UserAPI

1.00

Generated by Doxygen 1.8.13

Contents

1 Rozum Robotics User API & Servo Box **1**

1.1 Servo box 1

1.2 API Categories 1

1.3 API Tutorials 1

2 Module Index **3**

2.1 Modules 3

3 Class Index **5**

3.1 Class List 5

4 File Index **7**

4.1 File List 7

5 Module Documentation **9**

5.1 Initialization and deinitialization 9

5.1.1 Detailed Description 9

5.1.2 Function Documentation 9

5.1.2.1 rr_deinit_interface() 9

5.1.2.2 rr_deinit_servo() 10

5.1.2.3 rr_init_interface() 10

5.1.2.4 rr_init_servo() 11

5.2 Switching servo working states 12

5.2.1 Detailed Description 12

5.2.2 Function Documentation 12

5.2.2.1 rr_describe_nmt() 12
5.2.2.2 rr_net_reboot() 13
5.2.2.3 rr_net_reset_communication() 13
5.2.2.4 rr_net_set_state_operational() 14
5.2.2.5 rr_net_set_state_pre_operational() 14
5.2.2.6 rr_net_set_state_stopped() 14
5.2.2.7 rr_servo_reboot() 15
5.2.2.8 rr_servo_reset_communication() 15
5.2.2.9 rr_servo_set_state_operational() 15
5.2.2.10 rr_servo_set_state_pre_operational() 16
5.2.2.11 rr_servo_set_state_stopped() 16
5.2.2.12 rr_setup_nmt_callback() 17
5.3 Simple motion control (duty, current, velocity, position) 18
5.3.1 Detailed Description 18
5.3.2 Function Documentation 18
5.3.2.1 rr_set_current() 18
5.3.2.2 rr_set_duty() 19
5.3.2.3 rr_set_position() 19
5.3.2.4 rr_set_position_with_limits() 20
5.3.2.5 rr_set_velocity() 20
5.3.2.6 rr_set_velocity_with_limits() 20
5.3.2.7 rr_stop_and_freeze() 21
5.3.2.8 rr_stop_and_release() 21
5.4 Trajectory motion control (PVT) 23
5.4.1 Detailed Description 23
5.4.2 Function Documentation 23
5.4.2.1 rr_add_motion_point() 24
5.4.2.2 rr_clear_points() 24
5.4.2.3 rr_clear_points_all() 25
5.4.2.4 rr_get_points_free_space() 25

5.4.2.5 rr_get_points_size() 26

5.4.2.6 rr_get_time_calculation_result() 26

5.4.2.7 rr_invoke_time_calculation() 26

5.4.2.8 rr_start_motion() 27

5.5 Reading and writing servo configuration 29

5.5.1 Detailed Description 29

5.5.2 Function Documentation 29

5.5.2.1 rr_get_max_velocity() 29

5.5.2.2 rr_set_max_velocity() 30

5.5.2.3 rr_set_zero_position() 30

5.5.2.4 rr_set_zero_position_and_save() 30

5.6 Reading realtime parameter 32

5.6.1 Detailed Description 32

5.6.2 Function Documentation 32

5.6.2.1 rr_param_cache_setup_entry() 32

5.6.2.2 rr_param_cache_update() 33

5.6.2.3 rr_read_cached_parameter() 33

5.6.2.4 rr_read_parameter() 34

5.7 Error handling 35

5.7.1 Detailed Description 35

5.7.2 Function Documentation 35

5.7.2.1 rr_describe_emcy_bit() 35

5.7.2.2 rr_describe_emcy_code() 35

5.7.2.3 rr_read_error_status() 37

5.7.2.4 rr_setup_emcy_callback() 37

5.8 Debugging 39

5.8.1 Detailed Description 39

5.8.2 Function Documentation 39

5.8.2.1 rr_set_comm_log_stream() 39

5.8.2.2 rr_set_debug_log_stream() 39

5.9 Auxiliary functions . . . 41
5.9.1 Detailed Description . . . 41
5.9.2 Function Documentation . . . 41
5.9.2.1 rr_sleep_ms() . . . 41
5.10 Servo box specs & manual . . . 42
5.10.1 1. Product overview . . . 42
5.10.2 2. Integrating servos with a power supply and a servobox . . . 42
5.10.2.1 2.1. Power supply connection . . . 42
5.10.2.2 2.2. CAN connection . . . 43
5.10.3 3. Servobox components . . . 43
5.10.3.1 3.1 Energy eater . . . 43
5.10.3.2 3.2 Capacitor module . . . 44
5.11 One servo PVT move . . . 45
5.12 Two servos PVT move . . . 47
5.13 Three servos PVT move . . . 49
5.14 Device parameter cache programming and reading . . . 52
5.15 Device parameters reading . . . 54
5.16 Servo PVT point calculation . . . 55
5.17 Reading of the maximum velocity . . . 56
5.18 Reading of the motion queue parameters . . . 57
5.19 Reading device errors . . . 58

6 Class Documentation 59

6.1 param_cache_entry_t Struct Reference . . . 59
6.1.1 Detailed Description . . . 59
6.2 rr_can_interface_t Struct Reference . . . 59
6.2.1 Detailed Description . . . 60
6.3 rr_servo_t Struct Reference . . . 60
6.3.1 Detailed Description . . . 60

7 File Documentation 61
7.1 include/api.h File Reference 61
7.1.1 Detailed Description 66
7.1.2 Typedef Documentation 66
7.1.2.1 rr_emcy_cb_t 66
7.1.2.2 rr_nmt_cb_t 66
7.1.3 Enumeration Type Documentation 67
7.1.3.1 rr_nmt_state_t 67
7.1.3.2 rr_ret_status_t 67
7.1.3.3 rr_servo_param_t 67
7.2 src/api.c File Reference 69
7.2.1 Detailed Description 72
7.3 tutorial/control_servo_traj_1.c File Reference 73
7.3.1 Detailed Description 73
7.3.2 Function Documentation 73
7.3.2.1 main() 73
7.4 tutorial/control_servo_traj_2.c File Reference 74
7.4.1 Detailed Description 74
7.4.2 Function Documentation 74
7.4.2.1 main() 74
7.5 tutorial/control_servo_traj_3.c File Reference 75
7.5.1 Detailed Description 75
7.5.2 Function Documentation 75
7.5.2.1 main() 75
7.6 tutorial/read_any_param.c File Reference 76
7.6.1 Detailed Description 76
7.6.2 Function Documentation 76
7.6.2.1 main() 76
7.7 tutorial/read_any_param_cache.c File Reference 77
7.7.1 Detailed Description 77

7.7.2 Function Documentation 77
7.7.2.1 main() 77
7.8 tutorial/read_errors.c File Reference 78
7.8.1 Detailed Description 78
7.8.2 Function Documentation 79
7.8.2.1 main() 79
7.9 tutorial/read_servo_max_velocity.c File Reference 79
7.9.1 Detailed Description 80
7.9.2 Function Documentation 80
7.9.2.1 main() 80
7.10 tutorial/read_servo_motion_queue.c File Reference 80
7.10.1 Detailed Description 81
7.10.2 Function Documentation 81
7.10.2.1 main() 81
7.11 tutorial/read_servo_trajectory_time.c File Reference 82
7.11.1 Detailed Description 82
7.11.2 Function Documentation 82
7.11.2.1 main() 82
7.12 tutorial/tutorial.h File Reference 83
7.12.1 Detailed Description 83
7.13 /mnt/hdd/home/maksimkatkou/work/rr-git/UserAPI/hw/doc.c File Reference 83
7.13.1 Detailed Description 83

Index **85**

Chapter 1

Rozum Robotics User API & Servo Box

1.1 Servo box

- Servo box specs & manual

1.2 API Categories

- Auxiliary functions
- Initialization and deinitialization
- Switching servo working states
- Simple motion control (duty, current, velocity, position)
- Trajectory motion control (PVT)
- Reading and writing servo configuration
- Reading realtime parameter
- Error handling
- Debugging

1.3 API Tutorials

- C
  1. One servo PVT move
  2. Two servos PVT move
  3. Three servos PVT move
  4. Device parameters reading
  5. Device parameter cache programming and reading
  6. Reading device errors
  7. Servo PVT point calculation
  8. Reading of the motion queue parameters
  9. Reading of the maximum velocity
- Java
- Python
- Ruby

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Initialization and deinitialization 9
Switching servo working states 12
Simple motion control (duty, current, velocity, position) 18
Trajectory motion control (PVT) 23
Reading and writing servo configuration 29
Reading realtime parameter 32
Error handling 35
Debugging 39
Auxiliary functions 41
Servo box specs & manual 42
One servo PVT move 45
Two servos PVT move 47
Three servos PVT move 49
Device parameter cache programming and reading 52
Device parameters reading 54
Servo PVT point calculation 55
Reading of the maximum velocity 56
Reading of the motion queue parameters 57
Reading device errors 58

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

param_cache_entry_t
Device information source instance 59
rr_can_interface_t
Interface instance structure 59
rr_servo_t
Device instance structure 60

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

include/api.h
Rozum Robotics API Header File 61
src/api.c
Rozum Robotics API Source File 69
tutorial/control_servo_traj_1.c
Setting PVT points for one servo 73
tutorial/control_servo_traj_2.c
Setting PVT points for two servos 74
tutorial/control_servo_traj_3.c
Setting points for three servos 75
tutorial/read_any_param.c
Tutorial example of reading device parameters 76
tutorial/read_any_param_cache.c
Tutorial example of reading servo parameters from the cache 77
tutorial/read_errors.c
Tutorial example of reading device errors 78
tutorial/read_servo_max_velocity.c
Tutorial example of reading the maximum servo 79
tutorial/read_servo_motion_queue.c
Tutorial example of reading motion queue parameters 80
tutorial/read_servo_trajectory_time.c
Tutorial example of calculating a PVT point 82
tutorial/tutorial.h
Tutorial header file with config 83
/mnt/hdd/home/maksimkatkou/work/rr-git/UserAPI/hw/doc.c
Hardware manual 83

Chapter 5

Module Documentation

5.1 Initialization and deinitialization

Functions

- rr_can_interface_t ∗ rr_init_interface (const char ∗interface_name)

  The function is the first to call to be able to work with the user API. It opens the COM port where the corresponding CAN-USB dongle is connected, enabling communication between the user program and the servo motors on the respective CAN bus.
- int rr_deinit_interface (rr_can_interface_t ∗∗interface)

  The function closes the COM port where the corresponding CAN-USB dongle is connected, clearing all data associated with the interface descriptor. It is advisable to call the function every time before quitting the user program.
- rr_servo_t ∗ rr_init_servo (rr_can_interface_t ∗interface, const uint8_t id)

  The function determines whether the servo motor with the speficied ID is connected to the specified interface. It waits for 2 seconds to receive a Heartbeat message from the servo. When the message arrives within the interval, the servo is identified as successfully connected.
- int rr_deinit_servo (rr_servo_t ∗∗servo)

  The function deinitializes the servo, clearing all data associated with the servo descriptor.

5.1.1 Detailed Description

5.1.2 Function Documentation

5.1.2.1 rr_deinit_interface()

```
int rr_deinit_interface (
            rr_can_interface_t ** interface )
```

The function closes the COM port where the corresponding CAN-USB dongle is connected, clearing all data associated with the interface descriptor. It is advisable to call the function every time before quitting the user program.

Parameters

interface	Interface descriptor (see rr_init_interface).

Returns

int Status code (rr_ret_status_t)

5.1.2.2 rr_deinit_servo()

```
int rr_deinit_servo (
            rr_servo_t ** servo )
```

The function deinitializes the servo, clearing all data associated with the servo descriptor.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

int Status code (rr_ret_status_t)

5.1.2.3 rr_init_interface()

```
rr_can_interface_t* rr_init_interface (
            const char * interface_name )
```

The function is the first to call to be able to work with the user API. It opens the COM port where the corresponding CAN-USB dongle is connected, enabling communication between the user program and the servo motors on the respective CAN bus.

Example:

rr_can_interface_t *interface = rr_init_interface ("/dev/ttyACM0");

```
if(!interface)
{
... handle errors ...
}
```

Parameters

interface_name	Full path to the COM port to open. The path can vary, depending on the operating system.

Examples:

OS Linux: "/dev/ttyACM0"

mac OS: "/dev/cu.modem301"

Returns

Interface descriptor (rr_can_interface_t)
or NULL when an error occurs

5.1.2.4 rr_init_servo()

```
rr_servo_t* rr_init_servo (
            rr_can_interface_t * interface,
            const uint8_t id )
```

The function determines whether the servo motor with the speficied ID is connected to the specified interface. It waits for 2 seconds to receive a Heartbeat message from the servo. When the message arrives within the interval, the servo is identified as successfully connected.

The function returns the servo descriptor that you will need for subsequent API calls to the servo.

Parameters

interface	Descriptor of the interface (returned by the rr_init_interface function) where the servo is connected
id	Unique identifier of the servo in the specified interface. The available value range is from 0 to 127.

Returns

Servo descriptor (rr_servo_t)
or NULL when no Heartbeat message is received within the specified interval

5.2 Switching servo working states

Functions

- void rr_setup_nmt_callback (rr_can_interface_t ∗interface, rr_nmt_cb_t cb)

  The function sets a user callback to be intiated in connection with with changes of network management (NMT) states (e.g., a servo connected to/ disconnected from the CAN bus, the interface/ a servo going to the operational state, etc.). User callbacks are functions to execute specific user-defined operations, e.g., to display a warning about an NMT state change or stop the program.
- const char ∗ rr_describe_nmt (rr_nmt_state_t state)

  The function returns a string describing the NMT state code specified in the 'state' parameter. You can also use the function with rr_setup_nmt_callback, setting the callback to display a detailed message describing an NMT event.
- int rr_servo_reboot (const rr_servo_t ∗servo)

  The function reboots the servo specified in the 'servo' parameter of the function, resetting it to the power-on state.
- int rr_servo_reset_communication (const rr_servo_t ∗servo)

  The function resets communication with the servo specified in the 'servo' parameter without resetting the entire interface.
- int rr_servo_set_state_operational (const rr_servo_t ∗servo)

  The function sets the servo specified in the 'servo' parameter to the operational state. In the state, the servo is both available for communication and can execute commands.
- int rr_servo_set_state_pre_operational (const rr_servo_t ∗servo)

  The function sets the servo specified in the 'servo' parameter to the pre-operational state. In the state, the servo is available for communication, but cannot execute any commands.
- int rr_servo_set_state_stopped (const rr_servo_t ∗servo)

  The function sets the servo specified in the 'servo' parameter to the stopped state. In the state, only Heartbeats are available. You can neither communicate with the servo nor make it execute any commands.
- int rr_net_reboot (const rr_can_interface_t ∗interface)

  The function reboots all servos connected to the interface specified in the 'interface' parameter, resetting them back to the power-on state.
- int rr_net_reset_communication (const rr_can_interface_t ∗interface)

  The function resets communication via the interface specified in the 'interface' parameter. For instance, you may need to use the function when changing settings that require a reset after modification.
- int rr_net_set_state_operational (const rr_can_interface_t ∗interface)

  The function sets all servos connected to the interface (CAN bus) specified in the 'interface' parameter to the operational state. In the state, the servos can both communicate with the user program and execute commands.
- int rr_net_set_state_pre_operational (const rr_can_interface_t ∗interface)

  The function sets all servos connected to the interface specified in the 'interface' parameter to the pre-operational state. In the state, the servos are available for communication, but cannot execute commands.
- int rr_net_set_state_stopped (const rr_can_interface_t ∗interface)

  The function sets all servos connected to the interface specified in the 'interface' parameter to the stopped state. In the state, the servos are neither available for communication nor can execute commands.

