API Reference

The ur_rtde library API consists of the following parts:

- rtde_control_interface.h: The RTDE control interface
- rtde_receive_interface.h: The RTDE receive interface
- rtde_io_interface.h: The RTDE IO interface
- rtde.h: The RTDE class
- script_client.h: Script client
- dashboard_client.h: Dashboard client
- robotiq_gripper.h: Robotiq gripper interface

RTDE Control Interface API

class **RTDEControlInterface**

This class provides the interface to control the robot and to execute robot movements.

Note

Currently the RTDEControlInterface, should not be considered thread safe, since no measures (mutexes) are taken to ensure that a function is done, before another would be executed. It is up to the caller to provide protection using mutexes.

Public Types

enum **Flags**

Configuration flags for the RTDEControlInterface.

These flags modify the behavior of the RTDEControlInterface when passed to the constructor. Multiple flags can be combined using bitwise OR ('|') operations.

Values:

enumerator **FLAG_UPLOAD_SCRIPT**

enumerator **FLAG_USE_EXT_UR_CAP**

enumerator **FLAG_VERBOSE**

enumerator **FLAG_UPPER_RANGE_REGISTERS**

enumerator **FLAG_NO_WAIT**

enumerator **FLAG_CUSTOM_SCRIPT**

enumerator **FLAG_NO_EXT_FT**

enumerator **FLAG_DISABLE_REMOTE_CONTROL_CHECK**

enumerator **FLAGS_DEFAULT**

Public Functions

explicit **RTDEControlInterface**(std::string hostname, double frequency = -1.0, uint16_t flags = FLAGS_DEFAULT, int ur_cap_port = 50002, int rt_priority = RT_PRIORITY_UNDEFINED)

Constructor for the RTDEControlInterface class.

Creates an interface to control and execute robot movements on a Universal Robot.

See also

Flags for detailed information about available configuration flags

Note

The RTDEControlInterface is not thread-safe. The caller must provide protection using mutexes if needed.

Parameters:

- **hostname** – The IP address or hostname of the robot.
- **frequency** – The frequency at which RTDE data will be exchanged with the robot (-1.0 means use the robot's default frequency, 500Hz for e-Series and UR-Series, while its 125Hz for the CB-series).
- **flags** – Configuration flags that modify the behavior of the interface (see Flags).
- **ur_cap_port** – The port used for the External URCap interface (default: 50002)
- **rt_priority** – Real-time priority of the RTDEControlInterface thread (if supported by the OS).

void **disconnect**()

Returns: Can be used to disconnect from the robot. To reconnect you have to call the reconnect() function.

bool **reconnect**()

Returns: Can be used to reconnect to the robot after a lost connection.

bool **isConnected**()

Returns: Connection status for RTDE, useful for checking for lost connection.

bool **waitForNextState**()

Returns: When the next state has been received

void **waitPeriod**(*const* std::chrono::steady_clock::time_point &t_cycle_start)

Used for waiting the rest of the control period, set implicitly as dt = 1 / frequency.

A combination of sleeping and spinning are used to achieve the lowest possible jitter. The function is especially useful for a realtime control loop. NOTE: the function is to be used in combination with the initPeriod(). See the realtime_control_example.cpp.

Parameters: **t_cycle_start** – the start of the control period. Typically given as dt = 1 / frequency.

std::chrono::steady_clock::time_point **initPeriod**()

This function is used in combination with waitPeriod() and is used to get the start of a control period / cycle.

See the realtime_control_example.cpp.

bool **reuploadScript**()

In the event of an error, this function can be used to resume operation by reuploading the RTDE control script.

This will only happen if a script is not already running on the controller.

bool **sendCustomScriptFunction**(*const* std::string &function_name, *const* std::string &script)

Send a custom ur script to the controller.

Parameters:

- **function_name** – specify a name for the custom script function
- **script** – the custom ur script to be sent to the controller specified as a string, each line must be terminated with a newline. The code will automatically be indented with one tab to fit with the function body.

bool **sendCustomScript**(*const* std::string &script)

Send a custom ur script to the controller The function enables sending of short scripts which was defined inline within source code.

So you can write code like this:

```
const std::string inline_script =
           "def script_test():\n"
                   "\tdef test():\n"
                           "textmsg(\"test1\")\n"
                           "textmsg(\"test2\")\n"
                   "\tend\n"
                   "\twrite_output_integer_register(0, 1)\n"
                   "\ttest()\n"
                   "\ttest()\n"
                   "\twrite_output_integer_register(0, 2)\n"
           "end\n"
           "run program\n";
     bool result = rtde_c.sendCustomScript(inline_script);
```

Returns: Returns true if the script has been executed successfully and false on timeout

bool **sendCustomScriptFile**(*const* std::string &file_path)

Send a custom ur script file to the controller.

Parameters: **file_path** – the file path to the custom ur script file

void **setCustomScriptFile**(*const* std::string &file_path)

Assign a custom script file that will be sent to device as the main control script.

Setting an empty file_name will disable the custom script loading This eases debugging when modifying the control script because it does not require to recompile the whole library

void **stopScript**()

This function will terminate the script on controller.

void **stopL**(double a = 10.0, bool asynchronous = *false*)

Stop (linear in tool space) - decelerate tool speed to zero.

Parameters:

- **a** – tool acceleration [m/s^2] (rate of deceleration of the tool)
- **asynchronous** – a bool specifying if the stop command should be asynchronous. Stopping a fast move with a stopL with a low deceleration may block the caller for some seconds. To avoid blocking set asynchronous = true

void **stopJ**(double a = 2.0, bool asynchronous = *false*)

Stop (linear in joint space) - decelerate joint speeds to zero.

Parameters:

- **a** – joint acceleration [rad/s^2] (rate of deceleration of the leading axis).
- **asynchronous** – a bool specifying if the stop command should be asynchronous. Stopping a fast move with a stopJ with a low deceleration may block the caller for some seconds. To avoid blocking set asynchronous = true

bool **moveJ**(*const* std::vector<double> &q, double speed = 1.05, double acceleration = 1.4, bool asynchronous = *false*)

Move to joint position (linear in joint-space)

Parameters:

- **q** – joint positions
- **speed** – joint speed of leading axis [rad/s]
- **acceleration** – joint acceleration of leading axis [rad/s^2]
- **asynchronous** – a bool specifying if the move command should be asynchronous. If asynchronous is true it is possible to stop a move command using either the stopJ or stopL function. Default is false, this means the function will block until the movement has completed.

bool **moveJ**(*const* std::vector<std::vector<double>> &path, bool asynchronous = *false*)

Move to each joint position specified in a path.

Parameters:

- **path** – with joint positions that includes acceleration, speed and blend for each position
- **asynchronous** – a bool specifying if the move command should be asynchronous. If asynchronous is true it is possible to stop a move command using either the stopJ or stopL function. Default is false, this means the function will block until the movement has completed.

bool **moveJ_IK**(*const* std::vector<double> &pose, double speed = 1.05, double acceleration = 1.4, bool asynchronous = *false*)

Move to pose (linear in joint-space)

Parameters:

- **pose** – target pose
- **speed** – joint speed of leading axis [rad/s]
- **acceleration** – joint acceleration of leading axis [rad/s^2]
- **asynchronous** – a bool specifying if the move command should be asynchronous. If asynchronous is true it is possible to stop a move command using either the stopJ or stopL function. Default is false, this means the function will block until the movement has completed.

bool **moveL**(*const* std::vector<double> &pose, double speed = 0.25, double acceleration = 1.2, bool asynchronous = *false*)

Move to position (linear in tool-space)

Parameters:
- **pose** – target pose
- **speed** – tool speed [m/s]
- **acceleration** – tool acceleration [m/s^2]
- **asynchronous** – a bool specifying if the move command should be asynchronous. If asynchronous is true it is possible to stop a move command using either the stopJ or stopL function. Default is false, this means the function will block until the movement has completed.

bool **moveL**(*const* std::vector<std::vector<double>> &path, bool asynchronous = *false*)

Move to each pose specified in a path.

Parameters:
- **path** – with tool poses that includes acceleration, speed and blend for each position
- **asynchronous** – a bool specifying if the move command should be asynchronous. If asynchronous is true it is possible to stop a move command using either the stopJ or stopL function. Default is false, this means the function will block until the movement has completed.

bool **moveL_FK**(*const* std::vector<double> &q, double speed = 0.25, double acceleration = 1.2, bool asynchronous = *false*)

Move to position (linear in tool-space)

Parameters:
- **q** – joint positions
- **speed** – tool speed [m/s]
- **acceleration** – tool acceleration [m/s^2]
- **asynchronous** – a bool specifying if the move command should be asynchronous. If asynchronous is true it is possible to stop a move command using either the stopJ or stopL function. Default is false, this means the function will block until the movement has completed.

bool **speedJ**(*const* std::vector<double> &qd, double acceleration = 0.5, double time = 0.0)

Joint speed - Accelerate linearly in joint space and continue with constant joint speed.

Parameters:
- **qd** – joint speeds [rad/s]
- **acceleration** – joint acceleration [rad/s^2] (of leading axis)
- **time** – time [s] before the function returns (optional)

bool **speedL**(*const* std::vector<double> &xd, double acceleration = 0.25, double time = 0.0)

Tool speed - Accelerate linearly in Cartesian space and continue with constant tool speed.

