



UNIVERSAL ROBOTS

The URScript Programming Language

Version 3.6.0
May 28, 2018

The information contained herein is the property of Universal Robots A/S and shall not be reproduced in whole or in part without prior written approval of Universal Robots A/S. The information herein is subject to change without notice and should not be construed as a commitment by Universal Robots A/S. This manual is periodically reviewed and revised.

Universal Robots A/S assumes no responsibility for any errors or omissions in this document.

Copyright © 2009–2018 by Universal Robots A/S

The Universal Robots logo is a registered trademark of Universal Robots A/S.

Contents

Contents	2
1 The URScript Programming Language	3
1.1 Introduction	3
1.2 Connecting to URControl	3
1.3 Numbers, Variables, and Types	4
1.4 Flow of Control	4
1.4.1 Special keywords	5
1.5 Function	5
1.6 Remote Procedure Call (RPC)	6
1.7 Scoping rules	6
1.8 Threads	8
1.8.1 Threads and scope	10
1.8.2 Thread scheduling	10
1.9 Program Label	11
2 Module motion	12
2.1 Functions	12
2.2 Variables	28
3 Module internals	29
3.1 Functions	29
3.2 Variables	40
4 Module urmath	40
4.1 Functions	40
4.2 Variables	55
5 Module interfaces	56
5.1 Functions	56
5.2 Variables	94

1 The URScript Programming Language

1.1 Introduction

The Universal Robot can be controlled at two levels:

- The PolyScope or the Graphical User Interface Level
- Script Level

At the **Script Level**, the **URScript** is the programming language that controls the robot. The **URScript** includes variables, types, and the flow control statements. There are also built-in variables and functions that monitor and control I/O and robot movements.

1.2 Connecting to URControl

URControl is the low-level robot controller running on the Mini-ITX PC in the Control Box. When the PC boots up, the URControl starts up as a daemon (i.e., a service) and the PolyScope or Graphical User Interface connects as a client using a local TCP/IP connection.

Programming a robot at the *Script Level* is done by writing a client application (running at another PC) and connecting to URControl using a TCP/IP socket.

- **hostname**: ur-xx (or the IP address found in the **About Dialog-Box** in PolyScope if the robot is not in DNS).
- **port**: 30002

When a connection has been established URScript programs or commands are sent in clear text on the socket. Each line is terminated by "\n". Note that the text can only consist of extended ASCII characters.

The following conditions must be met to ensure that the URControl correctly recognizes the script:

- The script must start from a function definition or a secondary function definition (either "def" or "sec" keywords) in the first column
- All other script lines must be indented by at least one white space
- The last line of script must be an "end" keyword starting in the first column

1.3 Numbers, Variables, and Types

In **URScript** arithmetic expression syntax is standard:

```
1+2-3
4*5/6
(1+2)*3/(4-5)
```

In boolean expressions, boolean operators are spelled out:

```
True or False and (1 == 2)
1 > 2 or 3 != 4 xor 5 < -6
not 42 >= 87 and 87 <= 42
```

Variable assignment is done using the equal sign =:

```
foo = 42
bar = False or True and not False
baz = 87-13/3.1415
hello = "Hello, World!"
l = [1,2,4]
target = p[0.4,0.4,0.0,0.0,3.14159,0.0]
```

The fundamental type of a variable is deduced from the first assignment of the variable. In the example above `foo` is an `int` and `bar` is a `bool`. `target` is a `pose`: a combination of a position and orientation.

The fundamental types are:

- `none`
- `bool`
- `number` - either `int` or `float`
- `pose`
- `string`

A `pose` is given as `p[x, y, z, ax, ay, az]`, where `x, y, z` is the position of the TCP, and `ax, ay, az` is the orientation of the TCP, given in axis-angle notation.

1.4 Flow of Control

The flow of control of a program is changed by `if`-statements:

```
if a > 3:
    a = a + 1
elif b < 7:
    b = b * a
else:
    a = a + b
```

```
end

and while-loops:

l = [1,2,3,4,5]
i = 0
while i < 5:
    l[i] = l[i]*2
    i = i + 1
end
```

You can use `break` to stop a loop prematurely and `continue` to pass control to the next iteration of the nearest enclosing loop.

1.4.1 Special keywords

- `halt` terminates the program
- `return` returns from a function

1.5 Function

A function is declared as follows:

```
def add(a, b):
    return a+b
end
```

The function can then be called like this:

```
result = add(1, 4)
```

It is also possible to give function arguments default values:

```
def add(a=0,b=0):
    return a+b
end
```

Arguments can only be passed by value (including arrays). This means that any modification done to the content of the argument within the scope of the function will not be reflected outside that scope.

```
def myProg()

    a = [50,100]
    fun(a)

    def fun(p1):
        p1[0] = 25
        assert(p1[0] == 25)
```

```

    ...
end

    assert(a[0] == 50)
    ...
end

```

URScript also supports named parameters.

1.6 Remote Procedure Call (RPC)

Remote Procedure Calls (RPC) are similar to normal function calls, except that the function is defined and executed remotely. On the remote site, the **RPC** function being called must exist with the same number of parameters and corresponding types (together the function's signature). If the function is not defined remotely, it stops program execution. The controller uses the XMLRPC standard to send the parameters to the remote site and retrieve the result(s). During an **RPC** call, the controller waits for the remote function to complete. The XMLRPC standard is among others supported by C++ (xmlrpc-c library), Python and Java.

Creating a URScript program to initialize a camera, take a snapshot and retrieve a new target pose:

```

camera = rpc_factory("xmlrpc", "http://127.0.0.1/RPC2")
if (! camera.initialize("RGB")):
    popup("Camera was not initialized")
camera.takeSnapshot()
target = camera.getTarget()
...

```

First the `rpc_factory` (see `Interfaces` section) creates an XMLRPC connection to the specified **remote** server. The `camera` variable is the handle for the remote function calls. You must initialize the camera and therefore call `camera.initialize("RGB")`. The function returns a boolean value to indicate if the request was successful. In order to find a target position, the camera first takes a picture, hence the `camera.takeSnapshot()` call. Once the snapshot is taken, the image analysis in the remote site calculates the location of the target. Then the program asks for the exact target location with the function call `target = camera.getTarget()`. On return the `target` variable is assigned the result. The `camera.initialize("RGB")`, `takeSnapshot()` and `getTarget()` functions are the responsibility of the RPC server.

The `Technical support website` contains more examples of XMLRPC servers.

1.7 Scoping rules

A URScript program is declared as a function without parameters:

```
def myProg():
```

end

Every variable declared inside a program has a scope. The scope is the textual region where the variable is directly accessible. Two qualifiers are available to modify this visibility:

- `local` qualifier tells the controller to treat a variable inside a function, as being truly local, even if a global variable with the same name exists.
- `global` qualifier forces a variable declared inside a function, to be globally accessible.

For each variable the controller determines the scope binding, i.e. whether the variable is global or local. In case no `local` or `global` qualifier is specified (also called a free variable), the controller will first try to find the variable in the globals and otherwise the variable will be treated as local.

In the following example, the first `a` is a global variable and the second `a` is a local variable. Both variables are independent, even though they have the same name:

```
def myProg():  
  
    global a = 0  
  
    def myFun():  
        local a = 1  
        ...  
    end  
    ...  
end
```

Beware that the global variable is no longer accessible from within the function, as the local variable masks the global variable of the same name.

In the following example, the first `a` is a global variable, so the variable inside the function is the same variable declared in the program:

```
def myProg():  
  
    global a = 0  
  
    def myFun():  
        a = 1  
        ...  
    end  
    ...  
end
```

For each nested function the same scope binding rules hold. In the following example, the first `a` is global defined, the second local and the third implicitly global again:

```
def myProg():  
  
    global a = 0  
  
    def myFun():  
        local a = 1  
  
        def myFun2():  
            a = 2  
            ...  
        end  
        ...  
    end  
    ...  
end
```

The first and third `a` are one and the same, the second `a` is independent.

Variables on the first scope level (first indentation) are treated as global, even if the `global` qualifier is missing or the `local` qualifier is used:

```
def myProg():  
  
    a = 0  
  
    def myFun():  
        a = 1  
        ...  
    end  
    ...  
end
```

The variables `a` are one and the same.

1.8 Threads

Threads are supported by a number of special commands.

To declare a new thread a syntax similar to the declaration of functions are used:

```
thread myThread():  
    # Do some stuff  
    return False  
end
```

A couple of things should be noted. First of all, a thread cannot take any parameters, and so the parentheses in the declaration must be empty. Second, although a return statement is allowed in the thread, the value returned is discarded, and cannot be

accessed from outside the thread. A thread can contain other threads, the same way a function can contain other functions. Threads can in other words be nested, allowing for a thread hierarchy to be formed.

To run a thread use the following syntax:

```
thread myThread():
  # Do some stuff
  return False
end

thrd = run myThread()
```

The value returned by the `run` command is a handle to the running thread. This handle can be used to interact with a running thread. The `run` command spawns from the new thread, and then executes the instruction following the `run` instruction.

A thread can only wait for a running thread spawned by itself. To wait for a running thread to finish, use the `join` command:

```
thread myThread():
  # Do some stuff
  return False
end

thrd = run myThread()

join thrd
```

This halts the calling threads execution, until the specified thread finishes its execution. If the thread is already finished, the statement has no effect.

To kill a running thread, use the `kill` command:

```
thread myThread():
  # Do some stuff
  return False
end

thrd = run myThread()

kill thrd
```

After the call to `kill`, the thread is stopped, and the thread handle is no longer valid. If the thread has children, these are killed as well.

To protect against race conditions and other thread-related issues, support for critical sections is provided. A critical section ensures the enclosed code can finish running before another thread can start running. The previous statement is always true, unless a time-demanding command is present within the scope of the critical section. In such a case, another thread will be allowed to run. Time-demanding commands include `sleep`, `sync`, `move`-commands, and `socketRead`. Therefore, it is important to keep the critical section as short as possible. The syntax is as follows:

```
thread myThread():
  enter_critical
  # Do some stuff
  exit_critical
  return False
end
```

1.8.1 Threads and scope

The scoping rules for threads are exactly the same, as those used for functions. See 1.7 for a discussion of these rules.

1.8.2 Thread scheduling

Because the primary purpose of the URScript scripting language is to control the robot, the scheduling policy is largely based upon the realtime demands of this task.

The robot must be controlled a frequency of 125 Hz, or in other words, it must be told what to do every 0.008 second (each 0.008 second period is called a frame). To achieve this, each thread is given a “physical” (or robot) time slice of 0.008 seconds to use, and all threads in a runnable state is then scheduled in a round robin¹ fashion.

Each time a thread is scheduled, it can use a piece of its time slice (by executing instructions that control the robot), or it can execute instructions that do not control the robot, and therefore do not use any “physical” time. If a thread uses up its entire time slice, it is placed in a non-runnable state, and is not allowed to run until the next frame starts. If a thread does not use its time slice within a frame, it is expected to switch to a non-runnable state before the end of the frame². The reason for this state switching can be a join instruction or simply because the thread terminates.

It should be noted that even though the `sleep` instruction does not control the robot, it still uses “physical” time. The same is true for the `sync` instruction.

¹Before the start of each frame the threads are sorted, such that the thread with the largest remaining time slice is to be scheduled first.

²If this expectation is not met, the program is stopped.

1.9 Program Label

Program label code lines, with an "\$" as first symbol, are special lines in programs generated by PolyScope that make it possible to track the execution of a program.

```
$ 2 "var_1= True "  
global var_1= True
```

2 Module motion

2.1 Functions

conveyor_pulse_decode(*type*, *A*, *B*)

Tells the robot controller to treat digital inputs number *A* and *B* as pulses for a conveyor encoder. Only digital input 0, 1, 2 or 3 can be used.

```
>>> conveyor_pulse_decode(1, 0, 1)
```

This example shows how to set up quadrature pulse decoding with input *A* = digital_in(0) and input *B* = digital_in(1)

```
>>> conveyor_pulse_decode(2, 3)
```

This example shows how to set up rising and falling edge pulse decoding with input *A* = digital_in(3). Note that you do not have to set parameter *B* (as it is not used anyway).

Parameters

type: An integer determining how to treat the inputs on *A* and *B*

0 is no encoder, pulse decoding is disabled.

1 is quadrature encoder, input *A* and *B* must be square waves with 90 degree offset. Direction of the conveyor can be determined.

2 is rising and falling edge on single input (*A*).

3 is rising edge on single input (*A*).

4 is falling edge on single input (*A*).

The controller can decode inputs at up to 40kHz

A: Encoder input *A*, values of 0-3 are the digital inputs 0-3.

B: Encoder input *B*, values of 0-3 are the digital inputs 0-3.