5.2.1 Detailed Description

5.2.2 Function Documentation

5.2.2.1 rr_describe_nmt()

```
const char* rr_describe_nmt (
            rr_nmt_state_t state )
```

The function returns a string describing the NMT state code specified in the 'state' parameter. You can also use the function with rr_setup_nmt_callback, setting the callback to display a detailed message describing an NMT event.

Parameters

state	NMT state code to descibe

Returns

Pointer to the description string

5.2.2.2 rr_net_reboot()

```
int rr_net_reboot (
            const rr_can_interface_t * interface )
```

The function reboots all servos connected to the interface specified in the 'interface' parameter, resetting them back to the power-on state.

Parameters

interface	Interface descriptor returned by the rr_init_interface function

Returns

int Status code (rr_ret_status_t)

5.2.2.3 rr_net_reset_communication()

```
int rr_net_reset_communication (
            const rr_can_interface_t * interface )
```

The function resets communication via the interface specified in the 'interface' parameter. For instance, you may need to use the function when changing settings that require a reset after modification.

Parameters

interface	Interface descriptor returned by the rr_init_interface function

Returns

int Status code (rr_ret_status_t)

5.2.2.4 rr_net_set_state_operational()

```
int rr_net_set_state_operational (
            const rr_can_interface_t * interface )
```

The function sets all servos connected to the interface (CAN bus) specified in the 'interface' parameter to the operational state. In the state, the servos can both communicate with the user program and execute commands.

For instance, you may need to call the function to switch all servos on a specific bus from the pre-operational state to the operational one after an error (e.g., due to overcurrent).

Parameters

interface	Interface descriptor returned by the rr_init_interface function

Returns

int Status code (rr_ret_status_t)

5.2.2.5 rr_net_set_state_pre_operational()

```
int rr_net_set_state_pre_operational (
            const rr_can_interface_t * interface )
```

The function sets all servos connected to the interface specified in the 'interface' parameter to the pre-operational state. In the state, the servos are available for communication, but cannot execute commands.

For instance, you may need to call the function, if you want to force all servos on a specific bus to stop executing commands, e.g., in an emergency.

Parameters

interface	Interface descriptor returned by the rr_init_interface function

Returns

int Status code (rr_ret_status_t)

5.2.2.6 rr_net_set_state_stopped()

```
int rr_net_set_state_stopped (
            const rr_can_interface_t * interface )
```

The function sets all servos connected to the interface specified in the 'interface' parameter to the stopped state. In the state, the servos are neither available for communication nor can execute commands.

For instance, you may need to call the fuction to stop all servos on a specific bus without deinitializing them.

Parameters

interface	Interface descriptor returned by the rr_init_interface function.

Returns

int Status code (rr_ret_status_t)

5.2.2.7 rr_servo_reboot()

```
int rr_servo_reboot (
            const rr_servo_t * servo )
```

The function reboots the servo specified in the 'servo' parameter of the function, resetting it to the power-on state.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

int Status code (rr_ret_status_t)

5.2.2.8 rr_servo_reset_communication()

```
int rr_servo_reset_communication (
            const rr_servo_t * servo )
```

The function resets communication with the servo specified in the 'servo' parameter without resetting the entire interface.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

int Status code (rr_ret_status_t)

5.2.2.9 rr_servo_set_state_operational()

```
int rr_servo_set_state_operational (
            const rr_servo_t * servo )
```

The function sets the servo specified in the 'servo' parameter to the operational state. In the state, the servo is both available for communication and can execute commands.

For instance, you may need to call the function to switch the servo from the pre-operational state to the operational one after an error (e.g., due to overcurrent).

Parameters

servo	Servo descriptor returned by the rr_init_servo function If the parameter is set to 0, all servos connected to the interface will be set to the operational state.

Returns

int Status code (rr_ret_status_t)

5.2.2.10 rr_servo_set_state_pre_operational()

```
int rr_servo_set_state_pre_operational (
            const rr_servo_t * servo )
```

The function sets the servo specified in the 'servo' parameter to the pre-operational state. In the state, the servo is available for communication, but cannot execute any commands.

For instance, you may need to call the function, if you want to force the servo to stop executing commands, e.g., in an emergency.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

int Status code (rr_ret_status_t)

5.2.2.11 rr_servo_set_state_stopped()

```
int rr_servo_set_state_stopped (
            const rr_servo_t * servo )
```

The function sets the servo specified in the 'servo' parameter to the stopped state. In the state, only Heartbeats are available. You can neither communicate with the servo nor make it execute any commands.

For instance, you may need to call the fuction to reduce the workload of a CAN bus by disabling individual servos connected to it without deninitializing them.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

int Status code (rr_ret_status_t)

5.2.2.12 rr_setup_nmt_callback()

```
void rr_setup_nmt_callback (
            rr_can_interface_t * interface,
            rr_nmt_cb_t cb )
```

The function sets a user callback to be intiated in connection with with changes of network management (NMT) states (e.g., a servo connected to/ disconnected from the CAN bus, the interface/ a servo going to the operational state, etc.). User callbacks are functions to execute specific user-defined operations, e.g., to display a warning about an NMT state change or stop the program.

Parameters

interface	Descriptor of the interface (as returned by the rr_init_interface function)
cb	(rr_nmt_cb_t) Type of the callback to be initiated when an NMT event occurs. When the parameter is set to "NULL," the function is disabled.

Returns

void

5.3 Simple motion control (duty, current, velocity, position)

Functions

- int rr_stop_and_release (const rr_servo_t *servo)

    The function sets the specified servo to the released state. The servo is de-energized and stops without retaining its position.
- int rr_stop_and_freeze (const rr_servo_t *servo)

    The function sets the specified servo to the freeze state. The servo stops, retaining its last position.
- int rr_set_current (const rr_servo_t *servo, const float current_a)

    *The function sets the current supplied to the stator of the servo specified in the 'servo' parameter. Changing the 'current_a parameter' value, it is possible to adjust the servo's torque (Torque = stator current*Kt).*
- int rr_set_velocity (const rr_servo_t *servo, const float velocity_deg_per_sec)

    The function sets the velocity at which the specified servo should move at its maximum current. The maximum current is in accordance with the servo motor specification.
- int rr_set_position (const rr_servo_t *servo, const float position_deg)

    The function sets the position that the specified servo should reach as a result of executing the command. The velocity and current are maximum values in accordance with the servo motor specifications. For setting lower velocity and current limits, use the rr_set_position_with_limits function.
- int rr_set_velocity_with_limits (const rr_servo_t *servo, const float velocity_deg_per_sec, const float current_a)

    The function commands the specified servo to rotate at the specified velocity, while setting the maximum limit for the servo current (below the servo motor specifications).
- int rr_set_position_with_limits (const rr_servo_t *servo, const float position_deg, const float velocity_deg_per_sec, const float current_a)

    The function sets the position that the specified servo should reach at user-defined velocity and current as a result of executing the command.
- int rr_set_duty (const rr_servo_t *servo, float duty_percent)

    The function limits the input voltage supplied to the servo, enabling to adjust its motion velocity. For instance, when the input voltage is 20V, setting the duty_percent parameter to 40% will result in 8V supplied to the servo.

5.3.1 Detailed Description

5.3.2 Function Documentation

5.3.2.1 rr_set_current()

```
int rr_set_current (
            const rr_servo_t * servo,
            const float current_a )
```

The function sets the current supplied to the stator of the servo specified in the 'servo' parameter. Changing the 'current_a parameter' value, it is possible to adjust the servo's torque (Torque = stator current*Kt).

Parameters

servo	Servo descriptor returned by the rr_init_servo function
current_a	Phase current of the stator in Amperes

Returns

int Status code (rr_ret_status_t)

5.3.2.2 rr_set_duty()

```
int rr_set_duty (
            const rr_servo_t * servo,
            float duty_percent )
```

The function limits the input voltage supplied to the servo, enabling to adjust its motion velocity. For instance, when the input voltage is 20V, setting the duty_percent parameter to 40% will result in 8V supplied to the servo.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
duty_percent	User-defined percentage of the input voltage to be supplied to the servo

Returns

int Status code (rr_ret_status_t)

5.3.2.3 rr_set_position()

```
int rr_set_position (
            const rr_servo_t * servo,
            const float position_deg )
```

The function sets the position that the specified servo should reach as a result of executing the command. The velocity and current are maximum values in accordance with the servo motor specifications. For setting lower velocity and current limits, use the rr_set_position_with_limits function.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
position_deg	Position of the servo (in degrees) to be reached. The parameter is a multi-turn value (e.g., when set to 720, the servo will make two turns, 360 degrees each). When the parameter is set to a "-" sign value, the servo will rotate in the opposite direction.

Returns

int Status code (rr_ret_status_t)

5.3.2.4 rr_set_position_with_limits()

```
int rr_set_position_with_limits (
            const rr_servo_t * servo,
            const float position_deg,
            const float velocity_deg_per_sec,
            const float current_a )
```

The function sets the position that the specified servo should reach at user-defined velocity and current as a result of executing the command.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
position_deg	Final position of the servo flange (in degrees) to be reached
velocity_deg_per_sec	Velocity (in degrees/sec)at which the servo should move to the specified position
current_a	Maximum user-defined current limit in Amperes

Returns

int Status code (rr_ret_status_t)

5.3.2.5 rr_set_velocity()

```
int rr_set_velocity (
            const rr_servo_t * servo,
            const float velocity_deg_per_sec )
```

The function sets the velocity at which the specified servo should move at its maximum current. The maximum current is in accordance with the servo motor specification.

When you need to set a lower current limit, use the rr_set_velocity_with_limits function.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
velocity_deg_per_sec	Velocity (in degrees/sec) at the servo flange

Returns

int Status code (rr_ret_status_t)

5.3.2.6 rr_set_velocity_with_limits()

```
int rr_set_velocity_with_limits (
            const rr_servo_t * servo,
```

```
            const float velocity_deg_per_sec,
            const float current_a )
```

The function commands the specified servo to rotate at the specified velocity, while setting the maximum limit for the servo current (below the servo motor specifications).

Parameters

servo	Servo descriptor returned by the rr_init_servo function
velocity_deg_per_sec	Velocity (in degrees/sec) at the servo flange. The value can have a "-" sign, in which case the servo will rotate in the opposite direction.
current_a	Maximum user-defined current limit in Amperes.

Returns

int Status code (rr_ret_status_t)

5.3.2.7 rr_stop_and_freeze()

```
int rr_stop_and_freeze (
            const rr_servo_t * servo )
```

The function sets the specified servo to the freeze state. The servo stops, retaining its last position.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

int Status code (rr_ret_status_t)

5.3.2.8 rr_stop_and_release()

```
int rr_stop_and_release (
            const rr_servo_t * servo )
```

The function sets the specified servo to the released state. The servo is de-energized and stops without retaining its position.

Note: When there is an external force affecting the servo (e.g., inertia, gravity), the servo may continue rotating or begin rotating in the opposite direction.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

int Status code (rr_ret_status_t)

5.4 Trajectory motion control (PVT)

Functions

- int rr_add_motion_point (const rr_servo_t ∗servo, const float position_deg, const float velocity_deg_per_sec, const uint32_t time_ms)

  The function enables creating PVT (position-velocity-time) points to set the motion trajectory of the servo specified in the 'servo' parameter. PVT points define the following:

- int rr_start_motion (rr_can_interface_t ∗interface, uint32_t timestamp_ms)

  The function commands all servos connected to the specified interface (CAN bus) to move simultaneously through a number of preset PVT points (see rr_add_motion_point).

- int rr_clear_points_all (const rr_servo_t ∗servo)

  The function clears the entire motion queue of the servo specified in the 'servo' parameter of the function. The servo completes the move it started before the function call and then clears all the remaining PVT points in the queue.

- int rr_clear_points (const rr_servo_t ∗servo, const uint32_t num_to_clear)

  The function removes the number of PVT points indicated in the 'num_to_clear' parameter from the tail of the motion queue preset for the specified servo. When the indicated number of PVT points to be removed exceeds the actual remaining number of PVT points in the queue, the funtion clears only the actual remaining number of PVT points.

- int rr_get_points_size (const rr_servo_t ∗servo, uint32_t ∗num)

  The function returns the actual motion queue size of the specified servo. The return value indicates how many PVT points have already been added to the motion queue.

- int rr_get_points_free_space (const rr_servo_t ∗servo, uint32_t ∗num)

  The function returns how many more PVT points the user can add to the motion queue of the servo specified in the 'servo' parameter.

- int rr_invoke_time_calculation (const rr_servo_t ∗servo, const float start_position_deg, const float start_velocity_deg_per_sec, const float start_acceleration_deg_per_sec2, const uint32_t start_time_ms, const float end_position_deg, const float end_velocity_deg_per_sec, const float end_acceleration_deg_per_sec2, const uint32_t end_time_ms)

  *The function enables calculating the time it will take for the specified servo to get from one position to another at the specified motion parameters (e.g., velocity, acceleration). To read the calculation result, use the the rr_get_time_calculation_result function. **Note:**The function is executed without the servo moving.*

- int rr_get_time_calculation_result (const rr_servo_t ∗servo, uint32_t ∗time_ms)

  The function enables reading the result of the calculations made using the rr_invoke_time_calculation function. It returns the calculated time (in milliseconds) it will take the servo with the specified descriptor to go from one position to another.

5.4.1 Detailed Description

5.4.2 Function Documentation

5.4.2.1 rr_add_motion_point()

```
int rr_add_motion_point (
            const rr_servo_t * servo,
            const float position_deg,
            const float velocity_deg_per_sec,
            const uint32_t time_ms )
```

The function enables creating PVT (position-velocity-time) points to set the motion trajectory of the servo specified in the 'servo' parameter. PVT points define the following:

- what position the servo specified in the 'servo' parameter should reach
- what time the movement to the specified position should take
- how fast the servo should move to the specified position

.
Created PVT points are arranged into a motion queue that defines the motion trajectory of the specified servo. To execute the motion queue, use the rr_start_motion function.
When any of the parameter values (e.g., position, velocity) exceeds user-defined limits or the servo motor specifications (whichever is the smallest value), the function returns an error.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
position_deg	Position that the servo flange (in degrees) should reach as a result of executing the command
velocity_deg_per_sec	Velocity(in degrees/sec) at which the servo should move to reach the specified position
time_ms	Time (in milliseconds) it should take the servo to move from the previous position (PVT point in a motion trajectory or an initial point) to the commanded one. The maximum admissible value is (2^32-1)/10 (roughly equivalent to 4.9 days).

Returns

int Status code (rr_ret_status_t)

5.4.2.2 rr_clear_points()

```
int rr_clear_points (
            const rr_servo_t * servo,
            const uint32_t num_to_clear )
```

The function removes the number of PVT points indicated in the 'num_to_clear' parameter from the tail of the motion queue preset for the specified servo. When the indicated number of PVT points to be removed exceeds the actual remaining number of PVT points in the queue, the funtion clears only the actual remaining number of PVT points.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
num_to_clear	Number of PVT points to be removed from the motion queue of the specified servo

Returns

int Status code (rr_ret_status_t)

5.4.2.3 rr_clear_points_all()

```
int rr_clear_points_all (
            const rr_servo_t * servo )
```

The function clears the entire motion queue of the servo specified in the 'servo' parameter of the function. The servo completes the move it started before the function call and then clears all the remaining PVT points in the queue.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

int Status code (rr_ret_status_t)

5.4.2.4 rr_get_points_free_space()

```
int rr_get_points_free_space (
            const rr_servo_t * servo,
            uint32_t * num )
```

The function returns how many more PVT points the user can add to the motion queue of the servo specified in the 'servo' parameter.

