ESP32 JAMScript Port

1.0.0

Generated by Doxygen 1.11.0

Chapter 1

ESP32 Port of JAMScript

1.1 Goals of the Project

This project focuses on porting JAMScript to the ESP32 microcontroller, a low-cost, low-power device with built-in Wi-Fi and Bluetooth capabilities. Specifically, we are to implement a set of functions (which we will refer to as system calls) which are to be called by the JAMScript runtime and are needed for the proper execution of a JAMScript program. By enabling JAMScript on the ESP32, we aim to enhance the efficiency and performance of edge computing systems, facilitating more responsive and intelligent IoT applications. The motivation to port JAMScript to the ESP32 lies in the growing demand for efficient, lightweight, and scalable solutions for edge computing and IoT applications. JAMScript, designed to enable communication between devices and organize tasks, is a powerful framework for distributed systems. However, its current implementation has been largely focused on general-purpose computing platforms. The ESP32, with its dual-core architecture, built-in Wi-Fi and Bluetooth capabilities, low power consumption, and relatively cheap cost presents an ideal target for embedded systems requiring real-time responsiveness. By porting JAMScript to the ESP32, we aim to extend its capabilities to resource-constrained environments, enabling developers to deploy smart, distributed systems directly at the edge.

1.2 Application Example

The implementation of JAMScript on the ESP32 opens up a wide range of applications, particularly in the realm of IoT. One prominent example is in autonomous vehicles, where real-time data processing is crucial. With JAMScript running on ESP32, vehicles can communicate with each other and with infrastructure in real-time, enabling more efficient traffic control and contributing to the development of smart cities. This example illustrates the potential of integrating JAMScript with ESP32 to enhance the efficiency, responsiveness, and intelligence of various edge computing applications.

1.3 Dependencies

- zenoh-pico release version 1.0.0 is used
- espressif/cbor version v0.6.0$\sim$1 is used

1.4 Module Documentation

The documentation for the structs, enums, defines, and functions of the various components associated with this project are shown below.

1.4.1 Cnode

The cnode module includes the data structure which holds all of the information about the controller (c-side) node. It contains functions to initiate and stop the cnode, as well as to send and receive messages over the network using the zenoh protocol. It manages tasks using the tboard component.

1.4.2 Zenoh

The zenoh module is a wrapper of the zenoh-pico library. It is one of the components of the Cnode.

1.4.3 Command

The command modules contains structures to represent a JAMScript command as well as functions to help encode, decode commands using CBOR.

1.4.4 Core

The core module provides the storing and retrivial (into flash) of the cnode nodeID and serialID fields. It is one of the components of the Core module.

1.4.5 System_manager

The system manager module provides the ESP32 system init functionality as well as the initiation of the Wi-Fi module, which is needed for the zenoh protocol. It is one of the components of Cnode.

1.4.6 Task

The task module contains the structures that hold JAMScript tasks, which are to be run via commands.

1.4.7 Tboard

The tboard module provides a structure to manage all of the tasks which can be executed on the cnode, as well as tasks which can be executed remotely by the cnode. It uses FreeRTOS to manage tasks. It is one of the components of Cnode.

1.4.8 Nvoid

The nvoid module is simply a definition of a custom type, which stores a void pointer (a pointer to any possible structure) and a length n (hence the name nvoid). It is one of the types which can be passed through the JAMScript commands.

Chapter 2

Topic Documentation

2.1 Cnode

The cnode module includes the data structure which holds all of the information about the controller (c-side) node.

Data Structures

- struct cnode_args_t

  arguments structure created by process_args() More...
- struct cnode_t

  CNode type, which contains CNode substructures and taskboard. More...

Functions

- cnode_t * cnode_init (int argc, char **argv)

  Constructor.
- void cnode_destroy (cnode_t *cn)

  Frees memory allocated during cnode_init()
- bool cnode_start (cnode_t *cn)

  Starts a Zenoh session, along with sub and pub.
- bool cnode_stop (cnode_t *cn)

  Stops listening thread.
- bool cnode_process_received_cmd (cnode_t *cn, const char *buf, size_t buflen)

  Processes an incoming message received through Zenoh.
- bool cnode_send_cmd (cnode_t *cnode, command_t *cmd)

  Sends a command to the Zenoh network.

2.1.1 Detailed Description

It contains functions to initiate and stop the cnode, as well as to send and receive messages over the network using the zenoh protocol. It manages tasks using the tboard component.