Example command: `conveyor_pulse_decode(1, 2, 3)`

- Example Parameters:
 - *type* = 1 → is quadrature encoder, input *A* and *B* must be square waves with 90 degree offset. Direction of the conveyor can be determined.
 - *A* = 2 → Encoder output *A* is connected to digital input 2
 - *B* = 3 → Encoder output *B* is connected to digital input 3

end_force_mode()

Resets the robot mode from force mode to normal operation.

This is also done when a program stops.

end_freedrive_mode()

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

end_teach_mode()

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

force_mode(*task_frame*, *selection_vector*, *wrench*, *type*, *limits*)

Set robot to be controlled in force mode

Parameters

<code>task_frame</code> :	A pose vector that defines the force frame relative to the base frame.
<code>selection_vector</code> :	A 6d vector of 0s and 1s. 1 means that the robot will be compliant in the corresponding axis of the task frame.
<code>wrench</code> :	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. Actual wrench applied may be lower than requested due to joint safety limits. Actual forces and torques can be read using <code>get_tcp_force</code> function in a separate thread.
<code>type</code> :	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.
<code>limits</code> :	(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.

Note: Avoid movements parallel to compliant axes and high deceleration (consider inserting a short sleep command of at least 0.02s) just before entering force mode. Avoid high acceleration in force mode as this decreases the force control accuracy.

force_mode_set_damping(*damping*)

Sets the damping parameter in force mode.

Parameters

damping: Between 0 and 1, default value is 0.

A value of 1 is full damping, so the robot will decelerate 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. Add this to the beginning of your program to ensure it is called before force mode is entered (otherwise default value will be used).

freedrive_mode()

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.

get_conveyor_tick_count()

Tells the tick count of the encoder, note that the controller interpolates tick counts to get more accurate movements with low resolution encoders

Return Value

The conveyor encoder tick count

```
movec(pose_via, pose_to, a=1.2, v=0.25, r=0, mode=0)
```

Move Circular: Move to position (circular in tool-space)

TCP moves on the circular arc segment from current pose, through pose_via to pose_to. Accelerates to and moves with constant tool speed v. Use the mode parameter to define the orientation interpolation.

Parameters

- pose_via: path point (note: only position is used). Pose_via can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose.
- pose_to: target pose (note: only position is used in Fixed orientation mode). Pose_to can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose.
- a: tool acceleration (m/s²)
- v: tool speed (m/s)
- r: blend radius (of target pose) (m)
- mode: 0: Unconstrained mode. Interpolate orientation from current pose to target pose (pose_to)
1: Fixed mode. Keep orientation constant relative to the tangent of the circular arc (starting from current pose)

Example command: `movec(p[x,y,z,0,0,0], pose_to, a=1.2, v=0.25, r=0.05, mode=1)`

- Example Parameters:
 - Note: first position on circle is previous waypoint.
 - pose_via = p(x,y,z,0,0,0) → second position on circle.
 - * Note rotations are not used so they can be left as zeros.
 - * Note: This position can also be represented as joint angles (j0,j1,j2,j3,j4,j5) then forward kinematics is used to calculate the corresponding pose
 - pose_to → third (and final) position on circle
 - a = 1.2 → acceleration is 1.2 m/s/s
 - v = 0.25 → velocity is 250 mm/s
 - r = 0 → blend radius (at pose_to) is 50 mm.
 - mode = 1 → use fixed orientation relative to tangent of circular arc

movej($q, a=1.4, v=1.05, t=0, r=0$)

Move to position (linear in joint-space)

When using this command, the robot must be at a standstill or come from a movej or movel with a blend. The speed and acceleration parameters control the trapezoid speed profile of the move. Alternatively, the t parameter can be used to set the time for this move. Time setting has priority over speed and acceleration settings.

Parameters

- q : joint positions (q can also be specified as a pose, then inverse kinematics is used to calculate the corresponding joint positions)
- a : joint acceleration of leading axis (rad/s²)
- v : joint speed of leading axis (rad/s)
- t : time (S)
- r : blend radius (m)

If a blend radius is set, the robot arm trajectory will be modified to avoid the robot stopping at the point.

However, if the blend region of this move overlaps with the blend radius of previous or following waypoints, this move will be skipped, and an 'Overlapping Blends' warning message will be generated.

Example command: `movej([0, 1.57, -1.57, 3.14, -1.57, 1.57], a=1.4, v=1.05, t=0, r=0)`

- Example Parameters:
 - $q = (0, 1.57, -1.57, 3.14, -1.57, 1.57)$ → base is at 0 deg rotation, shoulder is at 90 deg rotation, elbow is at -90 deg rotation, wrist 1 is at 180 deg rotation, wrist 2 is at -90 deg rotation, wrist 3 is at 90 deg rotation. Note: joint positions (q can also be specified as a pose, then inverse kinematics is used to calculate the corresponding joint positions)
 - $a = 1.4$ → acceleration is 1.4 rad/s/s
 - $v = 1.05$ → velocity is 1.05 rad/s
 - $t = 0$ → the time (seconds) to make move is not specified. If it were specified the command would ignore the a and v values.
 - $r = 0$ → the blend radius is zero meters.

```
move1(pose, a=1.2, v=0.25, t=0, r=0)
```

Move to position (linear in tool-space)

See movej.

Parameters

pose: target pose (pose can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose)

a: tool acceleration (m/s²)

v: tool speed (m/s)

t: time (S)

r: blend radius (m)

Example command: `move1(pose, a=1.2, v=0.25, t=0, r=0)`

- Example Parameters:

- `pose = p(0.2,0.3,0.5,0,0,3.14)` → position in base frame of x = 200 mm, y = 300 mm, z = 500 mm, rx = 0, ry = 0, rz = 180 deg
- `a = 1.2` → acceleration of 1.2 m/s²
- `v = 0.25` → velocity of 250 mm/s
- `t = 0` → the time (seconds) to make the move is not specified. If it were specified the command would ignore the a and v values.
- `r = 0` → the blend radius is zero meters.

```
movep(pose, a=1.2, v=0.25, r=0)
```

Move Process

Blend circular (in tool-space) and move linear (in tool-space) to position. Accelerates to and moves with constant tool speed v .

Parameters

- `pose`: target pose (pose can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose)
- `a`: tool acceleration (m/s^2)
- `v`: tool speed (m/s)
- `r`: blend radius (m)

Example command: `movep(pose, a=1.2, v=0.25, r=0)`

- Example Parameters:
 - `pose = p(0.2,0.3,0.5,0,0,3.14)` → position in base frame of $x = 200 \text{ mm}$, $y = 300 \text{ mm}$, $z = 500 \text{ mm}$, $rx = 0$, $ry = 0$, $rz = 180 \text{ deg}$.
 - `a = 1.2` → acceleration of 1.2 m/s^2
 - `v = 0.25` → velocity of 250 mm/s
 - `r = 0` → the blend radius is zero meters.

position_deviation_warning(*enabled*, *threshold=0.8*)

When enabled, this function generates warning messages to the log when the robot deviates from the target position. This function can be called at any point in the execution of a program. It has no return value.

```
>>> position_deviation_warning(True)
```

In the above example, the function has been enabled. This means that log messages will be generated whenever a position deviation occurs. The optional "threshold" parameter can be used to specify the level of position deviation that triggers a log message.

Parameters

- enabled:** (Boolean) Enable or disable position deviation log messages.
- threshold:** (Float) Optional value in the range (0;1), where 0 is no position deviation and 1 is the maximum position deviation (equivalent to the amount of position deviation that causes a protective stop of the robot). If no threshold is specified by the user, a default value of 0.8 is used.

Example command: `position_deviation_warning(True, 0.8)`

- Example Parameters:
 - Enabled = True → Logging of warning is turned on
 - Threshold = 0.8 → 80% of deviation that causes a protective stop causes a warning to be logged in the log history file.

reset_revolution_counter(*qNear*=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0])

Reset the revolution counter, if no offset is specified. This is applied on joints which safety limits are set to "Unlimited" and are only applied when new safety settings are applied with limited joint angles.

```
>>> reset_revolution_counter()
```

Parameters

qNear: Optional parameter, reset the revolution counter to one close to the given *qNear* joint vector. If not defined, the joint's actual number of revolutions are used.

Example command: `reset_revolution_counter(qNear=[0.0, 0.0, 0.0, 0.0, 0.0, 0.0])`

- Example Parameters:
 - *qNear* = (0.0, 0.0, 0.0, 0.0, 0.0, 0.0) → Optional parameter, resets the revolution counter of wrist 3 to zero on UR3 robots to the nearest zero location to joint rotations represented by *qNear*.

servoc(*pose*, *a*=1.2, *v*=0.25, *r*=0)

Servo Circular

Servo to position (circular in tool-space). Accelerates to and moves with constant tool speed *v*.

Parameters

pose: target pose (pose can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose)

a: tool acceleration (m/s²)

v: tool speed (m/s)

r: blend radius (of target pose) (m)

Example command: `servoc(p[0.2,0.3,0.5,0,0,3.14], a=1.2, v=0.25, r=0)`

- Example Parameters:
 - *pose* = p(0.2,0.3,0.5,0,0,3.14) → position in base frame of x = 200 mm, y = 300 mm, z = 500 mm, rx = 0, ry = 0, rz = 180 deg.
 - *a* = 1.2 → acceleration of 1.2 m/s²
 - *v* = 0.25 → velocity of 250 mm/s
 - *r* = 0 → the blend radius at the target position is zero meters.

```
servoj(q, a, v, t=0.008, lookahead_time=0.1, gain=300)
```

Servo to position (linear in joint-space)

Servo function used for online control of the robot. The lookahead time and the gain can be used to smoothen or sharpen the trajectory.

Note: A high gain or a short lookahead time may cause instability. Preferred use is to call this function with a new setpoint (q) in each time step (thus the default t=0.008)

Parameters

q:	joint positions (rad)
a:	NOT used in current version
v:	NOT used in current version
t:	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) smoothenes the trajectory with this lookahead time
gain:	proportional gain for following target position, range (100,2000)

Example command: `servoj([0.0,1.57,-1.57,0,0,3.14], 0, 0, 0.1, 0.1, 300)`

- Example Parameters:

- q = (0.0,1.57,-1.57,0,0,3.14) → joint angles in radians representing rotations of base, shoulder, elbow, wrist1, wrist2 and wrist3
- a = 0 → not used in current version
- v = 0 → not used in current version
- t = .1 → time where the command is controlling the robot. The function is blocking for time t (S)
- lookahead time = .1 → time (S), range (0.03,0.2) smoothenes the trajectory with this lookahead time
- gain = 300 → proportional gain for following target position, range (100,2000)

set_conveyor_tick_count(*tick_count*, *absolute_encoder_resolution=0*)

Tells the robot controller the tick count of the encoder. This function is useful for absolute encoders, use `conveyor_pulse_decode()` for setting up an incremental encoder. For circular conveyors, the value must be between 0 and the number of ticks per revolution.

Parameters

<code>tick_count</code> :	Tick count of the conveyor (Integer)
<code>absolute_encoder_resolution</code> :	Resolution of the encoder, needed to handle wrapping nicely. (Integer)
	0 is a 32 bit signed encoder, range (-2147483648 ; 2147483647) (default)
	1 is a 8 bit unsigned encoder, range (0 ; 255)
	2 is a 16 bit unsigned encoder, range (0 ; 65535)
	3 is a 24 bit unsigned encoder, range (0 ; 16777215)
	4 is a 32 bit unsigned encoder, range (0 ; 4294967295)

Example command: `set_conveyor_tick_count(24543, 0)`

- Example Parameters:
 - `Tick_count = 24543` → a value read from e.g. a MODBUS register being updated by the absolute encoder
 - `Absolute_encoder_resolution = 0` → 0 is a 32 bit signed encoder, range (-2147483648 ; 2147483647) (default)

set_pos(*q*)

Set joint positions of simulated robot

Parameters

q: joint positions

Example command: `set_pos([0.0, 1.57, -1.57, 0, 0, 3.14])`

- Example Parameters:
 - $q = (0.0, 1.57, -1.57, 0, 0, 3.14)$ → the position of the simulated robot with joint angles in radians representing rotations of base, shoulder, elbow, wrist1, wrist2 and wrist3

set_safety_mode_transition_hardness(*type*)

Sets the transition hardness between normal mode, reduced mode and safeguard stop.

Parameters

type: An integer specifying transition hardness.

0 is hard transition between modes using maximum torque, similar to emergency stop.

1 is soft transition between modes.

speedj(*qd*, *a*, *t*)

Joint speed

Accelerate linearly in joint space and continue with constant joint speed. The time *t* is optional; if provided the function will return after time *t*, regardless of the target speed has been reached. If the time *t* is not provided, the function will return when the target speed is reached.