Note: Currently, the maximum motion queue size is 100 PVT.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
num	Pointer to the variable where the function will save the reading

Returns

int Status code (rr_ret_status_t)

5.4.2.5 rr_get_points_size()

```
int rr_get_points_size (
            const rr_servo_t * servo,
            uint32_t * num )
```

The function returns the actual motion queue size of the specified servo. The return value indicates how many PVT points have already been added to the motion queue.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
num	Pointer to the parameter where the function will save the reading

Returns

int Status code (rr_ret_status_t)

5.4.2.6 rr_get_time_calculation_result()

```
int rr_get_time_calculation_result (
            const rr_servo_t * servo,
            uint32_t * time_ms )
```

The function enables reading the result of the calculations made using the rr_invoke_time_calculation function. It returns the calculated time (in milliseconds) it will take the servo with the specified descriptor to go from one position to another.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
time_ms	Pointer to the variable where the function will save the calculated time

Returns

int Status code (rr_ret_status_t)

5.4.2.7 rr_invoke_time_calculation()

```
int rr_invoke_time_calculation (
            const rr_servo_t * servo,
            const float start_position_deg,
```

```
            const float start_velocity_deg_per_sec,
            const float start_acceleration_deg_per_sec2,
            const uint32_t start_time_ms,
            const float end_position_deg,
            const float end_velocity_deg_per_sec,
            const float end_acceleration_deg_per_sec2,
            const uint32_t end_time_ms )
```

The function enables calculating the time it will take for the specified servo to get from one position to another at the specified motion parameters (e.g., velocity, acceleration). To read the calculation result, use the the rr_get_time_calculation_result function. **Note:**The function is executed without the servo moving.

When the start time and the end time parameters are set to 0, the function returns the calculated time value. When the parameters are set to values other than 0, the function will either return OK or an error. 'OK' means the motion at the specified function parameters is possible, whereas an error indicates that the motion cannot be executed.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
start_position_deg	Position (in degrees) from where the specified servo should start moving
start_velocity_deg_per_sec	Servo velocity (in degrees/sec) at the start of motion
start_acceleration_deg_per_sec2	Servo acceleration (in degrees/sec^2) at the start of motion
start_time_ms	Initial time setting (in milliseconds)
end_position_deg	Position (in degrees) where the servo should arrive
end_velocity_deg_per_sec	Servo velocity (in degrees/sec) in the end of motion
end_acceleration_deg_per_sec2	Servo acceleration (in degrees/sec^2) in the end of motion
end_time_ms	Final time setting (in milliseconds)

Returns

int Status code (rr_ret_status_t)

5.4.2.8 rr_start_motion()

```
int rr_start_motion (
            rr_can_interface_t * interface,
            uint32_t timestamp_ms )
```

The function commands all servos connected to the specified interface (CAN bus) to move simultaneously through a number of preset PVT points (see rr_add_motion_point).

Note: When any of the servos fails to reach any of the PVT points due to an error, it will broadcast a "Go to Stopped State" command to all the other servos on the same bus. The servos will stop executing the preset PVT points and go to the stopped state. In the state, only Heartbeats are available. You can neither communicate with servos nor command them to execute any operations.

Note: Once servos execute the last PVT in their preset motion queue, the queue is cleared automatically.

Parameters

interface	Interface descriptor returned by the rr_init_interface function
timestamp_ms	Delay (in milliseconds) before the servos associated with the interface start to move. When the value is set to 0, the servos will start moving immediately. The available value range is from 0 to 2^24-1.

Returns

int Status code (rr_ret_status_t)

5.5 Reading and writing servo configuration

Functions

- int rr_set_zero_position (const rr_servo_t *servo, const float position_deg)

  The function enables setting the current position (in degrees) of the servo with the specified descriptor to any value defined by the user. For instance, when the current servo position is 101 degrees and the 'position_deg' parameter is set to 25 degrees, the servo is assumed to be positioned at 25 degrees.
- int rr_set_zero_position_and_save (const rr_servo_t *servo, const float position_deg)

  The function enables setting the current position (in degrees) of the servo with the specified descriptor to any value defined by the user and saving it to the FLASH memory. If you don't want to save the newly set position, use the rr_set_zero_position function.
- int rr_get_max_velocity (const rr_servo_t *servo, float *velocity_deg_per_sec)

  The function reads the maximum velocity of the servo at the current moment. It returns the smallest of the three values—the user-defined maximum velocity limit (rr_set_max_velocity), the maximum velocity value based on the servo specifications, or the calculated maximum velocity based on the supply voltage.
- int rr_set_max_velocity (const rr_servo_t *servo, const float max_velocity_deg_per_sec)

  The function sets the maximum velocity limit for the servo specified in the 'servo' parameter. The setting is volatile: after a reset or a power outage, it is no longer valid.

5.5.1 Detailed Description

5.5.2 Function Documentation

5.5.2.1 rr_get_max_velocity()

```
int rr_get_max_velocity (
            const rr_servo_t * servo,
            float * velocity_deg_per_sec )
```

The function reads the maximum velocity of the servo at the current moment. It returns the smallest of the three values—the user-defined maximum velocity limit (rr_set_max_velocity), the maximum velocity value based on the servo specifications, or the calculated maximum velocity based on the supply voltage.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
velocity_deg_per_sec	Maximum servo velocity (in degrees/sec)

Returns

int Status code (rr_ret_status_t)

5.5.2.2 rr_set_max_velocity()

```
int rr_set_max_velocity (
            const rr_servo_t * servo,
            const float max_velocity_deg_per_sec )
```

The function sets the maximum velocity limit for the servo specified in the 'servo' parameter. The setting is volatile: after a reset or a power outage, it is no longer valid.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
max_velocity_deg_per_sec	Velocity at the servo flange (in degrees/sec)

Returns

int Status code (rr_ret_status_t)

5.5.2.3 rr_set_zero_position()

```
int rr_set_zero_position (
            const rr_servo_t * servo,
            const float position_deg )
```

The function enables setting the current position (in degrees) of the servo with the specified descriptor to any value defined by the user. For instance, when the current servo position is 101 degrees and the 'position_deg' parameter is set to 25 degrees, the servo is assumed to be positioned at 25 degrees.

The setting is volatile: after a reset or a power outage, it is no longer valid.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
position_deg	User-defined position (in degrees) to replace the current position value

Returns

int Status code (rr_ret_status_t)

5.5.2.4 rr_set_zero_position_and_save()

```
int rr_set_zero_position_and_save (
            const rr_servo_t * servo,
            const float position_deg )
```

The function enables setting the current position (in degrees) of the servo with the specified descriptor to any value defined by the user and saving it to the FLASH memory. If you don't want to save the newly set position, use the rr_set_zero_position function.

Note: The FLASH memory limit is 1,000 write cycles. Therefore, it is not advisable to use the function on a regular basis.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
position_deg	User-defined position (in degrees) to replace the current position value

Returns

int Status code (rr_ret_status_t)

5.6 Reading realtime parameter

Functions

- int rr_param_cache_update (rr_servo_t *servo)

  The function is always used in combination with the rr_param_cache_setup_entry function. It retreives from the servo the array of parameters set up using rr_param_cache_setup_entry function and saves the array to the program cache. You can subsequently read the parameters from the program cache with the rr_read_cached_parameter function. For more information, see rr_param_cache_setup_entry.
- int rr_param_cache_setup_entry (rr_servo_t *servo, const rr_servo_param_t param, bool enabled)

  *The function is the fist one in the API call sequence that enables reading multiple servo paramaters (e.g., velocity, voltage, and position) as a data array. Using the sequence is advisable when you need to read **more than one parameter at a time**. The user can set up the array to include up to 50 parameters. In all, the sequence comprises the following functions:*
- int rr_read_parameter (rr_servo_t *servo, const rr_servo_param_t param, float *value)

  The function enables reading a single parameter directly from the servo specified in the 'servo' parameter of the function. The function returns the current value of the parameter. Additionally, the parameter is saved to the program cache, irrespective of whether it was enabled/ disabled with the rr_param_cache_setup_entry function.
- int rr_read_cached_parameter (rr_servo_t *servo, const rr_servo_param_t param, float *value)

  The function is always used in combination with the rr_param_cache_setup_entry and the::rr_param_cache_update functions. For more information, see rr_param_cache_setup_entry.

5.6.1 Detailed Description

5.6.2 Function Documentation

5.6.2.1 rr_param_cache_setup_entry()

```
int rr_param_cache_setup_entry (
            rr_servo_t * servo,
            const rr_servo_param_t param,
            bool enabled )
```

The function is the fist one in the API call sequence that enables reading multiple servo paramaters (e.g., velocity, voltage, and position) as a data array. Using the sequence is advisable when you need to read **more than one parameter at a time**. The user can set up the array to include up to 50 parameters. In all, the sequence comprises the following functions:

- rr_param_cache_setup_entry for setting up an array of servo parameters to read
- rr_param_cache_update for retreiving the parameters from the servo and saving them to the program cache
- rr_read_cached_parameter for reading parameters from the program cache

Using the sequence of API calls allows for speeding up data acquisition by nearly two times. Let's assume you need to read 49 parameters. At a bit rate of 1 MBit/s, reading them one by one will take about 35 ms, whereas reading them as an array will only take 10 ms.

Note: When you need to read a single parameter, it is better to use the rr_read_parameter function.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
param	Index of the parameter to read as indicated in the rr_servo_param_t list (e.g., APP_PARAM_POSITION_ROTOR)
enabled	Set True/False to enable/ disable the specified parameter for reading

Returns

int Status code (rr_ret_status_t)

5.6.2.2 rr_param_cache_update()

```
int rr_param_cache_update (
            rr_servo_t * servo )
```

The function is always used in combination with the rr_param_cache_setup_entry function. It retreives from the servo the array of parameters set up using rr_param_cache_setup_entry function and saves the array to the program cache. You can subsequently read the parameters from the program cache with the rr_read_cached_parameter function. For more information, see rr_param_cache_setup_entry.

Note: After you exit the program, the cache will be cleared.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

int Status code (rr_ret_status_t)

5.6.2.3 rr_read_cached_parameter()

```
int rr_read_cached_parameter (
            rr_servo_t * servo,
            const rr_servo_param_t param,
            float * value )
```

The function is always used in combination with the rr_param_cache_setup_entry and the::rr_param_cache_update functions. For more information, see rr_param_cache_setup_entry.

The function enables reading parameters from the program cache. If you want to read more than one parameter, you will need to make a separate API call for each of them.

Note: Prior to reading a parameter, make sure to update the program cache using the rr_param_cache_update function.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
param	Index of the parameter to read; you can find these indices in the rr_servo_param_t list (e.g., APP_PARAM_POSITION_ROTOR)
value	Pointer to the variable where the function will save the reading

Returns

int Status code (rr_ret_status_t)

5.6.2.4 rr_read_parameter()

```
int rr_read_parameter (
            rr_servo_t * servo,
            const rr_servo_param_t param,
            float * value )
```

The function enables reading a single parameter directly from the servo specified in the 'servo' parameter of the function. The function returns the current value of the parameter. Additionally, the parameter is saved to the program cache, irrespective of whether it was enabled/ disabled with the rr_param_cache_setup_entry function.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
param	Index of the parameter to read; you can find these indices in the rr_servo_param_t list (e.g., APP_PARAM_POSITION_ROTOR).
value	Pointer to the variable where the function will save the reading

Returns

int Status code (rr_ret_status_t)

5.7 Error handling

Functions

- void rr_setup_emcy_callback (rr_can_interface_t *interface, rr_emcy_cb_t cb)

  The function sets a user callback to be intiated in connection with emergency (EMCY) events (e.g., overcurrent, power outage, etc.). User callbacks are functions to execute specific user-defined operations, e.g., to display a warning about an EMCY event or stop the program.
- const char * rr_describe_emcy_bit (uint8_t bit)

  The function returns a string describing in detail a specific EMCY event based on the code in the 'bit' parameter (e.g., "CAN bus warning limit reached"). The function can be used in combination with rr_describe_emcy_code. The latter provides a more generic description of an EMCY event.
- const char * rr_describe_emcy_code (uint16_t code)

  The function returns a string descibing a specific EMCY event based on the error code in the 'code' parameter. The description in the string is a generic type of the occured emergency event (e.g., "Temperature"). For a more detailed description, use the function together with the rr_describe_emcy_bit one.
- int rr_read_error_status (const rr_servo_t *servo, uint32_t *const error_count, uint8_t *const error_array)

  The functions enables reading the total current count of servo errors and their codes.

5.7.1 Detailed Description

5.7.2 Function Documentation

5.7.2.1 rr_describe_emcy_bit()

```
const char* rr_describe_emcy_bit (
            uint8_t bit )
```

The function returns a string describing in detail a specific EMCY event based on the code in the 'bit' parameter (e.g., "CAN bus warning limit reached"). The function can be used in combination with rr_describe_emcy_code. The latter provides a more generic description of an EMCY event.

Parameters

bit	Error bit field of the corresponding EMCY message (according to the CanOpen standard)

Returns

Pointer to the description string

5.7.2.2 rr_describe_emcy_code()

```
const char* rr_describe_emcy_code (
            uint16_t code )
```

The function returns a string descibing a specific EMCY event based on the error code in the 'code' parameter. The description in the string is a generic type of the occured emergency event (e.g., "Temperature"). For a more detailed description, use the function together with the rr_describe_emcy_bit one.

Parameters

code	Error code from the corresponding EMCY message (according to the CanOpen standard)

Returns

Pointer to the description string

5.7.2.3 rr_read_error_status()

```
int rr_read_error_status (
            const rr_servo_t * servo,
            uint32_t *const error_count,
            uint8_t *const error_array )
```

The functions enables reading the total current count of servo errors and their codes.

Parameters

servo	Servo Servo descriptor returned by the rr_init_servo function
error_count	Pointer to the variable where the function will save the current servo error count
error_array	Pointer to the array where the function will save the codes of all errors**Note:** Call the rr_describe_emcy_bit function, to get a detailed error code description. If the array is not used, set the parameter to 0.

Returns

int Status code (rr_ret_status_t)

5.7.2.4 rr_setup_emcy_callback()

```
void rr_setup_emcy_callback (
            rr_can_interface_t * interface,
            rr_emcy_cb_t cb )
```

The function sets a user callback to be intiated in connection with emergency (EMCY) events (e.g., overcurrent, power outage, etc.). User callbacks are functions to execute specific user-defined operations, e.g., to display a warning about an EMCY event or stop the program.

Parameters

interface	Descriptor of the interface (as returned by the rr_init_interface function)
cb	(rr_emcy_cb_t) Type of the callback to be initiated when an NMT event occurs. When the parameter is set to "NULL," the function is disabled.

Returns

void

5.8 Debugging

Functions

- void rr_set_comm_log_stream (const rr_can_interface_t *interface, FILE *f)

    The function sets a stream for saving CAN communication dump from the specified interface. Subsequently, the user can look through the logs saved to the stream to identify causes of CAN communication failures.
- void rr_set_debug_log_stream (FILE *f)

    The function sets a stream for saving the debugging messages generated by the API library. Subsequently, the user can look through the logs to identify and locate the events associated with certain problems.