2.1.2 Data Structure Documentation

2.1.2.1 struct cnode_args_t

Data Fields

char *	appid	
int	groupid	
char *	host	
int	nexecs	
int	port	
char *	redhost	
int	redport	
int	snumber	
char *	tags	

2.1.2.2 struct cnode_t

Data Fields

corestate_t *	core_state	pointer to corestate_t object. used to store the node_id and serial_id in ROM.
bool	initialized	boolean representing if this cnode instance has been initialized with cnode_init() or not.
volatile bool	message_received	boolean representing if a message has been received, needs to be reset manually.
char *	node_id	randomly generated (snowflakeid) ID
system_manager_t *	system_manager	pointer to system_manager_t object. used to initiate system & wifi
zenoh_t *	zenoh	pointer to zenoh_t object. used to send messages over the network to other cnodes/controllers.

2.1.3 Function Documentation

2.1.3.1 cnode_destroy()

```
void cnode_destroy (
            cnode_t * cn)
```

Parameters

cn	- pointer to cnode_t struct

2.1.3.2 cnode_init()

```
cnode_t * cnode_init (
            int argc,
            char ** argv)
```

Initiates the cnode structure and initiates all of its components. E.g., we call system_manager_init(), zenoh_init(), ...

Parameters

argc	cmd line argument count
argv	cmd line args

Returns

pointer to cnode_t struct

2.1.3.3 cnode_process_received_cmd()

```
bool cnode_process_received_cmd (
            cnode_t * cn,
            const char * buf,
            size_t buflen)
```

Parameters

cnode	Pointer to the cnode_t instance representing the current node.
buf	Pointer to the raw character buffer containing the encoded message.
buflen	Length of the buffer.

Returns

True if the command was successfully processed, false otherwise.

2.1.3.4 cnode_send_cmd()

```
bool cnode_send_cmd (
            cnode_t * cnode,
            command_t * cmd)
```

Parameters

cnode	Pointer to the cnode_t instance representing the current node.
cmd	Pointer to the command_t object to be sent.

Returns

True if the command was successfully sent, false otherwise.

2.1.3.5 cnode_start()

```
bool cnode_start (
            cnode_t * cn)
```

Starts listening thread.

Parameters

zenoh	pointer to zenoh_t struct
cn	pointer to cnode_t struct

2.1.3.6 cnode_stop()

```
bool cnode_stop (
            cnode_t * cn)
```

Parameters

cn	pointer to cnode_t struct

2.2 Command

The command modules contains structures to represent a JAMScript command as well as functions to help encode, decode commands using CBOR.

Data Structures

- struct arg_t

  Structure representing a single command argument. More...
- struct command_t

  A structure to hold the outgoing and incoming command. More...
- struct internal_command_t

  Structure for handling internal commands within the system. More...

Macros

- #define **TINY_CMD_STR_LEN** 16

  Tiny command length (bytes)
- #define **SMALL_CMD_STR_LEN** 32

  Small command length (bytes)
- #define **LARGE_CMD_STR_LEN** 128

  Large command length (bytes)
- #define **HUGE_CMD_STR_LEN** 1024

  Huge command length (bytes)

Enumerations

- enum jamcommand_t {
  CMD_PING , **CMD_REXEC** , **CMD_REXEC_ACK** , **CMD_REXEC_RES** ,
  CMD_CLOSE_PORT , **CMD_GET_REXEC_RES** }

  Enumeration of command types used in the JAM protocol.
- enum argtype_t {
  NULL_TYPE , **STRING_TYPE** , **INT_TYPE** , **LONG_TYPE** ,
  DOUBLE_TYPE , **NVOID_TYPE** , **VOID_TYPE** }

  Enumeration of argument types that a command can contain.

Functions

- internal_command_t * internal_command_new (command_t *cmd)

  Creates a new internal command from an existing command.
- void internal_command_free (internal_command_t *ic)

  Frees an internal command.
- command_t * command_new (jamcommand_t cmd, int subcmd, const char *fn_name, uint64_t task_id, const char *node_id, const char *fn_argsig,...)

