Servo UserAPI

Rozum Robotics

Contents

Chapter 1

Rozum Robotics User API & Servo Box

1.1 Servo box

- Servo box specs & manual

1.2 API Categories

- Preparing for servo intialization
- 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 parameters
- Error handling
- Debugging

1.3 API Tutorials

1. PVT trajectory for one servo
2. PVT trajectory for two servos
3. PVT trajectory for three servos
4. Reading device parameters
5. Setting up parameter cache and reading cached parameters
6. Reading device errors

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Class Index

3.1 Class List

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

Chapter 4

File Index

4.1 File List

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

Chapter 5

Module Documentation

5.1 Preparing for servo intialization

Once you have completed the servo integration procedure in accordance with the User (or servobox) manual and before you can start motion or send the majority of API commands, **it is essential to switch the servo(s) to the OPERATIONAL state**. In this case, follow the instructions below:

1. Get the servo state, using rr_net_get_state.
2. Your further actions depend on the output of the rr_net_get_state function.

   A. In case the output is PRE-OPERATIONAL:
   Send the rr_net_set_state_operational. The servo switches to the OPERATIONAL STATE, and you can start motion or send other API commands.

   Important! When the servo has some critical error(s) from previous sessions or otherwise in the PRE-OPE↩
   RATIONAL state, it will not be able to switch to OPERATIONAL. In this case, you need to reset the errors, using the rr_clear_errors function. Once the errors are reset, you can re-send the rr_net_set_state_operational command to switch the servo to OPERATIONAL.

   B.In case the ouput is STOPPED:
   Switch the servo to the PRE-OPERATIONAL state using the rr_net_set_state_pre_operational function. Then, reset errors with the rr_clear_errors function. The final step is to switch to the OPERATIONAL with rr_net_set_state_operational.

5.2 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.
- rr_ret_status_t rr_deinit_interface (rr_can_interface_t ∗∗iface)

  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 ∗iface, const uint8_t id)

  The function determines whether the servo motor with the specified 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.
- rr_ret_status_t rr_deinit_servo (rr_servo_t ∗∗servo)

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

5.2.1 Detailed Description

5.2.2 Function Documentation

5.2.2.1 rr_deinit_interface()

```
rr_ret_status_t rr_deinit_interface (
            rr_can_interface_t ** iface )
```

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

iface	Interface descriptor (see rr_init_interface).

Returns

Status code (rr_ret_status_t)

5.2.2.2 rr_deinit_servo()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.2.2.3 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 *iface = rr_init_interface ("/dev/ttyACM0");

```
if(!iface)
{
...  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

5.2.2.4 rr_init_servo()

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

The function determines whether the servo motor with the specified 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

iface	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

5.3 Switching servo working states

Functions

- void rr_setup_nmt_callback (rr_can_interface_t ∗iface, 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t rr_servo_get_state (const rr_servo_t ∗servo, rr_nmt_state_t ∗state)

  The function retrieves the actual NMT state of a specified servo. The state is described as a status code (:: rr_nmt↩ _state_t).
- rr_ret_status_t rr_servo_get_hb_stat (const rr_servo_t ∗servo, int64_t ∗min_hb_ival, int64_t ∗max_hb_ival)

  The function retrieves statistics on minimal and maximal intervals between Heartbeat messages of a servo. The statistics is saved to the variables specified in the param min_hb_ival and param max_hb_ival parameters, from where they are available for the user to perform further operations (e.g., comparison).
  The Heartbeat statistics is helpful in diagnozing and troubleshooting servo failures. For instance, when the Heartbeat interval of a servo is too long, it may mean that the control device sees the servo as being offline.
  Note:Before using the function, it is advisable to clear Heartbeat statistics with rr_servo_clear_hb_stat.
- rr_ret_status_t rr_servo_clear_hb_stat (const rr_servo_t ∗servo)

  The function clears statistics on minimal and maximal intervals between Heartbeat messages of a servo. It is advisable to use the function before attempting to get the Heartbeat statistics with the rr_servo_get_hb_stat function.
- rr_ret_status_t rr_net_reboot (const rr_can_interface_t ∗iface)

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

  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.
- rr_ret_status_t rr_net_set_state_operational (const rr_can_interface_t ∗iface)

  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.
- rr_ret_status_t rr_net_set_state_pre_operational (const rr_can_interface_t ∗iface)

  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.

- rr_ret_status_t rr_net_set_state_stopped (const rr_can_interface_t ∗iface)

  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.
- rr_ret_status_t rr_net_get_state (const rr_can_interface_t ∗iface, int id, rr_nmt_state_t ∗state)

  The function retrieves the actual NMT state of any device (a servo motor or any other) connected to the specified CAN network. The state is described as a status code (:: rr_nmt_state_t).

5.3.1 Detailed Description

5.3.2 Function Documentation

5.3.2.1 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

5.3.2.2 rr_net_get_state()

```
rr_ret_status_t rr_net_get_state (
            const rr_can_interface_t ∗ iface,
            int id,
            rr_nmt_state_t ∗ state )
```

The function retrieves the actual NMT state of any device (a servo motor or any other) connected to the specified CAN network. The state is described as a status code (:: rr_nmt_state_t).

Parameters

iface	Interface descriptor returned by the rr_init_interface function
id	Identificator of the addressed device
state	Pointer to the variable where the state of the device is returned

Returns

Status code (rr_ret_status_t)

5.3.2.3 rr_net_reboot()

```
rr_ret_status_t rr_net_reboot (
            const rr_can_interface_t * iface )
```

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

Parameters

iface	Interface descriptor returned by the rr_init_interface function

Returns

Status code (rr_ret_status_t)

5.3.2.4 rr_net_reset_communication()

```
rr_ret_status_t rr_net_reset_communication (
            const rr_can_interface_t * iface )
```

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

iface	Interface descriptor returned by the rr_init_interface function

Returns

Status code (rr_ret_status_t)

5.3.2.5 rr_net_set_state_operational()

```
rr_ret_status_t rr_net_set_state_operational (
            const rr_can_interface_t * iface )
```

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

iface	Interface descriptor returned by the rr_init_interface function

Returns

Status code (rr_ret_status_t)

5.3.2.6 rr_net_set_state_pre_operational()

```
rr_ret_status_t rr_net_set_state_pre_operational (
             const rr_can_interface_t * iface )
```

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

iface	Interface descriptor returned by the rr_init_interface function

Returns

Status code (rr_ret_status_t)

5.3.2.7 rr_net_set_state_stopped()

```
rr_ret_status_t rr_net_set_state_stopped (
             const rr_can_interface_t * iface )
```

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

iface	Interface descriptor returned by the rr_init_interface function.

Returns

Status code (rr_ret_status_t)

5.3.2.8 rr_servo_clear_hb_stat()

```
rr_ret_status_t rr_servo_clear_hb_stat (
            const rr_servo_t * servo )
```

The function clears statistics on minimal and maximal intervals between Heartbeat messages of a servo. It is advisable to use the function before attempting to get the Heartbeat statistics with the rr_servo_get_hb_stat function.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

Status code (rr_ret_status_t)

5.3.2.9 rr_servo_get_hb_stat()

```
rr_ret_status_t rr_servo_get_hb_stat (
            const rr_servo_t * servo,
            int64_t * min_hb_ival,
            int64_t * max_hb_ival )
```

The function retrieves statistics on minimal and maximal intervals between Heartbeat messages of a servo. The statistics is saved to the variables specified in the param min_hb_ival and param max_hb_ival parameters, from where they are available for the user to perform further operations (e.g., comparison).
The Heartbeat statistics is helpful in diagnozing and troubleshooting servo failures. For instance, when the Heartbeat interval of a servo is too long, it may mean that the control device sees the servo as being offline.
Note:Before using the function, it is advisable to clear Heartbeat statistics with rr_servo_clear_hb_stat.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
min_hb_ival	Pointer to the variable where the minimal arrival interval is to be saved; when set to NULL, the variable is disabled.
max_hb_ival	Pointer to the variable where the maximal arrival interval is to be saved; when set to NULL, the variable is disabled.

Returns

Status code (rr_ret_status_t), min_hb_ival & max_hb_ival - min/max arrival intervals or -1 if no information available

5.3.2.10 rr_servo_get_state()

```
rr_ret_status_t rr_servo_get_state (
            const rr_servo_t * servo,
            rr_nmt_state_t * state )
```

The function retrieves the actual NMT state of a specified servo. The state is described as a status code (:: rr_nmt_state_t).

Parameters

servo	Servo descriptor returned by the rr_init_servo function
state	Pointer to the variable where the state of the servo is returned

Returns

Status code (rr_ret_status_t)

5.3.2.11 rr_servo_reboot()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.3.2.12 rr_servo_reset_communication()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.3.2.13 rr_servo_set_state_operational()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.3.2.14 rr_servo_set_state_pre_operational()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.3.2.15 rr_servo_set_state_stopped()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.3.2.16 rr_setup_nmt_callback()

```
void rr_setup_nmt_callback (
            rr_can_interface_t * iface,
            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

iface	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

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

Functions

- rr_ret_status_t rr_release (const rr_servo_t ∗servo)

  The function sets the specified servo to the released state. The servo is de-energized and continues rotating for as long as it is affected by external forces (e.g., inertia, gravity).
- rr_ret_status_t rr_freeze (const rr_servo_t ∗servo)

  The function sets the specified servo to the freeze state. The servo stops, retaining its last position.
- rr_ret_status_t 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).
- rr_ret_status_t rr_brake_engage (const rr_servo_t ∗servo, const bool en)

  The function applies or releases the servo's built-in brake. If a servo is supplied without a brake, the function will not work. In this case, to stop a servo, use either the rr_freeze() or rr_release() function.
- rr_ret_status_t rr_set_velocity (const rr_servo_t ∗servo, const float velocity_deg_per_sec)

  The function sets the output shaft velocity with 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.
- rr_ret_status_t rr_set_velocity_motor (const rr_servo_t ∗servo, const float velocity_rpm)

  The function sets the velocity with which the motor of the specified servo should move at its maximum current. The maximum current is in accordance with the servo motor specification.
- rr_ret_status_t 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.
- rr_ret_status_t 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). The velocity value is the velocity of the servo's output shaft.
- rr_ret_status_t rr_set_position_with_limits (rr_servo_t ∗servo, const float position_deg, const float velocity↩ _deg_per_sec, const float accel_deg_per_sec_sq, uint32_t ∗time_ms)

  The function sets the position that the servo should reach with velocity and acceleration limits on generated trajectory.
- rr_ret_status_t 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.

5.4.1 Detailed Description

5.4.2 Function Documentation

5.4.2.1 rr_brake_engage()

```
rr_ret_status_t rr_brake_engage (
            const rr_servo_t ∗ servo,
            const bool en )
```

The function applies or releases the servo's built-in brake. If a servo is supplied without a brake, the function will not work. In this case, to stop a servo, use either the rr_freeze() or rr_release() function.

Parameters

servo	Servo descriptor returned by the rr_init_servo function
en	Desired action: true - engage brake, false - disengage

Returns

Status code (rr_ret_status_t)

5.4.2.2 rr_freeze()

```
rr_ret_status_t rr_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

Status code (rr_ret_status_t)

5.4.2.3 rr_release()

```
rr_ret_status_t rr_release (
             const rr_servo_t * servo )
```

The function sets the specified servo to the released state. The servo is de-energized and continues rotating for as long as it is affected by external forces (e.g., inertia, gravity).

Note: Affected by external force, the servo may also begin rotating in the opposite direction.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

Status code (rr_ret_status_t)

5.4.2.4 rr_set_current()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.4.2.5 rr_set_duty()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.4.2.6 rr_set_position()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.4.2.7 rr_set_position_with_limits()

```
rr_ret_status_t rr_set_position_with_limits (
            rr_servo_t * servo,
            const float position_deg,
            const float velocity_deg_per_sec,
            const float accel_deg_per_sec_sq,
            uint32_t * time_ms )
```

The function sets the position that the servo should reach with velocity and acceleration limits on generated trajectory.

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	Maximum Velocity on generated trajectory (in degrees/sec)
accel_deg_per_sec_sq	Maximum acceleration on generated trajectory (in degrees/(sec∗sec))
time_ms	Trajectory execution time (im milliseconds): int

Returns

Status code (rr_ret_status_t)

5.4.2.8 rr_set_velocity()

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

The function sets the output shaft velocity with 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.

The rr_set_velocity() function works with geared servos only. When a servo includes no gearhead, use the rr_set_velocity_motor() for setting servo velocity.

Parameters

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

Returns

Status code (rr_ret_status_t)

5.4.2.9 rr_set_velocity_motor()

```
rr_ret_status_t rr_set_velocity_motor (
            const rr_servo_t * servo,
            const float velocity_rpm )
```

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

You can use the function for both geared servos and servos without a gearhead. When a servo is geared, the velocity at the output flange will depend on the applied gear ratio (refer to the servo motor specification).

Parameters

servo	Servo descriptor returned by the rr_init_servo function
velocity_rpm	Velocity of the motor (in revolutions per minute)

Returns

Status code (rr_ret_status_t)

5.4.2.10 rr_set_velocity_with_limits()

```
rr_ret_status_t 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). The velocity value is the velocity of the servo's output shaft.

Similarly to rr_set_velocity(), this function can be used for geared servos only.

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

Status code (rr_ret_status_t)

5.5 Trajectory motion control (PVT)

Functions

- rr_ret_status_t 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:

- rr_ret_status_t rr_add_motion_point_pvat (const rr_servo_t ∗servo, const float position_deg, const float velocity_deg_per_sec, const float accel_deg_per_sec2, const uint32_t time_ms)

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

- rr_ret_status_t rr_start_motion (rr_can_interface_t ∗iface, 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).

- rr_ret_status_t 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. If you call the function while the servo is executing a motion point command, the servo stops without completing the motion. All the remaining motion points, including the one where the servo has been moving, are removed from the queue.

- rr_ret_status_t 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. **Note:** In case the indicated number of PVT points to be removed exceeds the actual remaining number of PVT points in the queue, the effect of applying the function is similar to that of applying rr_clear_points_all.*

- rr_ret_status_t 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.

- rr_ret_status_t 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.