Parameters

qd: joint speeds (rad/s)

a: joint acceleration (rad/s²) (of leading axis)

t: time (s) before the function returns (optional)

Example command: `speedj([0.2, 0.3, 0.1, 0.05, 0, 0], 0.5, 0.5)`

- Example Parameters:
 - *qd* → Joint speeds of: base=0.2 rad/s, shoulder=0.3 rad/s, elbow=0.1 rad/s, wrist1=0.05 rad/s, wrist2 and wrist3=0 rad/s
 - *a* = 0.5 rad/s² → acceleration of the leading axis (shoulder in this case)
 - *t* = 0.5 s → time before the function returns

speedl(*xd, a, t, aRot=' a'*)

Tool speed

Accelerate linearly in Cartesian space and continue with constant tool speed. The time *t* is optional; if provided the function will return after time *t*, regardless of the target speed has been reached. If the time *t* is not provided, the function will return when the target speed is reached.

Parameters

- xd*: tool speed (m/s) (spatial vector)
- a*: tool position acceleration (m/s²)
- t*: time (s) before function returns (optional)
- aRot*: tool acceleration (rad/s²) (optional), if not defined *a*, position acceleration, is used

Example command: `speedl ([0.5, 0.4, 0.0, 0., 1.57, 0, 0], 0.5, 0.5)`

- Example Parameters:
 - *qd* → Tool speeds of: x=500 mm/s, y=400 mm/s, rx=90 deg/s, ry and rz=0 mm/s
 - *a* = 0.5 rad/s² → acceleration of the leading axis (shoulder is this case)
 - *t* = 0.5 s → time before the function returns

stop_conveyor_tracking(*a=15, aRot=' a'*)

Stop tracking the conveyor, started by `track_conveyor_linear()` or `track_conveyor_circular()`, and decelerate tool speed to zero.

Parameters

- a*: tool acceleration (m/s²) (optional)
- aRot*: tool acceleration (rad/s²) (optional), if not defined *a*, position acceleration, is used

Example command: `stop_conveyor_tracking(a=15)`

- Example Parameters:
 - *a* = 15 rad/s² → acceleration of the tool

stopj(a)

Stop (linear in joint space)

Decelerate joint speeds to zero

Parameters

a: joint acceleration (rad/s²) (of leading axis)

Example command: stopj(2)

- Example Parameters:
 - a = 2 rad/s² → rate of deceleration of the leading axis.

stopl(a, aRot='a')

Stop (linear in tool space)

Decelerate tool speed to zero

Parameters

a: tool acceleration (m/s²)

aRot: tool acceleration (rad/s²) (optional), if not defined
a, position acceleration, is used

Example command: stopl(20)

- Example Parameters:
 - a = 20 m/s² → rate of deceleration of the tool
 - aRot → tool deceleration (rad/s²) (optional), if not defined, position acceleration, is used. i.e. it supersedes the "a" deceleration.

teach_mode()

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.

```
track_conveyor_circular(center, ticks_per_revolution,  
rotate_tool='False')
```

Makes robot movement (movej() etc.) track a circular conveyor.

```
>>> track_conveyor_circular(p[0.5,0.5,0,0,0,0],500.0, false)
```

The example code makes the robot track a circular conveyor with center in p(0.5,0.5,0,0,0,0) of the robot base coordinate system, where 500 ticks on the encoder corresponds to one revolution of the circular conveyor around the center.

Parameters

center:	Pose vector that determines center of the conveyor in the base coordinate system of the robot.
ticks_per_revolution:	How many ticks the encoder sees when the conveyor moves one revolution.
rotate_tool:	Should the tool rotate with the conveyor or stay in the orientation specified by the trajectory (move() etc.).

Example command:

```
track_conveyor_circular(p[0.5,0.5,0,0,0,0], 500.0, false)
```

- Example Parameters:
 - center = p(0.5,0.5,0,0,0,0) → location of the center of the conveyor
 - ticks_per_revolution = 500 → the number of ticks the encoder sees when the conveyor moves one revolution
 - rotate_tool = false → the tool should not rotate with the conveyor, but stay in the orientation specified by the trajectory (move() etc.).

track_conveyor_linear(*direction*, *ticks_per_meter*)

Makes robot movement (movej() etc.) track a linear conveyor.

```
>>> track_conveyor_linear(p[1,0,0,0,0,0],1000.0)
```

The example code makes the robot track a conveyor in the x-axis of the robot base coordinate system, where 1000 ticks on the encoder corresponds to 1m along the x-axis.

Parameters

`direction`: Pose vector that determines the direction of the conveyor in the base coordinate system of the robot

`ticks_per_meter`: How many ticks the encoder sees when the conveyor moves one meter

Example command: `track_conveyor_linear(p[1,0,0,0,0,0],1000.0)`

- Example Parameters:
 - `direction = p(1,0,0,0,0,0)` → Pose vector that determines the direction of the conveyor in the base coordinate system of the robot
 - `ticks_per_meter = 1000.` → How many ticks the encoder sees when the conveyor moves one meter.

2.2 Variables

Name	Description
<code>__package__</code>	Value: 'Motion'
<code>a_joint_default</code>	Value: 1.4
<code>a_tool_default</code>	Value: 1.2
<code>v_joint_default</code>	Value: 1.05
<code>v_tool_default</code>	Value: 0.25

3 Module internals

3.1 Functions

force()

Returns the force exerted at the TCP

Return the current externally exerted force at the TCP. The force is the norm of Fx, Fy, and Fz calculated using `get_tcp_force()`.

Return Value

The force in Newtons (float)

get_actual_joint_positions()

Returns the actual angular positions of all joints

The angular actual positions are expressed in radians and returned as a vector of length 6. Note that the output might differ from the output of `get_target_joint_positions()`, especially during acceleration and heavy loads.

Return Value

The current actual joint angular position vector in rad : (Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3)

get_actual_joint_speeds()

Returns the actual angular velocities of all joints

The angular actual velocities are expressed in radians pr. second and returned as a vector of length 6. Note that the output might differ from the output of `get_target_joint_speeds()`, especially during acceleration and heavy loads.

Return Value

The current actual joint angular velocity vector in rad/s:
(Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3)

get_actual_tcp_pose()

Returns the current measured tool pose

Returns the 6d pose representing the tool position and orientation specified in the base frame. The calculation of this pose is based on the actual robot encoder readings.

Return Value

The current actual TCP vector (X, Y, Z, Rx, Ry, Rz)

get_actual_tcp_speed()

Returns the current measured TCP speed

The speed of the TCP returned in a pose structure. The first three values are the cartesian speeds along x,y,z, and the last three define the current rotation axis, rx,ry,rz, and the length |rz,ry,rz| defines the angular velocity in radians/s.

Return Value

The current actual TCP velocity vector (X, Y, Z, Rx, Ry, Rz)

get_actual_tool_flange_pose()

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.

Return Value

The current actual tool flange vector: (X, Y, Z, Rx, Ry, Rz)

Note: See `get_actual_tcp_pose` for the actual 6d pose including TCP offset.

get_controller_temp()

Returns the temperature of the control box

The temperature of the robot control box in degrees Celcius.

Return Value

A temperature in degrees Celcius (float)

```
get_forward_kin(q='current_joint_positions',  
tcp='active_tcp')
```

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.

Parameters

q: joint position vector (Optional)

tcp: tcp offset pose (Optional)

Return Value

tool pose

Example command: `get_forward_kin([0., 3.14, 1.57, .785, 0, 0],
p[0, 0, 0.01, 0, 0, 0])`

- Example Parameters:

- q = (0., 3.14, 1.57, .785, 0, 0) → joint angles of j0=0 deg, j1=180 deg, j2=90 deg, j3=45 deg, j4=0 deg, j5=0 deg.

- tcp = p(0,0,0.01,0,0,0) → tcp offset of x=0mm, y=0mm, z=10mm and rotation vector of rx=0 deg., ry=0 deg, rz=0 deg.

get_inverse_kin(*x*, *qnear*, *maxPositionError*=1e-10, *maxOrientationError*=1e-10, *tcp*='active_tcp')

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

<i>x</i> :	tool pose
<i>qnear</i> :	list of joint positions (Optional)
<i>maxPositionError</i> :	the maximum allowed position error (Optional)
<i>maxOrientationError</i> :	the maximum allowed orientation error (Optional)
<i>tcp</i> :	<i>tcp</i> offset pose (Optional)

Return Value

joint positions

Example command: `get_inverse_kin(p[.1, .2, .2, 0, 3.14, 0], [0., 3.14, 1.57, .785, 0, 0])`

- Example Parameters:
 - $x = p(.1, .2, .2, 0, 3.14, 0)$ → pose with position of $x=100\text{mm}$, $y=200\text{mm}$, $z=200\text{mm}$ and rotation vector of $r_x=0$ deg., $r_y=180$ deg, $r_z=0$ deg.
 - $qnear = (0., 3.14, 1.57, .785, 0, 0)$ → solution should be near to joint angles of $j_0=0$ deg, $j_1=180$ deg, $j_2=90$ deg, $j_3=45$ deg, $j_4=0$ deg, $j_5=0$ deg.
 - *maxPositionError* is by default $1e-10$ m
 - *maxOrientationError* is by default $1e-10$ rad

get_joint_temp(*j*)

Returns the temperature of joint *j*

The temperature of the joint house of joint *j*, counting from zero. $j=0$ is the base joint, and $j=5$ is the last joint before the tool flange.

Parameters

j: The joint number (int)

Return Value

A temperature in degrees Celcius (float)

get_joint_torques()

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.

Return Value

The joint torque vector in Nm: (Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3)

get_target_joint_positions()

Returns the desired angular position of all joints

The angular target positions are expressed in radians and returned as a vector of length 6. Note that the output might differ from the output of `get_actual_joint_positions()`, especially during acceleration and heavy loads.

Return Value

The current target joint angular position vector in rad: (Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3)

get_target_joint_speeds()

Returns the desired angular velocities of all joints

The angular target velocities are expressed in radians pr. second and returned as a vector of length 6. Note that the output might differ from the output of `get_actual_joint_speeds()`, especially during acceleration and heavy loads.

Return Value

The current target joint angular velocity vector in rad/s: (Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3)

get_target_tcp_pose()

Returns the current target tool pose

Returns the 6d pose representing the tool position and orientation specified in the base frame. The calculation of this pose is based on the current target joint positions.

Return Value

The current target TCP vector (X, Y, Z, Rx, Ry, Rz)

get_target_tcp_speed()

Returns the current target TCP speed

The desired speed of the TCP returned in a pose structure. The first three values are the cartesian speeds along x,y,z, and the last three define the current rotation axis, rx,ry,rz, and the length $|r_x,r_y,r_z|$ defines the angular velocity in radians/s.

Return Value

The TCP speed (pose)

get_tcp_force()

Returns the wrench (Force/Torque vector) at the TCP

The external wrench is computed based on the error between the joint torques required to stay on the trajectory and the expected joint torques. The function returns p(Fx (N), Fy(N), Fz(N), TRx (Nm), TRy (Nm), TRz (Nm)) where Fx, Fy, and Fz are the forces in the axes of the robot base coordinate system measured in Newtons, and TRx, TRy, and TRz are the torques around these axes measured in Newton times Meters.

The maximum force exerted along each axis is 300 Newtons.

Return Value

The wrench (pose)

get_tcp_offset()

Gets the active tcp offset, i.e. the transformation from the output flange coordinate system to the TCP as a pose.

Return Value

tcp offset pose

get_tool_accelerometer_reading()

Returns the current reading of the tool accelerometer as a three-dimensional vector.

The accelerometer axes are aligned with the tool coordinates, and pointing an axis upwards results in a positive reading.

Return Value

X, Y, and Z component of the measured acceleration in SI-units (m/s²).

get_tool_current()

Returns the tool current

The tool current consumption measured in ampere.

Return Value

The tool current in ampere.

is_steady()

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. It is useful in combination with the GUI's wait node, before starting the screwdriver or other actuators influencing the position of the robot.

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

Return Value

True when the robot is fully at rest. Returns False otherwise (bool)

is_within_safety_limits(pose)

Checks if the given pose 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

`pose`: Target pose (which can also be specified as joint positions)

Return Value

True if within limits, false otherwise (bool)

Example command:

```
is_within_safety_limits(p[.1, .2, .2, 0, 3.14, 0])
```

- Example Parameters:

- `pose = p(.1,.2,.2,0,3.14,0)` → target pose with position of `x=100mm, y=200mm, z=200mm` and rotation vector of `rx=0 deg., ry=180 deg, rz=0 deg.`

popup(*s*, *title*=' Popup' , *warning*=False, *error*=False, *blocking*=False)

Display popup on GUI

Display message in popup window on GUI.