  Creates a new command object with variable arguments.
- command_t * command_new_using_arg (jamcommand_t cmd, int opt, const char *fn_name, uint64_t taskid, const char *node_id, const char *fn_argsig, arg_t *args)

  Creates a new command object using an argument list.
- void command_init_using_arg (command_t *command, jamcommand_t cmd, int opt, const char *fn_name, uint64_t taskid, const char *node_id, const char *fn_argsig, arg_t *args)

  Initializes an existing command object using arguments.
- command_t * command_from_data (char *fn_argsig, void *data, int len)

  Constructs a command from raw data.
- void command_from_data_inplace (command_t *cmdo, const char *fn_argsig, int len)

  Parses raw data into an existing command object.
- void command_hold (command_t *cmd)

  Increments reference count of a command object.
- void command_free (command_t *cmd)

  Frees a command object.
- bool command_qargs_alloc (const char *fmt, arg_t **rargs, va_list args)

  Allocates and initializes argument structures based on format string.
- void command_arg_print (arg_t *arg)

  Prints argument details.
- void command_arg_inner_free (arg_t *arg)

  Frees an argument's internal resources.
- void command_args_free (arg_t *arg)

  Frees a list of arguments.
- arg_t * command_args_clone (arg_t *arg)

  Clones an argument structure.
- void command_print (command_t *cmd)

  Prints command details.
- const char * command_to_string (jamcommand_t cmd, char *output_str, size_t max_len)

  Converts a command to a string.

2.2.1 Detailed Description

2.2.2 Data Structure Documentation

2.2.2.1 struct arg_t

Each argument has a type and a value stored in a union.

Data Fields

int	nargs	Number of arguments.
argtype_t	type	Type of argument.
union _argvalue_t	val	Value contained in union.

2.2.2.2 struct command_t

An outgoing command is parsed into a CBOR formatted byte array and similarly a CBOR formatted byte array is decoded into a CBOR item handle. Also, information is extracted from the CBOR item and inserted into the command structure at the decoding process.

Data Fields

arg_t *	args	List of arguments.
unsigned char	buffer[HUGE_CMD_STR_LEN]	CBOR serialized data.
jamcommand_t	cmd	Command type.
char	fn_argsig[SMALL_CMD_STR_LEN]	Function argument signature.
char	fn_name[SMALL_CMD_STR_LEN]	Function name.
long	id	Unique command ID.
int	length	Length of CBOR data.
char	node_id[LARGE_CMD_STR_LEN]	Unique node identifier (UUID4)
int	refcount	Reference counter for memory management.
int	subcmd	Sub-command type.
uint64_t	task_id	Task identifier (execution ID)

2.2.2.3 struct internal_command_t

A simplified command representation used for internal processing.

Data Fields

arg_t *	args	List of arguments.
jamcommand_t	cmd	Command type.
uint32_t	task_id	Task identifier.

2.2.3 Enumeration Type Documentation

2.2.3.1 argtype_t

```
enum argtype_t
```

These define the type of each argument passed within a command.

2.2.3.2 jamcommand_t

```
enum jamcommand_t
```

These represent different message types exchanged between nodes.

2.2.4 Function Documentation

2.2.4.1 command_arg_inner_free()

```
void command_arg_inner_free (
            arg_t * arg)
```

Parameters

arg	Pointer to argument

2.2.4.2 command_arg_print()

```
void command_arg_print (
            arg_t * arg)
```

Parameters

arg	Pointer to argument to be printed

2.2.4.3 command_args_clone()

```
arg_t * command_args_clone (
            arg_t * arg)
```

Parameters

arg	Pointer to argument to be cloned

Returns

Pointer to newly allocated argument

2.2.4.4 command_args_free()

```
void command_args_free (
            arg_t * arg)
```

Parameters

arg	Pointer to first argument in list

2.2.4.5 command_free()

```
void command_free (
            command_t * cmd)
```

Parameters

cmd	Pointer to command object to be freed

2.2.4.6 command_from_data()

```
command_t * command_from_data (
            char * fn_argsig,
            void * data,
            int len)
```

Parameters

fn_argsig	Argument signature
data	Pointer to raw data
len	Length of data

Returns

Pointer to newly allocated command object

2.2.4.7 command_from_data_inplace()

```
void command_from_data_inplace (
            command_t * cmdo,
            const char * fn_argsig,
            int len)
```

Parameters

cmdo	Pointer to an existing command object
fn_argsig	Argument signature
len	Length of data

2.2.4.8 command_hold()

```
void command_hold (
            command_t * cmd)
```

Parameters

cmd	Pointer to command object

2.2.4.9 command_init_using_arg()

```
void command_init_using_arg (
            command_t * command,
            jamcommand_t cmd,
            int opt,
            const char * fn_name,
            uint64_t taskid,
            const char * node_id,
            const char * fn_argsig,
            arg_t * args)
```