- rr_ret_status_t 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, uint32_t ∗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). **Note:**The function is executed without the servo moving.*

5.5.1 Detailed Description

5.5.2 Function Documentation

5.5.2.1 rr_add_motion_point()

```
rr_ret_status_t 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
- how fast the servo should move to the specified position
- how long the movement to the specified position should take

The graphs below illustrate how a servo builds a trajectory based on the preset PVT points.
Note: In this case, the preset position values are 0, 45, 90, 45, and 0 degrees; the preset velocity values are 0, 30, 15, 30, 0 degrees per second; the time values are equal to delta time between two adjacent points on the Time axis (e.g., 2,000ms-3,000ms=1,000ms).

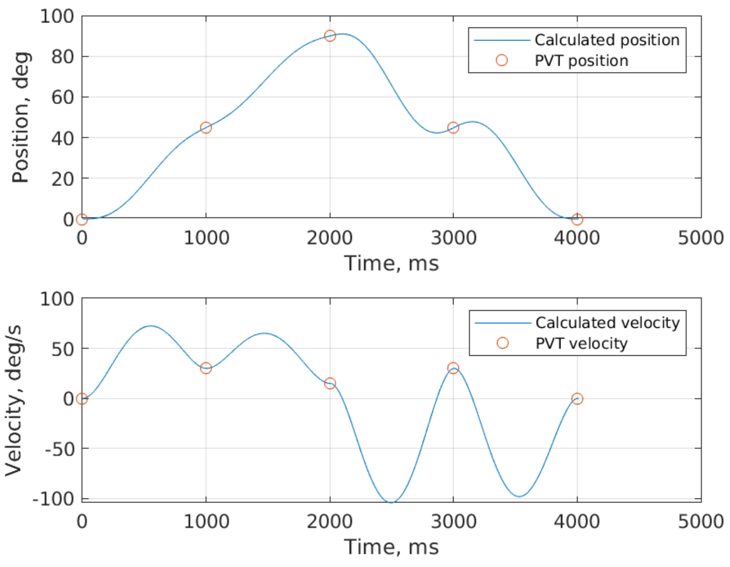


Figure 5.1 Example of calculated trajectory using PVT points

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.

Note: When you set a PVT trajectory to move more than one servo simultaneously, mind that the clock rate of the servos can differ by up to 2-3%. Therefore, if the preset PVT trajectory is rather long, servos can get desynchronized. To avoid the desynchronization, we have implemented the following mechanism:

- The device controlling the servos broadcasts a sync CAN frame to all servos on an interface. The frame should have the following format:
  ID = 0x27f, data = uint32 (4 bytes),
  Where: ‘data’ stands for the microseconds counter value by modulus of 600,000,000, starting from any value.

- Servos receive the frame and try to adjust their clock rates to that of the device using the PLL. The adjustment proper starts after the servos receive the second frame and can take up to 5 seconds, depending on the broadcasting frequency. The higher the broadcasting frequency, the less time the adjustment takes.

- The broadcasting frequency is 5 Hz minimum. The recommended frequency range is from 10 to 20 Hz. When the sync frames are not broadcast or the broadcast frequency is below 5 Hz, the clock rate of servos is as usual.

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	Servo velocity(in degrees/sec) at the point
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

Status code (rr_ret_status_t)

5.5.2.2 rr_add_motion_point_pvat()

```
rr_ret_status_t rr_add_motion_point_pvat (
                const rr_servo_t * servo,
                const float position_deg,
                const float velocity_deg_per_sec,
                const float accel_deg_per_sec2,
                const uint32_t time_ms )
```

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

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

For details of PVAT motion synchronization when running multiple servos, see rr_add_motion_point.

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	Servo velocity (in degrees/sec) at the point
accel_deg_per_sec2	Servo acceleration (in degrees/sec^2) at the point
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

Status code (rr_ret_status_t)

5.5.2.3 rr_clear_points()

```
rr_ret_status_t 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. **Note:** In case the indicated number of PVT points to be removed exceeds the actual remaining number of PVT points in the queue, the effect of applying the function is similar to that of applying rr_clear_points_all.

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. When the parameter is set to 0, the effect of applying the function is similar to that of applying rr_clear_points_all.

Returns

Status code (rr_ret_status_t)

5.5.2.4 rr_clear_points_all()

```
rr_ret_status_t 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. If you call the function while the servo is executing a motion point command, the servo stops without completing the motion. All the remaining motion points, including the one where the servo has been moving, are removed from the queue.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

Status code (rr_ret_status_t)

5.5.2.5 rr_get_points_free_space()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.5.2.6 rr_get_points_size()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.5.2.7 rr_invoke_time_calculation()

```
rr_ret_status_t 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,
            uint32_t * 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). **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)
time_ms	Pointer to the variable where the function will save the calculated time

Returns

Status code (rr_ret_status_t)

5.5.2.8 rr_start_motion()

```
rr_ret_status_t rr_start_motion (
            rr_can_interface_t * iface,
            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

iface	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

Status code (rr_ret_status_t)

5.6 Reading and writing servo configuration

Functions

- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.

5.6.1 Detailed Description

5.6.2 Function Documentation

5.6.2.1 rr_get_max_velocity()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.6.2.2 rr_set_max_velocity()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.6.2.3 rr_set_zero_position()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.6.2.4 rr_set_zero_position_and_save()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.7 Reading realtime parameters

Functions

- rr_ret_status_t 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 that was 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.
- rr_ret_status_t rr_param_cache_update_with_timestamp (rr_servo_t ∗servo)

  Same as rr_param_cache_update but with timestamp functionality.
- rr_ret_status_t 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_ret_status_t 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.
- rr_ret_status_t rr_read_parameter_with_timestamp (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value, uint32_t ∗timestamp)

  Same rr_read_parameter but with timestamp functionality.
- rr_ret_status_t 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.
- rr_ret_status_t rr_read_cached_parameter_with_timestamp (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value, uint32_t ∗timestamp)

  Same as rr_read_cached_parameter but with timestamp functionality.

5.7.1 Detailed Description

5.7.2 Function Documentation

5.7.2.1 rr_param_cache_setup_entry()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.7.2.2 rr_param_cache_update()

```
rr_ret_status_t 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 that was 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

Status code (rr_ret_status_t)

5.7.2.3 rr_param_cache_update_with_timestamp()

```
rr_ret_status_t rr_param_cache_update_with_timestamp (
            rr_servo_t * servo )
```

Same as rr_param_cache_update but with timestamp functionality.

Parameters

servo	Servo descriptor returned by the rr_init_servo function

Returns

Status code (rr_ret_status_t)

5.7.2.4 rr_read_cached_parameter()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.7.2.5 rr_read_cached_parameter_with_timestamp()

```
rr_ret_status_t rr_read_cached_parameter_with_timestamp (
            rr_servo_t * servo,
            const rr_servo_param_t param,
            float * value,
            uint32_t * timestamp )
```

Same as rr_read_cached_parameter but with timestamp functionality.

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
timestamp	pointer to variable to receive timestamp value (timestamp range is 0 to 599999999 microseconds)

Returns

Status code (rr_ret_status_t)

5.7.2.6 rr_read_parameter()

```
rr_ret_status_t 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

Status code (rr_ret_status_t)

5.7.2.7 rr_read_parameter_with_timestamp()

```
rr_ret_status_t rr_read_parameter_with_timestamp (
            rr_servo_t * servo,
            const rr_servo_param_t param,
            float * value,
            uint32_t * timestamp )
```

Same rr_read_parameter but with timestamp functionality.

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
timestamp	pointer to variable to receive timestamp value (timestamp range is 0 to 599999999 microseconds)

Returns

Status code (rr_ret_status_t)

5.8 Error handling

Functions

- void rr_setup_emcy_callback (rr_can_interface_t ∗iface, 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.
- int rr_emcy_log_get_size (rr_can_interface_t ∗iface)

  The function returns the total count of entries in the EMCY logging buffer. Each entry in the buffer contains an EMCY event that have occurred up to the moment on the servo specified in the descriptor. ***Note:****When the API library is disabled, no new entries are made in the buffer, irrespective of whether or not any events occur on the servo.*
  The function is used in combination with the rr_emcy_log_pop and rr_emcy_log_clear functions. The typical sequence is as follows:
- emcy_log_entry_t ∗ rr_emcy_log_pop (rr_can_interface_t ∗iface)

  The function enables reading entries from the EMCY logging buffer. Reading the entries is according to the "first in-first out" principle. Once an EMCY entry is read, the function removes it permenantly from the EMCY logging buffer.
  Note:*Typically, the rr_emcy_log_pop function is used in combination with the rr_emcy_log_get_size and rr_emcy_log_clear functions. For the sequence of using the functions, see rr_emcy_log_get_size.*
- void rr_emcy_log_clear (rr_can_interface_t ∗iface)

  The function clears the EMCY logging buffer, removing the total of entries from it. It is advisable to use the clearing function in the beginning of a new work session and before applying the rr_emcy_log_get_size and rr_emcy_log_pop functions. For the typical sequence of using the functions, see rr_emcy_log_get_size.
- 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_ret_status_t 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 actual count of servo hardware errors (e.g., no Heartbeats/overcurrent, etc.). In addition, the function returns the codes of all the detected errors as a single array.

5.8.1 Detailed Description

5.8.2 Function Documentation

5.8.2.1 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

5.8.2.2 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

5.8.2.3 rr_emcy_log_clear()

```
void rr_emcy_log_clear (
            rr_can_interface_t * iface )
```

The function clears the EMCY logging buffer, removing the total of entries from it. It is advisable to use the clearing function in the beginning of a new work session and before applying the rr_emcy_log_get_size and rr_emcy_log_pop functions. For the typical sequence of using the functions, see rr_emcy_log_get_size.

Parameters

iface	Descriptor of the interface returned by the rr_init_interface function

Returns

void

5.8.2.4 rr_emcy_log_get_size()

```
int rr_emcy_log_get_size (
             rr_can_interface_t * iface )
```

The function returns the total count of entries in the EMCY logging buffer. Each entry in the buffer contains an EMCY event that have occurred up to the moment on the servo specified in the descriptor. **Note:**When the API library is disabled, no new entries are made in the buffer, irrespective of whether or not any events occur on the servo.
The function is used in combination with the rr_emcy_log_pop and rr_emcy_log_clear functions. The typical sequence is as follows:

1. to clear the EMCY logging buffer with the rr_emcy_log_clear
2. to get the total count of entries in the EMCY logging buffer, using the rr_emcy_log_get_size function
3. to read the EMCY events from the buffer with the rr_emcy_log_pop function

Parameters

iface	Descriptor of the interface returned by the rr_init_interface function

Returns

int number of unread entries

5.8.2.5 rr_emcy_log_pop()

```
emcy_log_entry_t* rr_emcy_log_pop (
             rr_can_interface_t * iface )
```

The function enables reading entries from the EMCY logging buffer. Reading the entries is according to the "first in-first out" principle. Once an EMCY entry is read, the function removes it permenantly from the EMCY logging buffer.
Note: Typically, the rr_emcy_log_pop function is used in combination with the rr_emcy_log_get_size and rr_emcy_log_clear functions. For the sequence of using the functions, see rr_emcy_log_get_size.

Parameters

iface	Descriptor of the interface returned by the rr_init_interface function

Returns

emcy_log_entry_t pointer to the EMCY entry or NULL if the buffer contains no entries

5.8.2.6 rr_read_error_status()

```
rr_ret_status_t 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 actual count of servo hardware errors (e.g., no Heartbeats/overcurrent, etc.). In addition, the function returns the codes of all the detected errors as a single array.

Note: The rr_ret_status_t codes returned by API functions only indicate that an error occured during communication between the user program and a servo. If it is a hardware error, the rr_ret_status_t code will be RET_ERROR. Use rr_read_error_status to determine the cause of the error.

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. Default array size is ARRAY_ERROR_BITS_SIZE **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

Status code (rr_ret_status_t)

5.8.2.7 rr_setup_emcy_callback()

```
void rr_setup_emcy_callback (
             rr_can_interface_t * iface,
             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

iface	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

5.9 Debugging

Functions

- void rr_set_comm_log_stream (const rr_can_interface_t ∗iface, 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.

5.9.1 Detailed Description

5.9.2 Function Documentation

5.9.2.1 rr_set_comm_log_stream()

```
void rr_set_comm_log_stream (
            const rr_can_interface_t ∗ iface,
            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

iface	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

5.9.2.2 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

5.10 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.
- rr_ret_status_t rr_change_id_and_save (rr_can_interface_t ∗iface, rr_servo_t ∗∗servo, uint8_t new_can_id)

  *The function enables changing the default CAN identifier (ID) of the specified servo to avoid collisions on a bus line. **Important!** Each servo connected to a CAN bus must have **a unique ID**.*

5.10.1 Detailed Description

5.10.2 Function Documentation

5.10.2.1 rr_change_id_and_save()

```
rr_ret_status_t rr_change_id_and_save (
            rr_can_interface_t * iface,
            rr_servo_t ** servo,
            uint8_t new_can_id )
```

The function enables changing the default CAN identifier (ID) of the specified servo to avoid collisions on a bus line. **Important!** Each servo connected to a CAN bus must have **a unique ID**.

When called, the function resets CAN communication for the specified servo, checks that Heartbeats are generated for the new ID, and saves the new CAN ID to the EEPROM memory of the servo.

Note: The EEPROM memory limit is 1,000 write cycles. Therefore, it is advisable to use the function with discretion.

Parameters

iface	Descriptor of the interface (as returned by the rr_init_interface function)
servo	Servo descriptor returned by the rr_init_servo function. **Note**: All RDrive servos are supplied with **the same default CAN ID—32**.
new_can↩_id	New CAN ID. You can set any value within the range from 1 to 127, only make sure **no other servo has the same ID**.

Returns

Status code (rr_ret_status_t)

5.10.2.2 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.

Note: The user can also call system-specific sleep functions directly. However, using this sleep function is preferable to ensure compatibility with subsequent API library versions.

Parameters

ms	Idle period (in milleseconds)

Returns

void

5.11 Servo box specs & manual

5.11.1 1. Product overview

A **servobox** is a set of deliverables enabling users to easily connect, control, and operate RDrive servo motors in the designed range of loads. The set comprises the following components:

- an energy eater complete with a power supply connector (see Section 2.1)
- a capacitor module incorporating one or more capacitors (see Section 2.2)
- a CAN-USB dongle and a USB-A to Micro USB cable (1 m long) to provide communication with RDrive servos
- a 120 Ohm terminating resistor (see Section 3.2)
- a quick-start cable set comprising power supply (1 m long) and CAN (0.5 m long) cables (1 pcs of each type per servo)

It is the user's responsibility to additionally provide **a power supply unit**. The power supply unit must meet the following requirements:

- Supply voltage—48 V max
- Power equal to the total nominal power of all servo motors connected to it multiplied by a factor of 2.5 to 3

5.11.2 2. Servobox components

5.11.2.1 2.1 Energy eater

An energy eater is used to dissipate dynamic braking energy. When not dissipated, this energy can cause servos to generate voltages in excess of the power supply voltage, which can damage servos beyond repair.

The Rozum Robotics model range includes the following two models of energy eaters:

- **Model 1**—for applications with average dissipated power (at 60 degrees C max.) of **less than 25 W**

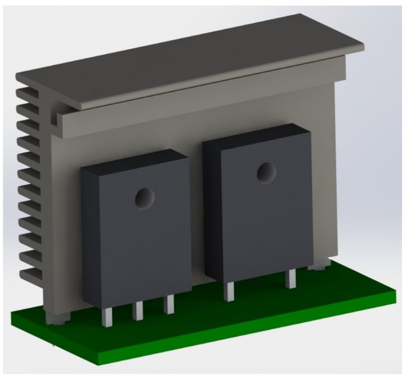


Figure 5.2 Energy eater—Model 1

- **Model 2**—for applications with average dissipated power (at 60 degrees C max.) **from 25 W to 120 W (1,000 W peak)**

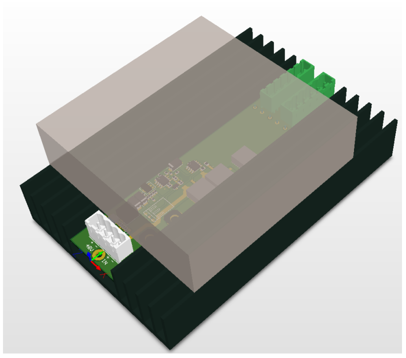


Figure 5.3 Energy eater—Model 2

When the average dissipated power of your application is less than 25 W (at 60 degrees C max.), you can also assemble an energy eater on your own based on the schematic below.