The time t is optional;

Parameters:
- **xd** – tool speed [m/s] (spatial vector)
- **acceleration** – tool position acceleration [m/s^2]
- **time** – time [s] before the function returns (optional)

bool **servoJ**(*const* std::vector<double> &q, double speed, double acceleration, double time, double lookahead_time, double gain)

Servo to position (linear in joint-space)

Parameters:
- **q** – joint positions [rad]
- **speed** – NOT used in current version
- **acceleration** – NOT used in current version
- **time** – time where the command is controlling the robot. The function is blocking for time t [S]
- **lookahead_time** – time [S], range [0.03,0.2] smoothens the trajectory with this lookahead time
- **gain** – proportional gain for following target position, range [100,2000]

bool **servoL**(*const* std::vector<double> &pose, double speed, double acceleration, double time, double lookahead_time, double gain)

Servo to position (linear in tool-space)

Parameters:
- **pose** – target pose
- **speed** – NOT used in current version
- **acceleration** – NOT used in current version
- **time** – time where the command is controlling the robot. The function is blocking for time t [S]
- **lookahead_time** – time [S], range [0.03,0.2] smoothens the trajectory with this lookahead time
- **gain** – proportional gain for following target position, range [100,2000]

bool **movePath**(*const* Path &path, bool asynchronous = *false*)

Move to each waypoint specified in the given path.

Parameters:
- **path** – The path with waypoints
- **asynchronous** – a bool specifying if the move command should be asynchronous. If asynchronous is true it is possible to stop a move command using either the stopJ or stopL function. Default is false, this means the function will block until the movement has completed.

bool **servoStop**(double a = 10.0)

Stop servo mode and decelerate the robot.

Parameters: **a** – rate of deceleration of the tool [m/s^2]

bool **speedStop**(double a = 10.0)

Stop speed mode and decelerate the robot.

Parameters: **a** – rate of deceleration of the tool [m/s^2] if using speedL, for speedJ its [rad/s^2] and rate of deceleration of leading axis.

bool **servoC**(*const* std::vector<double> &pose, double speed = 0.25, double acceleration = 1.2, double blend = 0.0)

Servo to position (circular in tool-space).

Accelerates to and moves with constant tool speed v.

Parameters:

- **pose** – target pose
- **speed** – tool speed [m/s]
- **acceleration** – tool acceleration [m/s^2]
- **blend** – blend radius (of target pose) [m]

bool **forceMode**(*const* std::vector<double> &task_frame, *const* std::vector<int> &selection_vector, *const* std::vector<double> &wrench, int type, *const* std::vector<double> &limits)

Set robot to be controlled in force mode.

Parameters:

- **task_frame** – A pose vector that defines the force frame relative to the base frame.
- **selection_vector** – A 6d vector of 0s and 1s. 1 means that the robot will be compliant in the corresponding axis of the task frame
- **wrench** – The forces/torques the robot will apply to its environment. The robot adjusts its position along/about compliant axis in order to achieve the specified force/torque. Values have no effect for non-compliant axes
- **type** – An integer [1;3] specifying how the robot interprets the force frame. 1: The force frame is transformed in a way such that its y-axis is aligned with a vector pointing from the robot tcp towards the origin of the force frame. 2: The force frame is not transformed. 3: The force frame is transformed in a way such that its x-axis is the projection of the robot tcp velocity vector onto the x-y plane of the force frame.
- **limits** – (Float) 6d vector. For compliant axes, these values are the maximum allowed tcp speed along/about the axis. For non-compliant axes, these values are the maximum allowed deviation along/about an axis between the actual tcp position and the one set by the program.

bool **forceModeStop**()

Resets the robot mode from force mode to normal operation.

bool **jogStart**(*const* std::vector<double> &speeds, int feature = FEATURE_BASE, double acc = 0.5, *const* std::vector<double> &custom_frame = {})

Starts jogging with the given speed vector with respect to the given feature.

When jogging has started, it is possible to provide new speed vectors by calling the jogStart() function over and over again. This makes it possible to use a joystick or a 3D Space Navigator to provide new speed vectors if the user moves the joystick or the Space Navigator cap.

Parameters:

- **speed** – Speed vector for translation and rotation. Translation values are given in mm / s and rotation values in rad / s.
- **feature** – Configures to move to move with respect to base frame (FEATURE_BASE), tool frame (FEATURE_TOOL) or custom frame (FEATURE_CUSTOM) If the feature is FEATURE_CUSTOM then the custom_frame parameter needs to be a valid pose.
- **acc** – Acceleration value. If you need to manually jog items that require a lower acceleration, then you can provide a custom value here.
- **custom_frame** – The custom_frame given as pose if the selected feature is FEATURE_CUSTOM

bool **jogStop**()

Stops jogging that has been started start_jog.

bool **zeroFtSensor**()

Zeroes the TCP force/torque measurement from the builtin force/torque sensor by subtracting the current measurement from the subsequent.

bool **setPayload**(double mass, *const* std::vector<double> &cog = {})

Set payload.

Parameters:

- **mass** – Mass in kilograms
- **cog** – Center of Gravity, a vector [CoGx, CoGy, CoGz] specifying the displacement (in meters) from the toolmount. If not specified the current CoG will be used.

bool **teachMode**()

Set robot in freedrive mode.

In this mode the robot can be moved around by hand in the same way as by pressing the “freedrive” button. The robot will not be able to follow a trajectory (eg. a movej) in this mode.

bool **endTeachMode**()

Set robot back in normal position control mode after freedrive mode.

bool **forceModeSetDamping**(double damping)

Sets the damping parameter in force mode.

A value of 1 is full damping, so the robot will decellerate quickly if no force is present. A value of 0 is no damping, here the robot will maintain the speed.

The value is stored until this function is called again. Call this function before force mode is entered (otherwise default value will be used).

Parameters: **damping** – Between 0 and 1, default value is 0.005

bool **forceModeSetGainScaling**(double scaling)

Scales the gain in force mode.

A value larger than 1 can make force mode unstable, e.g. in case of collisions or pushing against hard surfaces.

The value is stored until this function is called again. Call this function before force mode is entered (otherwise default value will be used)

Parameters: **scaling** – scaling parameter between 0 and 2, default is 1.

int **toolContact**(*const* std::vector<double> &direction)

Detects when a contact between the tool and an object happens.

Parameters: **direction** – The first three elements are interpreted as a 3D vector (in the robot base coordinate system) giving the direction in which contacts should be detected. If all elements of the list are zero, contacts from all directions are considered.

Returns: The returned value is the number of time steps back to just before the contact have started. A value larger than 0 means that a contact is detected. A value of 0 means no contact.

double **getStepTime**()

Returns the duration of the robot time step in seconds.

In every time step, the robot controller will receive measured joint positions and velocities from the robot, and send desired joint positions and velocities back to the robot. This happens with a predetermined frequency, in regular intervals. This interval length is the robot time step.

Returns: Duration of the robot step in seconds or 0 in case of an error

std::vector<double> **getActualJointPositionsHistory**(int steps = 0)

Returns the actual past angular positions of all joints This function returns the angular positions as reported by the function “get_actual_joint_

positions()” which indicates the number of controller time steps occurring before the current time step.

An exception is thrown if indexing goes beyond the buffer size.

Parameters: **steps** – The number of controller time steps required to go back. 0 corresponds to “get_actual_joint_positions()”

Returns: The joint angular position vector in rad : [Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3] that was actual at the provided number of steps before the current time step.

std::vector<double> **getTargetWaypoint**()

Returns the target waypoint of the active move.

This is different from the target tcp pose which returns the target pose for each time step. The get_target_waypoint() returns the same target pose for movel, movej, movep or movec during the motion. It returns the target tcp pose, if none of the mentioned move functions are running.

This method is useful for calculating relative movements where the previous move command uses blends.

Returns: The desired waypoint TCP vector [X, Y, Z, Rx, Ry, Rz] or and empty vector in case of an error.

bool **setTcp**(*const* std::vector<double> &tcp_offset)

Sets the active tcp offset, i.e.

the transformation from the output flange coordinate system to the TCP as a pose.

Parameters: **tcp_offset** – A pose describing the transformation of the tcp offset.

std::vector<double> **getInverseKinematics**(*const* std::vector<double> &x, *const* std::vector<double> &qnear = {}, double max_position_error = 1e-10, double max_orientation_error = 1e-10)

Calculate the inverse kinematic transformation (tool space -> jointspace).

If qnear is defined, the solution closest to qnear is returned.Otherwise, the solution closest to the current joint positions is returned. If no tcp is provided the currently active tcp of the controller will be used.

Parameters:
- **x** – tool pose
- **qnear** – list of joint positions (Optional)
- **maxPositionError** – the maximum allowed positionerror (Optional)
- **maxOrientationError** – the maximum allowed orientationerror (Optional)

Returns: joint positions

std::vector<double> **poseTrans**(*const* std::vector<double> &p_from, *const* std::vector<double> &p_from_to)

Pose transformation to move with respect to a tool or w.r.t.

a custom feature/frame The first argument, p_from, is used to transform the second argument, p_from_to, and the result is then returned. This means that the result is the resulting pose, when starting at the coordinate system of p_from, and then in that coordinate system moving p_from_to. This function can be seen in two different views. Either the function transforms, that is translates and rotates, p_from_to by the parameters of p_from. Or the function is used to get the resulting pose, when first making a move of p_from and then from there, a move of p_from_to. If the poses were regarded as transformation matrices, it would look like:

```
* T_world->to = T_world->from * T_from->to
* T_x->to = T_x->from * T_from->to
*
```

Parameters:
- **p_from** – starting pose (spatial vector)
- **p_from_to** – pose change relative to starting pose (spatial vector)

Returns: resulting pose (spatial vector)

bool **triggerProtectiveStop**()

Triggers a protective stop on the robot.