5.8.1 Detailed Description

5.8.2 Function Documentation

5.8.2.1 rr_set_comm_log_stream()

```
void rr_set_comm_log_stream (
            const rr_can_interface_t * interface,
            FILE * f )
```

The function sets a stream for saving CAN communication dump from the specified interface. Subsequently, the user can look through the logs saved to the stream to identify causes of CAN communication failures.

Parameters

interface	Descriptor of the interface where the logged CAN communication occurs (returned by the rr_init_interface function)
f	stdio stream for saving the communication log. When the parameter is set to "NULL," logging of CAN communication events in the interface is disabled.

Returns

void

5.8.2.2 rr_set_debug_log_stream()

```
void rr_set_debug_log_stream (
            FILE * f )
```

The function sets a stream for saving the debugging messages generated by the API library. Subsequently, the user can look through the logs to identify and locate the events associated with certain problems.

Parameters

f	stdio stream for saving the debugging log. When the parameter is set to "NULL," logging of debugging events is disabled.

Returns

void

5.9 Auxiliary functions

Functions

- void rr_sleep_ms (int ms)

  The function sets an idle period for the user program (e.g., to wait till a servo executes a motion trajectory). Until the period expires, the user program will not execute any further operations. However, the network management, CAN communication, emergency, and Heartbeat functions remain available.

5.9.1 Detailed Description

5.9.2 Function Documentation

5.9.2.1 rr_sleep_ms()

```
void rr_sleep_ms (
            int ms )
```

The function sets an idle period for the user program (e.g., to wait till a servo executes a motion trajectory). Until the period expires, the user program will not execute any further operations. However, the network management, CAN communication, emergency, and Heartbeat functions remain available.

Parameters

ms	Idle period (in milleseconds)

Returns

void

5.10 Servo box specs & manual

5.10.1 1. Product overview

A servobox is a solution designed to control motion of one or more RDrive servos. The solution comprises the following components:

- one or more energy eaters (see Section 3.1)
- one or more capacitor modules (see Section 3.2)
- a CAN-USB dongle to provide CANOpen communication between the servobox and the servos

Additionally, to ensure operation of the servobox, the user has to provide a power supply and USB-A to Micro USB cable to connect the CAN-USB dongle to PC.

The power supply should meet the following requirements:

- its supply voltage should be 48 V
- its power should be equal to the total peak power of all servo motors connected to it

5.10.2 2. Integrating servos with a power supply and a servobox

To integrate a RDrive servo into one circuit with a power supply and a servobox, you need to provide the following connections:

- power supply connection (two wires on the servo housing)
- CAN communication connection (two wires on the servo housing)

For connection diagrams and requirements, see Sections 2.1 and 2.2. For eater and capacitor requirements and schematic, see Section 3.1 and 3.2.

5.10.2.1 2.1. Power supply connection

Note: Never supply power before a servo (servos) is (are) fully integrated with a servobox and a power supply into one circuit. Charging current of the capacitor(s) can damage the power supply or injure the user!

The configuration of the servo box solution (e.g., how many eaters and capacitors it uses) and the electrical connection diagram depend on whether your intention is:

- to connect a single servo, in which case the configuration and the connection diagram are as below:
- to connect multiple servos, in which case the configuration and the connection diagram are as below:

In any case, make sure to meet the following electrical connection requirements:

- Typically, the total circuit length from the power supply to any servo motor must not exceed 10 meters.
- Length "L1" must not be longer than 10 meters.
- Length "L2" (from the eater to the capacitor) should not exceed the values from Table 1.
- Length "L3" (from the capacitor to any servo) should not exceed the values from Table 1.

Table 1: Line segment lengths vs. cross-sections

Servo model	L2						L3					
	0.75 mm2	1.0 mm2	1.5 mm2	2.5 mm2	4.0 mm2	6.0 mm2	0.75 mm2	1.0 mm2	1.5 mm2	2.5 mm2	4.0 mm2	6.0 mm2
R↩ D50	4 m	5 m	8 m	10 m	10 m	10 m	0,2 m	0,2 m	0,4 m	0,7 m	1,0 m	1,0 m
R↩ D60	2 m	3 m	5 m	9 m	10 m	10 m	0,1 m	0,1 m	0,2 m	0,4 m	1,0 m	1,0 m
R↩ D85	0,8 m	1 m	1 m	2 m	4 m	6 m	0,04 m	0,05 m	0,08 m	0,13 m	0,21 m	0,32 m

For length 1, make sure the cable cross-section is as specified below:

- When the total connected motor power is **less than 250 W**, the cable cross-section within the segment must be at least 1.00 mm2.
- When the total connected motor power is **less than 500 W**, the cable cross-section within the segment must be at least 2.00 mm2.

5.10.2.2 2.2. CAN connection

The CAN connection of RDrive servos is a two-wire bus line transmitting differential signals: CAN_HIGH and CA↩ N_LOW. The configuration of the bus line is as illustrated below:

Providing the CAN connection, make sure to comply with the following requirements:

- The CAN bus lines should be terminated with 120 Ohm resistors at both ends. You have to provide only one resistor because one is already integrated into the CAN-USB dongle supplied as part of the servobox solution.
- The bus line cable must be a twisted pair cable with the lay length of 2 to 4 cm.
- The cross section of the bus line cable must be between 0.12 mm2 to 0.3 mm2.
- To ensure the baud rate required for your application, LΣ should meet the specific values as indicated in Table 2.

Table 2: CAN line length vs. baud rate

Baud Rate	50 kbit/s	100 kbit/s	250 kbit/s	500 kbit/s	1 Mbit/s
Total line length, LΣ, m	$<$ 1000 m	$<$ 500 m	$<$ 200 m	$<$ 100 m	$<$ 40 m

5.10.3 3. Servobox components

5.10.3.1 3.1 Energy eater

An energy eater is used to dissipate the dynamic braking energy that can result from servos generating voltages in excess of the power supply voltage. Use the schematic below to assemble the device: **Required components:**

Component	Type	Other options	Comment
D1 - Diode	APT30S20BG	Schottky diode, $I_f \geq 20$ A, $V_r \geq$ 96 V	$I_f \geq 1.5 \times$ Total current of all connected servos
Q1 - Transistor	TIP147	PNP darlington transistor, $V_{ce} \geq$ 96V, $I_c \geq 10$ A	
R1 - Resistor 1	1K Ohm, 1 W		
R2 - Resistor 1	4.7 Ohm, $P_d \geq 25$ W		

Note: D1, Q1, and R2 should be connected to an appropriate heatsink. The maximum dissipated power of the heatsink should be equal to the maximum dynamic braking energy in your circuit. When the power to dissipate is too high (dynamic braking power is more than 50 W), it also is essential to provide active cooling, such as a fan.

5.10.3.2 3.2 Capacitor module

In the servobox solution, capacitors are intended to accumulate and supply electric energy to servos. The devices allow for compensating short-duration power consumption peaks that are due to servos located at a distance (usually quite long) from the power supply unit. For the same reason, make sure to place capacitors as close as possible to the servo. To assemble the device, use the schematic below. **Requirements:**

Component	Type	Comment
C1...Cn	Aluminum electrolytic capacitor or tantalum/polymer capacitor, $U \geq 80$ V, ESR ≤ 0.1 Ohm	Total capacitance should be ≥ 5 uF per 1 W of connected servo

5.11 One servo PVT move

The tutorial describes how to set up and execute a motion trajectory for one servo. In this example, the motion trajectory comprises two PVT (position-time-velocity) points:

- one PVT commanding the servo to move to the position of 100 degrees in 6,000 milliseconds
- one PVT commanding the servo to move to the position of -100 degrees in 6,000 milliseconds

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
```

2. Initialize the servo.

```
rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
```

3. Clear the motion queue.

```
rr_clear_points_all(servo);
```

Adding PVT points to form a motion queue

4. Set the first PVT point, commanding the servo to move to the position of 100 degrees in 6,000 milliseconds. **Note**: When a point is added successfully to the motion queue, the function will return OK. Otherwise, the function returns an error warning and quits the program.

```
int status = rr_add_motion_point(servo, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

5. Set the second PVT point, commanding the servo to move to the position of -100 degrees in 6,000 milliseconds. **Note**: When a point is added successfully to the motion queue, the function will return OK. Otherwise, the function returns an error warning and quits the program.

```
status = rr_add_motion_point(servo, -100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

Executing the resulting motion queue

6. Command the servo to move through the PVT points you added to the motion queue. Set the function parameter to 0 to get the servo moving without a delay.

```
rr_start_motion(iface, 0);
```

7. To ensure the program will not move on to execute another operation, set an idle period of 14,000 milliseconds.

```
rr_sleep_ms(14000); // wait till the movement ends
```

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);

rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);

API_DEBUG("========== Tutorial of the %s ==========\n", "controlling one servo");

rr_clear_points_all(servo);
API_DEBUG("Appending points\n");
int status = rr_add_motion_point(servo, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
status = rr_add_motion_point(servo, -100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
rr_start_motion(iface, 0);
rr_sleep_ms(14000); // wait till the movement ends
```

5.12 Two servos PVT move

The tutorial describes how to set up motion trajectories for two servos and to execute them simultaneously. In this example, each motion trajectory comprises two PVT (position-time-velocity) points:

- one PVT commanding servos to move to the position of 100 degrees in 6,000 milliseconds
- one PVT commanding servos to move to the position of -100 degrees in 6,000 milliseconds

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
```

2. Initialize servo 1.

```
rr_servo_t *servo1 = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
```

3. Initialize servo 2.

```
rr_servo_t *servo2 = rr_init_servo(iface,
  TUTORIAL_SERVO_1_ID);
```

4. Clear the motion queue of servo 1.

```
rr_clear_points_all(servo1);
```

5. Clear the motion queue of servo 2.

```
rr_clear_points_all(servo2);
```

Adding PVT points to form motion queues

6. Set the first PVT point for servo 1, commanding it to move to the position of 100 degrees in 6,000 milliseconds.

```
int status = rr_add_motion_point(servo1, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

7. Set the first PVT point for servo 2, commanding it to move to the position of 100 degrees in 6,000 milliseconds.

```
status = rr_add_motion_point(servo2, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

8. Set the second PVT point for servo 1, commanding it to move to the position of -100 degrees in 6,000 milliseconds.

```
status = rr_add_motion_point(servo1, -100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

9. Set the second PVT point for servo 2, commanding it to move to the position of -100 degrees in 6,000 milliseconds.

```
status = rr_add_motion_point(servo2, -100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

Executing the resulting motion queues

10. Command all servos to move simultaneously. Each of the two servos will execute their preset motion queues. Set the function parameter to 0 to get the servos moving without a delay.

```
rr_start_motion(iface, 0);
```

11. To ensure the program will not move on to execute another operation, set an idle period of 14,000 milliseconds.

```
rr_sleep_ms(14000); //wait till the movement ends
```

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
rr_servo_t *servo1 = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
rr_servo_t *servo2 = rr_init_servo(iface,
  TUTORIAL_SERVO_1_ID);

API_DEBUG("========== Tutorial of the %s ==========\n", "controlling two servos");

rr_clear_points_all(servo1);
rr_clear_points_all(servo2);

int status = rr_add_motion_point(servo1, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
status = rr_add_motion_point(servo2, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
status = rr_add_motion_point(servo1, -100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
status = rr_add_motion_point(servo2, -100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
rr_start_motion(iface, 0);

rr_sleep_ms(14000); //wait till the movement ends
```

5.13 Three servos PVT move

The tutorial describes how to set up motion trajectories for three servos and to execute them simultaneously. In this example, each motion trajectory comprises two PVT (position-time-velocity) points:

- one PVT commanding servos to move to the position of 100 degrees in 6,000 milliseconds
- one PVT commanding servos to move to the position of -100 degrees in 6,000 milliseconds

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
```

2. Initialize servo 1.

```
rr_servo_t *servo1 = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
```

3. Initialize servo 2.

```
rr_servo_t *servo2 = rr_init_servo(iface,
  TUTORIAL_SERVO_1_ID);
```

4. Initialize servo 3.

```
rr_servo_t *servo3 = rr_init_servo(iface,
  TUTORIAL_SERVO_2_ID);
```

5. Clear points servo ID0.

```
rr_clear_points_all(servo1);
```

6. Clear points servo ID1.

```
rr_clear_points_all(servo2);
```

7. Clear points servo ID2.

```
rr_clear_points_all(servo3);
```

Adding PVT ponts to form motion queues

8. Set the first PVT point for servo 1, commanding it to move to the position of 100 degrees in 6,000 milliseconds.

```
int status = rr_add_motion_point(servo1, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

9. Set the first PVT point for servo 2, commanding it to move to the position of 100 degrees in 6,000 milliseconds.

```
status = rr_add_motion_point(servo2, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

10. Set the first PVT point for servo 3, commanding it to move to the position of 100 degrees in 6,000 milliseconds.

```
status = rr_add_motion_point(servo3, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

11. Set the second PVT point for servo 1, commanding it to move to the position of -100 degrees in 6,000 milliseconds.

```
status = rr_add_motion_point(servo1, -100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

12. Set the second PVT point for servo 2, commanding it to move to the position of -100 degrees in 6,000 milliseconds.

```
status = rr_add_motion_point(servo2, -100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

13. Set the second PVT point for servo 3, commanding it to move to the position of -100 degrees in 6,000 milliseconds.

```
status = rr_add_motion_point(servo3, -100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
```

Executing the resulting motion queues

14. Command all servos to start moving simulateneously. Each of the three servos will execute their own motion queues. Set the function parameter to 0 to get the servos moving without a delay.

```
rr_start_motion(iface, 0);
```

15. To ensure the program will not move on to execute another operation, set an idle period of 14,000 milliseconds.

```
rr_sleep_ms(14000); // wait till the movement ends
```

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
rr_servo_t *servo1 = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
rr_servo_t *servo2 = rr_init_servo(iface,
  TUTORIAL_SERVO_1_ID);
rr_servo_t *servo3 = rr_init_servo(iface,
  TUTORIAL_SERVO_2_ID);

API_DEBUG("========== Tutorial of the %s ==========\n", "controlling three servos");

rr_clear_points_all(servo1);
rr_clear_points_all(servo2);
rr_clear_points_all(servo3);
int status = rr_add_motion_point(servo1, 100.0, 0.0, 6000);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation: %d\n", status);
    return 1;
}
status = rr_add_motion_point(servo2, 100.0, 0.0, 6000);
if(status != RET_OK)
{
```

```
        API_DEBUG("Error in the trjectory point calculation: %d\n", status);
        return 1;
    }
    status = rr_add_motion_point(servo3, 100.0, 0.0, 6000);
    if(status != RET_OK)
    {
        API_DEBUG("Error in the trjectory point calculation: %d\n", status);
        return 1;
    }
    status = rr_add_motion_point(servo1, -100.0, 0.0, 6000);
    if(status != RET_OK)
    {
        API_DEBUG("Error in the trjectory point calculation: %d\n", status);
        return 1;
    }
    status = rr_add_motion_point(servo2, -100.0, 0.0, 6000);
    if(status != RET_OK)
    {
        API_DEBUG("Error in the trjectory point calculation: %d\n", status);
        return 1;
    }
    status = rr_add_motion_point(servo3, -100.0, 0.0, 6000);
    if(status != RET_OK)
    {
        API_DEBUG("Error in the trjectory point calculation: %d\n", status);
        return 1;
    }
    rr_start_motion(iface, 0);

    rr_sleep_ms(14000); // wait till the movement ends
```

5.14 Device parameter cache programming and reading

This tutorial describes how to set up an array of servo parameters, save them to the program cache in one operation, and then read them one by one from the cache. In this example, we will work with four parameters: rotor position, rotor velocity, input voltage, and input current. **Note** : In general, it is advisable to use the function, when you need to read **more than one parameter** from the servo. If you need to read a single parameter, use the rr_read_parameter function (refer to the Reading device parameters tutorial).

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
```