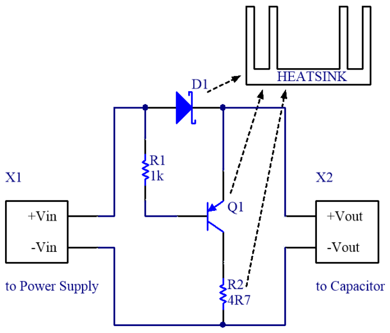


Figure 5.4 Eater module schematic

Required components:

Component	Type	Other options	Comments
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	1K Ohm, 1 W		
R2 - Resistor	4.7 Ohm, $P_d \geq 25$ W		
X1 - Connector			Input connector (from the power source to the power supply)
X2 - Connector			Output connector (from the power consumer to the capacitor and servo)

Heatsink requirements

As you can see in the Eater module schematic, D1, Q1, and R2 should be connected to an appropriate heatsink.

Because the energy eater as shown in the schematic dissipates 25 W of average power at the ambient temperature of 60 degrees C max, the heatsink should have thermal resistance of at least 1W/deg C.

To comply with the requirement, select a heatsink with the following characteristics:

- forced air convection with the flow rate of 15m3/h
- heat dissipating surface of 600 cm2

For example, this can be a heatsink as described below:

- heatsink size—100 x 100 mm
- fan size—40 x 40 mm
- fin quantity—12
- fin height—25 mm

5.11.2.2 2.2 Capacitor module

In the servobox solution, capacitors are intended to accumulate electric energy and supply it to servos. The devices allow for compensating short-term power consumption peaks that are due to inductive resistance. This is important because inductive resistance values in the servo's power supply circuit are high when the distance from a servo to the supply unit is long. Therefore, the general requirement is to place capacitors as close to servos as possible. For exact distances, refer to Section 3.1, Table 1 ("L3" column). As part of the servobox solution, capacitor modules are supplied attached to servo motors. However, you can detach them to subsequently mount at admissible distances (see Table 1, "L3" column).

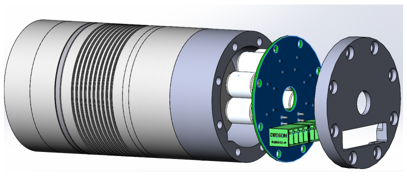


Figure 5.5 An RDrive servo with an attached capacitor module

Alternatively, you can assemble a capacitor module on your own, using the schematic below:

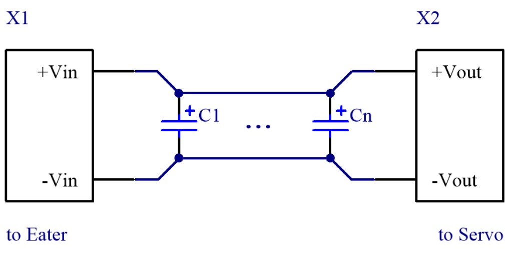


Figure 5.6 Capacitor module schematic

Requirements:

Component	Type	Comments
X1 - Connector		Input connector (power source)
X2 - Connector		Output connector (from the power consumer to the servo)
C1...Cn	Aluminum electrolytic capacitor or tantalum/polymer capacitor, U $\geq$ 80 V	Total capacitance should be $\geq$ 5 uF per 1 W of connected servo power

Note: Total capacitor ESR $\leq$ 0.1 Ohm

5.11.3 3. Connecting servos to a power supply and a servobox

To integrate an RDrive servo into one circuit with a power supply unit 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 3.1 and 3.2. For eater and capacitor requirements and schematic, see Section 2.1 and 2.2.

5.11.3.1 3.1. Power supply connection

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

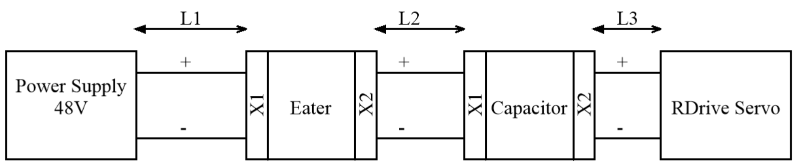


Figure 5.7 Connecting single RDrive servo to power supply

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

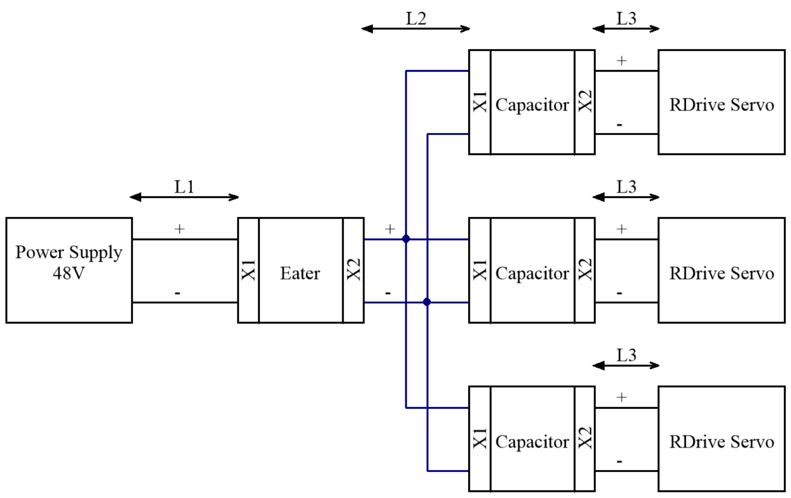


Figure 5.8 Connecting multiple RDrive servos to power supply

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

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

Table 1: Line segment lengths vs. cross-sections

Servo mode	L2						L3							
	0.75 mm^2	1.0 mm^2	1.5 mm^2	2.5 mm^2	4.0 mm^2	6.0 mm^2	0.25 mm^2	0.5 mm^2	0.75 mm^2	1.0 mm^2	1.5 mm^2	2.5 mm^2	4.0 mm^2	6.0 mm^2
R↩ D50	4 m	5 m	8 m	10 m	10 m	10 m	0.1 m	0.1 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	0.1 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	-	-	-	-	0.08 m	0.13 m	0.21 m	0.32 m

5.11.3.2 3.2. 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:

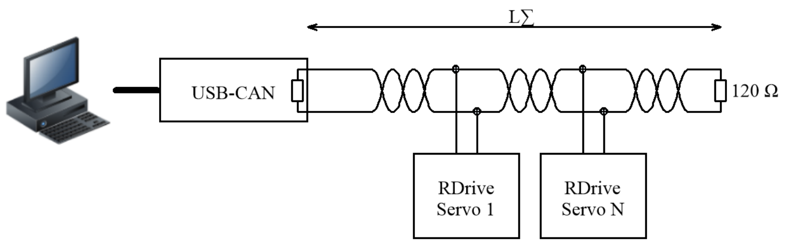


Figure 5.9 Connecting RDrive servos to USB-CAN

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

- CAN bus lines of less than 40 m long should be terminated with 120 Ohm resistors at both ends. For bus lines of over 40 m long, use 150-300 Ohm resistors.
  Note: 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.
- For the cross section of the bus line, see Table 2.
- To ensure the baud rate required for your application, LΣ should meet the specific values as indicated in Table 2.

Table 2: CAN line parameters

Baud Rate	**50 kbit/s**	**100 kbit/s**	**250 kbit/s**	**500 kbit/s**	**1 Mbit/s**
Total line length, LΣ, m	$<$ 1000	$<$ 500	$<$ 200	$<$ 100	$<$ 40
Cross-section, mm2	0.75 to 0.8	0.5 to 0.6	0.34 to 0.6	0.34 to 0.6	0.25 to 0.34

Connecting multiple servos to a CAN bus

To avoid collisions, each servo connected to a CAN bus line must have a unique CAN identifier. However, RDrive servo motors are all supplied with the same **default ID—32**. Therefore, an important step of connecting multiple RDrive servos to a single CAN bus line is to change their default IDs to unique ones.

Start with connecting each of your multiple servos, one by one, to a CAN bus line, following the sequence as described below:

Caution! Never connect or disconnect servos when power supply is on!

1. Take servo 1 and connect it to the CAN bus line as desribed in this section above.
2. Run the Changing CAN ID of a single servo tutorial to change the servo's default ID.
3. Remember or write down the newly assigned CAN ID and disconnect the servo.

4. Repeat Steps 1 to 3 for all the servos you want to connect to the same CAN bus line.

Once you have changed all CAN IDs as appropriate, you can connect all the servos back to the CAN bus line and start working with them.

Note: In total, you can connect up to 127 devices to a single CAN bus. So, the admissible CAN ID pool is from 1 to 127.

5.12 PVT trajectory for one servo

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-velocity-time) 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(argv[1]);
```

2. Initialize the servo.

```
rr_servo_t *servo = rr_init_servo(iface, 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(argv[1]);

rr_servo_t *servo = rr_init_servo(iface, id);

rr_servo_set_state_operational(servo);

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
```

5.13 PVT trajectory for two servos

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-velocity-time) 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

Important!Before setting up PVT points, make sure to change the default CAN ID of at least one of the servos (see the Changing CAN ID of a single servo tutorial).

Note: When you set a PVT trajectory to move more than one servo simultaneously, mind that the clock rate of the servos can differ by up to 2-3%. Therefore, if the preset PVT trajectory is rather long, servos can get desynchronized. To avoid the desynchronization, we have implemented the following mechanism:

- The device controlling the servos broadcasts a sync CAN frame to all servos on an interface. The frame should have the following format:
  ID = 0x27f, data = uint32 (4 bytes),
  Where: 'data' stands for the microseconds counter value by modulus of 600,000,000, starting from any value.
- Servos receive the frame and try to adjust their clock rates to that of the device using the PLL. The adjustment proper starts after the servos receive the second frame and can take up to 5 seconds, depending on the broadcasting frequency. The higher the broadcasting frequency, the less time the adjustment takes.
- The broadcasting frequency is 5 Hz minimum. The recommended frequency range is from 10 to 20 Hz. When the sync frames are not broadcast or the broadcast frequency is below 5 Hz, the clock rate of servos is as usual.

1. Initialize the interface.
   ```
   rr_can_interface_t *iface = rr_init_interface(argv[1]);
   ```
2. Initialize servo 1.
   ```
   rr_servo_t *servo1 = rr_init_servo(iface, id1);
   ```
3. Initialize servo 2.
   ```
   rr_servo_t *servo2 = rr_init_servo(iface, id2);
   ```
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(argv[1]);
rr_servo_t *servo1 = rr_init_servo(iface, id1);
rr_servo_t *servo2 = rr_init_servo(iface, id2);

rr_servo_set_state_operational(servo1);
rr_servo_set_state_operational(servo2);

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
```

5.14 PVT trajectory for three servos

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-velocity-time) 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

Important!Before setting up PVT points, make sure to change the default CAN IDs of the servos (see the Changing CAN ID of a single servo tutorial).

Note: When you set a PVT trajectory to move more than one servo simultaneously, mind that the clock rate of the servos can differ by up to 2-3%. Therefore, if the preset PVT trajectory is rather long, servos can get desynchronized. To avoid the desynchronization, we have implemented the following mechanism:

- The device controlling the servos broadcasts a sync CAN frame to all servos on an interface. The frame should have the following format:
  ID = 0x27f, data = uint32 (4 bytes),
  Where: ‘data’ stands for the microseconds counter value by modulus of 600,000,000, starting from any value.
- Servos receive the frame and try to adjust their clock rates to that of the device using the PLL. The adjustment proper starts after the servos receive the second frame and can take up to 5 seconds, depending on the broadcasting frequency. The higher the broadcasting frequency, the less time the adjustment takes.
- The broadcasting frequency is 5 Hz minimum. The recommended frequency range is from 10 to 20 Hz. When the sync frames are not broadcast or the broadcast frequency is below 5 Hz, the clock rate of servos is as usual.

1. Initialize the interface.
```
rr_can_interface_t *iface = rr_init_interface(argv[1]);
```
2. Initialize servo 1.
```
rr_servo_t *servo1 = rr_init_servo(iface, id1);
```
3. Initialize servo 2.
```
rr_servo_t *servo2 = rr_init_servo(iface, id2);
```
4. Initialize servo 3.
```
rr_servo_t *servo3 = rr_init_servo(iface, id3);
```
5. Clear points in the motion queue of servo 1.
```
rr_clear_points_all(servo1);
```
6. Clear points in the motion queue of servo 2.
```
rr_clear_points_all(servo2);
```
7. Clear points in the motion queue of servo 3.
```
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(argv[1]);
rr_servo_t *servo1 = rr_init_servo(iface, id1);
rr_servo_t *servo2 = rr_init_servo(iface, id2);
rr_servo_t *servo3 = rr_init_servo(iface, id3);

rr_servo_set_state_operational(servo1);
rr_servo_set_state_operational(servo2);
rr_servo_set_state_operational(servo3);

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
```

5.15 Setting up parameter cache and reading cached parameters

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(argv[1]);
   ```

2. Initialize the servo.

   ```
   rr_servo_t *servo = rr_init_servo(iface, 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(argv[1]);
rr_servo_t *servo = rr_init_servo(iface, id);

API_DEBUG("========== Tutorial of %s ==========\n", "programming and reading the device
   parameter cache");

rr_param_cache_setup_entry(servo,
  APP_PARAM_POSITION_ROTOR, true);
rr_param_cache_setup_entry(servo,
  APP_PARAM_VELOCITY_ROTOR, true);
rr_param_cache_setup_entry(servo,
  APP_PARAM_VOLTAGE_INPUT, true);
rr_param_cache_setup_entry(servo,
  APP_PARAM_CURRENT_INPUT, true);

rr_param_cache_update(servo);

float value;

rr_read_cached_parameter(servo,
  APP_PARAM_POSITION_ROTOR, &value);
API_DEBUG("\t%s value: %.3f\n", STRFY(APP_PARAM_POSITION_ROTOR),
  value);

rr_read_cached_parameter(servo,
  APP_PARAM_VELOCITY_ROTOR, &value);
API_DEBUG("\t%s value: %.3f\n", STRFY(APP_PARAM_VELOCITY_ROTOR),
  value);

rr_read_cached_parameter(servo,
  APP_PARAM_VOLTAGE_INPUT, &value);
API_DEBUG("\t%s value: %.3f\n", STRFY(APP_PARAM_VOLTAGE_INPUT),
  value);

rr_read_cached_parameter(servo,
  APP_PARAM_CURRENT_INPUT, &value);
API_DEBUG("\t%s value: %.3f\n", STRFY(APP_PARAM_CURRENT_INPUT),
  value);
```

5.16 Reading device parameters

The tutorial describes how to read a sequence of single variables representing actual servo 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 parameter cache and reading cached parameters**.

1. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(argv[1]);
```