Can be used for testing and debugging.

bool **isProgramRunning**()

Returns true if a program is running on the controller, otherwise it returns false This is just a shortcut for getRobotStatus() & RobotStatus::ROBOT_STATUS_PROGRAM_RUNNING.

uint32_t **getRobotStatus**()

Note

If you work with RTDE control and receive interface and you need to read the robot status or program running state, then you should always use the getRobotStatus() function from RTDE Control if you need a status that is in sync with the program uploading or reuploading of this object.

Returns: Robot status Bits 0-3: Is power on | Is program running | Is teach button pressed | Is power button pressed There is a synchronization gap between the three interfaces RTDE Control RTDE Receive and Dashboard Client. RTDE Control and RTDE Receive open its own RTDE connection and so the internal state is not in sync. That means, if RTDE Control reports, that program is running, RTDE Receive may still return that program is not running. The update of the Dashboard Client even needs more time. That means, the dashboard client still returns program not running after some milliseconds have passed after RTDE Control already reports program running.

int **getAsyncOperationProgress**()

Reads progress information for asynchronous operations that supports progress feedback (such as movePath).

Deprecated:

Use getAsyncOperationProgress() instead.

Return values:

- **<0** – Indicates that no async operation is running or that an async operation has finished. The returned values of two consecutive async operations is never equal. Normally the returned values are toggled between -1 and -2. This allows the application to clearly detect the end of an operation even if it is too short to see its start. That means, if the value returned by this function is less than 0 and is different from that last value returned by this function, then a new async operation has finished.
- **0** – Indicates that an async operation has started - progress 0
- **>=** – 0 Indicates the progress of an async operation. For example, if an operation has 3 steps, the progress ranges from 0 - 2. The progress value is updated, before a step is executed. When the last step has been executed, the value will change to -1 to indicate the end of the async operation.

AsyncOperationStatus **getAsyncOperationProgressEx**()

Returns extended async operation progress information for asynchronous operations that supports progress feedback (such as movePath).

It also returns the status of any async operation such as moveJ, moveL, stopJ or stopL.

See also

AsyncOperationStatus class for a detailed description of the progress status.

bool **setWatchdog**(double min_frequency = 10.0)

Enable a watchdog for the communication with a specified minimum frequency for which an input update is expected to arrive.

The watchdog is useful for safety critical realtime applications eg. servoing. The default action taken is to shutdown the control, if the watchdog is not kicked with the minimum frequency.

Preferably you would call this function right after the RTDEControlInterface has been constructed.

Parameters: **min_frequency** – The minimum frequency an input update is expected to arrive defaults to 10Hz.

bool **kickWatchdog**()

Kicks the watchdog safeguarding the communication.

Normally you would kick the watchdog in your control loop. Be sure to kick it as often as specified by the minimum frequency of the watchdog.

bool **isPoseWithinSafetyLimits**(*const* std::vector<double> &pose)

Checks if the given pose is reachable and within the current safety limits of the robot.

It checks safety planes limits, TCP orientation deviation limits and range of the robot. If a solution is found when applying the inverse kinematics to the given target TCP pose, this pose is considered reachable.

Parameters: **pose** – target pose
Returns: a bool indicating if the pose is within the safety limits.

bool **isJointsWithinSafetyLimits**(*const* std::vector<double> &q)

Checks if the given joint position is reachable and within the current safety limits of the robot.

This check considers joint limits (if the target pose is specified as joint positions), safety planes limits, TCP orientation deviation limits and range of the robot. If a solution is found when applying the inverse kinematics to the given target TCP pose, this pose is considered reachable

Parameters: **q** – joint positions
Returns: a bool indicating if the joint positions are within the safety limits.

std::vector<double> **getJointTorques**()

Returns the torques of all joints.

The torque on the joints, corrected by the torque needed to move the robot itself (gravity, friction, etc.), returned as a vector of length 6.

Returns: The joint torque vector in Nm: [Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3]

std::vector<double> **getTCPOffset**()

Gets the active tcp offset, i.e.

the transformation from the output flange coordinate system to the TCP as a pose.

Returns: the TCP offset as a pose

std::vector<double> **getForwardKinematics**(*const* std::vector<double> &q = {}, *const* std::vector<double> &tcp_offset = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0})

Calculate the forward kinematic transformation (joint space -> tool space) using the calibrated robot kinematics.

If no joint position vector is provided the current joint angles of the robot arm will be used. If no tcp is provided the currently active tcp of the controller will be used.

NOTICE! If you only specify the tcp_offset, and leave the q empty the current joint positions will be used. On the other hand if you specify the q but not the tcp_offset an offset of zeroes will be assumed see the default value.

Parameters:
- **q** – joint position vector (Optional)
- **tcp_offset** – tcp offset pose (Optional)

Returns: the forward kinematic transformation as a pose

bool **isSteady**()

Checks if robot is fully at rest.

True when the robot is fully at rest, and ready to accept higher external forces and torques, such as from industrial screwdrivers.

Note: This function will always return false in modes other than the standard position mode, e.g. false in force and teach mode.

Returns: True when the robot is fully at rest. Returns False otherwise.

bool **moveUntilContact**(*const* std::vector<double> &xd, *const* std::vector<double> &direction = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0}, double acceleration = 0.5)

Move the robot until contact, with specified speed and contact detection direction.

The robot will automatically retract to the initial point of contact.

See also

startContactDetection() function for async contact detection

Parameters:
- **xd** – tool speed [m/s] (spatial vector)
- **direction** – List of six floats. The first three elements are interpreted as a 3D vector (in the robot base coordinate system) giving the direction in which contacts should be detected. If all elements of the list are zero, contacts from all directions are considered. You can also set direction=get_target_tcp_speed() in which case it will detect contacts in the direction of the TCP movement.
- **acceleration** – tool position acceleration [m/s^2]

Returns: True once the robot is in contact.

bool **freedriveMode**(*const* std::vector<int> &free_axes = {1, 1, 1, 1, 1, 1}, *const* std::vector<double> &feature = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0})

Set robot in freedrive mode.

In this mode the robot can be moved around by hand in the same way as by pressing the “freedrive” button.

The robot will not be able to follow a trajectory (eg. a movej) in this mode.

The default parameters enables the robot to move freely in all directions. It is possible to enable constrained freedrive by providing user specific parameters.

Examples:

- freedrive_mode()
  - Robot can move freely in all directions.
- freedrive_mode(freeAxes=[1,0,0,0,0,0], feature=p[0.1,0,0,0,0.785])
  - Example Parameters:
    - freeAxes = [1,0,0,0,0,0] -> The robot is compliant in the x direction relative to the feature.
    - feature = p[0.1,0,0,0,0.785] -> This feature is offset from the base frame with 100 mm in the
      x direction and rotated 45 degrees in the rz direction.

Note: Immediately before entering freedrive mode, avoid:

- movements in the non-compliant axes
- high acceleration in freedrive mode
- high deceleration in freedrive mode

Parameters:
- **freeAxes** – A 6 dimensional vector that contains 0's and 1's, these indicates in which axes movement is allowed. The first three values represents the cartesian directions along x, y, z, and the last three defines the rotation axis, rx, ry, rz. All relative to the selected feature
- **feature** – A pose vector that defines a freedrive frame relative to the base frame. For base and tool reference frames predefined constants "base", and "tool" can be used in place of pose vectors.

Returns: true when the robot is in freedrive mode, false otherwise.

bool **endFreedriveMode**()

Set robot back in normal position control mode after freedrive mode.

Returns: true when the robot is not in freedrive anymore.

int **getFreedriveStatus**()

Returns status of freedrive mode for current robot pose.

Constrained freedrive usability is reduced near singularities. Value returned by this function corresponds to distance to the nearest singularity.

It can be used to advice operator to follow different path or switch to unconstrained freedrive.

Returns:
- 0 = Normal operation.
- 1 = Near singularity.
- 2 = Too close to singularity. High movement resistance in freedrive.

bool **setExternalForceTorque**(*const* std::vector<double> &external_force_torque)

Input external wrench when using ft_rtde_input_enable builtin.

Parameters: **external_force_torque** – A 6 dimensional vector that contains the external wrench.

bool **ftRtdeInputEnable**(bool enable, double sensor_mass = 0.0, *const* std::vector<double> &sensor_measuring_offset = {0.0, 0.0, 0.0}, *const* std::vector<double> &sensor_cog = {0.0, 0.0, 0.0})

This function is used for enabling and disabling the use of external F/T measurements in the controller.

Be aware that the following function is impacted:

- force_mode
- screw_driving
- freedrive_mode

The RTDE interface shall be used for feeding F/T measurements into the real-time control loop of the robot using input variable external_force_torque of type VECTOR6D. If no other RTDE watchdog has been configured (using script function rtde_set_watchdog), a default watchdog will be set to a 10Hz minimum update frequency when the external F/T sensor functionality is enabled. If the update frequency is not met the robot program will pause.

Notes: This function replaces the deprecated enable_external_ft_sensor. The TCP Configuration in the installation must also include the weight and offset contribution of the sensor. Only the enable parameter is required; sensor mass, offset and center of gravity are optional (zero if not provided).