2. Initialize the servo.

```
rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
```

Setting up and saving an array of parameters to the program cache

3. Add parameter 1 (rotor position) to the array of parameters you want to read from the servo.

```
rr_param_cache_setup_entry(servo,
  APP_PARAM_POSITION_ROTOR, true);
```

4. Add parameter 2 (rotor velocity) to the array of parameters you want to read from the servo.

```
rr_param_cache_setup_entry(servo,
  APP_PARAM_VELOCITY_ROTOR, true);
```

5. Add parameter 3 (input voltage) to the array of parameters you want to read from the servo.

```
rr_param_cache_setup_entry(servo,
  APP_PARAM_VOLTAGE_INPUT, true);
```

6. Add parameter 4 (input current) to the array of parameters you want to read from the servo.

```
rr_param_cache_setup_entry(servo,
  APP_PARAM_CURRENT_INPUT, true);
```

7. Save the parameters to the program cache.

```
rr_param_cache_update(servo);
```

Reading the parameters from the cache

8. Create a variable where the function will read the parameters from the cache.

```
float value;
```

9. Read parameter 1 (rotor position) from the cache.

```
rr_read_cached_parameter(servo,
  APP_PARAM_POSITION_ROTOR, &value);
```

10. Read parameter 2 (rotor velocity) from the cache.

```
rr_read_cached_parameter(servo,
  APP_PARAM_VELOCITY_ROTOR, &value);
```

11. Read parameter 3 (input voltage) from the cache.

```
rr_read_cached_parameter(servo,
  APP_PARAM_VOLTAGE_INPUT, &value);
```

12. Read parameter 4 (input current) from the cache.

```
rr_read_cached_parameter(servo,
  APP_PARAM_CURRENT_INPUT, &value);
```

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(/*TUTORIAL_DEVICE*/ "
  /dev/ttyS3");

rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error count");

uint32_t _size;
rr_read_error_status(servo, &_size, 0);
API_DEBUG("\tError count: %d %s\n", _size, _size ? "" : "(No errors)");

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error list");
uint8_t array[100];
uint32_t size;
rr_read_error_status(servo, &size, array);
API_DEBUG("\tError count: %d %s\n", size, size ? "" : "(No errors)");
for(int i = 0; i < size; i++)
{
    API_DEBUG("\t\tError: %s\n", rr_describe_emcy_bit(array[i]));
}
```

5.15 Device parameters reading

The tutorial describes how to read a sequence of single variables representing current device parameters (e.g., position, voltage, etc.) **Note** : For reference, the tutorial includes more than one parameter. In practice, however, if you need to read more than one parameter, refer to the tutorial **Setting up cache and reading cached parameters**.

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
```

2. Initialize the servo.

```
rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
```

Reading current device parameters

3. Create a variable where the function will save the parameters.

```
float value;
```

4. Read the rotor position.

```
rr_read_parameter(servo, APP_PARAM_POSITION_ROTOR, &value);
```

5. Read the rotor velocity.

```
rr_read_parameter(servo, APP_PARAM_VELOCITY_ROTOR, &value);
```

6. Read the input voltage.

```
rr_read_parameter(servo, APP_PARAM_VOLTAGE_INPUT, &value);
```

7. Read the input current.

```
rr_read_parameter(servo, APP_PARAM_CURRENT_INPUT, &value);
```

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(/*TUTORIAL_DEVICE*/ "
  /dev/ttyS3");

rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error count");

uint32_t _size;
rr_read_error_status(servo, &_size, 0);
API_DEBUG("\tError count: %d %s\n", _size, _size ? "" : "(No errors)");

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error list");
uint8_t array[100];
uint32_t size;
rr_read_error_status(servo, &size, array);
API_DEBUG("\tError count: %d %s\n", size, size ? "" : "(No errors)");
for(int i = 0; i < size; i++)
{
    API_DEBUG("\t\tError: %s\n", rr_describe_emcy_bit(array[i]));
}
```

5.16 Servo PVT point calculation

This tutorial describes how you can calculate and read the minimum time that it will take the servo to reach the position of 100 degrees. **Note:** Following the instructions in the tutorial, you can get the said travel time value without actually moving the servo.

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
```

2. Initialize the servo.

```
rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
```

Calculating the time to reach the specified position

3. Calculate the time it will take the servo to reach the position of 100 degrees when the other parameters are set to 0. The calculation result is the minumum time value.

```
int status = rr_invoke_time_calculation(servo, 0.0, 0.0, 0.0, 0, 100.0, 0.0,
  0.0, 0);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation\n");
    return 1;
}
```

Reading the calculation result

4. Create a variable where the function will return the calculation result.

```
uint32_t travel_time;
```

5. Read the calculation result.

```
rr_get_time_calculation_result(servo, &travel_time);
```

5.17 Reading of the maximum velocity

This tutorial describes how to read the maximum velocity at which the servo can move at the current moment.
Note: The function will return the least of the three limits: the servo motor specifications, the user-defined maximum velocity limit (see rr_set_velocity_with_limits), or the calculated value based on the input voltage.

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
```

2. Initialize the servo.

```
rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
```

Reading the maximum servo velocity

3. Create a variable where the function will return the maximum servo velocity.

```
float velocity;
```

4. Read the current maximum velocity value.

```
rr_get_max_velocity(servo, &velocity);
```

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(/*TUTORIAL_DEVICE*/ "
  /dev/ttyS3");

rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error count");

uint32_t _size;
rr_read_error_status(servo, &_size, 0);
API_DEBUG("\tError count: %d %s\n", _size, _size ? "" : "(No errors)");

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error list");
uint8_t array[100];
uint32_t size;
rr_read_error_status(servo, &size, array);
API_DEBUG("\tError count: %d %s\n", size, size ? "" : "(No errors)");
for(int i = 0; i < size; i++)
{
    API_DEBUG("\t\tError: %s\n", rr_describe_emcy_bit(array[i]));
}
```

5.18 Reading of the motion queue parameters

This tutorial describes how to determine the current size of a motion queue. In this example, we will read the number of free and occupied PVT points in a motion queue before and after adding PVT (position-velocity-time) points to the motion queue. **Note:** Currently, the maximum motion queue size is 100 PVT.

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(
  TUTORIAL_DEVICE);
```

2. Initialize the servo.

```
rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
```

3. Clear the motion queue of the servo.

```
rr_clear_points_all(servo);
```

Reading the initial motion queue size

4. Create a variable where the function will save the motion queue size values (free and occupied PVT points).

```
uint32_t num;
```

5. Read how many PVT points have been already added to the motion queue.

```
rr_get_points_size(servo, &num);
```

6. Read how many more PVT points can be added to the motion queue.

```
rr_get_points_free_space(servo, &num);
```

Reading the motion queue size after adding new PVT points to the motion queue

7. Add PVT point 1 to the motion queue, setting the time parameter to 10000000 ms.

```
rr_add_motion_point(servo, 0.0, 0.0, 10000000);
```

8. Add PVT point 2 to the motion queue, setting the time parameter to 10000000 ms.

```
rr_add_motion_point(servo, 0.0, 0.0, 10000000);
```

9. Read how many PVT points are already in the motion queue.

```
rr_get_points_size(servo, &num);
```

10. Read how many more PVT points can be added to the motion queue.

```
rr_get_points_free_space(servo, &num);
```

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(/*TUTORIAL_DEVICE*/ "
  /dev/ttyS3");

rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error count");

uint32_t _size;
rr_read_error_status(servo, &_size, 0);
API_DEBUG("\tError count: %d %s\n", _size, _size ? "" : "(No errors)");

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error list");
uint8_t array[100];
uint32_t size;
rr_read_error_status(servo, &size, array);
API_DEBUG("\tError count: %d %s\n", size, size ? "" : "(No errors)");
for(int i = 0; i < size; i++)
{
    API_DEBUG("\t\tError: %s\n", rr_describe_emcy_bit(array[i]));
}
```

5.19 Reading device errors

The tutorial describes how to read the total number of errors that occurred on the servo and to display their description.

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(/*TUTORIAL_DEVICE*/ "
  /dev/ttyS3");
```

1. Initialize the servo.

```
rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);
```

Reading the current error count

2. Create a variable where the function will read the current error count.

```
uint32_t _size;
```

3. Read the current error count. **Note**: The "array" argument is zero (we don't need to read error bits).

```
rr_read_error_status(servo, &_size, 0);
```

Reading the current error count and error bits

4. Create an array where the function will read the current error bits.

```
uint8_t array[100];
```

5. Create a variable where the function will read the current error count.

```
uint32_t size;
```

6. Read the current error count and error bits (if any).

```
rr_read_error_status(servo, &_size, 0);
```

7. Cycle print of error bits (described by rr_describe_emcy_bit function).

```
for(int i = 0; i < size; i++)
{
    API_DEBUG("\t\tError: %s\n", rr_describe_emcy_bit(array[i]));
}
```

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(/*TUTORIAL_DEVICE*/ "
  /dev/ttyS3");

rr_servo_t *servo = rr_init_servo(iface,
  TUTORIAL_SERVO_0_ID);

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error count");

uint32_t _size;
rr_read_error_status(servo, &_size, 0);
API_DEBUG("\tError count: %d %s\n", _size, _size ? "" : "(No errors)");

API_DEBUG("========== Tutorial of the %s ==========\n", "reading servo error list");
uint8_t array[100];
uint32_t size;
rr_read_error_status(servo, &size, array);
API_DEBUG("\tError count: %d %s\n", size, size ? "" : "(No errors)");
for(int i = 0; i < size; i++)
{
    API_DEBUG("\t\tError: %s\n", rr_describe_emcy_bit(array[i]));
}
```

Chapter 6

Class Documentation

6.1 param_cache_entry_t Struct Reference

Device information source instance.

```
#include <api.h>
```

Public Attributes

- float value

  Source value.
- uint8_t activated

  Source activation flag.

6.1.1 Detailed Description

Device information source instance.

The documentation for this struct was generated from the following file:

- include/api.h

6.2 rr_can_interface_t Struct Reference

Interface instance structure.

```
#include <api.h>
```

Public Attributes

- void ∗ iface
  Interface internals.
- void ∗ nmt_cb
  NMT callback pointer.
- void ∗ emcy_cb
  EMCY callback pointer.

6.2.1 Detailed Description

Interface instance structure.

The documentation for this struct was generated from the following file:

- include/api.h

6.3 rr_servo_t Struct Reference

Device instance structure.

```
#include <api.h>
```

Public Attributes

- void ∗ dev
  Device internals.
- param_cache_entry_t pcache [APP_PARAM_SIZE]
  Device sources cells.

6.3.1 Detailed Description

Device instance structure.

The documentation for this struct was generated from the following file:

- include/api.h

Chapter 7

File Documentation

7.1 include/api.h File Reference

Rozum Robotics API Header File.

```
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
```

Classes

- struct param_cache_entry_t

  Device information source instance.
- struct rr_servo_t

  Device instance structure.
- struct rr_can_interface_t

  Interface instance structure.

Macros

- #define API_DEBUG(...) fprintf(stderr, __VA_ARGS__)

  Standart debug.
- #define STRFY(x) #x

  Make string from the variable.

Typedefs

- typedef void(∗ rr_nmt_cb_t) (rr_can_interface_t ∗interface, int servo_id, rr_nmt_state_t nmt_state)

  Type of the intiated network management (NMT) callback
- typedef void(∗ rr_emcy_cb_t) (rr_can_interface_t ∗interface, int servo_id, uint16_t code, uint8_t reg, uint8_t bits, uint32_t info)

  Type of the intiated emergency (EMCY) callback

Enumerations

- enum rr_ret_status_t {
RET_OK = 0, RET_ERROR, RET_BAD_INSTANCE, RET_BUSY,
RET_WRONG_TRAJ, RET_LOCKED, RET_STOPPED, RET_TIMEOUT,
RET_ZERO_SIZE, RET_SIZE_MISMATCH }

  Return codes of the API functions.
- enum rr_servo_param_t {
APP_PARAM_NULL = 0, APP_PARAM_POSITION, APP_PARAM_VELOCITY, APP_PARAM_POSITIO←N_ROTOR,
APP_PARAM_VELOCITY_ROTOR, APP_PARAM_POSITION_GEAR_360, APP_PARAM_POSITION_G←EAR_EMULATED, APP_PARAM_CURRENT_INPUT,
APP_PARAM_CURRENT_OUTPUT, APP_PARAM_VOLTAGE_INPUT, APP_PARAM_VOLTAGE_OUT←PUT, APP_PARAM_CURRENT_PHASE,
APP_PARAM_TEMPERATURE_ACTUATOR, APP_PARAM_TEMPERATURE_ELECTRONICS, APP_P←ARAM_TORQUE, APP_PARAM_ACCELERATION,
APP_PARAM_ACCELERATION_ROTOR, APP_PARAM_CURRENT_PHASE_1, APP_PARAM_CURRE←NT_PHASE_2, APP_PARAM_CURRENT_PHASE_3,
APP_PARAM_CURRENT_RAW, APP_PARAM_CURRENT_RAW_2, APP_PARAM_CURRENT_RAW_3, APP_PARAM_ENCODER_MASTER_TRACK,
APP_PARAM_ENCODER_NONIUS_TRACK, APP_PARAM_ENCODER_MOTOR_MASTER_TRACK, A←PP_PARAM_ENCODER_MOTOR_NONIUS_TRACK, APP_PARAM_TORQUE_ELECTRIC_CALC,
APP_PARAM_CONTROLLER_VELOCITY_ERROR, APP_PARAM_CONTROLLER_VELOCITY_SETPO←INT, APP_PARAM_CONTROLLER_VELOCITY_FEEDBACK, APP_PARAM_CONTROLLER_VELOCITY←_OUTPUT,
APP_PARAM_CONTROLLER_POSITION_ERROR, APP_PARAM_CONTROLLER_POSITION_SETPOI←NT, APP_PARAM_CONTROLLER_POSITION_FEEDBACK, APP_PARAM_CONTROLLER_POSITION_←OUTPUT,
APP_PARAM_CONTROL_MODE, APP_PARAM_FOC_ANGLE, APP_PARAM_FOC_IA, APP_PARAM_←FOC_IB,
APP_PARAM_FOC_IQ_SET, APP_PARAM_FOC_ID_SET, APP_PARAM_FOC_IQ, APP_PARAM_FOC←_ID,
APP_PARAM_FOC_IQ_ERROR, APP_PARAM_FOC_ID_ERROR, APP_PARAM_FOC_UQ, APP_PARA←M_FOC_UD,
APP_PARAM_FOC_UA, APP_PARAM_FOC_UB, APP_PARAM_FOC_U1, APP_PARAM_FOC_U2,
APP_PARAM_FOC_U3, APP_PARAM_FOC_PWM1, APP_PARAM_FOC_PWM2, APP_PARAM_FOC_←PWM3,
APP_PARAM_FOC_TIMER_TOP, APP_PARAM_DUTY, APP_PARAM_CURRENT_PHASE_ABS, APP_←PARAM_CURRENT_RMS_ABS,
APP_PARAM_QUALITY, APP_PARAM_SIZE }

  Device parameter & source indexes.
- enum rr_nmt_state_t {
RR_NMT_INITIALIZING = 0, RR_NMT_BOOT = 2, RR_NMT_PRE_OPERATIONAL = 127, RR_NMT_OP←ERATIONAL = 5,
RR_NMT_STOPPED = 4, RR_NMT_HB_TIMEOUT = -1 }

  Network management (NMT) states.