2. Initialize the servo.

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

Reading current device parameters

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

```
float value;
```

4. Read the actual rotor position.

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

5. Read the actual rotor velocity.

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

6. Read the actual input voltage.

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

7. Read the actual input current.

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

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(argv[1]);

rr_servo_t *servo = rr_init_servo(iface, id);

API_DEBUG("========== Tutorial of %s ==========\n", "reading any device parameter (single)");

float value;

rr_read_parameter(servo, APP_PARAM_POSITION_ROTOR, &value);
API_DEBUG("\t%s value: %.3f\n", STRFY(APP_PARAM_POSITION_ROTOR),
  value);

rr_read_parameter(servo, APP_PARAM_VELOCITY_ROTOR, &value);
API_DEBUG("\t%s value: %.3f\n", STRFY(APP_PARAM_VELOCITY_ROTOR),
  value);

rr_read_parameter(servo, APP_PARAM_VOLTAGE_INPUT, &value);
API_DEBUG("\t%s value: %.3f\n", STRFY(APP_PARAM_VOLTAGE_INPUT),
  value);

rr_read_parameter(servo, APP_PARAM_CURRENT_INPUT, &value);
API_DEBUG("\t%s value: %.3f\n", STRFY(APP_PARAM_CURRENT_INPUT),
  value);
```

5.17 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(argv[1]);
```

2. Initialize the servo.

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

Calculating the time to reach the specified position

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

```
uint32_t travel_time;
```

4. 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, &travel_time);
if(status != RET_OK)
{
    API_DEBUG("Error in the trjectory point calculation\n");
    return 1;
}
```

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(argv[1]);
rr_servo_t *servo = rr_init_servo(iface, id);

API_DEBUG("========== Tutorial of the %s ==========\n", "trajectory calculation");

uint32_t travel_time;

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

API_DEBUG("\tCalculated travel time: %d ms.\n", travel_time);
```

5.18 Reading maximum servo 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(argv[1]);
```

2. Initialize the servo.

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

Reading the maximum servo velocity

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

```
float velocity;
```

4. Read the maximum servo velocity.

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

Complete tutorial code:

```
rr_can_interface_t *iface = rr_init_interface(argv[1]);
rr_servo_t *servo = rr_init_servo(iface, id);

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

float velocity;
rr_get_max_velocity(servo, &velocity);

API_DEBUG("\tMax velocity: %.3f Deg/sec\n", velocity);
```

5.19 Reading motion queue parameters

This tutorial describes how to determine the actual 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(argv[1]);
   ```

2. Initialize the servo.

   ```
   rr_servo_t *servo = rr_init_servo(iface, 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(argv[1]);
rr_servo_t *servo = rr_init_servo(iface, id);

API_DEBUG("========== Tutorial of the %s ==========\n", "reading motion queue parameters");

API_DEBUG("Clearing points\n");
rr_clear_points_all(servo);


uint32_t num;
rr_get_points_size(servo, &num);
API_DEBUG("\tPoints in the queue before: %d\n", num);

rr_get_points_free_space(servo, &num);
API_DEBUG("\tPoints queue free size before: %d\n", num);

API_DEBUG("Appending points\n");

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

rr_get_points_size(servo, &num);
API_DEBUG("\tPoints in the queue after: %d\n", num);

rr_get_points_free_space(servo, &num);
API_DEBUG("\tPoints queue free size after: %d\n", num);
```

5.20 Changing CAN ID of a single servo

The tutorial describes how to change the CAN identifier of a single servo and save it to the EEPROM memory.

Important! RDrive servos are all supplied with **the same default CAN IDs ranging from 32 to 37.**. To avoid collisions, you need to assign **a unique CAN ID** to each servo on the same CAN bus.

1. Create two variables—one for the default (current) CAN ID and one for the new CAN ID.

```
uint8_t id_old;
uint8_t id_new;
```

2. Set the default (current) ID and a new one to replace it. In the example below, we get the current and new ID values from the console return.

```
if(argc == 4)
{
    id_old = strtol(argv[2], NULL, 0);
    id_new = strtol(argv[3], NULL, 0);
}
else
{
    API_DEBUG("Wrong format!\nUsage: %s interface old_id new_id\n", argv[0]);
    return 1;
}
```

3. Initialize the interface.

```
rr_can_interface_t *iface = rr_init_interface(argv[1]);
```

4. Initialize the servo.

```
rr_servo_t *servo = rr_init_servo(iface, id_old);
if(servo == NULL)
{
    API_DEBUG("Failed to init servo with ID: %d\n", id_old);
    return 1;
}
```

5. Change the default (current) servo ID to a new one. You use any value from 1 to 127, only make sure no other servo on the CAN bus has the same ID.

```
rr_ret_status_t status = rr_change_id_and_save(iface, &servo,
  id_new);
if(status != RET_OK)
{
    API_DEBUG("Failed to change servo CAN ID and save it to the EEPROM: %d\n", status);
    return 1;
}
```

Complete tutorial code:

```
uint8_t id_old;
uint8_t id_new;

if(argc == 4)
{
    id_old = strtol(argv[2], NULL, 0);
    id_new = strtol(argv[3], NULL, 0);
}
else
{
    API_DEBUG("Wrong format!\nUsage: %s interface old_id new_id\n", argv[0]);
    return 1;
}
rr_can_interface_t *iface = rr_init_interface(argv[1]);
rr_servo_t *servo = rr_init_servo(iface, id_old);
if(servo == NULL)
{
    API_DEBUG("Failed to init servo with ID: %d\n", id_old);
    return 1;
}

API_DEBUG("========== Tutorial of the %s ==========\n", "changing CAN ID of one servo and
   saving it to the EEPROM");
rr_ret_status_t status = rr_change_id_and_save(iface, &servo,
  id_new);
if(status != RET_OK)
{
    API_DEBUG("Failed to change servo CAN ID and save it to the EEPROM: %d\n", status);
    return 1;
}
```

5.21 Calibrating to mitigate cogging effects

The tutorial describes how to calibrate a servo to optimize its motion trajectory, while compensating for torque fluctuations due to cogging effects.

1. Read the parameters as required to run the cogging calibration tutorial.

```
int main(int argc, char *argv[])
{
    uint8_t id;

    if(argc == 3)
    {
        id = strtol(argv[2], NULL, 0);
    }
    else
    {
        API_DEBUG("Wrong format!\nUsage: %s interface id\n", argv[0]);
        return 1;
    }
```

2. Initialize the interface.

```
    rr_can_interface_t *iface = rr_init_interface(argv[1]);
```

3. Initialize the servo.

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

4. Set the servo to the operational state.

```
    rr_servo_set_state_operational(servo);

    rr_nmt_state_t state = 0;
    for(int i = 0; i < 20; i++)
    {
        rr_sleep_ms(100);
        rr_servo_get_state(servo, &state);
        if(state == RR_NMT_OPERATIONAL)
        {
            break;
        }
    }
    if(state != RR_NMT_OPERATIONAL)
    {
        API_DEBUG("Can't switch tot operational mode\n");
        exit(1);
    }
```

5. Start calibration. During the procedure, the servo rotates to +/-45 degrees relative to its start position.

```
    uint8_t cogging_cmd[] = {0x1a, 0, 0, 0, 0};
    float value = 0, value_prev = 0;

    if(rr_write_raw_sdo(servo, 0x4010, 0, (uint8_t *)cogging_cmd, sizeof(cogging_cmd), 1, 100) !=
      RET_OK)
    {
        API_DEBUG("Can't start calibration\n");
        exit(1);
    }
```

6. Read the VELOCITY_SETPOINT parameter from time to time until the output is 0. As soon as the parameter reaches the value, calibration is over.

```
    while(true)
    {
        rr_sleep_ms(100);
        rr_read_parameter(servo,
      APP_PARAM_CONTROLLER_VELOCITY_SETPOINT, &value);
        if(value != value_prev)
        {
            API_DEBUG("APP_PARAM_CONTROLLER_VELOCITY_SETPOINT value: %.3f\n", value);
        }
        value_prev = value;

        if(fabsf(value) < 1)
        {
            API_DEBUG("Calibration finished\n");
            break;
        }
    }
```

7. Enable the resulting cogging compensation table.

```
    value = 1.0;
    if(rr_write_raw_sdo(servo, 0x41ff, 15, (uint8_t *)&value, sizeof(value), 1, 100) !=
      RET_OK)
    {
        API_DEBUG("Can't enable cogging map\n");
        exit(1);
    }

    if(rr_write_raw_sdo(servo, 0x41ff, 16, (uint8_t *)&value, sizeof(value), 1, 100) !=
      RET_OK)
    {
        API_DEBUG("Can't enable friction map\n");
        exit(1);
    }
```

8. Save the cogging compensation table to the FLASH memory.

```
    API_DEBUG("Saving to flash\n");

    if(rr_write_raw_sdo(servo, 0x1010, 1, (uint8_t *)"evas", 4, 1, 4000) !=
      RET_OK)
    {
        API_DEBUG("Can't save to flash\n");
    }
```

Complete tutorial code:

```
int main(int argc, char *argv[])
{
    uint8_t id;

    if(argc == 3)
    {
        id = strtol(argv[2], NULL, 0);
    }
    else
    {
        API_DEBUG("Wrong format!\nUsage: %s interface id\n", argv[0]);
        return 1;
    }

    rr_can_interface_t *iface = rr_init_interface(argv[1]);

    rr_servo_t *servo = rr_init_servo(iface, id);

    rr_servo_set_state_operational(servo);

    rr_nmt_state_t state = 0;
    for(int i = 0; i < 20; i++)
    {
        rr_sleep_ms(100);
        rr_servo_get_state(servo, &state);
        if(state == RR_NMT_OPERATIONAL)
        {
            break;
        }
    }
    if(state != RR_NMT_OPERATIONAL)
    {
        API_DEBUG("Can't switch tot operational mode\n");
        exit(1);
    }

    uint8_t cogging_cmd[] = {0x1a, 0, 0, 0, 0};
    float value = 0, value_prev = 0;

    if(rr_write_raw_sdo(servo, 0x4010, 0, (uint8_t *)cogging_cmd, sizeof(cogging_cmd), 1, 100) !=
      RET_OK)
    {
        API_DEBUG("Can't start calibration\n");
        exit(1);
    }

    while(true)
    {
        rr_sleep_ms(100);
        rr_read_parameter(servo,
      APP_PARAM_CONTROLLER_VELOCITY_SETPOINT, &value);
        if(value != value_prev)
        {
            API_DEBUG("APP_PARAM_CONTROLLER_VELOCITY_SETPOINT value: %.3f\n", value);
        }
        value_prev = value;

        if(fabsf(value) < 1)
        {
            API_DEBUG("Calibration finished\n");
            break;
        }
    }

    value = 1.0;
    if(rr_write_raw_sdo(servo, 0x41ff, 15, (uint8_t *)&value, sizeof(value), 1, 100) !=
      RET_OK)
    {
        API_DEBUG("Can't enable cogging map\n");
        exit(1);
    }

    if(rr_write_raw_sdo(servo, 0x41ff, 16, (uint8_t *)&value, sizeof(value), 1, 100) !=
      RET_OK)
    {
        API_DEBUG("Can't enable friction map\n");
        exit(1);
    }

    API_DEBUG("Saving to flash\n");

    if(rr_write_raw_sdo(servo, 0x1010, 1, (uint8_t *)"evas", 4, 1, 4000) !=
      RET_OK)
    {
        API_DEBUG("Can't save to flash\n");
    }
```

5.22 Checking calibration quality

The tutorial describes how to verify the results of the calibration to mitigate cogging effects. The verification is by reading rotor position and phase current at 4 RPM and -4RPM into CSV files. Based on the files, dependance curves are built, making it possible to evaluate whether current settings of the cogging compensation table provide the required mitigation effect.

1. Read the parameters as required to run the tutorial to check the calibration quality.

```
int main(int argc, char *argv[])
{
    uint8_t id;
    struct timespec tprev, tnow, tstart;
    float pos, curr;
    FILE *f;

    if(argc == 3)
    {
        id = strtol(argv[2], NULL, 0);
    }
    else
    {
        API_DEBUG("Wrong format!\nUsage: %s interface id\n", argv[0]);
        return 1;
    }
```

2. Initialize the interface.

```
    rr_can_interface_t *iface = rr_init_interface(argv[1]);
```

3. Initialize the servo.

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

4. Set the servo to the operational state.

```
    rr_servo_set_state_operational(servo);

    rr_nmt_state_t state = 0;
    for(int i = 0; i < 20; i++)
    {
        rr_sleep_ms(100);
        rr_servo_get_state(servo, &state);
        if(state == RR_NMT_OPERATIONAL)
        {
            break;
        }
    }
    if(state != RR_NMT_OPERATIONAL)
    {
        API_DEBUG("Can't switch tot operational mode\n");
        exit(1);
    }
```

5. Set up a parameter array to read, comprising:
   - rotor position
   - phase current

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

6. Disable the cogging compensation table.

```
    enable_compensation(servo, false);
```

7. Set the servo to rotate at 4 RPM.

```
rr_set_velocity_motor(servo, 4.0);
API_DEBUG("Setting 4 RPM @ motor without cogging comp.\n");
API_DEBUG("Collecting data (~60sec) ...\n");
```

8. Open a CSV file to save the measured values of the parameters you set at Step 5.

```
f = fopen("without_comp.csv", "w+");
```

9. Record the start time of the measurements.

```
fprintf(f, "time_us, mpos_deg, mcurr_amp\n");
clock_gettime(CLOCK_REALTIME, &tnow);
tprev = tnow;
tstart = tnow;
```

10. Start a measurement cycle, saving measured values. The total measurement cycle time is 1 minute.

```
while(true)
{
    clock_gettime(CLOCK_REALTIME, &tnow);
    rr_param_cache_update(servo);
    rr_read_cached_parameter(servo,
  APP_PARAM_POSITION_ROTOR, &pos);
    rr_read_cached_parameter(servo,
  APP_PARAM_CURRENT_PHASE, &curr);

    fprintf(f, "%" PRId64 ", %f, %f\n", calcdiff(tnow, tprev), pos, curr);

    if(calcdiff(tnow, tstart) >= 60000000)
    {
        break;
    }
}
```

11. Close the CSV file you opened at Step 8.

```
fclose(f);
```

12. Enable the cogging compensation table.

```
enable_compensation(servo, true);
```

13. Set the servo to rotate at -4RPM.

```
rr_set_velocity_motor(servo, -4.0);
API_DEBUG("Setting -4 RPM @ motor with cogging comp.\n");
API_DEBUG("Collecting data (~60sec) ...\n");
```

14. Open a new CSV file to save measurements.

```
f = fopen("with_comp.csv", "w+");
```

15. Record the start time of the measurements.

```
fprintf(f, "time_us, mpos_deg, mcurr_amp\n");
clock_gettime(CLOCK_REALTIME, &tnow);
tprev = tnow;
tstart = tnow;
```

16. Start a new measurement cycle, saving measured values. The total measurement cycle time is 1 minute.

```
while(true)
{
    clock_gettime(CLOCK_REALTIME, &tnow);
    rr_param_cache_update(servo);
    rr_read_cached_parameter(servo,
  APP_PARAM_POSITION_ROTOR, &pos);
    rr_read_cached_parameter(servo,
  APP_PARAM_CURRENT_PHASE, &curr);

    fprintf(f, "%" PRId64 ", %f, %f\n", calcdiff(tnow, tprev), pos, curr);

    if(calcdiff(tnow, tstart) >= 60000000)
    {
        break;
    }
}
```

17. Close the CSV file you opened at Step 14.

    ```
    fclose(f);
    ```

    As a result, you get two CSV files with measured rotor position ad phase current. Use the files to build dependance curves to evaluate whether current settings of the cogging compensation table have provided the required mitigation effect.