Parameters:

- **enable** – enable or disable feature (bool)
- **sensor_mass** – mass of the sensor in kilograms (float)
- **sensor_measuring_offset** – [x, y, z] measuring offset of the sensor in meters relative to the tool flange frame
- **sensor_cog** – [x, y, z] center of gravity of the sensor in meters relative to the tool flange frame

bool **enableExternalFtSensor**(bool enable, double sensor_mass = 0.0, *const* std::vector<double> &sensor_measuring_offset = {0.0, 0.0, 0.0}, *const* std::vector<double> &sensor_cog = {0.0, 0.0, 0.0})

this function is used for enabling and disabling the use of external F/T measurements in the controller.

(Deprecated, but can be used when ftRtdeInputEnable() is not available)

Be aware that the following function is impacted:

- force_mode
- screw_driving
- freedrive_mode

The RTDE interface shall be used for feeding F/T measurements into the real-time control loop of the robot using input variable external_force_torque of type VECTOR6D. If no other RTDE watchdog has been configured (using script function rtde_set_watchdog), a default watchdog will be set to a 10Hz minimum update frequency when the external F/T sensor functionality is enabled. If the update frequency is not met the robot program will pause.

Notes: When using this function, the sensor position is applied such that the resulting torques are computed with opposite sign. New programs should use ftRtdeInputEnable in place of this. The TCP Configuration in the installation must also include the weight and offset contribution of the sensor. Only the enable parameter is required; sensor mass, offset and center of gravity are optional (zero if not provided).

Parameters:
- **enable** – enable or disable feature (bool)
- **sensor_mass** – mass of the sensor in kilograms (float)
- **sensor_measuring_offset** – [x, y, z] measuring offset of the sensor in meters relative to the tool flange frame
- **sensor_cog** – [x, y, z] center of gravity of the sensor in meters relative to the tool flange frame

std::vector<double> **getActualToolFlangePose**()

Returns the current measured tool flange pose.

Returns the 6d pose representing the tool flange position and orientation specified in the base frame, without the Tool Center Point offset. The calculation of this pose is based on the actual robot encoder readings.

Returns: the current actual tool flange vector: [X, Y, Z, Rx, Ry, Rz]

bool **setGravity**(*const* std::vector<double> &direction)

Set the direction of the acceleration experienced by the robot.

When the robot mounting is fixed, this corresponds to an acceleration of g away from the earth's centre.

setGravity({0, 9.82*sin(theta), 9.82*cos(theta)}) will set the acceleration for a robot that is rotated "theta" radians around the x-axis of the robot base coordinate system.

Example command: setGravity({0, 9.82, 0}) Example Parameters:

- d is vector with a direction of y (direction of the robot cable) and a magnitude of 9.82 m/s^2 (1g).

Parameters: **d** – a 3D vector, describing the direction of the gravity, relative to the base of the robot.

bool **getInverseKinematicsHasSolution**(*const* std::vector<double> &x, *const* std::vector<double> &qnear = {}, double max_position_error = 1e-10, double max_orientation_error = 1e-10)

Check if get_inverse_kin has a solution and return boolean (true) or (false).

This can be used to avoid the runtime exception of getInverseKin() when no solution exists.

Parameters:
- **x** – tool pose
- **qnear** – list of joint positions (Optional)
- **maxPositionError** – the maximum allowed positionerror (Optional)
- **maxOrientationError** – the maximum allowed orientationerror (Optional)

Returns: true if getInverseKin has a solution, false otherwise (bool)

bool **startContactDetection**(*const* std::vector<double> &direction = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0})

Starts async contact detection thread.

Move the robot until contact, with specified speed and contact detection direction. The robot will automatically retract to the initial point of contact.

Contact monitoring should get started after a async move command has been started. That means, there should be no async move commands after this command because async move commands may cause a reupload of the script which in turn kills the contact monitoring thread. If a contact is detected, the contact monitoring thread is stopped, the robot is stopped and then retracted to the initial point of contact.

```
rtde_control.moveL(target, 0.25, 0.5, true);
rtde_control.startContactDetection(); // detect contact in direction of TCP movement

// now wait until the robot stops - it either stops if it has reached
// the target pose or if a contact has been detected
// you can use the readContactDetection() function, to check if a contact
// has been detected.
bool contact_detected = rtde_control.readContactDetection();


contact_detected = rtde_control.stopContactDetection();
```

Parameters: **direction** – List of six floats. The first three elements are interpreted as a 3D vector (in the robot base coordinate system) giving the direction in which contacts should be detected. If all elements of the list are zero, direction will be set to direction=get_target_tcp_speed() in which case it will detect contacts in the direction of the TCP movement.

Returns: Returns true, if a contact has been detected

bool **readContactDetection**()

Reads the current async contact detection state.

The function returns true, when a contact between the tool and an object has been detected.

bool **stopContactDetection**()

Stop contact monitoring This function stops contact monitoring and returns true, if a contact has been detected.

Normally the contact detection is stopped, as soon as a contact ha been detected. If you would like to stop the contact detection manually, i.e. because of a timeout, the you can use this function.

bool **setTargetPayload**(double mass, *const* std::vector<double> &cog = {}, *const* std::vector<double> &inertia = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0})

Sets the mass, center of gravity (abbr.

CoG) and the inertia matrix of the active payload.

This function must be called when the payload mass, the mass displacement (CoG) or the inertia matrix changes - (i.e. when the robot picks up or puts down a workpiece).

Notes:

- This script should be used instead of the deprecated set_payload, set_payload_mass, and set_payload_cog.
- The payload mass and CoG are required, the inertia matrix is optional. When left out a zero inertia matrix will be used.
- The maximum value allowed for each component of the inertia matrix is +/- 133 kg*m^2. An exception is thrown if limits are exceeded

Parameters:
- **mass** – Mass in kilograms
- **cog** – Center of Gravity, a vector [CoGx, CoGy, CoGz] specifying the displacement (in meters) from the toolmount.
- **inertia** – payload inertia matrix (in kg*m^2), as a vector with six elements [Ixx, Iyy, Izz, Ixy, Ixz, Iyz] with origin in the CoG and the axes aligned with the tool flange axes.

bool **directTorque**(*const* std::vector<double> &torque, bool friction_comp = *true*)

Set the joint torques of the robot using Direct Joint Torque Control.

The directTorque function will use one timestep no matter the speed scaling, similar to the sync() function. It needs to be called continuously at each robot times step. Otherwise the robot will return to position control mode.

It compensates for gravity and uses, by default, our friction compensation. The friction compensation can be disabled by the optional argument friction_comp=False .

Parameters:
- **torque** – vector of target torques to be commanded to the robot joints.
- **friction_comp** – Enable internal friction compensation; Enabled by default

std::vector<double> **getMassMatrix**(*const* std::vector<double> &q = {}, bool include_rotors_inertia = *false*)

Get mass matrix.

Parameters:
- **q** – Robot joint position; Default uses the current robot joint positions
- **include_rotors_inertia;** – also includes the inertia on the motor side of the gear; It is disabled by default.

Returns: The mass matrix

std::vector<double> **getCoriolisAndCentrifugalTorques**(*const* std::vector<double> &q = {}, *const* std::vector<double> &qd = {})

Get the coriolis and centrifugal torques.

Parameters:
- **q** – Robot joint position; Default uses the current robot joint positions
- **qd** – Robot joint velocities; Default uses the current robot joint velocities.

std::vector<double> **getTargetJointAccelerations**()

Get target joint accelerations.

Returns: List of the joints acceleration derived directly from the encoders. Please expect noise on the output.

std::vector<double> **getJacobian**(*const* std::vector<double> &pos = {}, *const* std::vector<double> &tcp = {})

Get Jacobian matrix.

Parameters:
- **pos** – Joint space position; Default uses the robots current pose for the tool flange.
- **tcp** – tcp offset; Default uses the active tcp offset

Returns: Return the Jacobian matrix.

std::vector<double> **getJacobianTimeDerivative**(*const* std::vector<double> &pos = {}, *const* std::vector<double> &vel = {}, *const* std::vector<double> &tcp = {})

Get time-derivative Jacobian matrix.

Parameters:
- **pos** – Joint space position; Default uses the robot's current pose for the tool flange.
- **vel** – Joint space velocity; Default uses the robot's current velocity of the tool flange.
- **tcp;** – tcp offset; Default uses the active tcp offset.

Returns: Return the Jacobian matrix.