Functions

- void rr_sleep_ms (int ms)

  The function sets an idle period for the user program (e.g., to wait till a servo executes a motion trajectory). Until the period expires, the user program will not execute any further operations. However, the network management, CAN communication, emergency, and Heartbeat functions remain available.
- void rr_set_debug_log_stream (FILE ∗f)

  The function sets a stream for saving the debugging messages generated by the API library. Subsequently, the user can look through the logs to identify and locate the events associated with certain problems.

- void rr_set_comm_log_stream (const rr_can_interface_t ∗interface, FILE ∗f)

  The function sets a stream for saving CAN communication dump from the specified interface. Subsequently, the user can look through the logs saved to the stream to identify causes of CAN communication failures.
- void rr_setup_nmt_callback (rr_can_interface_t ∗interface, rr_nmt_cb_t cb)

  The function sets a user callback to be intiated in connection with with changes of network management (NMT) states (e.g., a servo connected to/ disconnected from the CAN bus, the interface/ a servo going to the operational state, etc.). User callbacks are functions to execute specific user-defined operations, e.g., to display a warning about an NMT state change or stop the program.
- void rr_setup_emcy_callback (rr_can_interface_t ∗interface, rr_emcy_cb_t cb)

  The function sets a user callback to be intiated in connection with emergency (EMCY) events (e.g., overcurrent, power outage, etc.). User callbacks are functions to execute specific user-defined operations, e.g., to display a warning about an EMCY event or stop the program.
- const char ∗ rr_describe_nmt (rr_nmt_state_t state)

  The function returns a string describing the NMT state code specified in the 'state' parameter. You can also use the function with rr_setup_nmt_callback, setting the callback to display a detailed message describing an NMT event.
- const char ∗ rr_describe_emcy_code (uint16_t code)

  The function returns a string descibing a specific EMCY event based on the error code in the 'code' parameter. The description in the string is a generic type of the occured emergency event (e.g., "Temperature"). For a more detailed description, use the function together with the rr_describe_emcy_bit one.
- const char ∗ rr_describe_emcy_bit (uint8_t bit)

  The function returns a string describing in detail a specific EMCY event based on the code in the 'bit' parameter (e.g., "CAN bus warning limit reached"). The function can be used in combination with rr_describe_emcy_code. The latter provides a more generic description of an EMCY event.
- rr_can_interface_t ∗ rr_init_interface (const char ∗interface_name)

  The function is the first to call to be able to work with the user API. It opens the COM port where the corresponding CAN-USB dongle is connected, enabling communication between the user program and the servo motors on the respective CAN bus.
- int rr_deinit_interface (rr_can_interface_t ∗∗interface)

  The function closes the COM port where the corresponding CAN-USB dongle is connected, clearing all data associated with the interface descriptor. It is advisable to call the function every time before quitting the user program.
- rr_servo_t ∗ rr_init_servo (rr_can_interface_t ∗interface, const uint8_t id)

  The function determines whether the servo motor with the speficied ID is connected to the specified interface. It waits for 2 seconds to receive a Heartbeat message from the servo. When the message arrives within the interval, the servo is identified as successfully connected.
- int rr_deinit_servo (rr_servo_t ∗∗servo)

  The function deinitializes the servo, clearing all data associated with the servo descriptor.
- int rr_servo_reboot (const rr_servo_t ∗servo)

  The function reboots the servo specified in the 'servo' parameter of the function, resetting it to the power-on state.
- int rr_servo_reset_communication (const rr_servo_t ∗servo)

  The function resets communication with the servo specified in the 'servo' parameter without resetting the entire interface.
- int rr_servo_set_state_operational (const rr_servo_t ∗servo)

  The function sets the servo specified in the 'servo' parameter to the operational state. In the state, the servo is both available for communication and can execute commands.
- int rr_servo_set_state_pre_operational (const rr_servo_t ∗servo)

  The function sets the servo specified in the 'servo' parameter to the pre-operational state. In the state, the servo is available for communication, but cannot execute any commands.
- int rr_servo_set_state_stopped (const rr_servo_t ∗servo)

  The function sets the servo specified in the 'servo' parameter to the stopped state. In the state, only Heartbeats are available. You can neither communicate with the servo nor make it execute any commands.
- int rr_net_reboot (const rr_can_interface_t ∗interface)

  The function reboots all servos connected to the interface specified in the 'interface' parameter, resetting them back to the power-on state.
- int rr_net_reset_communication (const rr_can_interface_t ∗interface)

The function resets communication via the interface specified in the 'interface' parameter. For instance, you may need to use the function when changing settings that require a reset after modification.

- int rr_net_set_state_operational (const rr_can_interface_t ∗interface)

  The function sets all servos connected to the interface (CAN bus) specified in the 'interface' parameter to the operational state. In the state, the servos can both communicate with the user program and execute commands.
- int rr_net_set_state_pre_operational (const rr_can_interface_t ∗interface)

  The function sets all servos connected to the interface specified in the 'interface' parameter to the pre-operational state. In the state, the servos are available for communication, but cannot execute commands.
- int rr_net_set_state_stopped (const rr_can_interface_t ∗interface)

  The function sets all servos connected to the interface specified in the 'interface' parameter to the stopped state. In the state, the servos are neither available for communication nor can execute commands.
- int rr_stop_and_release (const rr_servo_t ∗servo)

  The function sets the specified servo to the released state. The servo is de-energized and stops without retaining its position.
- int rr_stop_and_freeze (const rr_servo_t ∗servo)

  The function sets the specified servo to the freeze state. The servo stops, retaining its last position.
- int rr_set_current (const rr_servo_t ∗servo, const float current_a)

  The function sets the current supplied to the stator of the servo specified in the 'servo' parameter. Changing the 'current_a parameter' value, it is possible to adjust the servo's torque (Torque = stator current∗Kt).
- int rr_set_velocity (const rr_servo_t ∗servo, const float velocity_deg_per_sec)

  The function sets the velocity at which the specified servo should move at its maximum current. The maximum current is in accordance with the servo motor specification.
- int rr_set_position (const rr_servo_t ∗servo, const float position_deg)

  The function sets the position that the specified servo should reach as a result of executing the command. The velocity and current are maximum values in accordance with the servo motor specifications. For setting lower velocity and current limits, use the rr_set_position_with_limits function.
- int rr_set_velocity_with_limits (const rr_servo_t ∗servo, const float velocity_deg_per_sec, const float current_a)

  The function commands the specified servo to rotate at the specified velocity, while setting the maximum limit for the servo current (below the servo motor specifications).
- int rr_set_position_with_limits (const rr_servo_t ∗servo, const float position_deg, const float velocity_deg_↩ per_sec, const float current_a)

  The function sets the position that the specified servo should reach at user-defined velocity and current as a result of executing the command.
- int rr_set_duty (const rr_servo_t ∗servo, float duty_percent)

  The function limits the input voltage supplied to the servo, enabling to adjust its motion velocity. For instance, when the input voltage is 20V, setting the duty_percent parameter to 40% will result in 8V supplied to the servo.
- int rr_add_motion_point (const rr_servo_t ∗servo, const float position_deg, const float velocity_deg, const uint32_t time_ms)

  The function enables creating PVT (position-velocity-time) points to set the motion trajectory of the servo specified in the 'servo' parameter. PVT points define the following:
- int rr_start_motion (rr_can_interface_t ∗interface, uint32_t timestamp_ms)

  The function commands all servos connected to the specified interface (CAN bus) to move simultaneously through a number of preset PVT points (see rr_add_motion_point).
- int rr_read_error_status (const rr_servo_t ∗servo, uint32_t ∗const error_count, uint8_t ∗const error_array)

  The functions enables reading the total current count of servo errors and their codes.
- int rr_param_cache_update (rr_servo_t ∗servo)

  The function is always used in combination with the rr_param_cache_setup_entry function. It retreives from the servo the array of parameters set up using rr_param_cache_setup_entry function and saves the array to the program cache. You can subsequently read the parameters from the program cache with the rr_read_cached_parameter function. For more information, see rr_param_cache_setup_entry.
- int rr_param_cache_setup_entry (rr_servo_t ∗servo, const rr_servo_param_t param, bool enabled)

*The function is the fist one in the API call sequence that enables reading multiple servo paramaters (e.g., velocity, voltage, and position) as a data array. Using the sequence is advisable when you need to read **more than one parameter at a time**. The user can set up the array to include up to 50 parameters. In all, the sequence comprises the following functions:*

- int rr_read_parameter (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value)

  The function enables reading a single parameter directly from the servo specified in the 'servo' parameter of the function. The function returns the current value of the parameter. Additionally, the parameter is saved to the program cache, irrespective of whether it was enabled/ disabled with the rr_param_cache_setup_entry function.

- int rr_read_cached_parameter (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value)

  The function is always used in combination with the rr_param_cache_setup_entry and the::rr_param_cache_update functions. For more information, see rr_param_cache_setup_entry.

- int rr_clear_points_all (const rr_servo_t ∗servo)

  The function clears the entire motion queue of the servo specified in the 'servo' parameter of the function. The servo completes the move it started before the function call and then clears all the remaining PVT points in the queue.

- int rr_clear_points (const rr_servo_t ∗servo, const uint32_t num_to_clear)

  The function removes the number of PVT points indicated in the 'num_to_clear' parameter from the tail of the motion queue preset for the specified servo. When the indicated number of PVT points to be removed exceeds the actual remaining number of PVT points in the queue, the funtion clears only the actual remaining number of PVT points.

- int rr_get_points_size (const rr_servo_t ∗servo, uint32_t ∗num)

  The function returns the actual motion queue size of the specified servo. The return value indicates how many PVT points have already been added to the motion queue.

- int rr_get_points_free_space (const rr_servo_t ∗servo, uint32_t ∗num)

  The function returns how many more PVT points the user can add to the motion queue of the servo specified in the 'servo' parameter.

- int rr_invoke_time_calculation (const rr_servo_t ∗servo, const float start_position_deg, const float start_↩ velocity_deg, const float start_acceleration_deg_per_sec2, const uint32_t start_time_ms, const float end↩ _position_deg, const float end_velocity_deg, const float end_acceleration_deg_per_sec2, const uint32_↩ t end_time_ms)

  *The function enables calculating the time it will take for the specified servo to get from one position to another at the specified motion parameters (e.g., velocity, acceleration). To read the calculation result, use the the rr_get_time_↩ calculation_result function. **Note:**The function is executed without the servo moving.*

- int rr_get_time_calculation_result (const rr_servo_t ∗servo, uint32_t ∗time_ms)

  The function enables reading the result of the calculations made using the rr_invoke_time_calculation function. It returns the calculated time (in milliseconds) it will take the servo with the specified descriptor to go from one position to another.

- int rr_set_zero_position (const rr_servo_t ∗servo, const float position_deg)

  The function enables setting the current position (in degrees) of the servo with the specified descriptor to any value defined by the user. For instance, when the current servo position is 101 degrees and the 'position_deg' parameter is set to 25 degrees, the servo is assumed to be positioned at 25 degrees.

- int rr_set_zero_position_and_save (const rr_servo_t ∗servo, const float position_deg)

  The function enables setting the current position (in degrees) of the servo with the specified descriptor to any value defined by the user and saving it to the FLASH memory. If you don't want to save the newly set position, use the rr_set_zero_position function.

- int rr_get_max_velocity (const rr_servo_t ∗servo, float ∗velocity_deg_per_sec)

  The function reads the maximum velocity of the servo at the current moment. It returns the smallest of the three values—the user-defined maximum velocity limit (rr_set_max_velocity), the maximum velocity value based on the servo specifications, or the calculated maximum velocity based on the supply voltage.

- int rr_set_max_velocity (const rr_servo_t ∗servo, const float max_velocity_deg_per_sec)

  The function sets the maximum velocity limit for the servo specified in the 'servo' parameter. The setting is volatile: after a reset or a power outage, it is no longer valid.

7.1.1 Detailed Description

Rozum Robotics API Header File.

Author

Rozum

Date

2018-06-01

7.1.2 Typedef Documentation

7.1.2.1 rr_emcy_cb_t

```
typedef void(* rr_emcy_cb_t) (rr_can_interface_t *interface, int servo_id, uint16_t code,
uint8_t reg, uint8_t bits, uint32_t info)
```

Type of the intiated emergency (EMCY) callback

Parameters

interface	Descriptor of the interface (see rr_init_interface) where the EMCY event occured
servo_id	Descriptor of the servo (see rr_init_servo) where the EMCY event occured
code	Error code
reg	Register field of the EMCY message (see CanOpen documentation)
bits	Bits field of the EMCY message (see CanOpen documentation)
info	Additional field (see CanOpen documentation)

7.1.2.2 rr_nmt_cb_t

```
typedef void(* rr_nmt_cb_t) (rr_can_interface_t *interface, int servo_id, rr_nmt_state_t nmt_state)
```

Type of the intiated network management (NMT) callback

Parameters

interface	Descriptor of the interface (see rr_init_interface) where the NMT event occured
servo_id	Descriptor of the servo (see rr_init_servo) where the NMT event occured
nmt_state	Network management state (rr_nmt_state_t) that the servo entered

7.1.3 Enumeration Type Documentation

7.1.3.1 rr_nmt_state_t

```
enum rr_nmt_state_t
```

Network management (NMT) states.

Enumerator

RR_NMT_INITIALIZING	Device is initializing
RR_NMT_BOOT	Device executing bootloader application
RR_NMT_PRE_OPERATIONAL	Device is in pre-operational state
RR_NMT_OPERATIONAL	Device is in operational state
RR_NMT_STOPPED	Device is stopped
RR_NMT_HB_TIMEOUT	Device heartbeat timeout (device disappeared from bus)

7.1.3.2 rr_ret_status_t

```
enum rr_ret_status_t
```

Return codes of the API functions.

Enumerator

RET_OK	Status OK.
RET_ERROR	Generic error.
RET_BAD_INSTANCE	Bad interface or servo instance (null)
RET_BUSY	Device is busy.
RET_WRONG_TRAJ	Wrong trajectory.
RET_LOCKED	Device is locked.
RET_STOPPED	Device is in STOPPED state.
RET_TIMEOUT	Communication timeout.
RET_ZERO_SIZE	Zero size.
RET_SIZE_MISMATCH	Received & target size mismatch.

7.1.3.3 rr_servo_param_t

```
enum rr_servo_param_t
```

Device parameter & source indexes.