18. Set the servo to the released state.

    ```
    rr_release(servo);
    ```

Complete tutorial code:

```
void enable_compensation(rr_servo_t *servo, bool en)
{
    float value = en ? 1.0 : 0.0;
    if(rr_write_raw_sdo(servo, 0x41ff, 15, (uint8_t *)&value, sizeof(value), 1, 100) !=
      RET_OK)
    {
        API_DEBUG("Can't enable cogging map\n");
        exit(1);
    }

    if(rr_write_raw_sdo(servo, 0x41ff, 16, (uint8_t *)&value, sizeof(value), 1, 100) !=
      RET_OK)
    {
        API_DEBUG("Can't enable friction map\n");
        exit(1);
    }
}

#define USEC_PER_SEC            1000000

int64_t calcdiff(struct timespec t1, struct timespec t2)
{
  int64_t diff;
  diff = USEC_PER_SEC * (long long)((int) t1.tv_sec - (int) t2.tv_sec);
  diff += ((int) t1.tv_nsec - (int) t2.tv_nsec) / 1000;
  return diff;
}
int main(int argc, char *argv[])
{
    uint8_t id;
    struct timespec tprev, tnow, tstart;
    float pos, curr;
    FILE *f;

    if(argc == 3)
    {
        id = strtol(argv[2], NULL, 0);
    }
    else
    {
        API_DEBUG("Wrong format!\nUsage: %s interface id\n", argv[0]);
        return 1;
    }
    rr_can_interface_t *iface = rr_init_interface(argv[1]);
    rr_servo_t *servo = rr_init_servo(iface, id);
    rr_servo_set_state_operational(servo);

    rr_nmt_state_t state = 0;
    for(int i = 0; i < 20; i++)
    {
        rr_sleep_ms(100);
        rr_servo_get_state(servo, &state);
        if(state == RR_NMT_OPERATIONAL)
        {
            break;
        }
    }
    if(state != RR_NMT_OPERATIONAL)
    {
        API_DEBUG("Can't switch tot operational mode\n");
        exit(1);
    }
    rr_param_cache_setup_entry (servo,
      APP_PARAM_POSITION_ROTOR, true);
    rr_param_cache_setup_entry (servo,
      APP_PARAM_CURRENT_PHASE, true);
    enable_compensation(servo, false);
    rr_set_velocity_motor(servo, 4.0);
```

```
API_DEBUG("Setting 4 RPM @ motor without cogging comp.\n");
API_DEBUG("Collecting data (~60sec) ...\n");
f = fopen("without_comp.csv", "w+");

fprintf(f, "time_us, mpos_deg, mcurr_amp\n");
clock_gettime(CLOCK_REALTIME, &tnow);
tprev = tnow;
tstart = tnow;

while(true)
{
    clock_gettime(CLOCK_REALTIME, &tnow);
    rr_param_cache_update(servo);
    rr_read_cached_parameter(servo,
  APP_PARAM_POSITION_ROTOR, &pos);
    rr_read_cached_parameter(servo,
  APP_PARAM_CURRENT_PHASE, &curr);

    fprintf(f, "%" PRId64 ", %f, %f\n", calcdiff(tnow, tprev), pos, curr);

    if(calcdiff(tnow, tstart) >= 60000000)
    {
        break;
    }
}

fclose(f);

enable_compensation(servo, true);

rr_set_velocity_motor(servo, -4.0);
API_DEBUG("Setting -4 RPM @ motor with cogging comp.\n");
API_DEBUG("Collecting data (~60sec) ...\n");

f = fopen("with_comp.csv", "w+");

fprintf(f, "time_us, mpos_deg, mcurr_amp\n");
clock_gettime(CLOCK_REALTIME, &tnow);
tprev = tnow;
tstart = tnow;

while(true)
{
    clock_gettime(CLOCK_REALTIME, &tnow);
    rr_param_cache_update(servo);
    rr_read_cached_parameter(servo,
  APP_PARAM_POSITION_ROTOR, &pos);
    rr_read_cached_parameter(servo,
  APP_PARAM_CURRENT_PHASE, &curr);

    fprintf(f, "%" PRId64 ", %f, %f\n", calcdiff(tnow, tprev), pos, curr);

    if(calcdiff(tnow, tstart) >= 60000000)
    {
        break;
    }
}

fclose(f);

rr_release(servo);
        }
```

5.23 Detecting available CAN devices

The tutorial demonstrates how to identify all devices connected to a CAN interface and to get device data, such as device name, as well as its hardware and software versions.

1. Create variables to save the states of potentially available CAN devices. The total number of variables is up to 128.

```
int main(int argc, char *argv[])
{
    rr_nmt_state_t states[MAX_CO_DEV];
    rr_nmt_state_t state;

    if(argc != 2)
    {
        API_DEBUG("Wrong format!\nUsage: %s interface\n", argv[0]);
        return 1;
    }
```

2. Initiate the interface.

```
    rr_can_interface_t *iface = rr_init_interface(argv[1]);
```

3. Enable the RR_NMT_HB_TIMEOUT variable.

```
    for(int i = 0; i < MAX_CO_DEV; i++)
    {
        states[i] = RR_NMT_HB_TIMEOUT;
    }
```

4. Run a work cycle to scan the CAN bus, reading the states of potentially available CAN devices every 100 ms. Compare the read values with the previous states.

```
    while(true)
    {
        rr_sleep_ms(100);

        for(int i = 0; i < MAX_CO_DEV; i++)
        {
            rr_net_get_state(iface, i, &state);
            if(states[i] != state)
            {
                states[i] = state;
                if(state != RR_NMT_HB_TIMEOUT)
                {
                    get_dev_info(iface, i);
                }
                else
                {
                    API_DEBUG("DEVICE %d disappeared\n", i);
                }
            }
        }
    }
```

5. Run an auxiliary function to display the following data about available CAN devices:

   - Hardware and software version
   - Device name

```
void get_dev_info(rr_can_interface_t *iface, int id)
{
    rr_servo_t *servo = rr_init_servo(iface, id);

    char name[1024];
    char hw[1024];
    char sw[1024];

        sdo_read_str(servo, 0x1008, 0, name, sizeof(name));
        sdo_read_str(servo, 0x1009, 0, hw, sizeof(hw));
        sdo_read_str(servo, 0x100a, 0, sw, sizeof(sw));

    API_DEBUG("Device %d:\n  NAME: %s\n  HW: %s\n  SW: %s\n", id,
            name[0] ? name : "N/A",
            hw[0] ? hw : "N/A",
            sw[0] ? sw : "N/A");

    rr_deinit_servo(&servo);
}
```

The same sequence of scanning the CAN bus repeats multiple times until you stop the program by pressing the Ctrl-C hotkey combination.

Complete tutorial code:

```
void sdo_read_str(rr_servo_t *servo, uint16_t idx, uint8_t sidx, char *buf, int
      max_sz)
{
    if(!buf)
    {
        return;
    }
    if(max_sz <= 0)
    {
        buf[0] = 0;
    }
    max_sz--;
    if(rr_read_raw_sdo(servo, idx, sidx, buf, &max_sz, 1, 200) == RET_OK)
    {
        buf[max_sz] = '\0';
    }
    else
    {
        buf[0] = '\0';
    }

}

void get_dev_info(rr_can_interface_t *iface, int id)
{
    rr_servo_t *servo = rr_init_servo(iface, id);

    char name[1024];
    char hw[1024];
    char sw[1024];

        sdo_read_str(servo, 0x1008, 0, name, sizeof(name));
        sdo_read_str(servo, 0x1009, 0, hw, sizeof(hw));
        sdo_read_str(servo, 0x100a, 0, sw, sizeof(sw));

    API_DEBUG("Device %d:\n  NAME: %s\n  HW: %s\n  SW: %s\n", id,
            name[0] ? name : "N/A",
            hw[0] ? hw : "N/A",
            sw[0] ? sw : "N/A");

    rr_deinit_servo(&servo);
}

int main(int argc, char *argv[])
{
    rr_nmt_state_t states[MAX_CO_DEV];
    rr_nmt_state_t state;

    if(argc != 2)
    {
        API_DEBUG("Wrong format!\nUsage: %s interface\n", argv[0]);
        return 1;
    }

    rr_can_interface_t *iface = rr_init_interface(argv[1]);

    for(int i = 0; i < MAX_CO_DEV; i++)
    {
        states[i] = RR_NMT_HB_TIMEOUT;
    }
    API_DEBUG("Waiting for devices ...\nPress Ctrl-C to stop\n");

    while(true)
    {
        rr_sleep_ms(100);

        for(int i = 0; i < MAX_CO_DEV; i++)
        {
            rr_net_get_state(iface, i, &state);
            if(states[i] != state)
            {
                states[i] = state;
                if(state != RR_NMT_HB_TIMEOUT)
                {
                    get_dev_info(iface, i);
                }
                else
                {
                    API_DEBUG("DEVICE %d disappeared\n", i);
                }
            }
```

```
                }
            }
        }
```

5.24 Reading emergency (EMY) log

The tutorial describes how to get and read messages from the emergency log buffer. The EMCY log stores EMCY messages about any events when a servo is out of normal operating state (e.g., overcurrent, high temperature)or goes back to the normal state.

1. Read the parameters as required to run the tutorial.

```
int main(int argc, char *argv[])
{
    uint8_t id;
    emcy_log_entry_t *emcy;

    if(argc == 3)
    {
        id = strtol(argv[2], NULL, 0);
    }
    else
    {
        API_DEBUG("Wrong format!\nUsage: %s interface id\n", argv[0]);
        return 1;
    }
```

2. Initiate the interface.

```
    rr_can_interface_t *iface = rr_init_interface(argv[1]);
```

3. Initiate the servo.

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

4. Read EMCY messages from the EMCY log.

```
    while((emcy = rr_emcy_log_pop(iface)))
    {
        printf("id: %d, code: 0x%." PRIx16 ", reg: 0x%.2" PRIx8 ", bits: 0x%.2" PRIx8 ", info: 0x%.8"
      PRIx32 "\n",
            (int)emcy->id, emcy->err_code, emcy->err_reg, emcy->err_bits, emcy->err_info);
    }
```

Reading the messages is according to the first in-fist out pinciple. The function continues running until it reads the last EMCY message from the log.

Complete tutorial code:

```
int main(int argc, char *argv[])
{
    uint8_t id;
    emcy_log_entry_t *emcy;

    if(argc == 3)
    {
        id = strtol(argv[2], NULL, 0);
    }
    else
    {
        API_DEBUG("Wrong format!\nUsage: %s interface id\n", argv[0]);
        return 1;
    }

    rr_can_interface_t *iface = rr_init_interface(argv[1]);

    rr_servo_t *servo = rr_init_servo(iface, id);

    while((emcy = rr_emcy_log_pop(iface)))
    {
        printf("id: %d, code: 0x%." PRIx16 ", reg: 0x%.2" PRIx8 ", bits: 0x%.2" PRIx8 ", info: 0x%.8"
      PRIx32 "\n",
            (int)emcy->id, emcy->err_code, emcy->err_reg, emcy->err_bits, emcy->err_info);
    }
```

5.25 Checking PVT points

The tutorial demonstrates how to verify validity of PVT points in a servo trajectory without actually connecting a servo, while checking for PVT velocities over maximum limits. Maximum velocity limits (degrees per seconds) for RDrive motors (@48V):

- RD50: 267.84 ~= 260
- RD60: 267.84 ~= 260
- RD85: 244.80 ~= 240

1. Create an array of PVT points to define the desired motion trajectory.

```
int main(int argc, char *argv[])
{
    typedef struct
    {
        float pos;
        float vel;
        float time;
    } point_t;

    point_t points[] = {
        {343.77, 0, 0},
        {202.57, 1.33, 1537.4},
        {205.63, 9.20, 1148.9},
        {210.00, 21.24, 277.9},
        {215.61, 32.27, 209.5},
        {222.35, 42.94, 178.5},
        {230.14, 52.62, 161.8},
        {238.85, 60.75, 152.6},
        {248.41, 67.20, 148.5},
        {258.72, 72.29, 147.1},
        {269.68, 76.51, 147.1},
        {281.23, 79.88, 147.1},
        {293.29, 81.69, 148.4},
        {305.8, 81.01, 152.4},
        {318.72, 77.22, 161.6},
        {331.98, 69.83, 178.0},
        {345.56, 57.72, 208.5},
        {359.42, 35.09, 275.4},
        {373.52, 0.00, 709.8}};
```

2. Check the PVT points for velocities over the maximum limit.

```
    for(uint32_t i = 0; i < sizeof(points) / sizeof(points[0]) - 1; i++)
    {
        float calc_velocity;
        bool over_speed = rr_check_point(velocity_limit, &calc_velocity,
                                         points[i].pos, points[i].vel,
                                         points[i + 1].pos, points[i + 1].vel,
                                         points[i + 1].time);

        API_DEBUG("Point Angles (Velocities): %.3f (%.3f) -> %.3f (%.3f) [%d ms] # Limit Vel: %.3f
       | Max Vel: %.3f %s\n",
                  points[i].pos, points[i].vel,
                  points[i + 1].pos, points[i + 1].vel,
                  (int)points[i + 1].time,
                  velocity_limit, calc_velocity,
                  over_speed ? "is too high" : "");
    }
```

Finally, the function displays calculation results for each of the PVT points.