RTDE Receive Interface API

class **RTDEReceiveInterface**

Public Types

enum **SafetyStatus**

Values:

enumerator **IS_NORMAL_MODE**

enumerator **IS_REDUCED_MODE**

enumerator **IS_PROTECTIVE_STOPPED**

enumerator **IS_RECOVERY_MODE**

enumerator **IS_SAFEGUARD_STOPPED**

enumerator **IS_SYSTEM_EMERGENCY_STOPPED**

enumerator **IS_ROBOT_EMERGENCY_STOPPED**

enumerator **IS_EMERGENCY_STOPPED**

enumerator **IS_VIOLATION**

enumerator **IS_FAULT**

enumerator **IS_STOPPED_DUE_TO_SAFETY**

enum **OutputIntRegisters**

Values:

enumerator **OUTPUT_INT_REGISTER_0**

enumerator **OUTPUT_INT_REGISTER_1**

enumerator **OUTPUT_INT_REGISTER_2**

enumerator **OUTPUT_INT_REGISTER_3**

enumerator **OUTPUT_INT_REGISTER_4**

enumerator **OUTPUT_INT_REGISTER_5**

enumerator **OUTPUT_INT_REGISTER_6**

enumerator **OUTPUT_INT_REGISTER_7**

enumerator **OUTPUT_INT_REGISTER_8**

enumerator **OUTPUT_INT_REGISTER_9**

enumerator **OUTPUT_INT_REGISTER_10**

enumerator **OUTPUT_INT_REGISTER_11**

enumerator **OUTPUT_INT_REGISTER_12**

enumerator **OUTPUT_INT_REGISTER_13**

enumerator **OUTPUT_INT_REGISTER_14**

enumerator **OUTPUT_INT_REGISTER_15**

enumerator **OUTPUT_INT_REGISTER_16**

enumerator **OUTPUT_INT_REGISTER_17**

enumerator **OUTPUT_INT_REGISTER_18**

enumerator **OUTPUT_INT_REGISTER_19**

enumerator **OUTPUT_INT_REGISTER_20**

enumerator **OUTPUT_INT_REGISTER_21**

enumerator **OUTPUT_INT_REGISTER_22**

enumerator **OUTPUT_INT_REGISTER_23**

enum **OutputDoubleRegisters**

Values:

enumerator **OUTPUT_DOUBLE_REGISTER_0**

enumerator **OUTPUT_DOUBLE_REGISTER_1**

enumerator **OUTPUT_DOUBLE_REGISTER_2**

enumerator **OUTPUT_DOUBLE_REGISTER_3**

enumerator **OUTPUT_DOUBLE_REGISTER_4**

enumerator **OUTPUT_DOUBLE_REGISTER_5**

enumerator **OUTPUT_DOUBLE_REGISTER_6**

enumerator **OUTPUT_DOUBLE_REGISTER_7**

enumerator **OUTPUT_DOUBLE_REGISTER_8**

enumerator **OUTPUT_DOUBLE_REGISTER_9**

enumerator **OUTPUT_DOUBLE_REGISTER_10**

enumerator **OUTPUT_DOUBLE_REGISTER_11**

enumerator **OUTPUT_DOUBLE_REGISTER_12**

enumerator **OUTPUT_DOUBLE_REGISTER_13**

enumerator **OUTPUT_DOUBLE_REGISTER_14**

enumerator **OUTPUT_DOUBLE_REGISTER_15**

enumerator **OUTPUT_DOUBLE_REGISTER_16**

enumerator **OUTPUT_DOUBLE_REGISTER_17**

enumerator **OUTPUT_DOUBLE_REGISTER_18**

enumerator **OUTPUT_DOUBLE_REGISTER_19**

enumerator **OUTPUT_DOUBLE_REGISTER_20**

enumerator **OUTPUT_DOUBLE_REGISTER_21**

enumerator **OUTPUT_DOUBLE_REGISTER_22**

enumerator **OUTPUT_DOUBLE_REGISTER_23**

enum **RuntimeState**

Values:

enumerator **STOPPING**

enumerator **STOPPED**

enumerator **PLAYING**

enumerator **PAUSING**

enumerator **PAUSED**

enumerator **RESUMING**

enum class **PausingState**

Values:

enumerator **PAUSED**

enumerator **RUNNING**

enumerator **RAMPUP**

Public Functions

explicit **RTDEReceiveInterface**(std::string hostname, double frequency = -1.0, std::vector<std::string> variables = {}, bool verbose = *false*, bool use_upper_range_registers = *false*, int rt_priority = RT_PRIORITY_UNDEFINED)

Constructor for the RTDEReceiveInterface class.

Creates an interface to receive data from a Universal Robot.

Note

The list of available variables depends on the robot model and software version. Refer to the UR RTDE documentation for a complete list of available variables.

Parameters:

- **hostname** – The IP address or hostname of the robot.
- **frequency** – The frequency at which RTDE data will be exchanged with the robot (-1.0 means use the robot's default frequency, 500Hz for e-Series and UR-Series, while its 125Hz for the CB-series).
- **variables** – A vector of variable names to be monitored (empty vector means use all default variables).
- **verbose** – Enable verbose output for debugging purposes.
- **use_upper_range_registers** – Use the upper range registers for RTDE communication.
- **rt_priority** – Real-time priority of the RTDEReceiveInterface thread (if supported by the OS).

virtual **~RTDEReceiveInterface**()

void **disconnect**()

Returns: Can be used to disconnect from the robot. To reconnect you have to call the reconnect() function.

bool **reconnect**()

Returns: Can be used to reconnect to the robot after a lost connection.

void **waitPeriod**(*const* std::chrono::steady_clock::time_point &t_cycle_start)

Used for waiting the rest of the control period, set implicitly as dt = 1 / frequency.

A combination of sleeping and spinning are used to achieve the lowest possible jitter. The function is especially useful for a realtime control loop. NOTE: the function is to be used in combination with the initPeriod(). See the realtime_control_example.cpp.

Parameters: **t_cycle_start** – the start of the control period. Typically given as dt = 1 / frequency.

std::chrono::steady_clock::time_point **initPeriod**()

This function is used in combination with waitPeriod() and is used to get the start of a control period / cycle.

See the realtime_control_example.cpp.

bool **startFileRecording**(*const* std::string &filename, *const* std::vector<std::string> &variables = {})

Returns: Can be used to reconnect to the robot after a lost connection.

bool **stopFileRecording**()

Returns: Can be used to reconnect to the robot after a lost connection.

`bool isConnected()`

Returns: Connection status for RTDE, useful for checking for lost connection.

`double getTimestamp()`

Returns: Time elapsed since the controller was started [s]

`std::vector<double> getTargetQ()`

Returns: Target joint positions

`std::vector<double> getTargetQd()`

Returns: Target joint velocities

`std::vector<double> getTargetQdd()`

Returns: Target joint accelerations

`std::vector<double> getTargetCurrent()`

Returns: Target joint currents

`std::vector<double> getTargetMoment()`

Returns: Target joint moments (torques)

`std::vector<double> getActualQ()`

Returns: Actual joint positions

`std::vector<double> getActualQd()`

Returns: Actual joint velocities

`std::vector<double> getActualCurrent()`

Returns: Actual joint currents

`std::vector<double> getJointControlOutput()`

Returns: Joint control currents

`std::vector<double> getActualTCPPose()`

Returns: Actual Cartesian coordinates of the tool: (x,y,z,rx,ry,rz), where rx, ry and rz is a rotation vector representation of the tool orientation

std::vector<double> **getActualTCPSpeed**()

Returns: Actual speed of the tool given in Cartesian coordinates

std::vector<double> **getActualTCPForce**()

Returns: Generalized forces in the TCP

std::vector<double> **getTargetTCPPose**()

Returns: Target Cartesian coordinates of the tool: (x,y,z,rx,ry,rz), where rx, ry and rz is a rotation vector representation of the tool orientation

std::vector<double> **getTargetTCPSpeed**()

Returns: Target speed of the tool given in Cartesian coordinates

uint64_t **getActualDigitalInputBits**()

Returns: Current state of the digital inputs. 0-7: Standard, 8-15: Configurable, 16-17: Tool

bool **getDigitalInState**(std::uint8_t input_id)

Test if a digital input is set 'high' or 'low' the range is 0-7: Standard, 8-15: Configurable, 16-17: Tool.

Parameters: **input_id** – the id of the digital input to test
Returns: a bool indicating the state of the digital input

std::vector<double> **getJointTemperatures**()

Returns: Temperature of each joint in degrees Celsius

double **getActualExecutionTime**()

Returns: Controller real-time thread execution time

int32_t **getRobotMode**()

Returns: Robot mode -1 = ROBOT_MODE_NO_CONTROLLER 0 = ROBOT_MODE_DISCONNECTED 1 = ROBOT_MODE_CONFIRM_SAFETY 2 = ROBOT_MODE_BOOTING 3 = ROBOT_MODE_POWER_OFF 4 = ROBOT_MODE_POWER_ON 5 = ROBOT_MODE_IDLE 6 = ROBOT_MODE_BACKDRIVE 7 = ROBOT_MODE_RUNNING 8 = ROBOT_MODE_UPDATING_FIRMWARE

uint32_t **getRobotStatus**()

Returns: Robot status Bits 0-3: Is power on | Is program running | Is teach button pressed | Is power button pressed

`std::vector<int32_t> getJointMode()`

Returns: Joint control modes

`int32_t getSafetyMode()`

Returns: Safety mode

`uint32_t getSafetyStatusBits()`

Returns: Safety status bits Bits 0-10: Is normal mode | Is reduced mode | Is protective stopped | Is recovery mode | Is safeguard stopped | Is system emergency stopped | Is robot emergency stopped | Is emergency stopped | Is violation | Is fault | Is stopped due to safety

`std::vector<double> getActualToolAccelerometer()`

Returns: Tool x, y and z accelerometer values

`double getSpeedScaling()`

Returns: Speed scaling of the trajectory limiter

`double getTargetSpeedFraction()`

Returns: Target speed fraction

`double getActualMomentum()`

Returns: Norm of Cartesian linear momentum

`double getActualMainVoltage()`

Returns: Safety Control Board: Main voltage

`double getActualRobotVoltage()`

Returns: Safety Control Board: Robot voltage (48V)

`double getActualRobotCurrent()`

Returns: Safety Control Board: Robot current

`std::vector<double> getActualJointVoltage()`

Returns: Actual joint voltages

uint64_t **getActualDigitalOutputBits**()

Returns: Current state of the digital outputs. 0-7: Standard, 8-15: Configurable, 16-17: Tool

bool **getDigitalOutState**(std::uint8_t output_id)

Test if a digital output is set 'high' or 'low' the range is 0-7: Standard, 8-15: Configurable, 16-17: Tool.