Enumerator

APP_PARAM_NULL	Not used.
APP_PARAM_POSITION	Actual multi-turn position of the output shaft (degrees)
APP_PARAM_VELOCITY	Actual velocity of the output shaft (degrees per second)
APP_PARAM_POSITION_ROTOR	Actual position of the motor shaft (degrees)
APP_PARAM_VELOCITY_ROTOR	Actual velocity of the motor shaft (degrees per second)
APP_PARAM_POSITION_GEAR_360	Actual single-turn position of the output shaft (from 0 to 360 degrees)
APP_PARAM_POSITION_GEAR_EMULATED	Actual multi-turn position of the motor shaft multiplied by gear ratio (degrees)
APP_PARAM_CURRENT_INPUT	Actual DC current (amperes)
APP_PARAM_CURRENT_OUTPUT	Not used.
APP_PARAM_VOLTAGE_INPUT	Actual DC voltage (volts)
APP_PARAM_VOLTAGE_OUTPUT	Not used.
APP_PARAM_CURRENT_PHASE	Actual magnitude of AC current (amperes)
APP_PARAM_TEMPERATURE_ACTUATOR	Not used.
APP_PARAM_TEMPERATURE_ELECTRONICS	Actual temperature of the motor controller.
APP_PARAM_TORQUE	Not used.
APP_PARAM_ACCELERATION	Not used.
APP_PARAM_ACCELERATION_ROTOR	Not used.
APP_PARAM_CURRENT_PHASE_1	Actual phase 1 current.
APP_PARAM_CURRENT_PHASE_2	Actual phase 2 current.
APP_PARAM_CURRENT_PHASE_3	Actual phase 3 current.
APP_PARAM_CURRENT_RAW	Not used.
APP_PARAM_CURRENT_RAW_2	Not used.
APP_PARAM_CURRENT_RAW_3	Not used.
APP_PARAM_ENCODER_MASTER_TRACK	Internal use only.
APP_PARAM_ENCODER_NONIUS_TRACK	Internal use only.
APP_PARAM_ENCODER_MOTOR_MASTER_TR↩ACK	Internal use only.
APP_PARAM_ENCODER_MOTOR_NONIUS_TR↩ACK	Internal use only.
APP_PARAM_TORQUE_ELECTRIC_CALC	Internal use only.
APP_PARAM_CONTROLLER_VELOCITY_ERROR	Velocity following error.
APP_PARAM_CONTROLLER_VELOCITY_SETP↩OINT	Velocity target.
APP_PARAM_CONTROLLER_VELOCITY_FEED↩BACK	Actual velocity (degrees per second)
APP_PARAM_CONTROLLER_VELOCITY_OUTPUT	Not used.
APP_PARAM_CONTROLLER_POSITION_ERROR	Position following error.
APP_PARAM_CONTROLLER_POSITION_SETP↩OINT	Position target.
APP_PARAM_CONTROLLER_POSITION_FEED↩BACK	Actual position (degrees)
APP_PARAM_CONTROLLER_POSITION_OUTPUT	Not used.
APP_PARAM_CONTROL_MODE	Internal use only.
APP_PARAM_FOC_ANGLE	Internal use only.
APP_PARAM_FOC_IA	Internal use only.

Enumerator

APP_PARAM_FOC_IB	Internal use only.
APP_PARAM_FOC_IQ_SET	Internal use only.
APP_PARAM_FOC_ID_SET	Internal use only.
APP_PARAM_FOC_IQ	Internal use only.
APP_PARAM_FOC_ID	Internal use only.
APP_PARAM_FOC_IQ_ERROR	Internal use only.
APP_PARAM_FOC_ID_ERROR	Internal use only.
APP_PARAM_FOC_UQ	Internal use only.
APP_PARAM_FOC_UD	Internal use only.
APP_PARAM_FOC_UA	Internal use only.
APP_PARAM_FOC_UB	Internal use only.
APP_PARAM_FOC_U1	Internal use only.
APP_PARAM_FOC_U2	Internal use only.
APP_PARAM_FOC_U3	Internal use only.
APP_PARAM_FOC_PWM1	Internal use only.
APP_PARAM_FOC_PWM2	Internal use only.
APP_PARAM_FOC_PWM3	Internal use only.
APP_PARAM_FOC_TIMER_TOP	Internal use only.
APP_PARAM_DUTY	Internal use only.
APP_PARAM_CURRENT_PHASE_ABS	Internal use only.
APP_PARAM_CURRENT_RMS_ABS	Internal use only.
APP_PARAM_QUALITY	Internal use only.
APP_PARAM_SIZE	Use when you need to define the total param arrray size.

7.2 src/api.c File Reference

Rozum Robotics API Source File.

```
#include "api.h"
#include "logging.h"
#include "usbcan_proto.h"
#include "usbcan_types.h"
#include "usbcan_util.h"
```

Functions

- void rr_sleep_ms (int ms)

  The function sets an idle period for the user program (e.g., to wait till a servo executes a motion trajectory). Until the period expires, the user program will not execute any further operations. However, the network management, CAN communication, emergency, and Heartbeat functions remain available.
- void rr_set_comm_log_stream (const rr_can_interface_t ∗interface, FILE ∗f)

  The function sets a stream for saving CAN communication dump from the specified interface. Subsequently, the user can look through the logs saved to the stream to identify causes of CAN communication failures.
- void rr_set_debug_log_stream (FILE ∗f)

The function sets a stream for saving the debugging messages generated by the API library. Subsequently, the user can look through the logs to identify and locate the events associated with certain problems.

- void rr_setup_nmt_callback (rr_can_interface_t ∗interface, rr_nmt_cb_t cb)

  The function sets a user callback to be intiated in connection with with changes of network management (NMT) states (e.g., a servo connected to/ disconnected from the CAN bus, the interface/ a servo going to the operational state, etc.). User callbacks are functions to execute specific user-defined operations, e.g., to display a warning about an NMT state change or stop the program.
- void rr_setup_emcy_callback (rr_can_interface_t ∗interface, rr_emcy_cb_t cb)

  The function sets a user callback to be intiated in connection with emergency (EMCY) events (e.g., overcurrent, power outage, etc.). User callbacks are functions to execute specific user-defined operations, e.g., to display a warning about an EMCY event or stop the program.
- const char ∗ rr_describe_nmt (rr_nmt_state_t state)

  The function returns a string describing the NMT state code specified in the 'state' parameter. You can also use the function with rr_setup_nmt_callback, setting the callback to display a detailed message describing an NMT event.
- const char ∗ rr_describe_emcy_bit (uint8_t bit)

  The function returns a string describing in detail a specific EMCY event based on the code in the 'bit' parameter (e.g., "CAN bus warning limit reached"). The function can be used in combination with rr_describe_emcy_code. The latter provides a more generic description of an EMCY event.
- const char ∗ rr_describe_emcy_code (uint16_t code)

  The function returns a string descibing a specific EMCY event based on the error code in the 'code' parameter. The description in the string is a generic type of the occured emergency event (e.g., "Temperature"). For a more detailed description, use the function together with the rr_describe_emcy_bit one.
- rr_can_interface_t ∗ rr_init_interface (const char ∗interface_name)

  The function is the first to call to be able to work with the user API. It opens the COM port where the corresponding CAN-USB dongle is connected, enabling communication between the user program and the servo motors on the respective CAN bus.
- int rr_deinit_interface (rr_can_interface_t ∗∗interface)

  The function closes the COM port where the corresponding CAN-USB dongle is connected, clearing all data associated with the interface descriptor. It is advisable to call the function every time before quitting the user program.
- rr_servo_t ∗ rr_init_servo (rr_can_interface_t ∗interface, const uint8_t id)

  The function determines whether the servo motor with the speficied ID is connected to the specified interface. It waits for 2 seconds to receive a Heartbeat message from the servo. When the message arrives within the interval, the servo is identified as successfully connected.
- int rr_deinit_servo (rr_servo_t ∗∗servo)

  The function deinitializes the servo, clearing all data associated with the servo descriptor.
- int rr_servo_reboot (const rr_servo_t ∗servo)

  The function reboots the servo specified in the 'servo' parameter of the function, resetting it to the power-on state.
- int rr_servo_reset_communication (const rr_servo_t ∗servo)

  The function resets communication with the servo specified in the 'servo' parameter without resetting the entire interface.
- int rr_servo_set_state_operational (const rr_servo_t ∗servo)

  The function sets the servo specified in the 'servo' parameter to the operational state. In the state, the servo is both available for communication and can execute commands.
- int rr_servo_set_state_pre_operational (const rr_servo_t ∗servo)

  The function sets the servo specified in the 'servo' parameter to the pre-operational state. In the state, the servo is available for communication, but cannot execute any commands.
- int rr_servo_set_state_stopped (const rr_servo_t ∗servo)

  The function sets the servo specified in the 'servo' parameter to the stopped state. In the state, only Heartbeats are available. You can neither communicate with the servo nor make it execute any commands.
- int rr_net_reboot (const rr_can_interface_t ∗interface)

  The function reboots all servos connected to the interface specified in the 'interface' parameter, resetting them back to the power-on state.
- int rr_net_reset_communication (const rr_can_interface_t ∗interface)

  The function resets communication via the interface specified in the 'interface' parameter. For instance, you may need to use the function when changing settings that require a reset after modification.

- int rr_net_set_state_operational (const rr_can_interface_t ∗interface)

  The function sets all servos connected to the interface (CAN bus) specified in the 'interface' parameter to the operational state. In the state, the servos can both communicate with the user program and execute commands.
- int rr_net_set_state_pre_operational (const rr_can_interface_t ∗interface)

  The function sets all servos connected to the interface specified in the 'interface' parameter to the pre-operational state. In the state, the servos are available for communication, but cannot execute commands.
- int rr_net_set_state_stopped (const rr_can_interface_t ∗interface)

  The function sets all servos connected to the interface specified in the 'interface' parameter to the stopped state. In the state, the servos are neither available for communication nor can execute commands.
- int rr_stop_and_release (const rr_servo_t ∗servo)

  The function sets the specified servo to the released state. The servo is de-energized and stops without retaining its position.
- int rr_stop_and_freeze (const rr_servo_t ∗servo)

  The function sets the specified servo to the freeze state. The servo stops, retaining its last position.
- int rr_set_current (const rr_servo_t ∗servo, const float current_a)

  The function sets the current supplied to the stator of the servo specified in the 'servo' parameter. Changing the 'current_a parameter' value, it is possible to adjust the servo's torque (Torque = stator current∗Kt).
- int rr_set_velocity (const rr_servo_t ∗servo, const float velocity_deg_per_sec)

  The function sets the velocity at which the specified servo should move at its maximum current. The maximum current is in accordance with the servo motor specification.
- int rr_set_position (const rr_servo_t ∗servo, const float position_deg)

  The function sets the position that the specified servo should reach as a result of executing the command. The velocity and current are maximum values in accordance with the servo motor specifications. For setting lower velocity and current limits, use the rr_set_position_with_limits function.
- int rr_set_velocity_with_limits (const rr_servo_t ∗servo, const float velocity_deg_per_sec, const float current_a)

  The function commands the specified servo to rotate at the specified velocity, while setting the maximum limit for the servo current (below the servo motor specifications).
- int rr_set_position_with_limits (const rr_servo_t ∗servo, const float position_deg, const float velocity_deg_↩ per_sec, const float current_a)

  The function sets the position that the specified servo should reach at user-defined velocity and current as a result of executing the command.
- int rr_set_duty (const rr_servo_t ∗servo, float duty_percent)

  The function limits the input voltage supplied to the servo, enabling to adjust its motion velocity. For instance, when the input voltage is 20V, setting the duty_percent parameter to 40% will result in 8V supplied to the servo.
- int rr_add_motion_point (const rr_servo_t ∗servo, const float position_deg, const float velocity_deg_per_sec, const uint32_t time_ms)

  The function enables creating PVT (position-velocity-time) points to set the motion trajectory of the servo specified in the 'servo' parameter. PVT points define the following:
- int rr_start_motion (rr_can_interface_t ∗interface, uint32_t timestamp_ms)

  The function commands all servos connected to the specified interface (CAN bus) to move simultaneously through a number of preset PVT points (see rr_add_motion_point).
- int rr_read_error_status (const rr_servo_t ∗servo, uint32_t ∗const error_count, uint8_t ∗const error_array)

  The functions enables reading the total current count of servo errors and their codes.
- int rr_param_cache_update (rr_servo_t ∗servo)

  The function is always used in combination with the rr_param_cache_setup_entry function. It retreives from the servo the array of parameters set up using rr_param_cache_setup_entry function and saves the array to the program cache. You can subsequently read the parameters from the program cache with the rr_read_cached_parameter function. For more information, see rr_param_cache_setup_entry.
- int rr_param_cache_setup_entry (rr_servo_t ∗servo, const rr_servo_param_t param, bool enabled)

  *The function is the fist one in the API call sequence that enables reading multiple servo paramaters (e.g., velocity, voltage, and position) as a data array. Using the sequence is advisable when you need to read **more than one parameter at a time**. The user can set up the array to include up to 50 parameters. In all, the sequence comprises the following functions:*

- int rr_read_parameter (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value)

  The function enables reading a single parameter directly from the servo specified in the 'servo' parameter of the function. The function returns the current value of the parameter. Additionally, the parameter is saved to the program cache, irrespective of whether it was enabled/ disabled with the rr_param_cache_setup_entry function.
- int rr_read_cached_parameter (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value)

  The function is always used in combination with the rr_param_cache_setup_entry and the::rr_param_cache_update functions. For more information, see rr_param_cache_setup_entry.
- int rr_clear_points_all (const rr_servo_t ∗servo)

  The function clears the entire motion queue of the servo specified in the 'servo' parameter of the function. The servo completes the move it started before the function call and then clears all the remaining PVT points in the queue.
- int rr_clear_points (const rr_servo_t ∗servo, const uint32_t num_to_clear)

  The function removes the number of PVT points indicated in the 'num_to_clear' parameter from the tail of the motion queue preset for the specified servo. When the indicated number of PVT points to be removed exceeds the actual remaining number of PVT points in the queue, the funtion clears only the actual remaining number of PVT points.
- int rr_get_points_size (const rr_servo_t ∗servo, uint32_t ∗num)

  The function returns the actual motion queue size of the specified servo. The return value indicates how many PVT points have already been added to the motion queue.
- int rr_get_points_free_space (const rr_servo_t ∗servo, uint32_t ∗num)

  The function returns how many more PVT points the user can add to the motion queue of the servo specified in the 'servo' parameter.
- int rr_invoke_time_calculation (const rr_servo_t ∗servo, const float start_position_deg, const float start←_velocity_deg_per_sec, const float start_acceleration_deg_per_sec2, const uint32_t start_time_ms, const float end_position_deg, const float end_velocity_deg_per_sec, const float end_acceleration_deg_per_sec2, const uint32_t end_time_ms)

  *The function enables calculating the time it will take for the specified servo to get from one position to another at the specified motion parameters (e.g., velocity, acceleration). To read the calculation result, use the the rr_get_time_←calculation_result function. **Note:**The function is executed without the servo moving.*
- int rr_get_time_calculation_result (const rr_servo_t ∗servo, uint32_t ∗time_ms)

  The function enables reading the result of the calculations made using the rr_invoke_time_calculation function. It returns the calculated time (in milliseconds) it will take the servo with the specified descriptor to go from one position to another.
- int rr_set_zero_position (const rr_servo_t ∗servo, const float position_deg)

  The function enables setting the current position (in degrees) of the servo with the specified descriptor to any value defined by the user. For instance, when the current servo position is 101 degrees and the 'position_deg' parameter is set to 25 degrees, the servo is assumed to be positioned at 25 degrees.
- int rr_set_zero_position_and_save (const rr_servo_t ∗servo, const float position_deg)

  The function enables setting the current position (in degrees) of the servo with the specified descriptor to any value defined by the user and saving it to the FLASH memory. If you don't want to save the newly set position, use the rr_set_zero_position function.
- int rr_get_max_velocity (const rr_servo_t ∗servo, float ∗velocity_deg_per_sec)

  The function reads the maximum velocity of the servo at the current moment. It returns the smallest of the three values—the user-defined maximum velocity limit (rr_set_max_velocity), the maximum velocity value based on the servo specifications, or the calculated maximum velocity based on the supply voltage.
- int rr_set_max_velocity (const rr_servo_t ∗servo, const float max_velocity_deg_per_sec)

  The function sets the maximum velocity limit for the servo specified in the 'servo' parameter. The setting is volatile: after a reset or a power outage, it is no longer valid.