Parameters

- s*: message string
- title*: title string
- warning*: warning message?
- error*: error message?
- blocking*: if True, program will be suspended until "continue" is pressed

Example command: `popup("here I am", title="Popup #1", blocking=True)`

- Example Parameters:
 - *s* popup text is "here I am"
 - *title* popup title is "Popup #1"
 - *blocking* = true → popup must be cleared before other actions will be performed.

powerdown()

Shutdown the robot, and power off the robot and controller.

set_gravity(*d*)

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.

```
>>> set_gravity([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

Parameters

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

Example command: `set_gravity(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/s² (1g).

set_payload(m, cog)

Set payload mass and center of gravity

Alternatively one could use `set_payload_mass` and `set_payload_cog`.

Sets the mass and center of gravity (abbr. cog) of the payload.

This function must be called, when the payload weight or weight distribution changes - i.e when the robot picks up or puts down a heavy workpiece.

The cog argument is optional - if not provided, the Tool Center Point (TCP) will be used as the Center of Gravity (cog). If the cog argument is omitted, later calls to `set_tcp(pose)` will change CoG to the new TCP.

The cog is specified as a vector, (CoGx, CoGy, CoGz), displacement, from the toolmount.

Parameters

`m`: mass in kilograms

`cog`: Center of Gravity: (CoGx, CoGy, CoGz) in meters.
Optional.

Example command: `set_payload(3., [0,0,.3])`

- Example Parameters:
 - `m = 3` → mass is set to 3 kg payload
 - `cog = (0,0,.3)` → Center of Gravity is set to x=0 mm, y=0 mm, z=300mm from the center of the tool mount in tool coordinates

set_payload_cog(CoG)

Set center of gravity

See also `set_payload`.

Sets center of gravity (abbr. CoG) of the payload.

This function must be called, when the weight distribution changes - i.e. when the robot picks up or puts down a heavy workpiece.

The CoG is specified as a vector, (CoGx, CoGy, CoGz), displacement, from the toolmount.

Parameters

CoG: Center of Gravity: (CoGx, CoGy, CoGz) in meters.

Example command: `set_payload_cog([0, 0, .3])`

- Example Parameters:
 - CoG = (0,0,.3) → Center of Gravity is set to x=0 mm, y=0 mm, z=300mm from the center of the tool mount in tool coordinates

set_payload_mass(m)

Set payload mass

See also `set_payload`.

Sets the mass of the payload.

This function must be called, when the payload weight changes - i.e. when the robot picks up or puts down a heavy workpiece.

Parameters

m: mass in kilograms

Example command: `set_payload_mass(3.)`

- Example Parameters:
 - m = 3 → mass is set to 3 kg payload

set_tcp(*pose*)

Sets the active tcp offset, i.e. the transformation from the output flange coordinate system to the TCP as a pose.

Parameters

pose: A pose describing the transformation.

Example command: `set_tcp(p[0.,.2,.3,0.,3.14,0.])`

- Example Parameters:
 - `pose = p(0.,.2,.3,0.,3.14,0.)` → tool center point is set to `x=0mm, y=200mm, z=300mm`, rotation vector is `rx=0 deg, ry=180 deg, rz=0 deg`. In tool coordinates

sleep(*t*)

Sleep for an amount of time

Parameters

t: time (s)

Example command: `sleep(3.)`

- Example Parameters:
 - `t = 3.` → time to sleep

sync()

Uses up the remaining "physical" time a thread has in the current frame.

textmsg(*s1*, *s2*=' ')

Send text message to log

Send message with *s1* and *s2* concatenated to be shown on the GUI log-tab

Parameters

s1: message string, variables of other types (int, bool poses etc.) can also be sent

s2: message string, variables of other types (int, bool poses etc.) can also be sent

Example command: `textmsg("value=", 3)`

- Example Parameters:
 - *s1* set first part of message to "value="
 - *s2* set second part of message to 3
 - * message in the log is "value=3"

3.2 Variables

Name	Description
<code>__package__</code>	Value: None

4 Module urmath

4.1 Functions

acos(f)

Returns the arc cosine of f

Returns the principal value of the arc cosine of f, expressed in radians. A runtime error is raised if f lies outside the range (-1, 1).

Parameters

f: floating point value

Return Value

the arc cosine of f.

Example command: `acos(0.707)`

- Example Parameters:
 - f is the cos of 45 deg. (.785 rad)
 - * Returns .785

asin(f)

Returns the arc sine of f

Returns the principal value of the arc sine of f, expressed in radians. A runtime error is raised if f lies outside the range (-1, 1).

Parameters

f: floating point value

Return Value

the arc sine of f.

Example command: `asin(0.707)`

- Example Parameters:
 - f is the sin of 45 deg. (.785 rad)
 - * Returns .785

atan(f)

Returns the arc tangent of f

Returns the principal value of the arc tangent of f, expressed in radians.

Parameters

f: floating point value

Return Value

the arc tangent of f.

Example command: `atan(1.)`

- Example Parameters:
 - f is the tan of 45 deg. (.785 rad)
 - * Returns .785

atan2(x, y)

Returns the arc tangent of x/y

Returns the principal value of the arc tangent of x/y, expressed in radians. To compute the value, the function uses the sign of both arguments to determine the quadrant.

Parameters

x: floating point value

y: floating point value

Return Value

the arc tangent of x/y.

Example command: `atan2(.5, .5)`

- Example Parameters:
 - x is the one side of the triangle
 - y is the second side of a triangle
 - * Returns $\text{atan}(.5/.5) = .785$

binary_list_to_integer(l)

Returns the value represented by the content of list l

Returns the integer value represented by the bools contained in the list l when evaluated as a signed binary number.

Parameters

- l: The list of bools to be converted to an integer. The bool at index 0 is evaluated as the least significant bit. False represents a zero and True represents a one. If the list is empty this function returns 0. If the list contains more than 32 bools, the function returns the signed integer value of the first 32 bools in the list.

Return Value

The integer value of the binary list content.

Example command:

```
binary_list_to_integer([True,False,False,True])
```

- Example Parameters:
 - l represents the binary values 1001
 - * Returns 9

ceil(f)

Returns the smallest integer value that is not less than f

Rounds floating point number to the smallest integer no greater than f.

Parameters

- f: floating point value

Return Value

rounded integer

Example command: `ceil(1.43)`

- Example Parameters:
 - Returns 2

cos(*f*)

Returns the cosine of *f*

Returns the cosine of an angle of *f* radians.

Parameters

f: floating point value

Return Value

the cosine of *f*.

Example command: `cos(1.57)`

- Example Parameters:
 - *f* is angle of 1.57 rad (90 deg)
 - * Returns 0.0

d2r(*d*)

Returns degrees-to-radians of *d*

Returns the radian value of '*d*' degrees. Actually: $(d/180)*\text{MATH_PI}$

Parameters

d: The angle in degrees

Return Value

The angle in radians

Example command: `d2r(90)`

- Example Parameters:
 - *d* angle in degrees
 - * Returns 1.57 angle in radians

floor(*f*)

Returns largest integer not greater than *f*

Rounds floating point number to the largest integer no greater than *f*.

Parameters

f: floating point value

Return Value

rounded integer

Example command: `floor(1.53)`

- Example Parameters:
 - Returns 1

get_list_length(v)

Returns the length of a list variable

The length of a list is the number of entries the list is composed of.

Parameters

v: A list variable

Return Value

An integer specifying the length of the given list

Example command: `get_list_length([1, 3, 3, 6, 2])`

- Example Parameters:
 - v is the list 1,3,3,6,2
 - * Returns 5

integer_to_binary_list(x)

Returns the binary representation of x

Returns a list of bools as the binary representation of the signed integer value x.

Parameters

x: The integer value to be converted to a binary list.

Return Value

A list of 32 bools, where False represents a zero and True represents a one. The bool at index 0 is the least significant bit.

Example command: `integer_to_binary_list(57)`

- Example Parameters:
 - x integer 57
 - * Returns binary list

interpolate_pose(*p_from*, *p_to*, *alpha*)

Linear interpolation of tool position and orientation.

When alpha is 0, returns *p_from*. When alpha is 1, returns *p_to*. As alpha goes from 0 to 1, returns a pose going in a straight line (and geodesic orientation change) from *p_from* to *p_to*. If alpha is less than 0, returns a point before *p_from* on the line. If alpha is greater than 1, returns a pose after *p_to* on the line.

Parameters

- p_from*: tool pose (pose)
- p_to*: tool pose (pose)
- alpha*: Floating point number

Return Value

interpolated pose (pose)

Example command: `interpolate_pose(p[.2, .2, .4, 0, 0, 0], p[.2, .2, .6, 0, 0, 0], .5)`

- Example Parameters:
 - *p_from* = p(.2,.2,.4,0,0,0)
 - *p_to* = p(.2,.2,.6,0,0,0)
 - *alpha* = .5
 - * Returns p(.2,.2,.5,0,0,0)

length(*v*)

Returns the length of a list variable or a string

The length of a list or string is the number of entries or characters it is composed of.

Parameters

- v*: A list or string variable

Return Value

An integer specifying the length of the given list or string

Example command: `length("here I am")`

- Example Parameters:
 - *v* equals string "here I am"
 - * Returns 9

log(b, f)

Returns the logarithm of f to the base b

Returns the logarithm of f to the base b. If b or f is negative, or if b is 1 a runtime error is raised.

Parameters

b: floating point value

f: floating point value

Return Value

the logarithm of f to the base of b.

Example command: `log(10., 4.)`

- Example Parameters:
 - b is base 10
 - f is log of 4
 - * Returns 0.60206

norm(a)

Returns the norm of the argument

The argument can be one of four different types:

Pose: In this case the euclidian norm of the pose is returned.

Float: In this case `fabs(a)` is returned.

Int: In this case `abs(a)` is returned.

List: In this case the euclidian norm of the list is returned, the list elements must be numbers.

Parameters

a: Pose, float, int or List

Return Value

norm of a

Example command:

- `norm(-5.3)` → Returns 5.3
- `norm(-8)` → Returns 8
- `norm(p[-.2, .2, -.2, -1.57, 0, 3.14])` → Returns 3.52768

point_dist(p_from, p_to)

Point distance

Parameters

p_from: tool pose (pose)

p_to: tool pose (pose)

Return Value

Distance between the two tool positions (without considering rotations)

Example command: `point_dist(p[.2, .5, .1, 1.57, 0, 3.14], p[.2, .5, .6, 0, 1.57, 3.14])`

• Example Parameters:

- p_from = p(.2,.5,.1,1.57,0,3.14) → The first point

- p_to = p(.2,.5,.6,0,1.57,3.14) → The second point

* Returns distance between the points regardless of rotation

pose_add(p_1, p_2)

Pose addition

Both arguments contain three position parameters (x, y, z) jointly called P, and three rotation parameters (R_x, R_y, R_z) jointly called R. This function calculates the result x_3 as the addition of the given poses as follows:

$$p_3.P = p_1.P + p_2.P$$

$$p_3.R = p_1.R * p_2.R$$

Parameters

p_1: tool pose 1 (pose)

p_2: tool pose 2 (pose)

Return Value

Sum of position parts and product of rotation parts (pose)

Example command: `pose_add(p[.2, .5, .1, 1.57, 0, 0], p[.2, .5, .6, 1.57, 0, 0])`

• Example Parameters:

- p_1 = p(.2,.5,.1,1.57,0,0) → The first point

- p_2 = p(.2,.5,.6,1.57,0,0) → The second point

* Returns p(0.4,1.0,0.7,3.14,0,0)

pose_dist(p_from, p_to)

Pose distance

Parameters

p_from: tool pose (pose)

p_to: tool pose (pose)

Return Value

distance

Example command: `pose_dist(p[.2, .5, .1, 1.57, 0, 3.14],
p[.2, .5, .6, 0, 1.57, 3.14])`

- Example Parameters:
 - p_from = p(.2,.5,.1,1.57,0,3.14) → The first point
 - p_to = p(.2,.5,.6,0,1.57,3.14) → The second point
 - * Returns distance between two poses including rotation

pose_inv(p_from)

Get the inverse of a pose

Parameters

p_from: tool pose (spatial vector)

Return Value

inverse tool pose transformation (spatial vector)

Example command: `pose_inv(p[.2, .5, .1, 1.57, 0, 3.14])`

- Example Parameters:
 - p_from = p(.2,.5,.1,1.57,0,3.14) → The point
 - * Returns p(0.19324,0.41794,-0.29662,1.23993,0.0,2.47985)

pose_sub(p_to, p_from)

Pose subtraction

Parameters

p_to: tool pose (spatial vector)

p_from: tool pose (spatial vector)

Return Value

tool pose transformation (spatial vector)

Example command: `pose_sub(p[.2, .5, .1, 1.57, 0, 0],
p[.2, .5, .6, 1.57, 0, 0])`

- Example Parameters:
 - p_1 = p(.2,.5,.1,1.57,0,0) → The first point
 - p_2 = p(.2,.5,.6,1.57,0,0) → The second point
 - * Returns p(0.0,0.0,-0.5,0.0,.0,.0.0)

pose_trans(*p_from*, *p_from_to*)

Pose transformation

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:

$$T_{\text{world} \rightarrow \text{to}} = T_{\text{world} \rightarrow \text{from}} * T_{\text{from} \rightarrow \text{to}}$$

$$T_{x \rightarrow \text{to}} = T_{x \rightarrow \text{from}} * T_{\text{from} \rightarrow \text{to}}$$

Parameters

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

Return Value

resulting pose (spatial vector)

Example command: `pose_trans(p[.2, .5, .1, 1.57, 0, 0],
p[.2, .5, .6, 1.57, 0, 0])`

- Example Parameters:
 - *p_1* = `p(.2,.5,.1,1.57,0,0)` → The first point
 - *p_2* = `p(.2,.5,.6,1.57,0,0)` → The second point
 - * Returns `p(0.4,-0.0996,0.60048,3.14,0,0,0)`

pow(*base, exponent*)

Returns base raised to the power of exponent

Returns the result of raising base to the power of exponent. If base is negative and exponent is not an integral value, or if base is zero and exponent is negative, a runtime error is raised.

Parameters

base: floating point value

exponent: floating point value

Return Value

base raised to the power of exponent

Example command: `pow(5., 3)`

- Example Parameters:
 - Base = 5
 - Exponent = 3
 - * Returns 125.

r2d(*r*)

Returns radians-to-degrees of r

Returns the degree value of 'r' radians.

Parameters

r: The angle in radians

Return Value

The angle in degrees

Example command: `r2d(1.57)`

- Example Parameters:
 - r 1.5707 rad
 - * Returns 90 deg

random()

Random Number

Return Value

pseudo-random number between 0 and 1 (float)

rotvec2rpy(*rotation_vector*)

Returns RPY vector corresponding to *rotation_vector*

Returns the RPY vector corresponding to 'rotation_vector' where the rotation vector is the axis of rotation with a length corresponding to the angle of rotation in radians.

Parameters

rotation_vector: The rotation vector (Vector3d) in radians, also called the Axis-Angle vector (unit-axis of rotation multiplied by the rotation angle in radians).

Return Value

The RPY vector (Vector3d) in radians, describing a roll-pitch-yaw sequence of extrinsic rotations about the X-Y-Z axes, (corresponding to intrinsic rotations about the Z-Y'-X'' axes). In matrix form the RPY vector is defined as $R_{rpy} = R_z(\text{yaw})R_y(\text{pitch})R_x(\text{roll})$.

Example command: `rotvec2rpy([3.14, 1.57, 0])`

- Example Parameters:
 - `rotation_vector = (3.14, 1.57, 0)` → `rx=3.14, ry=1.57, rz=0`
 - * Returns `(2.80856, .16202, 0.9)` → `roll=2.80856, pitch=.16202, yaw=0.9`

rpy2rotvec(*rpy_vector*)

Returns rotation vector corresponding to *rpy_vector*

Returns the rotation vector corresponding to 'rpy_vector' where the RPY (roll-pitch-yaw) rotations are extrinsic rotations about the X-Y-Z axes (corresponding to intrinsic rotations about the Z-Y'-X'' axes).

Parameters

rpy_vector: The RPY vector (Vector3d) in radians, describing a roll-pitch-yaw sequence of extrinsic rotations about the X-Y-Z axes, (corresponding to intrinsic rotations about the Z-Y'-X'' axes). In matrix form the RPY vector is defined as $R_{rpy} = R_z(\text{yaw})R_y(\text{pitch})R_x(\text{roll})$.

Return Value

The rotation vector (Vector3d) in radians, also called the Axis-Angle vector (unit-axis of rotation multiplied by the rotation angle in radians).

Example command: `rpy2rotvec([3.14, 1.57, 0])`

- Example Parameters:
 - *rpy_vector* = (3.14,1.57,0) → roll=3.14, pitch=1.57, yaw=0
 - * Returns (2.22153, 0.00177, -2.21976) → rx=2.22153, ry=0.00177, rz=-2.21976

sin(*f*)

Returns the sine of *f*

Returns the sine of an angle of *f* radians.

Parameters

f: floating point value

Return Value

the sine of *f*.

Example command: `sin(1.57)`

- Example Parameters:
 - *f* is angle of 1.57 rad (90 deg)
 - * Returns 1.0

sqrt(*f*)

Returns the square root of *f*

Returns the square root of *f*. If *f* is negative, a runtime error is raised.

Parameters

f: floating point value

Return Value

the square root of *f*.

Example command: `sqrt(9)`

- Example Parameters:
 - *f* = 9
 - * Returns 3

tan(*f*)

Returns the tangent of *f*

Returns the tangent of an angle of *f* radians.

Parameters

f: floating point value

Return Value

the tangent of *f*.

Example command: `tan(.7854)`

- Example Parameters:
 - *f* is angle of .7854 rad (45 deg)
 - * Returns 1.0

<p>wrench_trans(<i>T_from_to</i>, <i>w_from</i>)</p> <hr/> <p>Wrench transformation</p> <p>Move the point of view of a wrench.</p> <p>Note: Transforming wrenches is not as trivial as transforming poses as the torque scales with the length of the translation.</p> <p>$w_{to} = T_{from \rightarrow to} * w_{from}$</p> <p>Parameters</p> <p><i>T_from_to</i>: The transformation to the new point of view (Pose)</p> <p><i>w_from</i>: wrench to transform in list format (<i>F_x</i>, <i>F_y</i>, <i>F_z</i>, <i>M_x</i>, <i>M_y</i>, <i>M_z</i>)</p> <p>Return Value</p> <p>resulting wrench, <i>w_to</i> in list format (<i>F_x</i>, <i>F_y</i>, <i>F_z</i>, <i>M_x</i>, <i>M_y</i>, <i>M_z</i>)</p>
--

4.2 Variables

Name	Description
<code>__package__</code>	Value: None

5 Module interfaces

5.1 Functions

get_analog_in(n)

Deprecated: Get analog input signal level

Parameters

n: The number (id) of the input, integer: (0:3)

Return Value

float, The signal level in Amperes, or Volts

Deprecated: The `get_standard_analog_in` and `get_tool_analog_in` replace this function. Ports 2-3 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

Note: For backwards compatibility n:2-3 go to the tool analog inputs.

Example command: `get_analog_in(1)`

- Example Parameters:
 - n is analog input 1
 - * Returns value of analog output #1

get_analog_out(n)

Deprecated: Get analog output signal level

Parameters

n: The number (id) of the output, integer: (0:1)

Return Value

float, The signal level in Amperes, or Volts

Deprecated: The `get_standard_analog_out` replaces this function. This function might be removed in the next major release.

Example command: `get_analog_out(1)`

- Example Parameters:
 - n is analog output 1
 - * Returns value of analog output #1

get_configurable_digital_in(*n*)

Get configurable digital input signal level

See also `get_standard_digital_in` and `get_tool_digital_in`.

Parameters

n: The number (id) of the input, integer: (0:7)

Return Value

boolean, The signal level.

Example command: `get_configurable_digital_in(1)`

- Example Parameters:
 - *n* is configurable digital input 1
 - * Returns True or False

get_configurable_digital_out(*n*)

Get configurable digital output signal level

See also `get_standard_digital_out` and `get_tool_digital_out`.

Parameters

n: The number (id) of the output, integer: (0:7)

Return Value

boolean, The signal level.

Example command: `get_configurable_digital_out(1)`

- Example Parameters:
 - *n* is configurable digital output 1
 - * Returns True or False

get_digital_in(n)

Deprecated: Get digital input signal level

Parameters

n: The number (id) of the input, integer: (0:9)

Return Value

boolean, The signal level.

Deprecated: The `get_standard_digital_in` and `get_tool_digital_in` replace this function. Ports 8-9 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

Note: For backwards compatibility n:8-9 go to the tool digital inputs.

Example command: `get_digital_in(1)`

- Example Parameters:
 - n is digital input 1
 - * Returns True or False

get_digital_out(n)

Deprecated: Get digital output signal level

Parameters

n: The number (id) of the output, integer: (0:9)

Return Value

boolean, The signal level.

Deprecated: The `get_standard_digital_out` and `get_tool_digital_out` replace this function. Ports 8-9 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

Note: For backwards compatibility n:8-9 go to the tool digital outputs.

Example command: `get_digital_out(1)`

- Example Parameters:
 - n is digital output 1
 - * Returns True or False

get_euomap_input(*port_number*)

Reads the current value of a specific Euomap67 input signal. See <http://universal-robots.com/support> for signal specifications.

```
>>> var = get_euomap_input(3)
```

Parameters

`port_number`: An integer specifying one of the available Euomap67 input signals.

Return Value

A boolean, either True or False

Example command: `get_euomap_input(1)`

- Example Parameters:
 - `port_number` is euomap digital input on port 1
 - * Returns True or False

get_euomap_output(*port_number*)

Reads the current value of a specific Euomap67 output signal. This means the value that is sent from the robot to the injection moulding machine. See <http://universal-robots.com/support> for signal specifications.

```
>>> var = get_euomap_output(3)
```

Parameters

`port_number`: An integer specifying one of the available Euomap67 output signals.

Return Value

A boolean, either True or False

Example command: `get_euomap_output(1)`

- Example Parameters:
 - `port_number` is euomap digital output on port 1
 - * Returns True or False

get_flag(n)

Flags behave like internal digital outputs. They keep information between program runs.

Parameters

n: The number (id) of the flag, integer: (0:32)

Return Value

Boolean, The stored bit.

Example command: `get_flag(1)`

- Example Parameters:
 - n is flag number 1
 - * Returns True or False

get_standard_analog_in(n)

Get standard analog input signal level

See also `get_tool_analog_in`.

Parameters

n: The number (id) of the input, integer: (0:1)

Return Value

float, The signal level in Amperes, or Volts

Example command: `get_standard_analog_in(1)`

- Example Parameters:
 - n is standard analog input 1
 - * Returns value of standard analog input #1

get_standard_analog_out(n)

Get standard analog output signal level

Parameters

n: The number (id) of the output, integer: (0:1)

Return Value

float, The signal level in Amperes, or Volts

Example command: `get_standard_analog_out(1)`

- Example Parameters:
 - n is standard analog output 1
 - * Returns value of standard analog output #1

get_standard_digital_in(n)

Get standard digital input signal level

See also `get_configurable_digital_in` and `get_tool_digital_in`.

Parameters

n: The number (id) of the input, integer: (0:7)

Return Value

boolean, The signal level.

Example command: `get_standard_digital_in(1)`

- Example Parameters:
 - n is standard digital input 1
 - * Returns True or False

get_standard_digital_out(n)

Get standard digital output signal level

See also `get_configurable_digital_out` and `get_tool_digital_out`.

Parameters

n: The number (id) of the output, integer: (0:7)

Return Value

boolean, The signal level.

Example command: `get_standard_digital_out(1)`

- Example Parameters:
 - n is standard digital output 1
 - * Returns True or False

get_tool_analog_in(*n*)

Get tool analog input signal level

See also `get_standard_analog_in`.

Parameters

n: The number (id) of the input, integer: (0:1)

Return Value

float, The signal level in Amperes, or Volts

Example command: `get_tool_analog_in(1)`

- Example Parameters:
 - *n* is tool analog input 1
 - * Returns value of tool analog input #1

get_tool_digital_in(*n*)

Get tool digital input signal level

See also `get_configurable_digital_in` and `get_standard_digital_in`.

Parameters

n: The number (id) of the input, integer: (0:1)

Return Value

boolean, The signal level.

Example command: `get_tool_digital_in(1)`

- Example Parameters:
 - *n* is tool digital input 1
 - * Returns True or False

get_tool_digital_out(n)

Get tool digital output signal level

See also `get_standard_digital_out` and `get_configurable_digital_out`.

Parameters

n: The number (id) of the output, integer: (0:1)

Return Value

boolean, The signal level.

Example command: `get_tool_digital_out(1)`

- Example Parameters:
 - n is tool digital out 1
 - * Returns True or False

modbus_add_signal(*IP, slave_number, signal_address, signal_type, signal_name, sequential_mode=False*)

Adds a new modbus signal for the controller to supervise. Expects no response.

```
>>> modbus_add_signal("172.140.17.11", 255, 5, 1, "output1")
```

Parameters

IP:	A string specifying the IP address of the modbus unit to which the modbus signal is connected.
slave_number:	An integer normally not used and set to 255, but is a free choice between 0 and 255.
signal_address:	An integer specifying the address of the either the coil or the register that this new signal should reflect. Consult the configuration of the modbus unit for this information.
signal_type:	An integer specifying the type of signal to add. 0 = digital input, 1 = digital output, 2 = register input and 3 = register output.
signal_name:	A string uniquely identifying the signal. If a string is supplied which is equal to an already added signal, the new signal will replace the old one.
sequential_mode:	Setting to True forces the modbus client to wait for a response before sending the next request. This mode is required by some fieldbus units (Optional).

Example command: `modbus_add_signal("172.140.17.11", 255, 5, 1, "output1")`

- Example Parameters:
 - IP address = 172.140.17.11
 - Slave number = 255
 - Signal address = 5
 - Signal type = 1 digital output
 - Signal name = output 1

modbus_delete_signal(*signal_name*)

Deletes the signal identified by the supplied signal name.