Parameters: **output_id** – the id of the digital output to test
Returns: a bool indicating the state of the digital output

uint32_t **getRuntimeState**()

Returns: Program state

double **getStandardAnalogInput0**()

Returns: Standard analog input 0 [A or V]

double **getStandardAnalogInput1**()

Returns: Standard analog input 1 [A or V]

double **getStandardAnalogOutput0**()

Returns: Standard analog output 0 [A or V]

double **getStandardAnalogOutput1**()

Returns: Standard analog output 1 [A or V]

bool **isProtectiveStopped**()

Returns: a bool indicating if the robot is in 'Protective stop'

bool **isEmergencyStopped**()

Returns: a bool indicating if the robot is in 'Emergency stop'

int **getOutputIntRegister**(int output_id)

Get the specified output integer register in either lower range [18-22] or upper range [42-46].

Parameters: **output_id** – the id of the register to read, current supported range is: [18-22] or [42-46], this can be adjusted by changing the RTDEReceiveInterface output recipes and by using the use_upper_range_registers constructor flag to switch between lower and upper range.

Returns: an integer from the specified output register

double **getOutputDoubleRegister**(int output_id)

Get the specified output double register in either lower range [18-22] or upper range [42-46].

Parameters: **output_id** – the id of the register to read, current supported range is: [18-22] or [42-46], this can be adjusted by changing the RTDEReceiveInterface output recipes and by using the use_upper_range_registers constructor flag to switch between lower and upper range.

Returns: a double from the specified output register

double **getSpeedScalingCombined**()

Get the combined speed scaling The combined speed scaling is the speed scaling resulting from multiplying the speed scaling with the target speed fraction.

The combined speed scaling takes the runtime_state of the controller into account. If eg. a motion is paused on the teach pendant, and later continued, the speed scaling will be ramped up from zero and return to speed_scaling * target_speed_fraction when the runtime_state is RUNNING again.

This is useful for scaling trajectories with the slider speed scaling currently set on the teach pendant.

Returns: the actual combined speed scaling

std::vector<double> **getFtRawWrench**()

Get the raw force and torque measurement, not compensated for forces and torques caused by the payload.

Returns: the raw force and torque measurement

std::vector<double> **getActualCurrentAsTorque**()

Get the actual joint currents converted to torques.

Returns: actual joint currents converted to torque

double **getPayload**()

Get the payload of the robot in [kg].

Returns: the payload in [kg]

std::vector<double> **getPayloadCog**()

Get the payload Center of Gravity (CoGx, CoGy, CoGz)

Returns: the payload Center of Gravity (CoGx, CoGy, CoGz) in [m]

std::vector<double> **getPayloadInertia**()

Get the payload inertia matrix elements (Ixx,Iyy,Izz,Ixy,Ixz,Iyz) expressed in kg*m^2.

Returns: the payload inertia matrix elements (Ixx,Iyy,Izz,Ixy,Ixz,Iyz) expressed in kg*m^2

void **receiveCallback**(std::atomic<bool> *stop_thread)

void **recordCallback**(std::atomic<bool> *stop_thread)

inline const std::shared_ptr<RobotState> &**robot_state**() *const*

RTDE IO Interface API

class **RTDEIOInterface**

Public Functions

void **disconnect**()

Can be used to disconnect the RTDE IO client.

bool **reconnect**()

Can be used to reconnect to the robot after a lost connection.

bool **isConnected**()

Returns: Connection status for RTDE, useful for checking for lost connection.

bool **setStandardDigitalOut**(std::uint8_t output_id, bool signal_level)

Set standard digital output signal level.

Parameters:
- **output_id** – The number (id) of the output, integer: [0:7]
- **signal_level** – The signal level. (boolean)

bool **setConfigurableDigitalOut**(std::uint8_t output_id, bool signal_level)

Set configurable digital output signal level.

Parameters:
- **output_id** – The number (id) of the output, integer: [0:7]
- **signal_level** – The signal level. (boolean)

bool **setToolDigitalOut**(std::uint8_t output_id, bool signal_level)

Set tool digital output signal level.

Parameters:
- **output_id** – The number (id) of the output, integer: [0:1]
- **signal_level** – The signal level. (boolean)

bool **setSpeedSlider**(double speed)

Set the speed slider on the controller.

Parameters: **speed** – set the speed slider on the controller as a fraction value between 0 and 1 (1 is 100%)

bool **setAnalogOutputVoltage**(std::uint8_t output_id, double voltage_ratio)

Set Analog output voltage.

Parameters:
- **output_id** – The number (id) of the output, integer: [0:1]
- **voltage_ratio** – voltage set as a (ratio) of the voltage span [0..1], 1 means full voltage.

bool **setAnalogOutputCurrent**(std::uint8_t output_id, double current_ratio)

Set Analog output current.

Parameters:
- **output_id** – The number (id) of the output, integer: [0:1]
- **current_ratio** – current set as a (ratio) of the current span [0..1], 1 means full current.

bool **setInputIntRegister**(int input_id, int value)

Set the specified input integer register in either lower range [18-22] or upper range [42-46].

Parameters:
- **input_id** – the id of the register to set, current supported range is: [18-22] or [42-46], this can be adjusted by changing the RTDEControlInterface input recipes and by using the use_upper_range_registers constructor flag to switch between lower and upper range.
- **value** – the desired integer value

Returns: true if the register is successfully set, false otherwise.

bool **setInputDoubleRegister**(int input_id, double value)

Set the specified input double register in either lower range [18-22] or upper range [42-46].

Parameters:
- **input_id** – the id of the register to set, current supported range is: [18-22] or [42-46], this can be adjusted by changing the RTDEControlInterface input recipes and by using the use_upper_range_registers constructor flag to switch between lower and upper range.
- **value** – the desired double value

Returns: true if the register is successfully set, false otherwise.

RTDE Class API

class **RTDE**

Public Types

enum **RTDECommand**

Values:

enumerator **RTDE_REQUEST_PROTOCOL_VERSION**

enumerator **RTDE_GET_URCONTROL_VERSION**

enumerator **RTDE_TEXT_MESSAGE**

enumerator **RTDE_DATA_PACKAGE**

enumerator **RTDE_CONTROL_PACKAGE_SETUP_OUTPUTS**

enumerator **RTDE_CONTROL_PACKAGE_SETUP_INPUTS**

enumerator **RTDE_CONTROL_PACKAGE_START**

enumerator **RTDE_CONTROL_PACKAGE_PAUSE**

enum class **ConnectionState** : std::uint8_t

Values:

enumerator **DISCONNECTED**

enumerator **CONNECTED**

enumerator **STARTED**

enumerator **PAUSED**

Public Functions

explicit **RTDE**(*const* std::string hostname, int port = 30004, bool verbose = *false*)

virtual **~RTDE**()

void **connect**()

void **disconnect**(bool send_pause = *true*)

bool **isConnected**()

bool **isStarted**()

bool **isDataAvailable**()

bool **negotiateProtocolVersion**()

std::tuple<std::uint32_t, std::uint32_t, std::uint32_t, std::uint32_t> **getControllerVersion**()

void **receive**()

boost::system::error_code **receiveData**(std::shared_ptr<RobotState> &robot_state)

void **send**(*const* RobotCommand &robot_cmd)

void **sendAll**(*const* std::uint8_t &command, std::string payload = "")

void **sendStart**()

void **sendPause**()

bool **sendOutputSetup**(*const* std::vector<std::string> &output_names, double frequency)

bool **sendInputSetup**(*const* std::vector<std::string> &input_names)

class **RobotCommand**

Public Types

enum **Type**

Values:

enumerator **NO_CMD**

enumerator **MOVEJ**

enumerator **MOVEJ_IK**

enumerator **MOVEL**

enumerator **MOVEL_FK**

enumerator **FORCE_MODE**

enumerator **FORCE_MODE_STOP**

enumerator **ZERO_FT_SENSOR**

enumerator **SPEEDJ**

enumerator **SPEEDL**

enumerator **SERVOJ**

enumerator **SERVOC**

enumerator **SET_STD_DIGITAL_OUT**

enumerator **SET_TOOL_DIGITAL_OUT**

enumerator **SPEED_STOP**

enumerator **SERVO_STOP**

enumerator **SET_PAYLOAD**

enumerator **TEACH_MODE**

enumerator **END_TEACH_MODE**

enumerator **FORCE_MODE_SET_DAMPING**

enumerator **FORCE_MODE_SET_GAIN_SCALING**

enumerator **SET_SPEED_SLIDER**

enumerator **SET_STD_ANALOG_OUT**

enumerator **SERVOL**

enumerator **TOOL_CONTACT**

enumerator **GET_STEPTIME**

enumerator **GET_ACTUAL_JOINT_POSITIONS_HISTORY**

enumerator **GET_TARGET_WAYPOINT**

enumerator **SET_TCP**

enumerator **GET_INVERSE_KINEMATICS_ARGS**

enumerator **PROTECTIVE_STOP**

enumerator **STOPL**

enumerator **STOPJ**

enumerator **SET_WATCHDOG**

enumerator **IS_POSE_WITHIN_SAFETY_LIMITS**

enumerator **IS_JOINTS_WITHIN_SAFETY_LIMITS**

enumerator **GET_JOINT_TORQUES**

enumerator **POSE_TRANS**

enumerator **GET_TCP_OFFSET**

enumerator **JOG_START**

enumerator **JOG_STOP**

enumerator **GET_FORWARD_KINEMATICS_DEFAULT**

enumerator **GET_FORWARD_KINEMATICS_ARGS**

enumerator **MOVE_PATH**

enumerator **GET_INVERSE_KINEMATICS_DEFAULT**

enumerator **IS_STEADY**

enumerator **SET_CONF_DIGITAL_OUT**

enumerator **SET_INPUT_INT_REGISTER**

enumerator **SET_INPUT_DOUBLE_REGISTER**

enumerator **MOVE_UNTIL_CONTACT**

enumerator **FREEDRIVE_MODE**

enumerator **END_FREEDRIVE_MODE**

enumerator **GET_FREEDRIVE_STATUS**

enumerator **SET_EXTERNAL_FORCE_TORQUE**

enumerator **FT_RTDE_INPUT_ENABLE**

enumerator **ENABLE_EXTERNAL_FT_SENSOR**

enumerator **GET_ACTUAL_TOOL_FLANGE_POSE**

enumerator **SET_GRAVITY**

enumerator **GET_INVERSE_KINEMATICS_HAS_SOLUTION_DEFAULT**

enumerator **GET_INVERSE_KINEMATICS_HAS_SOLUTION_ARGS**

enumerator **START_CONTACT_DETECTION**

enumerator **STOP_CONTACT_DETECTION**

enumerator **READ_CONTACT_DETECTION**

enumerator **SET_TARGET_PAYLOAD**

enumerator **DIRECT_TORQUE**

enumerator **GET_MASS_MATRIX**

enumerator **GET_CORIOLIS_AND_CENTRIFUGAL_TORQUES**

enumerator **GET_TARGET_JOINT_ACCELERATIONS**

enumerator **GET_JACOBIAN**

enumerator **GET_JACOBIAN_TIME_DERIVATIVE**

enumerator **WATCHDOG**

enumerator **STOP_SCRIPT**

enum **Recipe**

Values:

enumerator **RECIPE_1**

enumerator **RECIPE_2**

enumerator **RECIPE_3**

enumerator **RECIPE_4**

enumerator **RECIPE_5**

enumerator **RECIPE_6**

enumerator **RECIPE_7**

enumerator **RECIPE_8**

enumerator **RECIPE_9**

enumerator **RECIPE_10**

enumerator **RECIPE_11**

enumerator **RECIPE_12**

enumerator **RECIPE_13**

enumerator **RECIPE_14**

enumerator **RECIPE_15**

enumerator **RECIPE_16**

enumerator **RECIPE_17**

enumerator **RECIPE_18**

enumerator **RECIPE_19**

enumerator **RECIPE_20**

enumerator **RECIPE_21**

enumerator **RECIPE_22**

enumerator **RECIPE_23**

Public Functions

inline **RobotCommand**()

Public Members

Type **type_** = NO_CMD

std::uint8_t **recipe_id_**

std::int32_t **async_**

std::int32_t **friction_comp_**

std::int32_t **include_rotors_inertia_**

std::int32_t **ft_rtde_input_enable_**

std::int32_t **reg_int_val_**

double **reg_double_val_**

std::vector<double> **val_**

std::vector<int> **selection_vector_**

std::vector<int> **free_axes_**

std::int32_t **force_mode_type_**

std::uint8_t **std_digital_out_**

std::uint8_t **std_digital_out_mask_**

std::uint8_t **configurable_digital_out_**

std::uint8_t **configurable_digital_out_mask_**

std::uint8_t **std_tool_out_**

std::uint8_t **std_tool_out_mask_**

std::uint8_t **std_analog_output_mask_**

std::uint8_t **std_analog_output_type_**

double **std_analog_output_0_**

double **std_analog_output_1_**

std::int32_t **speed_slider_mask_**

double **speed_slider_fraction_**

std::uint32_t **steps_**

Script Client API

class **ScriptClient**

Public Types

enum class **ConnectionState** : std::uint8_t

Values:

enumerator **DISCONNECTED**

enumerator **CONNECTED**

Public Functions

explicit **ScriptClient**(std::string hostname, uint32_t major_control_version, uint32_t minor_control_version, int port = 30003, bool verbose = *false*)

virtual **~ScriptClient**()

void **connect**()

void **disconnect**()

bool **isConnected**()

void **setScriptFile**(*const* std::string &file_name)

Assign a custom script file that will be sent to device if the sendScript() function is called.

Setting an empty file_name will disable the custom script loading This eases debugging when modifying the control script because it does not require to recompile the whole library

bool **sendScript**()

Send the internal control script that is compiled into the library or the assigned control script file.

bool **sendScript**(*const* std::string &file_name)

Send the script file with the given file_name.

bool **sendScriptCommand**(*const* std::string &cmd_str)

void **setScriptInjection**(*const* std::string &search_string, *const* std::string &inject_string)

std::string **getScript**()

Get the corrected rtde_control script as a std::string.

Dashboard Client API

class **DashboardClient**

This class provides the interface for access to the UR dashboard server.

Public Types

enum class **ConnectionState** : std::uint8_t

Values:

enumerator **DISCONNECTED**

enumerator **CONNECTED**

Public Functions

explicit **DashboardClient**(std::string hostname, int port = 29999, bool verbose = *false*)

virtual **~DashboardClient**()

void **connect**(uint32_t timeout_ms = 2000)

Connects to the dashboard server with the given timeout value.

bool **isConnected**()

Returns true if the dashboard client is connected to the server.

void **disconnect**()

void **send**(*const* std::string &str)

std::string **receive**()

void **loadURP**(*const* std::string &urp_name)

Returns when both program and associated installation has loaded.

The load command fails if the associated installation requires confirmation of safety. In this case an exception with an error message is thrown.

void **play**()

Throws exception if program fails to start.

void **stop**()

Throws exception if the program fails to stop.

void **pause**()

Throws exception if the program fails to pause.

void **quit**()

Closes connection.

void **shutdown**()

Shuts down and turns off robot and controller.

bool **running**()

Execution state enquiry.

Returns: Returns true if program is running.

void **popup**(*const* std::string &message)

The popup-text will be translated to the selected language, if the text exists in the language file.

void **closePopup**()

Closes the popup.

void **closeSafetyPopup**()

Closes a safety popup.

void **powerOn**()

Powers on the robot arm.

void **powerOff**()

Powers off the robot arm.

void **brakeRelease**()

Powers off the robot arm.

void **unlockProtectiveStop**()

Closes the current popup and unlocks protective stop.

The unlock protective stop command fails with an exception if less than 5 seconds has passed since the protective stop occurred.

void **restartSafety**()

Use this when robot gets a safety fault or violation to restart the safety.

After safety has been rebooted the robot will be in Power Off.

Attention

> You should always ensure it is okay to restart the system. It is highly recommended to check the error log before using this command (either via PolyScope or e.g. ssh connection).

std::string **polyscopeVersion**()

std::string **programState**()

std::string **robotmode**()

std::string **getRobotModel**()

std::string **getLoadedProgram**()

std::string **safetymode**()

Safety mode inquiry.

A Safeguard Stop resulting from any type of safeguard I/O or a configurable I/O three position enabling device result in SAFEGUARD_STOP. This function is deprecated. Instead, use safetystatus().

std::string **safetystatus**()

Safety status inquiry.

This differs from 'safetymode' by specifying if a given Safeguard Stop was caused by the permanent safeguard I/O stop, a configurable I/O automatic mode safeguard stop

or a configurable I/O three position enabling device stop. Thus, this is strictly more detailed than safetymode()..

void **addToLog**(*const* std::string &message)

Adds log-message to the Log history.

bool **isProgramSaved**()

Returns the save state of the active program.

bool **isInRemoteControl**()

Returns the remote control status of the robot.

If the robot is in remote control it returns false and if remote control is disabled or robot is in local control it returns false.

void **setUserRole**(*const* UserRole &role)

std::string **getSerialNumber**()

Returns serial number of the robot.

(Serial number like “20175599999”)

Returns: serial number as a std::string

Robotiq Gripper API

class **RobotiqGripper**

C++ driver for Robot IQ grippers Communicates with the gripper directly, via socket with string commands, leveraging string names for variables.

- WRITE VARIABLES (CAN ALSO READ):
  - ACT = ‘ACT’ # act : activate (1 while activated, can be reset to clear fault status)
  - GTO = ‘GTO’ # gto : go to (will perform go to with the actions set in pos, for, spe)
  - ATR = ‘ATR’ # atr : auto-release (emergency slow move)
  - ARD = ‘ARD’ # ard : auto-release direction (open(1) or close(0) during auto-release)
  - FOR = ‘FOR’ # for : force (0-255)
  - SPE = ‘SPE’ # spe : speed (0-255)
  - POS = ‘POS’ # pos : position (0-255), 0 = open
- READ VARIABLES
  - STA = ‘STA’ # status (0 = is reset, 1 = activating, 3 = active)
  - PRE = ‘PRE’ # position request (echo of last commanded position)
  - OBJ = ‘OBJ’ # object detection (0 = moving, 1 = outer grip, 2 = inner grip, 3 = no object at rest)

- FLT = ‘FLT’ # fault (0=ok, see manual for errors if not zero)

Public Types

enum **eStatus**

Gripper status reported by the gripper.

The integer values have to match what the gripper sends.

Values:

enumerator **RESET**

RESET.

enumerator **ACTIVATING**

ACTIVATING.

enumerator **ACTIVE**

ACTIVE.

enum **eObjectStatus**

Object status reported by the gripper.

The integer values have to match what the gripper sends.

Values:

enumerator **MOVING**

gripper is opening or closing

enumerator **STOPPED_OUTER_OBJECT**

outer object detected while opening the gripper

enumerator **STOPPED_INNER_OBJECT**

inner object detected while closing the gripper

enumerator **AT_DEST**

requested target position reached - no object detected

enum class **ConnectionState** : std::uint8_t

Connection status.

