_config\_get (C function), 17  
 zp\_config\_insert (C function), 18  
 zp\_keyexpr\_canon\_status\_t (C enum), 3  
 zp\_keyexpr\_canon\_status\_t.Z\_KEYEXPR\_CANON\_CONTAINS\_SHARP\_C (C enumerator), 3  
 zp\_keyexpr\_canon\_status\_t.Z\_KEYEXPR\_CANON\_CONTAINS\_UNBOUNDED (C enumerator), 4  
 zp\_keyexpr\_canon\_status\_t.Z\_KEYEXPR\_CANON\_DOLLAR\_AFTER\_DOT (C enumerator), 3  
 zp\_keyexpr\_canon\_status\_t.Z\_KEYEXPR\_CANON\_DOUBLE\_STAR\_AFTER\_DOT (C enumerator), 3  
 zp\_keyexpr\_canon\_status\_t.Z\_KEYEXPR\_CANON\_EMPTY\_CHUNK (C enumerator), 3  
 zp\_keyexpr\_canon\_status\_t.Z\_KEYEXPR\_CANON\_LONE\_DOLLAR\_STAR\_AFTER\_DOT (C enumerator), 3  
 zp\_keyexpr\_canon\_status\_t.Z\_KEYEXPR\_CANON\_SINGLE\_STAR\_AFTER\_DOT (C enumerator), 3  
 zp\_keyexpr\_canon\_status\_t.Z\_KEYEXPR\_CANON\_STARS\_IN\_CHUNK (C enumerator), 3  
 zp\_keyexpr\_canon\_status\_t.Z\_KEYEXPR\_CANON\_SUCCESS (C enumerator), 3  
 zp\_keyexpr\_canonize\_null\_terminated (C function), 15  
 zp\_keyexpr\_equals\_null\_terminated (C function), 17  
 zp\_keyexpr\_includes\_null\_terminated (C function), 16  
 zp\_keyexpr\_intersect\_null\_terminated (C function), 16  
 zp\_keyexpr\_is\_canon\_null\_terminated (C function), 15  
 zp\_keyexpr\_resolve (C function), 14  
 zp\_read (C function), 31  
 zp\_read\_options\_default (C function), 31  
 zp\_read\_options\_t (C type), 11  
 zp\_scouting\_config\_get (C function), 19  
 zp\_scouting\_config\_insert (C function), 19  
 zp\_send\_keep\_alive (C function), 32  
 zp\_send\_keep\_alive\_options\_default (C function), 31  
 zp\_send\_keep\_alive\_options\_t (C type), 11  
 zp\_start\_lease\_task (C function), 31  
 zp\_start\_read\_task (C function), 30  
 zp\_stop\_lease\_task (C function), 31  
 zp\_stop\_read\_task (C function), 30  
 zp\_task\_lease\_options\_default (C function), 31  
 zp\_task\_lease\_options\_t (C type), 10  
 zp\_task\_read\_options\_default (C function), 30

`zp_task_read_options_t` (*C type*), [10](#)