Parameters

command	Pointer to command object
cmd	Command type

opt	Optional parameters
fn_name	Function name
taskid	Task identifier
node_id	Node UUID
fn_argsig	Argument signature
args	Pointer to argument list

2.2.4.10 command_new()

```
command_t * command_new (
             jamcommand_t cmd,
             int subcmd,
             const char * fn_name,
             uint64_t task_id,
             const char * node_id,
             const char * fn_argsig,
              ...)
```

Parameters

cmd	Command type
subcmd	Sub-command type
fn_name	Function name
task_id	Task identifier
node_id	Node UUID
fn_argsig	Argument signature

Returns

Pointer to newly allocated command object

2.2.4.11 command_new_using_arg()

```
command_t * command_new_using_arg (
             jamcommand_t cmd,
             int opt,
             const char * fn_name,
             uint64_t taskid,
             const char * node_id,
             const char * fn_argsig,
             arg_t * args)
```

Parameters

cmd	Command type
opt	Optional parameters
fn_name	Function name
taskid	Task identifier

node_id	Node UUID
fn_argsig	Argument signature
args	Pointer to argument list

Returns

Pointer to newly allocated command object

2.2.4.12 command_print()

```
void command_print (
            command_t * cmd)
```

Parameters

cmd	Pointer to command to be printed

2.2.4.13 command_qargs_alloc()

```
bool command_qargs_alloc (
            const char * fmt,
            arg_t ** rargs,
            va_list args)
```

Parameters

fmt	Format string describing argument types
rargs	Pointer to allocated argument list
args	Variable argument list

Returns

Boolean indicating success or failure

2.2.4.14 command_to_string()

```
const char * command_to_string (
            jamcommand_t cmd,
            char * output_str,
            size_t max_len)
```

Parameters

cmd	Command to be converted
output_str	Buffer to store the output string
max_len	Maximum length of the output string

Returns

String representation of the command

2.2.4.15 internal_command_free()

```
void internal_command_free (
            internal_command_t * ic)
```

Parameters

ic	Pointer to the internal command to be freed

2.2.4.16 internal_command_new()

```
internal_command_t * internal_command_new (
            command_t * cmd)
```

Parameters

cmd	Pointer to an existing command_t object

Returns

Pointer to a newly allocated internal_command_t object

2.3 Core

The core module provides the storing and retrivial (into flash) of the cnode nodeID and serialID fields.

Data Structures

- struct corestate_t

  Struct representing the core state. More...

Functions

- corestate_t * core_init (int serialnum)

  Constructor.
- void core_destroy (corestate_t *cs)

  Frees memory allocated during core_init()
- void core_setup (corestate_t *cs)

  Does the UUID4 generation (for node ID) and stores serial & node ID into flash memory.
- int discountflake (char *buffer)

  Does the discount snowflake generation.

2.3.1 Detailed Description

It is one of the components of the Core module.

2.3.2 Data Structure Documentation

2.3.2.1 struct corestate_t

Data Fields

char *	device_id	device ID (nodeID). This is a snowflakeID.
int	serial_num	serial ID (0, 1, ...)

2.3.3 Function Documentation

2.3.3.1 core_destroy()

```
void core_destroy (
            corestate_t * cs)
```

Parameters

cs	pointer to corestate_t struct

2.3.3.2 core_init()

```
corestate_t * core_init (
            int serialnum)
```

Initiates the core. Calls core_setup() to generate serial & node ID

Parameters

serialnum	Serial number of the node

Returns

pointer to corestate_t struct

2.3.3.3 core_setup()

```
void core_setup (
            corestate_t * cs)
```

Parameters

cs	pointer to corestate_t struct

2.3.3.4 discountflake()

```
int discountflake (
            char * buffer)
```

Parameters

buffer	pointer to a buffer able to contain ID

Return values

-1	error occured during generation
0	ID generation successful

2.4 Nvoid

The nvoid module is simply a definition of a custom type, which stores a void pointer (a pointer to any possible structure) and a length n (hence the name nvoid).

Data Structures

- struct nvoid_t

  Struct defining the nvoid type. More...

Macros

- #define nvoid_free(n)

  Free the memory allocated to the nvoid object.