Complete tutorial code:

```
float velocity_limit = 80.0;

int main(int argc, char *argv[])
{
    typedef struct
    {
        float pos;
        float vel;
        float time;
    } point_t;

    point_t points[] = {
        {343.77, 0, 0},
        {202.57, 1.33, 1537.4},
        {205.63, 9.20, 1148.9},
        {210.00, 21.24, 277.9},
        {215.61, 32.27, 209.5},
        {222.35, 42.94, 178.5},
        {230.14, 52.62, 161.8},
        {238.85, 60.75, 152.6},
        {248.41, 67.20, 148.5},
        {258.72, 72.29, 147.1},
        {269.68, 76.51, 147.1},
        {281.23, 79.88, 147.1},
        {293.29, 81.69, 148.4},
        {305.8, 81.01, 152.4},
        {318.72, 77.22, 161.6},
        {331.98, 69.83, 178.0},
        {345.56, 57.72, 208.5},
        {359.42, 35.09, 275.4},
        {373.52, 0.00, 709.8}};

    for(uint32_t i = 0; i < sizeof(points) / sizeof(points[0]) - 1; i++)
    {
        float calc_velocity;
        bool over_speed = rr_check_point(velocity_limit, &calc_velocity,
                                         points[i].pos, points[i].vel,
                                         points[i + 1].pos, points[i + 1].vel,
                                         points[i + 1].time);

        API_DEBUG("Point Angles (Velocities): %.3f (%.3f) -> %.3f (%.3f) [%d ms] # Limit Vel: %.3f
       | Max Vel: %.3f %s\n",
                  points[i].pos, points[i].vel,
                  points[i + 1].pos, points[i + 1].vel,
                  (int)points[i + 1].time,
                  velocity_limit, calc_velocity,
                  over_speed ? "is too high" : "");
    }
}
```

5.26 Setting position with limits

The tutorial demonstrates how to use the rr_set_position_with_limits function.

1. Read the parameters as required to run the tutorial.

```
int main(int argc, char *argv[])
{
    uint8_t id;
    float ang, vel, acc;

    if(argc == 6)
    {
        id = strtol(argv[2], NULL, 0);
        ang = strtof(argv[3], NULL);
        vel = strtof(argv[4], NULL);
        acc = strtof(argv[5], NULL);
    }
    else
    {
        API_DEBUG("Wrong format!\nUsage: %s interface id angle max_vel max_acc\n", argv[0]);
        return 1;
    }
```

2. Initiate the interface.

```
    rr_can_interface_t *iface = rr_init_interface(argv[1]);
```

3. Initiate the servo.

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

4. Set the servo to the operational state.

```
    rr_servo_set_state_operational(servo);
```

5. Clear the existing motion queue.

```
    rr_clear_points_all(servo);
```

6. Run the rr_set_position_with_limits function, setting its parameters to the values read at Step 1.

```
    API_DEBUG("Appending points\n");
    uint32_t time_to_get_ms = 0;
    int status = rr_set_position_with_limits(servo, ang, vel, acc, &
      time_to_get_ms);
    if(status != RET_OK)
    {
        API_DEBUG("Error in point appending: %d\n", status);
        return 1;
    }
```

7. Set a variable to save data about how long the servo completes the preset motion.

```
    API_DEBUG("Time to get: %d ms\n", time_to_get_ms);
```

8. Set the rr_sleep_ms function to the time as requred to keep the program running until the servo completes the preset motion.

```
    rr_sleep_ms(time_to_get_ms + 100); // wait till the movement ends
```

Complete tutorial code:

```
int main(int argc, char *argv[])
{
    uint8_t id;
    float ang, vel, acc;

    if(argc == 6)
    {
        id = strtol(argv[2], NULL, 0);
        ang = strtof(argv[3], NULL);
        vel = strtof(argv[4], NULL);
        acc = strtof(argv[5], NULL);
    }
    else
    {
        API_DEBUG("Wrong format!\nUsage: %s interface id angle max_vel max_acc\n", argv[0]);
        return 1;
    }
    rr_can_interface_t *iface = rr_init_interface(argv[1]);
    rr_servo_t *servo = rr_init_servo(iface, id);

    rr_servo_set_state_operational(servo);

    rr_clear_points_all(servo);

    API_DEBUG("Appending points\n");
    uint32_t time_to_get_ms = 0;
    int status = rr_set_position_with_limits(servo, ang, vel, acc, &
      time_to_get_ms);
    if(status != RET_OK)
    {
        API_DEBUG("Error in point appending: %d\n", status);
        return 1;
    }

    API_DEBUG("Time to get: %d ms\n", time_to_get_ms);
    rr_sleep_ms(time_to_get_ms + 100); // wait till the movement ends
}
```

5.27 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(argv[1]);
```

2. Initialize the servo.

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

Reading the current error count

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

```
uint32_t _size;
```

4. 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

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

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

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

```
uint32_t size;
```

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

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

8. 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(argv[1]);

rr_servo_t *servo = rr_init_servo(iface, 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 emcy_log_entry_t Struct Reference

Emergency (EMCY) log entry structure.

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

Public Attributes

- uint8_t **id**
- uint16_t **err_code**
- uint8_t **err_reg**
- uint8_t **err_bits**
- int32_t **err_info**

6.1.1 Detailed Description

Emergency (EMCY) log entry structure.

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

- include/api.h

6.2 param_cache_entry_t Struct Reference

Device information source instance.

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

Public Attributes

- float value

    Source value.
- uint32_t timestamp

    Timestamp 0 - 599999999 us (value outside this range is invalid)
- bool activated

    Source activation flag.

6.2.1 Detailed Description

Device information source instance.

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

- include/api.h

6.3 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.
-

    struct {
      emcy_log_entry_t ∗ **d**
      int **head**
      int **tail**
      int **sz**
    } **emcy_log**

6.3.1 Detailed Description

Interface instance structure.

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

- include/api.h

6.4 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 source cells.

6.4.1 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 doc-src/doc.c File Reference

Hardware manual.

7.1.1 Detailed Description

Hardware manual.

Author

Rozum

7.2 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 emcy_log_entry_t

  Emergency (EMCY) log entry structure.
- struct rr_servo_t

  Device instance structure.
- struct rr_can_interface_t

  Interface instance structure.

Macros

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

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

    Make a string from the variable.
- #define ARRAY_ERROR_BITS_SIZE (64)

    Default size of the error bits array.
- #define EMCY_LOG_DEPTH 1024

    Size of the emergency (EMCY) log entry.
- #define MAX_CO_DEV 128

    Maximal number of CanOpen devoces on bus.
- #define RR_TIMESTAMP_INVALID 0xffffffffu

    Invalid timestamp value.

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, RET_WRONG_ARG }

    Return codes of the API functions.
- enum rr_servo_param_t {
  APP_PARAM_NULL = 0, APP_PARAM_POSITION, APP_PARAM_VELOCITY, APP_PARAM_POSITION_ROTOR,
  APP_PARAM_VELOCITY_ROTOR, APP_PARAM_POSITION_GEAR_360, APP_PARAM_POSITION_GEAR_EMULATED,
  APP_PARAM_CURRENT_INPUT,
  APP_PARAM_CURRENT_OUTPUT, APP_PARAM_VOLTAGE_INPUT, APP_PARAM_VOLTAGE_OUTPUT,
  APP_PARAM_CURRENT_PHASE,
  APP_PARAM_TEMPERATURE_ACTUATOR, APP_PARAM_TEMPERATURE_ELECTRONICS, APP_PARAM_TORQUE,
  APP_PARAM_ACCELERATION,
  APP_PARAM_ACCELERATION_ROTOR, APP_PARAM_CURRENT_PHASE_1, APP_PARAM_CURRENT_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,
  APP_PARAM_ENCODER_MOTOR_NONIUS_TRACK, APP_PARAM_TORQUE_ELECTRIC_CALC,
  APP_PARAM_CONTROLLER_VELOCITY_ERROR, APP_PARAM_CONTROLLER_VELOCITY_SETPOINT,
  APP_PARAM_CONTROLLER_VELOCITY_FEEDBACK, APP_PARAM_CONTROLLER_VELOCITY_OUTPUT,
  APP_PARAM_CONTROLLER_POSITION_ERROR, APP_PARAM_CONTROLLER_POSITION_SETPOINT,
  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_PARAM_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_BRAKE_TEMPERATURE = 75, APP_PARAM_SIZE = 80 }

  Device parameter and source indices.

- enum rr_nmt_state_t {
  RR_NMT_INITIALIZING = 0, RR_NMT_BOOT = 2, RR_NMT_PRE_OPERATIONAL = 127, RR_NMT_OPERATIONAL = 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.
- rr_ret_status_t **rr_write_raw_sdo** (const rr_servo_t ∗servo, uint16_t idx, uint8_t sidx, uint8_t ∗data, int sz, int retry, int tout)
- rr_ret_status_t **rr_read_raw_sdo** (const rr_servo_t ∗servo, uint16_t idx, uint8_t sidx, uint8_t ∗data, int ∗sz, int retry, int tout)
- 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.
- int rr_emcy_log_get_size (rr_can_interface_t ∗iface)

  The function returns the total count of entries in the EMCY logging buffer. Each entry in the buffer contains an EMCY event that have occurred up to the moment on the servo specified in the descriptor. ***Note:****When the API library is disabled, no new entries are made in the buffer, irrespective of whether or not any events occur on the servo.*
  The function is used in combination with the rr_emcy_log_pop and rr_emcy_log_clear functions. The typical sequence is as follows:
- emcy_log_entry_t ∗ rr_emcy_log_pop (rr_can_interface_t ∗iface)

The function enables reading entries from the EMCY logging buffer. Reading the entries is according to the "first in-first out" principle. Once an EMCY entry is read, the function removes it permenantly from the EMCY logging buffer.
Note:*Typically, the rr_emcy_log_pop function is used in combination with the rr_emcy_log_get_size and rr_emcy_log_clear functions. For the sequence of using the functions, see rr_emcy_log_get_size.*

- void rr_emcy_log_clear (rr_can_interface_t ∗iface)

  The function clears the EMCY logging buffer, removing the total of entries from it. It is advisable to use the clearing function in the beginning of a new work session and before applying the rr_emcy_log_get_size and rr_emcy_log_pop functions. For the typical sequence of using the functions, see rr_emcy_log_get_size.
- 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.
- rr_ret_status_t 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 specified 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.
- rr_ret_status_t rr_deinit_servo (rr_servo_t ∗∗servo)

  The function deinitializes the servo, clearing all data associated with the servo descriptor.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t rr_servo_get_state (const rr_servo_t ∗servo, rr_nmt_state_t ∗state)

  The function retrieves the actual NMT state of a specified servo. The state is described as a status code (:: rr_nmt←_state_t).
- rr_ret_status_t rr_servo_get_hb_stat (const rr_servo_t ∗servo, int64_t ∗min_hb_ival, int64_t ∗max_hb_ival)

  The function retrieves statistics on minimal and maximal intervals between Heartbeat messages of a servo. The statistics is saved to the variables specified in the param min_hb_ival and param max_hb_ival parameters, from where they are available for the user to perform further operations (e.g., comparison).
  The Heartbeat statistics is helpful in diagnozing and troubleshooting servo failures. For instance, when the Heartbeat interval of a servo is too long, it may mean that the control device sees the servo as being offline.
  Note:Before using the function, it is advisable to clear Heartbeat statistics with rr_servo_clear_hb_stat.
- rr_ret_status_t rr_servo_clear_hb_stat (const rr_servo_t ∗servo)

  The function clears statistics on minimal and maximal intervals between Heartbeat messages of a servo. It is advisable to use the function before attempting to get the Heartbeat statistics with the rr_servo_get_hb_stat function.
- rr_ret_status_t 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.

- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t rr_net_get_state (const rr_can_interface_t ∗interface, int id, rr_nmt_state_t ∗state)

  The function retrieves the actual NMT state of any device (a servo motor or any other) connected to the specified CAN network. The state is described as a status code (:: rr_nmt_state_t).
- rr_ret_status_t rr_release (const rr_servo_t ∗servo)

  The function sets the specified servo to the released state. The servo is de-energized and continues rotating for as long as it is affected by external forces (e.g., inertia, gravity).
- rr_ret_status_t rr_freeze (const rr_servo_t ∗servo)

  The function sets the specified servo to the freeze state. The servo stops, retaining its last position.
- rr_ret_status_t rr_brake_engage (const rr_servo_t ∗servo, const bool en)

  The function applies or releases the servo's built-in brake. If a servo is supplied without a brake, the function will not work. In this case, to stop a servo, use either the rr_freeze() or rr_release() function.
- rr_ret_status_t 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).
- rr_ret_status_t rr_set_velocity (const rr_servo_t ∗servo, const float velocity_deg_per_sec)

  The function sets the output shaft velocity with 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.
- rr_ret_status_t rr_set_velocity_motor (const rr_servo_t ∗servo, const float velocity_rpm)

  The function sets the velocity with which the motor of the specified servo should move at its maximum current. The maximum current is in accordance with the servo motor specification.
- rr_ret_status_t 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.
- rr_ret_status_t 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). The velocity value is the velocity of the servo's output shaft.
- rr_ret_status_t rr_set_position_with_limits (rr_servo_t ∗servo, const float position_deg, const float velocity↩_deg_per_sec, const float accel_deg_per_sec_sq, uint32_t ∗time_ms)

  The function sets the position that the servo should reach with velocity and acceleration limits on generated trajectory.
- rr_ret_status_t 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.
- rr_ret_status_t 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:
- rr_ret_status_t rr_add_motion_point_pvat (const rr_servo_t ∗servo, const float position_deg, const float velocity_deg_per_sec, const float accel_deg_per_sec2, const uint32_t time_ms)

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

- rr_ret_status_t 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).

- rr_ret_status_t 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 actual count of servo hardware errors (e.g., no Heartbeats/overcurrent, etc.). In addition, the function returns the codes of all the detected errors as a single array.
- rr_ret_status_t 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 that was 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.
- rr_ret_status_t rr_param_cache_update_with_timestamp (rr_servo_t ∗servo)

  Same as rr_param_cache_update but with timestamp functionality.
- rr_ret_status_t 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_ret_status_t 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.
- rr_ret_status_t rr_read_parameter_with_timestamp (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value, uint32_t ∗timestamp)

  Same rr_read_parameter but with timestamp functionality.
- rr_ret_status_t 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.
- rr_ret_status_t rr_read_cached_parameter_with_timestamp (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value, uint32_t ∗timestamp)

  Same as rr_read_cached_parameter but with timestamp functionality.
- rr_ret_status_t 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. If you call the function while the servo is executing a motion point command, the servo stops without completing the motion. All the remaining motion points, including the one where the servo has been moving, are removed from the queue.
- rr_ret_status_t 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. ***Note:*** *In case the indicated number of PVT points to be removed exceeds the actual remaining number of PVT points in the queue, the effect of applying the function is similar to that of applying rr_clear_points_all.*
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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, uint32_t ∗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). ***Note:****The function is executed without the servo moving.*

- rr_ret_status_t 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.
- rr_ret_status_t 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.

- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t rr_change_id_and_save (rr_can_interface_t ∗interface, rr_servo_t ∗∗servo, uint8_t new_↩
can_id)

  The function enables changing the default CAN identifier (ID) of the specified servo to avoid collisions on a bus line. ***Important!*** *Each servo connected to a CAN bus must have* ***a unique ID****.*

- rr_ret_status_t rr_clear_errors (const rr_servo_t ∗servo)

  Clears the error bits in the servo.
- rr_ret_status_t rr_get_hardware_version (const rr_servo_t ∗servo, char ∗version_string, int ∗version_string↩
_size)

  The function reads the hardware version of a servo (unique ID of the MCU + hardware type + hardware revision).
- rr_ret_status_t rr_get_software_version (const rr_servo_t ∗servo, char ∗version_string, int ∗version_string↩
_size)

  The function reads the software version of a servo (minor + major + firmware build date).
- bool rr_check_point (const float velocity_limit_deg_per_sec, float ∗velocity_max_calc_deg_per_sec, const float position_deg_start, const float velocity_deg_per_sec_start, const float position_deg_end, const float velocity_deg_per_sec_end, const uint32_t time_ms)

  The function enables verifying validity (reachability) of a trajectory point without actually initializing an interface or a servo. To verify, the maximum velocity limit is compared against the calculation output of the function.

7.2.1 Detailed Description

Rozum Robotics API Header File.

Author

Rozum

Date

2018-06-01

7.2.2 Typedef Documentation

7.2.2.1 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)

7.2.2.2 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

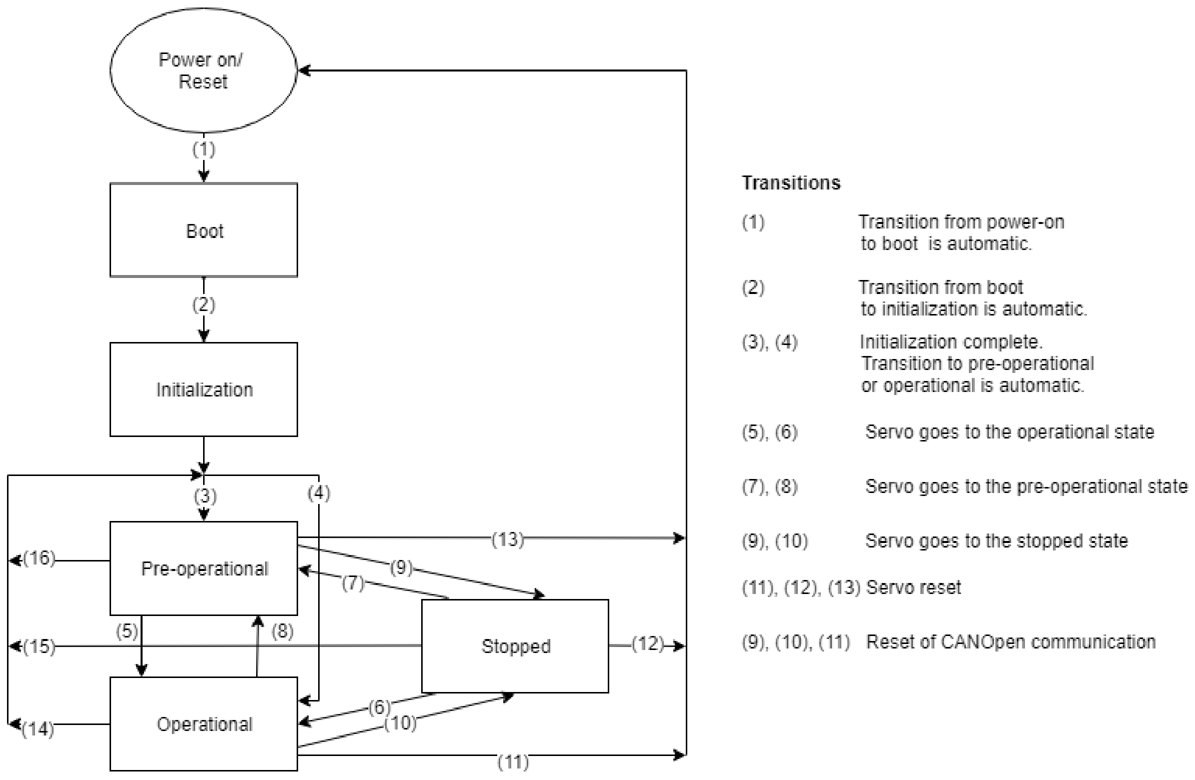
interface	Descriptor of the interface (see rr_init_interface) where the NMT event occured
servo_id	Descriptor of the servo (see rr_init_servo) where the NMT event occured
nmt_state	Network management state (rr_nmt_state_t) that the servo entered

7.2.3 Enumeration Type Documentation

7.2.3.1 rr_nmt_state_t