Values:

enumerator **DISCONNECTED**

enumerator **CONNECTED**

enum **eMoveMode**

For synchronous or asynchronous moves.

Values:

enumerator **START_MOVE**

returns immediately after move started

enumerator **WAIT_FINISHED**

returns if the move finished, that means if an object is gripped or if requested target position is reached

enum **eUnit**

Unit identifiers for the configuration of the units for position, speed and flow.

Values:

enumerator **UNIT_DEVICE**

device unit in the range 0 - 255 - 255 meansd fully closed and 0 means fully open

enumerator **UNIT_NORMALIZED**

normalized value in the range 0.0 - 1.0, for position 0.0 means fully closed and 1.0 means fully open

enumerator **UNIT_PERCENT**

percent in the range from 0 - 100 % for position 0% means fully closed and 100% means fully open

enumerator **UNIT_MM**

position value in mm - only for position values

enum **eMoveParameter**

Identifier for move parameters.

Values:

enumerator **POSITION**

enumerator **SPEED**

enumerator **FORCE**

enum **ePostionId**

Position identifiers.

Values:

enumerator **OPEN**

enumerator **CLOSE**

enum **eFaultCode**

Fault code set in FLT.

Values:

enumerator **NO_FAULT**

No fault (solid blue LED)

enumerator **FAULT_ACTION_DELAYED**

Action delayed; the activation (re-activation) must be completed prior to perform the action.

enumerator **FAULT_ACTIVATION_BIT**

The activation bit must be set prior to performing the action.

enumerator **FAULT_TEMPERATURE**

Maximum operating temperature exceeded (≥ 85 °C internally); let cool down (below 80 °C).

enumerator **FAULT_COMM**

No communication during at least 1 second.

enumerator **FAULT_UNDER_VOLTAGE**

Under minimum operating voltage.

enumerator **FAULT_EMCY_RELEASE_ACTIVE**

Automatic release in progress.

enumerator **FAULT_INTERNAL**

Internal fault, contact support@robotiq.com.

enumerator **FAULT_ACTIVATION**

Activation fault, verify that no interference or other error occurred.

enumerator **FAULT_OVERCURRENT**

Overcurrent triggered.

enumerator **FAULT_EMCY_RELEASE_FINISHED**

Automatic release completed.

Public Functions

RobotiqGripper(*const* std::string &Hostname, int Port = 63352, bool verbose = *false*)

Constructor - creates a RobotiqGripper object with the given Hostname/IP and Port.

Parameters:
- **Hostname** – The hostname or ip address to use for connection
- **Port** – The port to use for connection
- **verbose** – Prints additional debug information if true

void **connect**(uint32_t timeout_ms = 2000)

Connects to the gripper server with the given millisecond timeout.

void **disconnect**()

Disconnects from the gripper server.

bool **isConnected**() *const*

Returns true if connected.

void **activate**(bool auto_calibrate = *false*)

Resets the activation flag in the gripper, and sets it back to one, clearing previous fault flags.

This is required after an emergency stop or if you just powered on your robot.

Parameters: **auto_calibrate** – Whether to calibrate the minimum and maximum positions based on actual motion.

void **autoCalibrate**(float Speed = -1.0)

Attempts to calibrate the open and closed positions, by closing and opening the gripper.

The function returns if calibration has been finished.

Parameters: **Speed – [in]** Optional speed parameter. If the speed parameter is less than 0, then is ignored and the calibration move is executed with default calibration speed (normally 1/4 of max speed).

bool **isActive**()

Returns whether the gripper is active.

If the function returns false, you need to activate the gripper via the activate() function.

See also

activate()

float **getOpenPosition**() *const*

Returns the fully open position in the configured position unit.

If you work with UNIT_DEVICE, then the open position is the position value near 0. If you work with any other unit, then the open position is the bigger value (that means 1.0 for UNIT_NORMALIZED, 100 for UNIT_PERCENT or the opening in mm for UNIT_MM).

float **getClosedPosition**() *const*

Returns what is considered the closed position for gripper in the configured position unit.

If you work with UNIT_DEVICE, then the closed position is the position value near 255. If you work with any other unit, then the closed position is always 0. That means the position value defines the opening of the gripper.

float **getCurrentPosition**()

Returns the current position as returned by the physical hardware in the configured position unit.

bool **isOpen**()

Returns whether the current position is considered as being fully open.

bool **isClosed**()

Returns whether the current position is considered as being fully closed.

int **move**(float Position, float Speed = -1.0, float Force = -1.0, eMoveMode MoveMode = START_MOVE)

Sends command to start moving towards the given position, with the specified speed

and force.

If the Speed and Force parameters are -1.0, then they are ignored and the pre configured speed and force parameters set via setSpeed() and setForce() function are used. So this gives you the option to pass in the speed parameter each time or to use preset speed and force parameters.

Parameters:
- **Position** – Position to move to [getClosedPosition() to getOpenPosition()]
- **Speed** – Speed to move at [min_speed, max_speed]
- **Force** – Force to use [min_force, max_force]
- **MoveMode** – START_MOVE - starts the move and returns immediately WAIT_FINISHED - waits until the move has finished

Returns: : Returns the object detection status.

int **open**(float Speed = -1.0, float Force = -1.0, eMoveMode MoveMode = START_MOVE)

Moves the gripper to its fully open position.

See also

See move() function for a detailed description of all other parameters

int **close**(float Speed = -1.0, float Force = -1.0, eMoveMode MoveMode = START_MOVE)

Moves the gripper to its fully closed position.

See also

See move() function for a detailed description of all other parameters

void **emergencyRelease**(ePostionId Direction, eMoveMode MoveMode = WAIT_FINISHED)

The emergency release is meant to disengage the gripper after an emergency stop of the robot.

The emergency open is not intended to be used under normal operating conditions.

Parameters:
- **Direction** – OPEN - moves to fully open position, CLOSE - moves to
- **MoveMode** – WAIT_FINISHED - waits until emergency release has finished START_MOVE - returns as soon as the emergency release has started Use faultStatus() function o check when emergency release has finished. (faultStatus() == FAULT_EMCY_RELEASE_FINISHED) fully close position

int **faultStatus**()

Returns the current fault status code.

See also

eFaultCode

void **setUnit**(eMoveParameter Param, eUnit Unit)

Set the units to use for passing position, speed and force values to the gripper functions and for reading back values like current position.

The default unit for position is UNIT_NORM (0.0 - 1.0). That means gripper closed is 0.0 and gripper fully open is 1.0 The default unit for speed and force is also UNIT_NORM (0.0 - 1.0). That means 1.0 means maximum force and maximum speed and 0.5 means half speed and half force.

See also

eUnit

void **setPositionRange_mm**(int Range)

Configure the position range of your gripper.

If you would like to use the unit UNIT_MM for position values, then you need to properly configure the position range of your gripper for proper value conversion.

Parameters: **Range – [in]** The position range in mm from the device position 0 to the device position 255

float **setSpeed**(float Speed)

Sets the speed to use for future move commands in the configured speed unit.

See also

move()

Returns: Returns the adjusted speed value.

float **setForce**(float Force)

Sets the force for future move commands in the configured force unit.

See also

move()

Returns: Returns the adjusted force value

eObjectStatus **objectDetectionStatus**()

Returns the current object detection status if a move is active.

Use this function for polling the state.

eObjectStatus **waitForMotionComplete**()

Call this function after a move command to wait for completion of the commanded move.

bool **setVars**(*const* std::vector<std::pair<std::string, int>> Vars)

Sends the appropriate command via socket to set the value of n variables, and waits for its 'ack' response.

Parameters: **Vars** – Dictionary of variables to set (variable_name, value).
Returns: : True on successful reception of ack, false if no ack was received, indicating the set may not have been effective.

bool **setVar**(*const* std::string &Var, int Value)

Sends the appropriate command via socket to set the value of a variable, and waits for its 'ack' response.

Parameters:
- **Var** – Variable to set.
- **Value** – Value to set for the variable.

Returns: : True on successful reception of ack, false if no ack was received, indicating the set may not have been effective.

int **getVar**(*const* std::string &var)

Sends the appropriate command to retrieve the value of a variable from the gripper, blocking until the response is received or the socket times out.

Parameters: **var** – Name of the variable to retrieve.
Returns: : Value of the variable as integer.

std::vector<int> **getVars**(*const* std::vector<std::string> &Vars)

This function enables the reading of a number of variables into a vector of values:

```
std::vector<std::string> Vars{"STA", "OBJ", "ACT", "POS"};
auto Result = Gripper.getVars(Vars);
for (int i = 0; i < Vars.size(); ++i)
{
   std::cout << Vars[i] << ": " << Result[i] << std::endl;
}
```

void **getNativePositionRange**(int &MinPosition, int &MaxPosition)

Returns the native positions range in device units.

The native position range is properly initialized after an auto calibration.

Parameters:
- **MinPosition – [out]** Returns the detected minimum position
- **MaxPosition – [out]** Returns the detected maximum position

void **setNativePositionRange**(int MinPosition, int MaxPosition)

Sets the native position range in device units.

Normally the native position range is properly initialized after an auto calibration. If you would like to avoid a calibration move and if you have previously determined or calculated the naticve position range then you can set it via this function.

Parameters:
- **MinPosition – [in]** Returns the detected minimum position
- **MaxPosition – [in]** Returns the detected maximum position