Functions

- nvoid_t ∗ nvoid_new (void ∗data, int len)

  Constructor.
- nvoid_t ∗ nvoid_null ()

  Creates a new nvoid object which by default points to null.

2.4.1 Detailed Description

It is one of the types which can be passed through the JAMScript commands.

2.4.2 Data Structure Documentation

2.4.2.1 struct nvoid_t

Data Fields

void *	data	pointer to the data
int	len	length of the nvoid object

2.4.3 Macro Definition Documentation

2.4.3.1 nvoid_free

```
#define nvoid_free(
                n)
```

Value:

```
    do {            \
    free(n);                        \
} while (0)
```

2.4.4 Function Documentation

2.4.4.1 nvoid_new()

```
nvoid_t * nvoid_new (
             void * data,
             int len)
```

Allocates memory dynamically to create a new nvoid object.

Parameters

data	void pointer to the data
len	length of that data

Returns

pointer to the newly created nvoid object.

2.4.4.2 nvoid_null()

```
nvoid_t * nvoid_null ()
```

(&data=0, len=0).

Returns

pointer to the newly created nvoid object.

2.5 System_manager

The system manager module provides the ESP32 system init functionality as well as the initiation of the Wi-Fi module, which is needed for the zenoh protocol.

Data Structures

- struct system_manager_t

  Struct representing the system manager. More...

Functions

- system_manager_t * system_manager_init ()

  Constructor.
- bool system_manager_destroy (system_manager_t *system_manager)

  Frees memory associated with the system_manager_t struct.
- bool system_manager_wifi_init (system_manager_t *system_manager)

  Initializes the Wifi module and connects to a preset network.

2.5.1 Detailed Description

It is one of the components of Cnode.

2.5.2 Data Structure Documentation

2.5.2.1 struct system_manager_t

Data Fields

int	_connection_attempts	number of connections attempted
esp_event_handler_instance_t	got_ip_event_handle	event handle for got ip event
esp_event_handler_instance_t	wifi_any_event_handle	event handle for wifi events
bool	wifi_connection	if we are connected to the wifi or not

2.5.3 Function Documentation

2.5.3.1 system_manager_destroy()

```
bool system_manager_destroy (
            system_manager_t * system_manager)
```

Parameters

system_manager	pointer to system_manager_t struct

2.5.3.2 system_manager_init()

```
system_manager_t * system_manager_init ()
```

Initializes the system manager.

Returns

pointer to system_manager_t struct

2.5.3.3 system_manager_wifi_init()

```
bool system_manager_wifi_init (
            system_manager_t * system_manager)
```

Return values

true	If wifi initiation successful
false	If error occured during wifi init

2.6 Task

The task module contains the structures that hold JAMScript tasks, which are to be run via commands.

Data Structures

- struct execution_context_t

  Structure containing the execution context of a currently executing task. More...
- struct task_t

  Structure representing one task that is to be run by tboard. More...

Macros

- #define **MAX_ARGS** 20

  Maximum number of arguments.
- #define **MAX_TASKS** 20

  Maximum number of tasks.

Typedefs

- typedef void(∗ **function_stub_t**) (execution_context_t ∗)

  Function pointer to a function that returns void and takes in a execution_context_t∗ (function_stub)

Functions

- task_t * task_create (char *name, uint32_t serial_id, argtype_t return_type, char *fn_argsig, function_stub_t entry_point)

  Constructor.
- void task_destroy (task_t *task)

  Destructor.
- arg_t ** task_get_args (task_t *task)

  Returns the arguments of the task.
- void task_set_return_arg (task_t *task, arg_t *return_arg)

  Set the return argument of the task.
- void task_set_args (task_t *task, int num_args,...)

  Set the arguments of the task.
- void task_print (task_t *task)

  Print out information about task to the terminal.

2.6.1 Detailed Description

2.6.2 Data Structure Documentation

2.6.2.1 struct execution_context_t

Data Fields

arg_t **	query_args	query arguments to the task
arg_t *	return_arg	return argument

2.6.2.2 struct task_t

Data Fields

arg_t *	args[MAX_ARGS]	array of arg_t objects for the arguments
function_stub_t	entry_point	function pointer; represents the entry point to the stub of this function
char *	fn_argsig	string representing the argument signature in compact form. i.e., "iis" => (int, int, string)
volatile bool	has_finished	if the task has finished or not
volatile bool	is_running	if the task is running or not
char *	name	string: name of the task
arg_t *	return_arg	return value and type
uint32_t	serial_id	id starting at 0
TaskHandle_t	task_handle_frtos	task handle from free rtos

2.6.3 Function Documentation

2.6.3.1 task_create()

```
task_t * task_create (
            char * name,
            uint32_t serial_id,
            argtype_t return_type,
            char * fn_argsig,
            function_stub_t entry_point)
```

Initializes the task_t struct. The task_handle, args and return_arg (value) are set to NULL.