```
enum rr_nmt_state_t
```

Network management (NMT) states.



Transitions

(1)	Transition from power-on to boot is automatic.
(2)	Transition from boot to initialization is automatic.
(3), (4)	Initialization complete. Transition to pre-operational or operational is automatic.
(5), (6)	Servo goes to the operational state
(7), (8)	Servo goes to the pre-operational state
(9), (10)	Servo goes to the stopped state
(11), (12), (13)	Servo reset
(9), (10), (11)	Reset of CANOpen communication

Enumerator

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

7.2.3.2 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 the STOPPED state.
RET_TIMEOUT	CAN communication timeout.

Enumerator

RET_ZERO_SIZE	Zero data size.
RET_SIZE_MISMATCH	Mismatch of received and target data size.
RET_WRONG_ARG	Wrong function argument.

7.2.3.3 rr_servo_param_t

```
enum rr_servo_param_t
```

Device parameter and source indices.

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) in the servo's supply circuit.
APP_PARAM_CURRENT_OUTPUT	Not used.
APP_PARAM_VOLTAGE_INPUT	Actual DC voltage (Volts) supplied to the servo.
APP_PARAM_VOLTAGE_OUTPUT	Not used.
APP_PARAM_CURRENT_PHASE	Actual magnitude of AC current (Amperes) on the motor.
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 to the motor.
APP_PARAM_CURRENT_PHASE_2	Actual phase 2 current to the motor.
APP_PARAM_CURRENT_PHASE_3	Actual phase 3 current to the motor.
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.

Enumerator

APP_PARAM_TORQUE_ELECTRIC_CALC	Internal use only.
APP_PARAM_CONTROLLER_VELOCITY_ERROR	Velocity following error (difference in degrees per second between the setpoint and feedback velocities)
APP_PARAM_CONTROLLER_VELOCITY_SETP←↩ OINT	Velocity target (degrees per second)
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 (difference in degrees per second between the setpoint and feedback velocities)
APP_PARAM_CONTROLLER_POSITION_SETP←↩ OINT	Position target (degrees)
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.
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_BRAKE_TEMPERATURE	Brake coil temperature.
APP_PARAM_SIZE	Use to define the total parameter array size.

7.2.4 Function Documentation

7.2.4.1 rr_check_point()

```
bool rr_check_point (
            const float velocity_limit_deg_per_sec,
            float * velocity_max_calc_deg_per_sec,
            const float position_deg_start,
            const float velocity_deg_per_sec_start,
            const float position_deg_end,
            const float velocity_deg_per_sec_end,
            const uint32_t time_ms )
```

The function enables verifying validity (reachability) of a trajectory point without actually initializing an interface or a servo. To verify, the maximum velocity limit is compared against the calculation output of the function.

Parameters

velocity_limit_deg_per_sec	Velocity limit (in degrees/sec)
velocity_max_calc_deg_per_sec	Pointer to the maximum velocity calculated for the point (in degrees/sec)
position_deg_start	Start position preset for the point (in degrees)
velocity_deg_per_sec_start	Start velocity preset for the point (in degrees/sec)
position_deg_end	End position preset for the point (in degrees)
velocity_deg_per_sec_end	End velocity preset for the point (in degrees/sec)
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

true If the maximum calculated velocity at the point is greater than the velocity limit
false If the maximum calculated velocity at the point is equal or lower than the velocity limit

7.2.4.2 rr_clear_errors()

```
rr_ret_status_t rr_clear_errors (
            const rr_servo_t * servo )
```

Clears the error bits in the servo.

Parameters

servo	Servo descriptor returned by the rr_init_servo function.

Returns

Status code (rr_ret_status_t)

7.2.4.3 rr_get_hardware_version()

```
rr_ret_status_t rr_get_hardware_version (
            const rr_servo_t * servo,
            char * version_string,
            int * version_string_size )
```

The function reads the hardware version of a servo (unique ID of the MCU + hardware type + hardware revision).

Parameters

servo	Servo descriptor returned by the rr_init_servo function.
version_string	Pointer to the ASCII string to read
version_string_size	Input: size of the ::version_string, Output: size of the read string

Returns

Status code (rr_ret_status_t)

7.2.4.4 rr_get_software_version()

```
rr_ret_status_t rr_get_software_version (
            const rr_servo_t * servo,
            char * version_string,
            int * version_string_size )
```

The function reads the software version of a servo (minor + major + firmware build date).

Parameters

servo	Servo descriptor returned by the rr_init_servo function.
version_string	Pointer to the ASCII string to read
version_string_size	Input: size of the ::version_string, Output: size of the read string

Returns

Status code (rr_ret_status_t)

7.3 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"
```

Macros

- #define **PARAM_STORE_PASSWORD** 0x73617665

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 ∗iface, 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 ∗iface, 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 ∗iface, 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.
- int rr_emcy_log_get_size (rr_can_interface_t ∗iface)

  The function returns the total count of entries in the EMCY logging buffer. Each entry in the buffer contains an EMCY event that have occurred up to the moment on the servo specified in the descriptor. ***Note:****When the API library is disabled, no new entries are made in the buffer, irrespective of whether or not any events occur on the servo.*
  The function is used in combination with the rr_emcy_log_pop and rr_emcy_log_clear functions. The typical sequence is as follows:
- void **rr_emcy_log_push** (rr_can_interface_t ∗iface, uint8_t id, uint16_t err_code, uint8_t err_reg, uint8_t err_bits, int32_t err_info)
- emcy_log_entry_t ∗ rr_emcy_log_pop (rr_can_interface_t ∗iface)

  The function enables reading entries from the EMCY logging buffer. Reading the entries is according to the "first in-first out" principle. Once an EMCY entry is read, the function removes it permenantly from the EMCY logging buffer.
  Note:*Typically, the rr_emcy_log_pop function is used in combination with the rr_emcy_log_get_size and rr_emcy_log_clear functions. For the sequence of using the functions, see rr_emcy_log_get_size.*
- void rr_emcy_log_clear (rr_can_interface_t ∗iface)

  The function clears the EMCY logging buffer, removing the total of entries from it. It is advisable to use the clearing function in the beginning of a new work session and before applying the rr_emcy_log_get_size and rr_emcy_log_pop functions. For the typical sequence of using the functions, see rr_emcy_log_get_size.
- 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.
- rr_ret_status_t rr_deinit_interface (rr_can_interface_t ∗∗iface)

  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 ∗iface, const uint8_t id)

  The function determines whether the servo motor with the specified 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.
- rr_ret_status_t rr_deinit_servo (rr_servo_t ∗∗servo)

  The function deinitializes the servo, clearing all data associated with the servo descriptor.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t rr_servo_get_state (const rr_servo_t ∗servo, rr_nmt_state_t ∗state)

  The function retrieves the actual NMT state of a specified servo. The state is described as a status code (:: rr_nmt←_state_t).
- rr_ret_status_t rr_servo_get_hb_stat (const rr_servo_t ∗servo, int64_t ∗min_hb_ival, int64_t ∗max_hb_ival)

  The function retrieves statistics on minimal and maximal intervals between Heartbeat messages of a servo. The statistics is saved to the variables specified in the param min_hb_ival and param max_hb_ival parameters, from where they are available for the user to perform further operations (e.g., comparison).
  The Heartbeat statistics is helpful in diagnozing and troubleshooting servo failures. For instance, when the Heartbeat interval of a servo is too long, it may mean that the control device sees the servo as being offline.
  Note:Before using the function, it is advisable to clear Heartbeat statistics with rr_servo_clear_hb_stat.
- rr_ret_status_t rr_servo_clear_hb_stat (const rr_servo_t ∗servo)

  The function clears statistics on minimal and maximal intervals between Heartbeat messages of a servo. It is advisable to use the function before attempting to get the Heartbeat statistics with the rr_servo_get_hb_stat function.
- rr_ret_status_t rr_net_reboot (const rr_can_interface_t ∗iface)

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

  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.
- rr_ret_status_t rr_net_set_state_operational (const rr_can_interface_t ∗iface)

  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.
- rr_ret_status_t rr_net_set_state_pre_operational (const rr_can_interface_t ∗iface)

  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.

- rr_ret_status_t rr_net_set_state_stopped (const rr_can_interface_t ∗iface)

  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.
- rr_ret_status_t rr_net_get_state (const rr_can_interface_t ∗iface, int id, rr_nmt_state_t ∗state)

  The function retrieves the actual NMT state of any device (a servo motor or any other) connected to the specified CAN network. The state is described as a status code (:: rr_nmt_state_t).
- rr_ret_status_t rr_release (const rr_servo_t ∗servo)

  The function sets the specified servo to the released state. The servo is de-energized and continues rotating for as long as it is affected by external forces (e.g., inertia, gravity).
- rr_ret_status_t rr_freeze (const rr_servo_t ∗servo)

  The function sets the specified servo to the freeze state. The servo stops, retaining its last position.
- rr_ret_status_t 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).
- rr_ret_status_t rr_brake_engage (const rr_servo_t ∗servo, const bool en)

  The function applies or releases the servo's built-in brake. If a servo is supplied without a brake, the function will not work. In this case, to stop a servo, use either the rr_freeze() or rr_release() function.
- rr_ret_status_t rr_set_velocity (const rr_servo_t ∗servo, const float velocity_deg_per_sec)

  The function sets the output shaft velocity with 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.
- rr_ret_status_t rr_set_velocity_motor (const rr_servo_t ∗servo, const float velocity_rpm)

  The function sets the velocity with which the motor of the specified servo should move at its maximum current. The maximum current is in accordance with the servo motor specification.
- rr_ret_status_t 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.
- rr_ret_status_t 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). The velocity value is the velocity of the servo’s output shaft.
- rr_ret_status_t rr_set_position_with_limits (rr_servo_t ∗servo, const float position_deg, const float velocity↩
_deg_per_sec, const float accel_deg_per_sec_sq, uint32_t ∗time_ms)

  The function sets the position that the servo should reach with velocity and acceleration limits on generated trajectory.
- rr_ret_status_t 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.
- rr_ret_status_t 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:
- rr_ret_status_t rr_add_motion_point_pvat (const rr_servo_t ∗servo, const float position_deg, const float velocity_deg_per_sec, const float accel_deg_per_sec2, const uint32_t time_ms)

  The function enables creating PVAT (position-velocity-acceleration-time) points to set the motion trajectory of the servo specified in the 'servo' parameter. PVAT points define the following:
- rr_ret_status_t rr_start_motion (rr_can_interface_t ∗iface, 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).
- rr_ret_status_t 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 actual count of servo hardware errors (e.g., no Heartbeats/overcurrent, etc.). In addition, the function returns the codes of all the detected errors as a single array.

- rr_ret_status_t 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 that was 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.

- rr_ret_status_t rr_param_cache_update_with_timestamp (rr_servo_t ∗servo)

  Same as rr_param_cache_update but with timestamp functionality.

- rr_ret_status_t 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_ret_status_t 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.

- rr_ret_status_t rr_read_parameter_with_timestamp (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value, uint32_t ∗timestamp)

  Same rr_read_parameter but with timestamp functionality.

- rr_ret_status_t 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.