```
>>> modbus_delete_signal("output1")
```

Parameters

signal_name: A string equal to the name of the signal that should be deleted.

modbus_get_signal_status(*signal_name, is_secondary_program*)

Reads the current value of a specific signal.

```
>>> modbus_get_signal_status("output1", False)
```

Parameters

signal_name: A string equal to the name of the signal for which the value should be gotten.

is_secondary_program: A boolean for internal use only. Must be set to False.

Return Value

An integer or a boolean. For digital signals: True or False. For register signals: The register value expressed as an unsigned integer.

Example command:

```
modbus_get_signal_status("output1", False)
```

- Example Parameters:
 - Signal name = output 1
 - *is_secondary_program* = False (Note: must be set to False)

modbus_send_custom_command(*IP, slave_number, function_code, data*)

Sends a command specified by the user to the modbus unit located on the specified IP address. Cannot be used to request data, since the response will not be received. The user is responsible for supplying data which is meaningful to the supplied function code. The builtin function takes care of constructing the modbus frame, so the user should not be concerned with the length of the command.

```
>>> modbus_send_custom_command("172.140.17.11", 103, 6,  
>>>                               [17, 32, 2, 88])
```

The above example sets the watchdog timeout on a Beckhoff BK9050 to 600 ms. That is done using the modbus function code 6 (preset single register) and then supplying the register address in the first two bytes of the data array ((17,32) = (0x1120)) and the desired register content in the last two bytes ((2,88) = (0x0258) = dec 600).

Parameters

IP:	A string specifying the IP address locating the modbus unit to which the custom command should be send.
slave_number:	An integer specifying the slave number to use for the custom command.
function_code:	An integer specifying the function code for the custom command.
data:	An array of integers in which each entry must be a valid byte (0-255) value.

Example command:

```
modbus_send_custom_command("172.140.17.11", 103, 6,  
[17, 32, 2, 88])
```

• Example Parameters:

- IP address = 172.140.17.11
- Slave number = 103
- Function code = 6
- Data = (17,32,2,88)

* Function code and data are specified by the manufacturer of the slave Modbus device connected to the UR controller

modbus_set_output_register(*signal_name*, *register_value*,
is_secondary_program)

Sets the output register signal identified by the given name to the given value.

```
>>> modbus_set_output_register("output1", 300, False)
```

Parameters

<code>signal_name</code> :	A string identifying an output register signal that in advance has been added.
<code>register_value</code> :	An integer which must be a valid word (0-65535) value.
<code>is_secondary_program</code> :	A boolean for internal use only. Must be set to False.

Example command: `modbus_set_output_register("output1", 300, False)`

- Example Parameters:
 - Signal name = output1
 - Register value = 300
 - `is_secondary_program` = False (Note: must be set to False)

modbus_set_output_signal(*signal_name*, *digital_value*,
is_secondary_program)

Sets the output digital signal identified by the given name to the given value.

```
>>> modbus_set_output_signal("output2", True, False)
```

Parameters

<code>signal_name</code> :	A string identifying an output digital signal that in advance has been added.
<code>digital_value</code> :	A boolean to which value the signal will be set.
<code>is_secondary_program</code> :	A boolean for internal use only. Must be set to False.

Example command: `modbus_set_output_signal("output1", True, False)`

- Example Parameters:
 - Signal name = output1
 - Digital value = True
 - Is_secondary_program = False (Note: must be set to False)

modbus_set_runstate_dependent_choice(*signal_name*,
runstate_choice)

Sets whether an output signal must preserve its state from a program, or it must be set either high or low when a program is not running.

```
>>> modbus_set_runstate_dependent_choice("output2", 1)
```

Parameters

<code>signal_name</code> :	A string identifying an output digital signal that in advance has been added.
<code>runstate_choice</code> :	An integer: 0 = preserve program state, 1 = set low when a program is not running, 2 = set high when a program is not running.

Example command:

```
modbus_set_runstate_dependent_choice("output2", 1)
```

- Example Parameters:
 - Signal name = output2
 - Runstate dependent choice = 1 → set low when a program is not running

modbus_set_signal_update_frequency(*signal_name*,
update_frequency)

Sets the frequency with which the robot will send requests to the Modbus controller to either read or write the signal value.

```
>>> modbus_set_signal_update_frequency("output2", 20)
```

Parameters

signal_name: A string identifying an output digital signal that in advance has been added.

update_frequency: An integer in the range 0-125 specifying the update frequency in Hz.

Example command:

```
modbus_set_signal_update_frequency("output2", 20)
```

- Example Parameters:
 - Signal name = output2
 - Signal update frequency = 20 Hz

read_input_boolean_register(*address*)

Reads the boolean from one of the input registers, which can also be accessed by a Field bus. Note, uses it's own memory space.

```
>>> bool_val = read_input_boolean_register(3)
```

Parameters

address: Address of the register (0:63)

Return Value

The boolean value held by the register (True, False)

Example command: `read_input_boolean_register(3)`

- Example Parameters:
 - Address = input boolean register 3

read_input_float_register(*address*)

Reads the float from one of the input registers, which can also be accessed by a Field bus. Note, uses it's own memory space.

```
>>> float_val = read_input_float_register(3)
```

Parameters

address: Address of the register (0:23)

Return Value

The value held by the register (float)

Example command: `read_input_float_register(3)`

- Example Parameters:
 - Address = input float register 3

read_input_integer_register(*address*)

Reads the integer from one of the input registers, which can also be accessed by a Field bus. Note, uses it's own memory space.

```
>>> int_val = read_input_integer_register(3)
```

Parameters

address: Address of the register (0:23)

Return Value

The value held by the register (-2,147,483,648 : 2,147,483,647)

Example command: `read_input_integer_register(3)`

- Example Parameters:
 - Address = input integer register 3

read_output_boolean_register(*address*)

Reads the boolean from one of the output registers, which can also be accessed by a Field bus. Note, uses it's own memory space.

```
>>> bool_val = read_output_boolean_register(3)
```

Parameters

address: Address of the register (0:63)

Return Value

The boolean value held by the register (True, False)

Example command: `read_output_boolean_register(3)`

- Example Parameters:
 - Address = output boolean register 3

read_output_float_register(address)

Reads the float from one of the output registers, which can also be accessed by a Field bus. Note, uses it's own memory space.

```
>>> float_val = read_output_float_register(3)
```

Parameters

address: Address of the register (0:23)

Return Value

The value held by the register (float)

Example command: `read_output_float_register(3)`

- Example Parameters:
 - Address = output float register 3

read_output_integer_register(address)

Reads the integer from one of the output registers, which can also be accessed by a Field bus. Note, uses it's own memory space.

```
>>> int_val = read_output_integer_register(3)
```

Parameters

address: Address of the register (0:23)

Return Value

The int value held by the register (-2,147,483,648 : 2,147,483,647)

Example command: `read_output_integer_register(3)`

- Example Parameters:
 - Address = output integer register 3

read_port_bit(address)

Reads one of the ports, which can also be accessed by Modbus clients

```
>>> boolval = read_port_bit(3)
```

Parameters

address: Address of the port (See portmap on Support site, page "UsingModbusServer")

Return Value

The value held by the port (True, False)

Example command: `read_port_bit(3)`

- Example Parameters:
 - Address = port bit 3

read_port_register(*address*)

Reads one of the ports, which can also be accessed by Modbus clients

```
>>> intval = read_port_register(3)
```

Parameters

address: Address of the port (See portmap on Support site, page "UsingModbusServer")

Return Value

The signed integer value held by the port (-32768 : 32767)

Example command: `read_port_register(3)`

- Example Parameters:
 - Address = port register 3

rpc_factory(*type, url*)

Creates a new Remote Procedure Call (RPC) handle. Please read the subsection `ef{Remote Procedure Call (RPC)}` for a more detailed description of RPCs.

```
>>> proxy = rpc_factory("xmlrpc", "http://127.0.0.1:8080/RPC2")
```

Parameters

- type:** The type of RPC backed to use. Currently only the "xmlrpc" protocol is available.
- url:** The URL to the RPC server. Currently two protocols are supported: pstream and http. The pstream URL looks like "<ip-address>:<port>", for instance "127.0.0.1:8080" to make a local connection on port 8080. A http URL generally looks like "http://<ip-address>:<port>/<path>", whereby the <path> depends on the setup of the http server. In the example given above a connection to a local Python webserver on port 8080 is made, which expects XMLRPC calls to come in on the path "RPC2".

Return Value

A RPC handle with a connection to the specified server using the designated RPC backend. If the server is not available the function and program will fail. Any function that is made available on the server can be called using this instance. For example "bool isTargetAvailable(int number, ...)" would be "proxy.isTargetAvailable(var_1, ...)", whereby any number of arguments are supported (denoted by the ...).

Note: Giving the RPC instance a good name makes programs much more readable (i.e. "proxy" is not a very good name).

Example command: `rpc_factory("xmlrpc", "http://127.0.0.1:8080/RPC2")`

- Example Parameters:
 - type = xmlrpc
 - url = http://127.0.0.1:8080/RPC2

rtde_set_watchdog(*variable_name*, *min_frequency*, *action='pause'*)

This function will activate a watchdog for a particular input variable to the RTDE. When the watchdog did not receive an input update for the specified variable in the time period specified by *min_frequency* (Hz), the corresponding action will be taken. All watchdogs are removed on program stop.

```
>>> rtde_set_watchdog("input_int_register_0", 10, "stop")
```

Parameters

- variable_name:** Input variable name (string), as specified by the RTDE interface
- min_frequency:** The minimum frequency (float) an input update is expected to arrive.
- action:** Optional: Either "ignore", "pause" or "stop" the program on a violation of the minimum frequency. The default action is "pause".

Return Value

None

Note: Only one watchdog is necessary per RTDE input package to guarantee the specified action on missing updates.

Example command: `rtde set watchdog("input int register 0" , 10, "stop")`

- Example Parameters:
 - variable name = input int register 0
 - min frequency = 10 hz
 - action = stop the program

set_analog_inputrange(port, range)

Deprecated: Set range of analog inputs

Port 0 and 1 is in the controller box, 2 and 3 is in the tool connector.

Parameters

port: analog input port number, 0,1 = controller, 2,3 = tool

range: *Controller* analog input range 0: 0-5V (maps automatically onto range 2) and range 2: 0-10V.

range: *Tool* analog input range 0: 0-5V (maps automatically onto range 1), 1: 0-10V and 2: 4-20mA.

Deprecated: The `set_standard_analog_input_domain` and `set_tool_analog_input_domain` replace this function. Ports 2-3 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

Note: For *Controller* inputs ranges 1: -5-5V and 3: -10-10V are no longer supported and will show an exception in the GUI.

set_analog_out(n, f)

Deprecated: Set analog output signal level

Parameters

n: The number (id) of the output, integer: (0;1)

f: The relative signal level (0;1) (float)

Deprecated: The `set_standard_analog_out` replaces this function. This function might be removed in the next major release.

Example command: `set_analog_out(1, 0.5)`

- Example Parameters:
 - n is standard analog output port 1
 - f = 0.5, that corresponds to 5V (or 12mA depending on domain setting) on the output port

set_analog_outputdomain(*port, domain*)

Set domain of analog outputs

Parameters

- port*: analog output port number
domain: analog output domain: 0: 4-20mA, 1: 0-10V

Example command: `set_analog_outputdomain(1,1)`

- Example Parameters:
 - *port* is analog output port 1 (on controller)
 - *domain* = 1 (0-10 volts)

set_configurable_digital_out(*n, b*)

Set configurable digital output signal level

See also `set_standard_digital_out` and `set_tool_digital_out`.

Parameters

- n*: The number (id) of the output, integer: (0:7)
b: The signal level. (boolean)

Example command: `set_configurable_digital_out(1,True)`

- Example Parameters:
 - *n* is configurable digital output 1
 - *b* = True

set_digital_out(*n, b*)

Deprecated: Set digital output signal level

Parameters

- n*: The number (id) of the output, integer: (0:9)
b: The signal level. (boolean)

Deprecated: The `set_standard_digital_out` and `set_tool_digital_out` replace this function. Ports 8-9 should be changed to 0-1 for the latter function. This function might be removed in the next major release.

Example command: `set_digital_out(1,True)`

- Example Parameters:
 - *n* is digital output 1
 - *b* = True

set_euomap_output(*port_number, signal_value*)

Sets the value of a specific Euomap67 output signal. This means the value that is sent from the robot to the injection moulding machine. See <http://universal-robots.com/support> for signal specifications.

```
>>> set_euomap_output (3, True)
```

Parameters

port_number: An integer specifying one of the available Euomap67 output signals.

signal_value: A boolean, either True or False

Example command: `set_euomap_output (1, True)`

- Example Parameters:
 - port_number is euomap digital output on port 1
 - signal_value = True

set_euomap_runstate_dependent_choice(*port_number, runstate_choice*)

Sets whether an Euomap67 output signal must preserve its state from a program, or it must be set either high or low when a program is not running. See <http://universal-robots.com/support> for signal specifications.