7.2.1 Detailed Description

Rozum Robotics API Source File.

Author

Rozum

Date

2018-06-01

7.3 tutorial/control_servo_traj_1.c File Reference

Setting PVT points for one servo.

```
#include "api.h"
#include "tutorial.h"
```

Functions

- int main (int argc, char *argv[ ])

7.3.1 Detailed Description

Setting PVT points for one servo.

Author

Rozum

Date

2018-06-25

7.3.2 Function Documentation

7.3.2.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

[cccode 1] [Adding the interface1]

[Adding the interface1]

[Adding the servo1]

[Adding the servo1]

[Clear points all1]

[Clear points all1]

[Add motion point first]

[Add motion point first] [Add motion point second]

[Add motion point second] [Start motion1]

[Start motion1] [Sleep1]

[Sleep1] [cccode 1]

7.4 tutorial/control_servo_traj_2.c File Reference

Setting PVT points for two servos.

```
#include "api.h"
#include "tutorial.h"
```

Functions

- int main (int argc, char *argv[ ])

7.4.1 Detailed Description

Setting PVT points for two servos.

Author

Rozum

Date

2018-06-25

7.4.2 Function Documentation

7.4.2.1 main()

```
int main (
        int argc,
        char * argv[ ] )
```

[cccode 1] [Adding the interface2]

[Adding the interface2] [Adding servo ID0]

[Adding servo ID0] [Adding servo ID1]

[Adding servo ID1]

[Clear points servo ID0]

[Clear points servo ID0] [Clear points servo ID1]

[Clear points servo ID1]

[Add point1 servo ID0]

[Add point1 servo ID0] [Add point1 servo ID1]

[Add point1 servo ID1] [Add point2 servo ID0]

[Add point2 servo ID0] [Add point2 servo ID1]

[Add point2 servo ID1] [Start motion2]

[Start motion2]

[Sleep2]

[Sleep2] [cccode 1]

7.5 tutorial/control_servo_traj_3.c File Reference

Setting points for three servos.

```
#include "api.h"
#include "tutorial.h"
```

Functions

- int main (int argc, char *argv[ ])

7.5.1 Detailed Description

Setting points for three servos.

Author

Rozum

Date

2018-06-25

7.5.2 Function Documentation

7.5.2.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

[cccode 1] [Adding the interface3]

[Adding the interface3] [Adding servo ID0]

[Adding servo ID0] [Adding servo ID1]

[Adding servo ID1] [Adding servo ID2]

[Adding servo ID2]

[Clear points servo ID0]

[Clear points servo ID0] [Clear points servo ID1]

[Clear points servo ID1] [Clear points servo ID2]

[Clear points servo ID2] [Add motion point 1 servo ID0]

[Add motion point 1 servo ID0] [Add motion point 1 servo ID1]

[Add motion point 1 servo ID1] [Add motion point 1 servo ID2]

[Add motion point 1 servo ID2] [Add motion point 2 servo ID0]

[Add motion point 2 servo ID0] [Add motion point 2 servo ID1]

[Add motion point 2 servo ID1] [Add motion point 2 servo ID2]

[Add motion point 2 servo ID2] [Start motion]

[Start motion]

[Sleep]

[Sleep] [cccode 1]

7.6 tutorial/read_any_param.c File Reference

Tutorial example of reading device parameters.

```
#include "api.h"
#include "tutorial.h"
```

Functions

- int main (int argc, char *argv[ ])

7.6.1 Detailed Description

Tutorial example of reading device parameters.

Author

your name

Date

2018-06-25

7.6.2 Function Documentation

7.6.2.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

[cccode 1] [Adding the interface]

[Adding the interface] [Adding the servo]

[Adding the servo]

[Read parameter variable]

[Read parameter variable]

[Read rotor position]

[Read rotor position]

[Read rotor velocity]

[Read rotor velocity]

[Read voltage]

[Read voltage]

[Read current]

[Read current]

[cccode 1]

7.7 tutorial/read_any_param_cache.c File Reference

Tutorial example of reading servo parameters from the cache.

```
#include "api.h"
#include "tutorial.h"
```

Functions

- int main (int argc, char ∗argv[ ])

7.7.1 Detailed Description

Tutorial example of reading servo parameters from the cache.

Author

Rozum

Date

2018-06-25

7.7.2 Function Documentation

7.7.2.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

[cccode 1] [Adding the interface]

[Adding the interface] [Adding the servo]

[Adding the servo]

[Cache setup entry 1]

[Cache setup entry 1] [Cache setup entry 2]

[Cache setup entry 2] [Cache setup entry 3]

[Cache setup entry 3] [Cache setup entry 4]

[Cache setup entry 4]

[Cache update]

[Cache update]

[Parameter array]

[Parameter array]

[Read cached parameter 1]

[Read cached parameter 1]

[Read cached parameter 2]

[Read cached parameter 2]

[Read cached parameter 3]

[Read cached parameter 3]

[Read cached parameter 4]

[Read cached parameter 4]

[cccode 1]

7.8 tutorial/read_errors.c File Reference

Tutorial example of reading device errors.

```
#include "api.h"
#include "tutorial.h"
```

Functions

- int main (int argc, char ∗argv[ ])

7.8.1 Detailed Description

Tutorial example of reading device errors.

Author

Rozum

Date

2018-06-25

7.8.2 Function Documentation

7.8.2.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

[cccode 1] [Adding the interface 1]

[Adding the interface 1]

[Adding the servo]

[Adding the servo]

[Error count var]

[Error count var] [Error count read]

[Error count read]

[Error array 2]

[Error array 2] [Error count var 2]

[Error count var 2] [Error count and array read]

[Error count and array read]

[Cyclic read]

[Cyclic read] [cccode 1]

7.9 tutorial/read_servo_max_velocity.c File Reference

Tutorial example of reading the maximum servo.

```
#include "api.h"
#include "tutorial.h"
```

Functions

- int main (int argc, char *argv[ ])

7.9.1 Detailed Description

Tutorial example of reading the maximum servo.

Author

Rozum

Date

2018-06-25

7.9.2 Function Documentation

7.9.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

[cccode 1] [Adding the interface]

[Adding the interface] [Adding the servo]

[Adding the servo]

[Velocity variable]

[Velocity variable] [Read max velocity]

[Read max velocity]

[cccode 1]

7.10 tutorial/read_servo_motion_queue.c File Reference

Tutorial example of reading motion queue parameters.

```
#include "api.h"
#include "tutorial.h"
```

Functions

- int main (int argc, char *argv[ ])

7.10.1 Detailed Description

Tutorial example of reading motion queue parameters.

Author

Rozum

Date

2018-06-25

7.10.2 Function Documentation

7.10.2.1 main()

```
int main (
            int argc,
            char * argv[] )
```

[cccode 1] [Adding the interface]

[Adding the interface] [Adding the servo]

[Adding the servo]

[Clear points]

[Clear points]

[Points size variable]

[Points size variable] [Points size1]

[Points size1]

[Points free1]

[Points free1]

[Add point1]

[Add point1] [Add point2]

[Add point2]

[Points size2]

[Points size2]

[Points free2]

[Points free2]

[cccode 1]

7.11 tutorial/read_servo_trajectory_time.c File Reference

Tutorial example of calculating a PVT point.

```
#include "api.h"
#include "tutorial.h"
```

Functions

- int main (int argc, char *argv[ ])

7.11.1 Detailed Description

Tutorial example of calculating a PVT point.

Author

Rozum

Date

2018-06-25

7.11.2 Function Documentation

7.11.2.1 main()

```
int main (
            int argc,
            char * argv[ ] )
```

[cccode 1] [Adding the interface]

[Adding the interface] [Adding the servo]

[Adding the servo]

[Time calculation]

[Time calculation] [Travel time variable]

[Travel time variable] [Get calculation result]

[Get calculation result]

[cccode 1]

7.12 tutorial/tutorial.h File Reference

Tutorial header file with config.

Macros

- #define TUTORIAL_DEVICE "192.168.0.123"

  Tutorial interface address.
- #define TUTORIAL_SERVO_0_ID 64

  Tutorial servo #0 CAN ID.
- #define TUTORIAL_SERVO_1_ID 65

  Tutorial servo #1 CAN ID.
- #define TUTORIAL_SERVO_2_ID 66

  Tutorial servo #2 CAN ID.

7.12.1 Detailed Description

Tutorial header file with config.

Author

Rozum

Date

2018-06-25

7.13 /mnt/hdd/home/maksimkatkou/work/rr-git/UserAPI/hw/doc.c File Reference

Hardware manual.

7.13.1 Detailed Description

Hardware manual.

Author

Rozum

Index

/mnt/hdd/home/maksimkatkou/work/rr-git/UserAPI/hw/doc.c, 83

api.h
- rr_emcy_cb_t, 66
- rr_nmt_cb_t, 66
- rr_nmt_state_t, 67
- rr_ret_status_t, 67
- rr_servo_param_t, 67

Auxiliary functions, 41
- rr_sleep_ms, 41

control_servo_traj_1.c
- main, 73

control_servo_traj_2.c
- main, 74

control_servo_traj_3.c
- main, 75

Debugging, 39
- rr_set_comm_log_stream, 39
- rr_set_debug_log_stream, 39

Device parameter cache programming and reading, 52

Device parameters reading, 54

Error handling, 35
- rr_describe_emcy_bit, 35
- rr_describe_emcy_code, 35
- rr_read_error_status, 37
- rr_setup_emcy_callback, 37

include/api.h, 61

Initialization and deinitialization, 9
- rr_deinit_interface, 9
- rr_deinit_servo, 10
- rr_init_interface, 10
- rr_init_servo, 11

main
- control_servo_traj_1.c, 73
- control_servo_traj_2.c, 74
- control_servo_traj_3.c, 75
- read_any_param.c, 76
- read_any_param_cache.c, 77
- read_errors.c, 79
- read_servo_max_velocity.c, 80
- read_servo_motion_queue.c, 81
- read_servo_trajectory_time.c, 82

One servo PVT move, 45

param_cache_entry_t, 59

read_any_param.c
- main, 76

read_any_param_cache.c
- main, 77

read_errors.c
- main, 79

read_servo_max_velocity.c
- main, 80

read_servo_motion_queue.c
- main, 81

read_servo_trajectory_time.c
- main, 82

Reading and writing servo configuration, 29
- rr_get_max_velocity, 29
- rr_set_max_velocity, 29
- rr_set_zero_position, 30
- rr_set_zero_position_and_save, 30

Reading device errors, 58

Reading of the maximum velocity, 56

Reading of the motion queue parameters, 57

Reading realtime parameter, 32
- rr_param_cache_setup_entry, 32
- rr_param_cache_update, 33
- rr_read_cached_parameter, 33
- rr_read_parameter, 34

rr_add_motion_point
- Trajectory motion control (PVT), 23

rr_can_interface_t, 59

rr_clear_points
- Trajectory motion control (PVT), 24

rr_clear_points_all
- Trajectory motion control (PVT), 25

rr_deinit_interface
- Initialization and deinitialization, 9

rr_deinit_servo
- Initialization and deinitialization, 10

rr_describe_emcy_bit
- Error handling, 35

rr_describe_emcy_code
- Error handling, 35

rr_describe_nmt
- Switching servo working states, 12

rr_emcy_cb_t
- api.h, 66

rr_get_max_velocity
- Reading and writing servo configuration, 29

rr_get_points_free_space
- Trajectory motion control (PVT), 25

rr_get_points_size
Trajectory motion control (PVT), 26
rr_get_time_calculation_result
Trajectory motion control (PVT), 26
rr_init_interface
Initialization and deinitialization, 10
rr_init_servo
Initialization and deinitialization, 11
rr_invoke_time_calculation
Trajectory motion control (PVT), 26
rr_net_reboot
Switching servo working states, 13
rr_net_reset_communication
Switching servo working states, 13
rr_net_set_state_operational
Switching servo working states, 13
rr_net_set_state_pre_operational
Switching servo working states, 14
rr_net_set_state_stopped
Switching servo working states, 14
rr_nmt_cb_t
api.h, 66
rr_nmt_state_t
api.h, 67
rr_param_cache_setup_entry
Reading realtime parameter, 32
rr_param_cache_update
Reading realtime parameter, 33
rr_read_cached_parameter
Reading realtime parameter, 33
rr_read_error_status
Error handling, 37
rr_read_parameter
Reading realtime parameter, 34
rr_ret_status_t
api.h, 67
rr_servo_param_t
api.h, 67
rr_servo_reboot
Switching servo working states, 15
rr_servo_reset_communication
Switching servo working states, 15
rr_servo_set_state_operational
Switching servo working states, 15
rr_servo_set_state_pre_operational
Switching servo working states, 16
rr_servo_set_state_stopped
Switching servo working states, 16
rr_servo_t, 60
rr_set_comm_log_stream
Debugging, 39
rr_set_current
Simple motion control (duty, current, velocity, position), 18
rr_set_debug_log_stream
Debugging, 39
rr_set_duty
Simple motion control (duty, current, velocity, position), 19
rr_set_max_velocity
Reading and writing servo configuration, 29
rr_set_position
Simple motion control (duty, current, velocity, position), 19
rr_set_position_with_limits
Simple motion control (duty, current, velocity, position), 19
rr_set_velocity
Simple motion control (duty, current, velocity, position), 20
rr_set_velocity_with_limits
Simple motion control (duty, current, velocity, position), 20
rr_set_zero_position
Reading and writing servo configuration, 30
rr_set_zero_position_and_save
Reading and writing servo configuration, 30
rr_setup_emcy_callback
Error handling, 37
rr_setup_nmt_callback
Switching servo working states, 17
rr_sleep_ms
Auxiliary functions, 41
rr_start_motion
Trajectory motion control (PVT), 27
rr_stop_and_freeze
Simple motion control (duty, current, velocity, position), 21
rr_stop_and_release
Simple motion control (duty, current, velocity, position), 21

Servo box specs & manual, 42
Servo PVT point calculation, 55
Simple motion control (duty, current, velocity, position), 18
rr_set_current, 18
rr_set_duty, 19
rr_set_position, 19
rr_set_position_with_limits, 19
rr_set_velocity, 20
rr_set_velocity_with_limits, 20
rr_stop_and_freeze, 21
rr_stop_and_release, 21
src/api.c, 69
Switching servo working states, 12
rr_describe_nmt, 12
rr_net_reboot, 13
rr_net_reset_communication, 13
rr_net_set_state_operational, 13
rr_net_set_state_pre_operational, 14
rr_net_set_state_stopped, 14
rr_servo_reboot, 15
rr_servo_reset_communication, 15
rr_servo_set_state_operational, 15
rr_servo_set_state_pre_operational, 16

rr_servo_set_state_stopped, 16
rr_setup_nmt_callback, 17

Three servos PVT move, 49
Trajectory motion control (PVT), 23
rr_add_motion_point, 23
rr_clear_points, 24
rr_clear_points_all, 25
rr_get_points_free_space, 25
rr_get_points_size, 26
rr_get_time_calculation_result, 26
rr_invoke_time_calculation, 26
rr_start_motion, 27
tutorial/control_servo_traj_1.c, 73
tutorial/control_servo_traj_2.c, 74
tutorial/control_servo_traj_3.c, 75
tutorial/read_any_param.c, 76
tutorial/read_any_param_cache.c, 77
tutorial/read_errors.c, 78
tutorial/read_servo_max_velocity.c, 79
tutorial/read_servo_motion_queue.c, 80
tutorial/read_servo_trajectory_time.c, 82
tutorial/tutorial.h, 83
Two servos PVT move, 47