Note

Does not start the task.

Parameters

name	string describing name of function
serial_id	id of the function
return_type	enum value describing one of several possible return types of the function
fn_argsig	string, argument signature (see task_t)
entry_point	function pointer to stub of the function to be run (see function_stub_t)

Returns

pointer to initialized task_t struct

Return values

NULL	if could not allocate

2.6.3.2 task_destroy()

```
void task_destroy (
            task_t * task)
```

Frees memory allocated for the task_t struct.

Parameters

task	pointer to task_t struct

2.6.3.3 task_get_args()

```
arg_t ** task_get_args (
            task_t * task)
```

Parameters

task	pointer to task_t struct

Returns

pointer to arguments (arg_t)

2.6.3.4 task_print()

```
void task_print (
            task_t * task)
```

Parameters

task	pointer to task_t struct

2.6.3.5 task_set_args()

```
void task_set_args (
            task_t * task,
            int num_args,
             ...)
```

Parameters

task	pointer to task_t struct
num_args	number of arguments

Note

This function takes in variable arguments, each of which must be an arg_t* object

2.6.3.6 task_set_return_arg()

```
void task_set_return_arg (
            task_t * task,
            arg_t * return_arg)
```

Parameters

task	pointer to task_t struct
return_arg	pointer to arg_t struct

2.7 Tboard

The tboard module provides a structure to manage all of the tasks which can be executed on the cnode, as well as tasks which can be executed remotely by the cnode.

Data Structures

- struct tboard_t

  Structure representing the tboard itself. More...

Functions

- tboard_t * tboard_create ()

  Constructor.
- void tboard_destroy (tboard_t *tboard)

  Destructor.
- void tboard_register_task (tboard_t *tboard, task_t *task)

  Registers a task to the tboard.
- bool tboard_start_task (tboard_t *tboard, int task_serial_id, arg_t **args)

  Starts a task registered to the tboard with given arguments using the task serial id.
- void tboard_shutdown (tboard_t *tboard)

  Shutdown the tboard.

2.7.1 Detailed Description

It uses FreeRTOS to manage tasks. It is one of the components of Cnode.

2.7.2 Data Structure Documentation

2.7.2.1 struct tboard_t

Note

Can be accessed by various tasks (need to be careful about race conditions).

Data Fields

uint32_t	last_dead_task_id	the id of the last dead task
uint32_t	num_dead_tasks	number of dead tasks
uint32_t	num_tasks	number of tasks to run
SemaphoreHandle_t	task_management_mutex	
StaticSemaphore_t	task_management_mutex_data	
task_t *	tasks[MAX_TASKS]	pointer to array of tasks

2.7.3 Function Documentation

2.7.3.1 tboard_create()

```
tboard_t * tboard_create ()
```

Initializes the tboard structure. Should allocate memory to the array of tasks.

Returns

pointer to initialized tboard struct

2.7.3.2 tboard_destroy()

```
void tboard_destroy (
            tboard_t * tboard)
```

Frees memory allocated during creation of the tboard structure.

Parameters

tboard	pointer to tboard_t struct

2.7.3.3 tboard_register_task()

```
void tboard_register_task (
            tboard_t * tboard,
            task_t * task)
```

Parameters

tboard	pointer to tboard_t struct
task	pointer to task_t struct

2.7.3.4 tboard_shutdown()

```
void tboard_shutdown (
            tboard_t * tboard)
```

Parameters

tboard	pointer to tboard_t struct

2.7.3.5 tboard_start_task()

```
bool tboard_start_task (
            tboard_t * tboard,
            int task_serial_id,
            arg_t ** args)
```

Note

The task needs to have already been registered using tboard_register_task()

Parameters

tboard	pointer to tboard_t struct
task_serial_id	serial id of the task that is to be ran
args	pointer to an array of arg_t pointers (each of which represents an argument)

Return values

true	if the task was started successfully
false	an error occured during task starting (i.e., task not found)

2.8 Zenoh

The zenoh module is a wrapper of the zenoh-pico library.