```
>>> set_euomap_runstate_dependent_choice (3, 0)
```

Parameters

port_number: An integer specifying a Euomap67 output signal.

runstate_choice: An integer: 0 = preserve program state, 1 = set low when a program is not running, 2 = set high when a program is not running.

Example command:

```
set_euomap_runstate_dependent_choice (1, 1)
```

- Example Parameters:
 - port_number is euomap digital output on port 1
 - runstate_choice = 0 → set low when a program is not running

set_flag(*n, b*)

Flags behave like internal digital outputs. They keep information between program runs.

Parameters

n: The number (id) of the flag, integer: (0:32)

b: The stored bit. (boolean)

Example command: `set_flag(1, True)`

- Example Parameters:
 - *n* is flag number 1
 - *b* = True will set the bit to True

set_runstate_configurable_digital_output_to_value(*outputId, state*)

Sets the output signal levels depending on the state of the program (running or stopped).

Example: Set configurable digital output 5 to high when program is not running.

```
>>> set_runstate_configurable_digital_output_to_value(5, 2)
```

Parameters

outputId: The output signal number (id), integer: (0:7)

state: The state of the output, integer: 0 = Preserve state, 1 = Low when program is not running, 2 = High when program is not running, 3 = High when program is running and low when it is stopped.

Example command:

```
set_runstate_configurable_digital_output_to_value(5, 2)
```

- Example Parameters:
 - *outputid* = configurable digital output on port 5
 - Runstate choice = 2 → High when program is not running

set_runstate_gp_boolean_output_to_value(outputId, state)

Sets the output value depending on the state of the program (running or stopped).

Example: Set general purpose bit output 5 to high when program is not running.

```
>>> set_runstate_gp_bool_output_to_value(5, 2)
```

Parameters

outputId: The output signal number (id), integer: (0:63)

state: The state of the output, integer: 0 = Preserve state, 1 = Low when program is not running, 2 = High when program is not running, 3 = High when program is running and low when it is stopped.

Example command:

```
set_runstate_gp_boolean_output_to_value(5, 2)
```

- Example Parameters:
 - outputid = output on port 5
 - Runstate choice = 2 → High when program is not running

set_runstate_standard_analog_output_to_value(outputId, state)

Sets the output signal levels depending on the state of the program (running or stopped).

Example: Set standard analog output 1 to high when program is not running.

```
>>> set_runstate_standard_analog_output_to_value(1, 2)
```

Parameters

outputId: The output signal number (id), integer: (0:1)

state: The state of the output, integer: 0 = Preserve state, 1 = Min when program is not running, 2 = Max when program is not running, 3 = Max when program is running and Min when it is stopped.

Example command:

```
set_runstate_standard_analog_output_to_value(1, 2)
```

- Example Parameters:
 - outputid = standard analog output on port 1
 - Runstate choice = 2 → High when program is not running

set_runstate_standard_digital_output_to_value(outputId, state)

Sets the output signal level depending on the state of the program (running or stopped).

Example: Set standard digital output 5 to high when program is not running.

```
>>> set_runstate_standard_digital_output_to_value(5, 2)
```

Parameters

outputId: The output signal number (id), integer: (0:7)
state: The state of the output, integer: 0 = Preserve state, 1 = Low when program is not running, 2 = High when program is not running, 3 = High when program is running and low when it is stopped.

Example command:

```
set_runstate_standard_digital_output_to_value(5, 2)
```

- Example Parameters:
 - outputid = standard digital output on port 1
 - Runstate choice = 2 → High when program is not running

set_runstate_tool_digital_output_to_value(outputId, state)

Sets the output signal level depending on the state of the program (running or stopped).

Example: Set tool digital output 1 to high when program is not running.

```
>>> set_runstate_tool_digital_output_to_value(1, 2)
```

Parameters

outputId: The output signal number (id), integer: (0:1)
state: The state of the output, integer: 0 = Preserve state, 1 = Low when program is not running, 2 = High when program is not running, 3 = High when program is running and low when it is stopped.

Example command:

```
set_runstate_tool_digital_output_to_value(1, 2)
```

- Example Parameters:
 - outputid = tool digital output on port 1
 - Runstate choice = 2 → High when program is not running

set_standard_analog_input_domain(*port, domain*)

Set domain of standard analog inputs in the controller box

For the tool inputs see `set_tool_analog_input_domain`.

Parameters

`port`: analog input port number: 0 or 1

`domain`: analog input domains: 0: 4-20mA, 1: 0-10V

Example command: `set_standard_analog_input_domain(1,0)`

- Example Parameters:
 - port = analog input port 1
 - domain = 0 (4-20 mA)

set_standard_analog_out(*n, f*)

Set standard analog output signal level

Parameters

`n`: The number (id) of the output, integer: (0:1)

`f`: The relative signal level (0;1) (float)

Example command: `set_standard_analog_out(1,1.0)`

- Example Parameters:
 - n is standard analog output port 1
 - f = 1.0, that corresponds to 10V (or 20mA depending on domain setting) on the output port

set_standard_digital_out(*n, b*)

Set standard digital output signal level

See also `set_configurable_digital_out` and `set_tool_digital_out`.

Parameters

`n`: The number (id) of the output, integer: (0:7)

`b`: The signal level. (boolean)

Example command: `set_standard_digital_out(1,True)`

- Example Parameters:
 - n is standard digital output 1
 - f = True

set_tool_analog_input_domain(*port, domain*)

Set domain of analog inputs in the tool

For the controller box inputs see

`set_standard_analog_input_domain`.

Parameters

`port`: analog input port number: 0 or 1

`domain`: analog input domains: 0: 4-20mA, 1: 0-10V

Example command: `set_tool_analog_input_domain(1,1)`

- Example Parameters:
 - port = tool analog input 1
 - domain = 1 (0-10 volts)

set_tool_digital_out(*n, b*)

Set tool digital output signal level

See also `set_configurable_digital_out` and

`set_standard_digital_out`.

Parameters

`n`: The number (id) of the output, integer: (0:1)

`b`: The signal level. (boolean)

Example command: `set_tool_digital_out(1, True)`

- Example Parameters:
 - n is tool digital output 1
 - b = True

set_tool_voltage(*voltage*)

Sets the voltage level for the power supply that delivers power to the connector plug in the tool flange of the robot. The voltage can be 0, 12 or 24 volts.

Parameters

`voltage`: The voltage (as an integer) at the tool connector, integer: 0, 12 or 24.

Example command: `set_tool_voltage(24)`

- Example Parameters:
 - voltage = 24 volts

socket_close(*socket_name*=' socket_0')

Closes TCP/IP socket communication

Closes down the socket connection to the server.

```
>>> socket_comm_close()
```

Parameters

socket_name: Name of socket (string)

Example command: `socket_close(socket_name="socket_0")`

- Example Parameters:
 - `socket_name = socket_0`

socket_get_var(*name*, *socket_name*=' socket_0')

Reads an integer from the server

Sends the message "get <name> " through the socket, expects the response "<name> <int> " within 2 seconds. Returns 0 after timeout

```
>>> x_pos = socket_get_var("POS_X")
```

Parameters

name: Variable name (string)

socket_name: Name of socket (string)

Return Value

an integer from the server (int), 0 is the timeout value

Example command: `socket_get_var("POS.X", socket_name="socket_0")`

- Example Parameters:
 - `socket_name = socket_0`

socket_open(*address, port, socket_name=' socket_0'*)

Open TCP/IP ethernet communication socket

Attempts to open a socket connection, times out after 2 seconds.

Parameters

`address`: Server address (string)
`port`: Port number (int)
`socket_name`: Name of socket (string)

Return Value

False if failed, True if connection successfully established

Note: The used network setup influences the performance of client/server communication. For instance, TCP/IP communication is buffered by the underlying network interfaces.

Example command: `socket_open("192.168.5.1", 50000, "socket_10")`

- Example Parameters:
 - `address = 192.168.5.1`
 - `socket = 50000`
 - `socket_name = socket_10`

```
socket_read_ascii_float(number, socket_name=' socket_0' ,  
timeout=2)
```

Reads a number of ascii formatted floats from the socket. A maximum of 30 values can be read in one command.

```
>>> list_of_four_floats = socket_read_ascii_float(4)
```

The format of the numbers should be in parantheses, and seperated by ", ". An example list of four numbers could look like "(1.414 , 3.14159, 1.616, 0.0)".

The returned list contains the total numbers read, and then each number in succession. For example a read_ascii_float on the example above would return (4, 1.414, 3.14159, 1.616, 0.0).

A failed read or timeout will return the list with 0 as first element and then "Not a number (nan)" in the following elements (ex. (0, nan., nan, nan) for a read of three numbers).

Parameters

- | | |
|--------------|---|
| number: | The number of variables to read (int) |
| socket_name: | Name of socket (string) |
| timeout: | The number of seconds until the read action times out (float). A timeout of 0 or negative number indicates that the function should not return until a read is completed. |

Return Value

A list of numbers read (list of floats, length=number+1)

Example command: `socket_read_ascii_float(4, "socket10")`

- Example Parameters:
 - Number = 4 → Number of floats to read
 - socket_name = socket_10

socket_read_binary_integer(*number*, *socket_name*=' socket_0' ,
timeout=2)

Reads a number of 32 bit integers from the socket. Bytes are in network byte order. A maximum of 30 values can be read in one command.

```
>>> list_of_three_ints = socket_read_binary_integer(3)
```

Returns (for example) (3,100,2000,30000), if there is a timeout or the reply is invalid, (0,-1,-1,-1) is returned, indicating that 0 integers have been read

Parameters

<code>number</code> :	The number of variables to read (int)
<code>socket_name</code> :	Name of socket (string)
<code>timeout</code> :	The number of seconds until the read action times out (float). A timeout of 0 or negative number indicates that the function should not return until a read is completed.

Return Value

A list of numbers read (list of ints, length=number+1)

Example command: `socket_read_binary_integer(4, "socket10")`

- Example Parameters:
 - Number = 4 → Number of integers to read
 - `socket_name` = `socket_10`

socket_read_byte_list(*number*, *socket_name*=' socket_0' , *timeout*=2)

Reads a number of bytes from the socket. Bytes are in network byte order. A maximum of 30 values can be read in one command.

```
>>> list_of_three_ints = socket_read_byte_list(3)
```

Returns (for example) (3,100,200,44), if there is a timeout or the reply is invalid, (0,-1,-1,-1) is returned, indicating that 0 bytes have been read

Parameters

number: The number of variables to read (int)
socket_name: Name of socket (string)
timeout: The number of seconds until the read action times out (float). A timeout of 0 or negative number indicates that the function should not return until a read is completed.

Return Value

A list of numbers read (list of ints, length=number+1)

Example command: `socket_read_byte_list(4, "socket10")`

- Example Parameters:
 - Number = 4 → Number of byte variables to read
 - *socket_name* = `socket_10`

socket_read_line(*socket_name*=' socket_0' , *timeout*=2)

Deprecated: Reads the socket buffer until the first "\r\n" (carriage return and newline) characters or just the "\n" (newline) character, and returns the data as a string. The returned string will not contain the "\n" nor the "\r\n" characters. Bytes are in network byte order.

```
>>> line_from_server = socket_read_line()
```

Returns (for example) "reply from the server:", if there is a timeout or the reply is invalid, an empty line is returned (""). You can test if the line is empty with an if-statement.

```
>>> if(line_from_server) :  
>>>     popup("the line is not empty")  
>>> end
```

Parameters

socket_name: Name of socket (string)

timeout: The number of seconds until the read action times out (float). A timeout of 0 or negative number indicates that the function should not return until a read is completed.

Return Value

One line string

Deprecated: The `socket_read_string` replaces this function. Set flag "interpret_escape" to "True" to enable the use of escape sequences "\n" "\r" and "\t" as a prefix or suffix.

Example command: `socket_read_line("socket10")`

- Example Parameters:
 - `socket_name = socket_10`


```
socket_read_string(socket_name=' socket_0' , prefix=' ' , suffix=' ' ,
interpret_escape=' False' , timeout=2)
```

Reads all data from the socket and returns the data as a string. Bytes are in network byte order.

```
>>> string_from_server = socket_read_string()
```

Returns (for example) "reply from the server:\n Hello World". if there is a timeout or the reply is invalid, an empty string is returned (""). You can test if the string is empty with an if-statement.