- rr_ret_status_t rr_read_cached_parameter_with_timestamp (rr_servo_t ∗servo, const rr_servo_param_t param, float ∗value, uint32_t ∗timestamp)

  Same as rr_read_cached_parameter but with timestamp functionality.

- rr_ret_status_t 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. If you call the function while the servo is executing a motion point command, the servo stops without completing the motion. All the remaining motion points, including the one where the servo has been moving, are removed from the queue.

- rr_ret_status_t 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. **Note:** In case the indicated number of PVT points to be removed exceeds the actual remaining number of PVT points in the queue, the effect of applying the function is similar to that of applying rr_clear_points_all.*

- rr_ret_status_t 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.

- rr_ret_status_t 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.

- rr_ret_status_t 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, uint32_t ∗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). **Note:**The function is executed without the servo moving.*

- rr_ret_status_t 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.

- rr_ret_status_t 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.

- rr_ret_status_t 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.
- rr_ret_status_t 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.
- rr_ret_status_t rr_change_id_and_save (rr_can_interface_t ∗iface, rr_servo_t ∗∗servo, uint8_t new_can_id)

  *The function enables changing the default CAN identifier (ID) of the specified servo to avoid collisions on a bus line. **Important!** Each servo connected to a CAN bus must have **a unique ID**.*
- rr_ret_status_t rr_clear_errors (const rr_servo_t ∗servo)

  Clears the error bits in the servo.
- rr_ret_status_t rr_get_hardware_version (const rr_servo_t ∗servo, char ∗version_string, int ∗version_string↩ _size)

  The function reads the hardware version of a servo (unique ID of the MCU + hardware type + hardware revision).
- rr_ret_status_t rr_get_software_version (const rr_servo_t ∗servo, char ∗version_string, int ∗version_string↩ _size)

  The function reads the software version of a servo (minor + major + firmware build date).
- bool rr_check_point (const float velocity_limit_deg_per_sec, float ∗velocity_max_calc_deg_per_sec, const float position_deg_start, const float velocity_deg_per_sec_start, const float position_deg_end, const float velocity_deg_per_sec_end, const uint32_t time_ms)

  The function enables verifying validity (reachability) of a trajectory point without actually initializing an interface or a servo. To verify, the maximum velocity limit is compared against the calculation output of the function.

7.3.1 Detailed Description

Rozum Robotics API Source File.

Author

Rozum

Date

2018-06-01

7.3.2 Function Documentation

7.3.2.1 rr_check_point()

```
bool rr_check_point (
            const float velocity_limit_deg_per_sec,
            float * velocity_max_calc_deg_per_sec,
            const float position_deg_start,
            const float velocity_deg_per_sec_start,
            const float position_deg_end,
            const float velocity_deg_per_sec_end,
            const uint32_t time_ms )
```

The function enables verifying validity (reachability) of a trajectory point without actually initializing an interface or a servo. To verify, the maximum velocity limit is compared against the calculation output of the function.

Parameters

velocity_limit_deg_per_sec	Velocity limit (in degrees/sec)
velocity_max_calc_deg_per_sec	Pointer to the maximum velocity calculated for the point (in degrees/sec)
position_deg_start	Start position preset for the point (in degrees)
velocity_deg_per_sec_start	Start velocity preset for the point (in degrees/sec)
position_deg_end	End position preset for the point (in degrees)
velocity_deg_per_sec_end	End velocity preset for the point (in degrees/sec)
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

true If the maximum calculated velocity at the point is greater than the velocity limit
false If the maximum calculated velocity at the point is equal or lower than the velocity limit

7.3.2.2 rr_clear_errors()

```
rr_ret_status_t rr_clear_errors (
            const rr_servo_t * servo )
```

Clears the error bits in the servo.

Parameters

servo	Servo descriptor returned by the rr_init_servo function.

Returns

Status code (rr_ret_status_t)

7.3.2.3 rr_get_hardware_version()

```
rr_ret_status_t rr_get_hardware_version (
            const rr_servo_t * servo,
            char * version_string,
            int * version_string_size )
```

The function reads the hardware version of a servo (unique ID of the MCU + hardware type + hardware revision).

Parameters

servo	Servo descriptor returned by the rr_init_servo function.
version_string	Pointer to the ASCII string to read
version_string_size	Input: size of the ::version_string, Output: size of the read string

Returns

Status code (rr_ret_status_t)

7.3.2.4 rr_get_software_version()

```
rr_ret_status_t rr_get_software_version (
             const rr_servo_t * servo,
             char * version_string,
             int * version_string_size )
```

The function reads the software version of a servo (minor + major + firmware build date).

Parameters

servo	Servo descriptor returned by the rr_init_servo function.
version_string	Pointer to the ASCII string to read
version_string_size	Input: size of the ::version_string, Output: size of the read string

Returns

Status code (rr_ret_status_t)

7.4 tutorial/calibrate_cogging.c File Reference

Calibrating to mitigate cogging effects.

```
#include "api.h"
#include <stdlib.h>
#include <math.h>
```

Functions

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

7.4.1 Detailed Description

Calibrating to mitigate cogging effects.

Author

Rozum

Date

2018-07-11

7.4.2 Function Documentation

7.4.2.1 main()

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

[calibrate_cogging_code_full] [Read tutorial param1] [Read tutorial param1]

[Init interface31]

[Init interface31]

[Init servo31]

[Init servo31]

[Switching to operational state1]

[Switching to operational state1]

[Start calibration]

[Start calibration]

[Read vel setpoint]

[Read vel setpoint]

[Enable cog table]

[Enable cog table]

[Save to flash]

[Save to flash] [calibrate_cogging_code_full]

7.5 tutorial/calibration_quality.c File Reference

Calibration quality.

```
#include "api.h"
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <inttypes.h>
#include <stdio.h>
```

Macros

- #define **USEC_PER_SEC** 1000000

Functions

- void enable_compensation (rr_servo_t ∗servo, bool en)

  [calibration_quality_code_full]
- int64_t **calcdiff** (struct timespec t1, struct timespec t2)
- int main (int argc, char ∗argv[ ])

  [Read tutorial param2]

7.5.1 Detailed Description

Calibration quality.

Author

Rozum

Date

2018-07-11

7.5.2 Function Documentation

7.5.2.1 main()

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

[Read tutorial param2]

[Read tutorial param2]

[Init interface32]

[Init interface32] [Init servo32]

[Init servo32] [Switching to operational]

[Switching to operational] [Set up param entry]

[Set up param entry] [Disable cog table]

[Disable cog table] [Set velocity1]

[Set velocity1] [Open file1]

[Open file1]

[Record time1]

[Record time1]

[Measure param1]

[Measure param1]

[Close file1]

[Close file1]

[Enable cog table]

[Enable cog table]

[Set velocity2]

[Set velocity2]

[Open file2]

[Open file2]

[Record time2]

[Record time2]

[Measure param2]

[Measure param2]

[Close file2]

[Close file2]

[Release]

[Release]

7.6 tutorial/change_servo_id.c File Reference

Changing CAN ID of a single servo.

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

Functions

- int main (int argc, char ∗argv[ ])

7.6.1 Detailed Description

Changing CAN ID of a single servo.

Author

Rozum

Date

2018-07-11

7.6.2 Function Documentation

7.6.2.1 main()

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

[change_id_code_full] [Create 2 variables]

[Create 2 variables]

[Check arguments]

[Check arguments] [Add interface10]

[Add interface10] [Add servo10]

[Add servo10]

[Change ID]

[Change ID] [change_id_code_full]

7.7 tutorial/check_motion_points.c File Reference

Checking PVT points.

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

Functions

- int main (int argc, char ∗argv[ ])

  [Create PVT]

Variables

- float velocity_limit = 80.0

  [check_motion_points_code_full]

7.7.1 Detailed Description

Checking PVT points.

Author

Rozum

Date

2018-12-05

7.7.2 Function Documentation

7.7.2.1 main()

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

[Create PVT]

[Create PVT]

[Check velocity at PVT]

[Check velocity at PVT]

7.8 tutorial/control_servo_traj_1.c File Reference

Setting PVT points for one servo.

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

Functions

- int main (int argc, char ∗argv[ ])

7.8.1 Detailed Description

Setting PVT points for one servo.

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 interface1]

[Adding the interface1]

[Adding the servo1]

[Adding the servo1]

[Switching to operational state]

[Switching to operational state]

[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]

7.9 tutorial/control_servo_traj_2.c File Reference

Setting PVT points for two servos.

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

Functions

- int main (int argc, char ∗argv[ ])

7.9.1 Detailed Description

Setting PVT points for two servos.

Author

Rozum

Date

2018-06-25

7.9.2 Function Documentation

7.9.2.1 main()

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

[cccode 2] [Adding the interface2]

[Adding the interface2] [Adding servo ID0]

[Adding servo ID0] [Adding servo ID1]

[Adding servo ID1]

[Switching to operational state]

[Switching to operational state]

[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 2]

7.10 tutorial/control_servo_traj_3.c File Reference

Setting points for three servos.

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

Functions

- int main (int argc, char ∗argv[ ])

7.10.1 Detailed Description

Setting points for three servos.

Author

Rozum

Date

2018-06-25

7.10.2 Function Documentation

7.10.2.1 main()

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

[cccode 3] [Adding the interface3]

[Adding the interface3] [Adding servo one]

[Adding servo one] [Adding servo two]

[Adding servo two] [Adding servo three]

[Adding servo three]

[Switching to operational state]

[Switching to operational state]

[Clear points servo one]

[Clear points servo one] [Clear points servo two]

[Clear points servo two] [Clear points servo three]

[Clear points servo three] [Add point one servo one]

[Add point one servo one] [Add point one servo two]

[Add point one servo two] [Add point one servo three]

[Add point one servo three] [Add point two servo one]

[Add point two servo one] [Add point two servo two]

[Add point two servo two] [Add point two servo three]

[Add point two servo three] [Start motion3]

[Start motion3]

[Sleep3]

[Sleep3] [cccode 3]

7.11 tutorial/discovery.c File Reference

Detecting available CAN devices and their states.

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

Functions

- void sdo_read_str (rr_servo_t ∗servo, uint16_t idx, uint8_t sidx, char ∗buf, int max_sz)

  [discovery_code_full]
- void get_dev_info (rr_can_interface_t ∗iface, int id)

  [auxiliary to display]
- int main (int argc, char ∗argv[ ])

  [auxiliary to display]

7.11.1 Detailed Description

Detecting available CAN devices and their states.

Author

Rozum

Date

2018-07-11

7.11.2 Function Documentation

7.11.2.1 main()

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

[auxiliary to display]

[Create variables] [Create variables]

[Init interface33]

[Init interface33]

[Enable hb variable]

[Enable hb variable]

[Scan]

[Scan]

7.12 tutorial/read_any_param.c File Reference

Tutorial example of reading device parameters.

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

Functions

- int main (int argc, char ∗argv[ ])

7.12.1 Detailed Description

Tutorial example of reading device parameters.

Author

your name

Date

2018-06-25

7.12.2 Function Documentation

7.12.2.1 main()

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

[cccode 4] [Adding interface 4]

[Adding interface 4] [Adding servo 4]

[Adding servo 4]

[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 4]

7.13 tutorial/read_any_param_cache.c File Reference

Tutorial example of reading servo parameters from the cache.

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

Functions

- int main (int argc, char ∗argv[ ])

7.13.1 Detailed Description

Tutorial example of reading servo parameters from the cache.

Author

Rozum

Date

2018-06-25

7.13.2 Function Documentation

7.13.2.1 main()

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

[cccode 5] [Adding interface 5]

[Adding interface 5] [Adding servo 5]

[Adding servo 5]

[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 5]

7.14 tutorial/read_emcy_log.c File Reference

Reading emergency (EMY) log buffer.

```
#include "api.h"
#include <stdlib.h>
#include <inttypes.h>
```

Functions

- int main (int argc, char ∗argv[ ])
  [read_emcy_log_code_full]

7.14.1 Detailed Description

Reading emergency (EMY) log buffer.

Author

Rozum

Date

2018-06-25

7.14.2 Function Documentation

7.14.2.1 main()

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

[read_emcy_log_code_full]

[Read tutorial param3] [Read tutorial param3]

[Init interface34]

[Init interface34]

[Init servo34]

[Init servo34]

[read emcy]

[read emcy]
[read_emcy_log_code_full]

7.15 tutorial/read_errors.c File Reference

Tutorial example of reading device errors.

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

Functions

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

7.15.1 Detailed Description

Tutorial example of reading device errors.

Author

Rozum

Date

2018-06-25

7.15.2 Function Documentation

7.15.2.1 main()

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

[cccode 6] [Adding interface 6]

[Adding interface 6]

[Adding servo 6]

[Adding servo 6]

[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 6]

7.16 tutorial/read_servo_max_velocity.c File Reference

Tutorial example of reading the maximum servo.

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

Functions

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

7.16.1 Detailed Description

Tutorial example of reading the maximum servo.

Author

Rozum

Date

2018-06-25

7.16.2 Function Documentation

7.16.2.1 main()

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

[cccode 7] [Adding interface 7]

[Adding interface 7] [Adding servo 7]

[Adding servo 7]

[Velocity variable]

[Velocity variable] [Read max velocity]

[Read max velocity]

[cccode 7]

7.17 tutorial/read_servo_motion_queue.c File Reference

Tutorial example of reading motion queue parameters.

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

Functions

- int main (int argc, char ∗argv[ ])

7.17.1 Detailed Description

Tutorial example of reading motion queue parameters.

Author

Rozum

Date

2018-06-25

7.17.2 Function Documentation

7.17.2.1 main()

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

[cccode 8] [Adding interface 8]

[Adding interface 8] [Adding servo 8]

[Adding servo 8]

[Clear points 8]

[Clear points 8]

[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 8]

7.18 tutorial/read_servo_trajectory_time.c File Reference

Tutorial example of calculating a PVT point.

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

Functions

- int main (int argc, char ∗argv[ ])

7.18.1 Detailed Description

Tutorial example of calculating a PVT point.

Author

Rozum

Date

2018-06-25

7.18.2 Function Documentation

7.18.2.1 main()

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

[cccode 9] [Adding interface 9]

[Adding interface 9] [Adding servo 9]

[Adding servo 9]

[Travel time variable]

[Travel time variable]

[Time calculation]

[Time calculation]

[cccode 9]

7.19 tutorial/time_optimal_movement.c File Reference

Setting positions with limits.

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

Functions

- int main (int argc, char ∗argv[ ])
  [Read param tutorial2]

7.19.1 Detailed Description

Setting positions with limits.

Author

Rozum

Date

2018-12-05

7.19.2 Function Documentation

7.19.2.1 main()

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

[Read param tutorial2]

[time_optimal_movement_code_full] [Read param tutorial2] [Init interface35]

[Init interface35] [Init servo35]

[Init servo35]

[Switching to operational2]

[Switching to operational2]

[Clear points all]

[Clear points all]

[Set position with limits]

[Set position with limits]

[Set variable to save time]

[Set variable to save time] [Set sleep time]
[Set sleep time]

Index