Data Structures

- struct zenoh_t

  Struct representing a zenoh object. More...

Typedefs

- typedef void(∗ **zenoh_callback_t**) (z_loaned_sample_t ∗, void ∗)

  Function signature of the callback function which must be defined for zenoh_declare_sub().

Functions

- zenoh_t ∗ zenoh_init ()

  Constructor.
- void zenoh_destroy (zenoh_t ∗zenoh)

  Frees memory associated with the zenoh_t struct.
- bool zenoh_scout ()

  Scouts for JNodes.
- bool zenoh_declare_sub (zenoh_t ∗zenoh, const char ∗key_expression, zenoh_callback_t ∗callback, void ∗cb_arg)

  Declare a zenoh subscriber on a specific topic.
- bool zenoh_declare_pub (zenoh_t ∗zenoh, const char ∗key_expression)

  Declare a zenoh publisher on a specific topic.
- void zenoh_start_read_task (zenoh_t ∗zenoh)

  Start the zenoh read task by calling zp_start_read_task()
- void zenoh_start_lease_task (zenoh_t ∗zenoh)

  Start the zenoh lease task by calling zp_start_lease_task()
- bool zenoh_publish (zenoh_t ∗zenoh, const char ∗message)

  Publish a message over zenoh.
- bool zenoh_publish_encoded (zenoh_t ∗zenoh, const uint8_t ∗buffer, size_t buffer_len)

  Publish an encoded CBOR message over zenoh.

2.8.1 Detailed Description

It is one of the components of the Cnode.

Note

zenoh-pico version 1.0.0 is used

2.8.2 Data Structure Documentation

2.8.2.1 struct zenoh_t

Data Fields

z_owned_publisher_t	z_pub	zenoh publisher instance. used when publishing messages.
z_owned_session_t	z_session	zenoh session instance.
z_owned_subscriber_t	z_sub	zenoh subscriber instance. used when receiving messages.

2.8.3 Function Documentation

2.8.3.1 zenoh_declare_pub()

```
bool zenoh_declare_pub (
            zenoh_t * zenoh,
            const char * key_expression)
```

Parameters

zenoh	pointer to zenoh_t struct
key_expression	string describing the 'subscription topic'

Return values

true	If publish declaration returned without error
false	If an error occured

2.8.3.2 zenoh_declare_sub()

```
bool zenoh_declare_sub (
            zenoh_t * zenoh,
            const char * key_expression,
            zenoh_callback_t * callback,
            void * cb_arg)
```

Assign callback function.

Parameters

zenoh	pointer to zenoh_t struct
key_expression	string describing the 'subscription topic'
callback	pointer to zenoh callback function
cb_arg	pointer to argument passed to callback function

Return values

true	If subscription declaration returned without error
false	If an error occured

2.8.3.3 zenoh_destroy()

```
void zenoh_destroy (
            zenoh_t * zenoh)
```

Parameters

zenoh	pointer to zenoh_t struct

2.8.3.4 zenoh_init()

```
zenoh_t * zenoh_init ()
```

Initializes zenoh objects and starts a Zenoh session.

Returns

pointer to unitialized zenoh_t struct

2.8.3.5 zenoh_publish()

```
bool zenoh_publish (
            zenoh_t * zenoh,
            const char * message)
```

Parameters

zenoh	pointer to zenoh_t struct
message	string consisting of message

Return values

true	If publish successful
false	If an error occured

2.8.3.6 zenoh_publish_encoded()

```
bool zenoh_publish_encoded (
            zenoh_t * zenoh,
            const uint8_t * buffer,
            size_t buffer_len)
```

Parameters

zenoh	pointer to zenoh_t struct
buffer	pointer to buffer containing CBOR encoded message
buffer_len	length of buffer

Return values

true	If publish successful, false otherwise

2.8.3.7 zenoh_scout()

```
bool zenoh_scout ()
```

Note that JNodes must be using Zenoh.

Return values

true	If a JNode is found
false	If a JNode is not found

2.8.3.8 zenoh_start_lease_task()

```
void zenoh_start_lease_task (
            zenoh_t * zenoh)
```

Parameters

zenoh	pointer to zenoh_t struct

2.8.3.9 zenoh_start_read_task()

```
void zenoh_start_read_task (
            zenoh_t * zenoh)
```

Parameters

zenoh	pointer to zenoh_t struct

Index