```
>>> if(string_from_server) :
>>>     popup ("the string is not empty")
>>> end
```

The optional parameters "prefix" and "suffix", can be used to express what is extracted from the socket. The "prefix" specifies the start of the substring (message) extracted from the socket. The data up to the end of the "prefix" will be ignored and removed from the socket. The "suffix" specifies the end of the substring (message) extracted from the socket. Any remaining data on the socket, after the "suffix", will be preserved. E.g. if the socket server sends a string "noise>hello<", the controller can receive the "hello" by calling this script function with the prefix=">" and suffix="<".

By using the "prefix" and "suffix" it is also possible send multiple string to the controller at once, because the suffix defines where the message ends. E.g. sending ">hello<>world<"

```
>>> hello = socket_read_string(prefix=">", suffix="<")
>>> world = socket_read_string(prefix=">", suffix="<")
```

The optional parameter "interpret_escape" can be used to allow the use of escape sequences "\n", "\t" and "\r" as part of the prefix or suffix.

Parameters

socket_name:	Name of socket (string)
prefix:	Defines a prefix (string)
suffix:	Defines a suffix (string)
interpret_escape:	Enables the interpretation of escape sequences (bool)
timeout:	The number of seconds until the read action times out (float). A timeout of 0 or negative number indicates that the function should not return until a read is completed.

Return Value

String



- Example Parameters:
 - socket_name = socket_10

socket_send_byte(*value*, *socket_name*=' socket_0')

Sends a byte to the server

Sends the byte <value> through the socket. Expects no response. Can be used to send special ASCII characters; 10 is newline, 2 is start of text, 3 is end of text.

Parameters

value: The number to send (byte)
socket_name: Name of socket (string)

Return Value

a boolean value indicating whether the send operation was successful

Example command: `socket_send_byte(2, "socket10")`

- Example Parameters:
 - `value = 2`
 - `socket_name = socket_10`
 - * Returns True or False (sent or not sent)

socket_send_int(*value*, *socket_name*=' socket_0')

Sends an int (int32_t) to the server

Sends the int <value> through the socket. Send in network byte order. Expects no response.

Parameters

value: The number to send (int)
socket_name: Name of socket (string)

Return Value

a boolean value indicating whether the send operation was successful

Example command: `socket_send_int(2, "socket10")`

- Example Parameters:
 - `value = 2`
 - `socket_name = socket_10`
 - * Returns True or False (sent or not sent)

socket_send_line(*str*, *socket_name*=' socket_0')

Sends a string with a newline character to the server - useful for communicatin with the UR dashboard server

Sends the string <str> through the socket in ASCII coding. Expects no response.

Parameters

str: The string to send (ascii)
socket_name: Name of socket (string)

Return Value

a boolean value indicating whether the send operation was successful

Example command: `socket_send_line("hello", "socket10")`

- Example Parameters:
 - `str = hello`
 - `socket_name = socket_10`
 - * Returns True or False (sent or not sent)

socket_send_string(*str*, *socket_name*=' socket_0')

Sends a string to the server

Sends the string <str> through the socket in ASCII coding. Expects no response.

Parameters

str: The string to send (ascii)
socket_name: Name of socket (string)

Return Value

a boolean value indicating whether the send operation was successful

Example command: `socket_send_string("hello", "socket10")`

- Example Parameters:
 - `str = hello`
 - `socket_name = socket_10`
 - * Returns True or False (sent or not sent)

socket_set_var(*name, value, socket_name=' socket_0'*)

Sends an integer to the server

Sends the message "set <name> <value> " through the socket. Expects no response.

```
>>> socket_set_var("POS_Y", 2200)
```

Parameters

name: Variable name (string)
value: The number to send (int)
socket_name: Name of socket (string)

Example command: `socket_set_var("POS_Y", 2200, "socket10")`

- Example Parameters:
 - name = POS_Y
 - value = 2
 - socket_name = socket_10

write_output_boolean_register(*address, value*)

Writes the boolean value into one of the output registers, which can also be accessed by a Field bus. Note, uses it's own memory space.

```
>>> write_output_boolean_register(3, True)
```

Parameters

address: Address of the register (0:63)
value: Value to set in the register (True, False)

Example command: `write_output_boolean_register(3, True)`

- Example Parameters:
 - address = 3
 - value = True

write_output_float_register(*address, value*)

Writes the float value into one of the output registers, which can also be accessed by a Field bus. Note, uses it's own memory space.

```
>>> write_output_float_register(3, 37.68)
```

Parameters

address: Address of the register (0:23)

value: Value to set in the register (float)

Example command: `write_output_float_register(3,37.68)`

- Example Parameters:
 - address = 3
 - value = 37.68

write_output_integer_register(*address, value*)

Writes the integer value into one of the output registers, which can also be accessed by a Field bus. Note, uses it's own memory space.

```
>>> write_output_integer_register(3, 12)
```

Parameters

address: Address of the register (0:23)

value: Value to set in the register (-2,147,483,648 :
2,147,483,647)

Example command: `write_output_integer_register(3,12)`

- Example Parameters:
 - address = 3
 - value = 12

write_port_bit(*address, value*)

Writes one of the ports, which can also be accessed by Modbus clients

```
>>> write_port_bit(3, True)
```

Parameters

address: Address of the port (See portmap on Support site,
page "UsingModbusServer")

value: Value to be set in the register (True, False)

Example command: `write_port_bit(3, True)`

- Example Parameters:
 - Address = 3
 - Value = True

write_port_register(*address, value*)

Writes one of the ports, which can also be accessed by Modbus clients

```
>>> write_port_register(3,100)
```

Parameters

- address:** Address of the port (See portmap on Support site, page "UsingModbusServer")
- value:** Value to be set in the port (0 : 65536) or (-32768 : 32767)

Example command: `write_port_bit(3,100)`

- Example Parameters:
 - Address = 3
 - Value = 100

5.2 Variables

Name	Description
__package__	Value: None

Index

interfaces (*module*), 55–94

- interfaces.get_analog_in (*function*), 56
- interfaces.get_analog_out (*function*), 56
- interfaces.get_configurable_digital_in (*function*), 56
- interfaces.get_configurable_digital_out (*function*), 57
- interfaces.get_digital_in (*function*), 57
- interfaces.get_digital_out (*function*), 58
- interfaces.get_euomap_input (*function*), 58
- interfaces.get_euomap_output (*function*), 59
- interfaces.get_flag (*function*), 59
- interfaces.get_standard_analog_in (*function*), 60
- interfaces.get_standard_analog_out (*function*), 60
- interfaces.get_standard_digital_in (*function*), 60
- interfaces.get_standard_digital_out (*function*), 61
- interfaces.get_tool_analog_in (*function*), 61
- interfaces.get_tool_digital_in (*function*), 62
- interfaces.get_tool_digital_out (*function*), 62
- interfaces.modbus_add_signal (*function*), 63
- interfaces.modbus_delete_signal (*function*), 64
- interfaces.modbus_get_signal_status (*function*), 65
- interfaces.modbus_send_custom_command (*function*), 65
- interfaces.modbus_set_output_register (*function*), 66
- interfaces.modbus_set_output_signal (*function*), 67
- interfaces.modbus_set_runstate_dependent_choice (*function*), 68
- interfaces.modbus_set_signal_update_frequency (*function*), 68
- interfaces.read_input_boolean_register (*function*), 69
- interfaces.read_input_float_register (*function*), 69
- interfaces.read_input_integer_register (*function*), 70
- interfaces.read_output_boolean_register (*function*), 70
- interfaces.read_output_float_register (*function*), 70
- interfaces.read_output_integer_register (*function*), 71
- interfaces.read_port_bit (*function*), 71
- interfaces.read_port_register (*function*), 71
- interfaces.rpc_factory (*function*), 72
- interfaces.rtde_set_watchdog (*function*), 73
- interfaces.set_analog_inputrange (*function*), 74
- interfaces.set_analog_out (*function*), 75
- interfaces.set_analog_outputdomain (*function*), 75
- interfaces.set_configurable_digital_out (*function*), 76
- interfaces.set_digital_out (*function*), 76
- interfaces.set_euomap_output (*function*), 76
- interfaces.set_euomap_runstate_dependent_choice (*function*), 77
- interfaces.set_flag (*function*), 77
- interfaces.set_runstate_configurable_digital_output_to_value (*function*), 78
- interfaces.set_runstate_gp_boolean_output_to_value (*function*), 78
- interfaces.set_runstate_standard_analog_output_to_value (*function*), 79
- interfaces.set_runstate_standard_digital_output_to_value (*function*), 79

interfaces.set_runstate_tool_digital_output_to_value (function), 80
interfaces.set_standard_analog_input_domain (function), 80
interfaces.set_standard_analog_out (function), 81
interfaces.set_standard_digital_out (function), 81
interfaces.set_tool_analog_input_domain (function), 81
interfaces.set_tool_digital_out (function), 82
interfaces.set_tool_voltage (function), 82
interfaces.socket_close (function), 82
interfaces.socket_get_var (function), 83
interfaces.socket_open (function), 83
interfaces.socket_read_ascii_float (function), 84
interfaces.socket_read_binary_integer (function), 85
interfaces.socket_read_byte_list (function), 86
interfaces.socket_read_line (function), 87
interfaces.socket_read_string (function), 88
interfaces.socket_send_byte (function), 89
interfaces.socket_send_int (function), 90
interfaces.socket_send_line (function), 90
interfaces.socket_send_string (function), 91
interfaces.socket_set_var (function), 91
interfaces.write_output_boolean_register (function), 92
interfaces.write_output_float_register (function), 92
interfaces.write_output_integer_register (function), 93
interfaces.write_port_bit (function), 93
interfaces.write_port_register (function), 93
internals (module), 28–40
internals.force (function), 29
internals.get_actual_joint_positions (function), 29
internals.get_actual_joint_speeds (function), 29
internals.get_actual_tcp_pose (function), 29
internals.get_actual_tcp_speed (function), 30
internals.get_actual_tool_flange_pose (function), 30
internals.get_controller_temp (function), 30
internals.get_forward_kin (function), 30
internals.get_inverse_kin (function), 31
internals.get_joint_temp (function), 32
internals.get_joint_torques (function), 32
internals.get_target_joint_positions (function), 33
internals.get_target_joint_speeds (function), 33
internals.get_target_tcp_pose (function), 33
internals.get_target_tcp_speed (function), 33
internals.get_tcp_force (function), 34
internals.get_tcp_offset (function), 34
internals.get_tool_accelerometer_reading (function), 34
internals.get_tool_current (function), 34
internals.is_steady (function), 35
internals.is_within_safety_limits (function), 35
internals.popup (function), 35
internals.powerdown (function), 36

- internals.set_gravity (function), 36
 - internals.set_payload (function), 36
 - internals.set_payload_cog (function), 37
 - internals.set_payload_mass (function), 38
 - internals.set_tcp (function), 38
 - internals.sleep (function), 39
 - internals.sync (function), 39
 - internals.textmsg (function), 39
- motion (module), 11–28
- motion.conveyor_pulse_decode (function), 12
 - motion.end_force_mode (function), 12
 - motion.end_freedrive_mode (function), 13
 - motion.end_teach_mode (function), 13
 - motion.force_mode (function), 13
 - motion.force_mode_set_damping (function), 14
 - motion.freedrive_mode (function), 15
 - motion.get_conveyor_tick_count (function), 15
 - motion.movec (function), 15
 - motion.movej (function), 16
 - motion.movel (function), 17
 - motion.movep (function), 18
 - motion.position_deviation_warning (function), 19
 - motion.reset_revolution_counter (function), 20
 - motion.servoc (function), 21
 - motion.servoj (function), 21
 - motion.set_conveyor_tick_count (function), 22
 - motion.set_pos (function), 23
 - motion.set_safety_mode_transition_hardness (function), 24
 - motion.speedj (function), 24
 - motion.speedl (function), 24
 - motion.stop_conveyor_tracking (function), 25
 - motion.stopj (function), 25
 - motion.stopl (function), 26
 - motion.teach_mode (function), 26
 - motion.track_conveyor_circular (function), 26
 - motion.track_conveyor_linear (function), 27
- urmath (module), 40–55
- urmath.acos (function), 40
 - urmath.asin (function), 40
 - urmath.atan (function), 40
 - urmath.atan2 (function), 41
 - urmath.binary_list_to_integer (function), 41
 - urmath.ceil (function), 42
 - urmath.cos (function), 42
 - urmath.d2r (function), 43
 - urmath.floor (function), 43
 - urmath.get_list_length (function), 43
 - urmath.integer_to_binary_list (function), 44

urmath.interpolate_pose (function), 44
urmath.length (function), 45
urmath.log (function), 45
urmath.norm (function), 46
urmath.point_dist (function), 46
urmath.pose_add (function), 47
urmath.pose_dist (function), 47
urmath.pose_inv (function), 48
urmath.pose_sub (function), 48
urmath.pose_trans (function), 49
urmath.pow (function), 50
urmath.r2d (function), 51
urmath.random (function), 51
urmath.rotvec2rpy (function), 51
urmath.rpy2rotvec (function), 52
urmath.sin (function), 53
urmath.sqrt (function), 53
urmath.tan (function), 54
urmath.wrench_trans (function), 54