UNIVERSAL ROBOTS

The URScript Programming Language

For version 1.5

Contents

1 The URScript Programming Language

1.1 Introduction

The Universal Robot can be controlled a three different levels: The *Graphical User-Interface Level*, the *Script Level* and the *C-API Level*. URScript is the robot programming languange used to control the robot at the *Script Level*. Like any other programming language URScript has variables, types, flow of control statements, function etc. In addition URScript has a number of built-in variables and functions which monitors and controls the I/O and the movements of the robot.

1.2 Connecting to URControl

URControl is the low-level robot controller running on the Mini-ITX PC in the controller cabinet. When the PC boots up URControl starts up as a daemon (like a service) and PolyScope 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-adresse found in the about dialog-box in PolyScope if the robot is not in dns.)
- **port**: 30002

When connected URScript programs or commands are sent i clear text on the socket. Each line is terminated by '\n'.

1.3 Numbers, Variables and Types

The syntax of arithmetic expressions in URScript is very standard:

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

In boolean expressions the 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
end
```

To stop a loop prematurely the `break` statement can be used. Similarly the `continue` statement can be used to pass control to the next iteration of the nearest enclosing loop.

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
```

URScript also supports named parameters. These will not be described here, as the implementation is still somewhat broken.

1.6 Scoping rules

A urscript program is declared as a function without parameters:

```
def myProg():

end
```

Every variable declared inside a program exits at a global scope, except when they are declared inside a function. I that case the variable are local to that function. Two qualifiers are available to modify this behaviour. The `local` qualifier tells the runtime to treat a variable inside a function, as being truly local, even if a global variable with the same name exists. The `global` qualifier forces a variable declared inside a function, to be globally accessible.

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

```
def myProg():

  a = 0

  def myFun():
    a = 1
    return a
  end

  r = myFun()
end
```

In this next example, `a` is declared `local` inside the function, so the two variables are different, even though they have the same name:

```
def myProg():

  a = 0

  def myFun():
    local a = 1
```

```
    return a
  end

  r = myFun()
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.

1.7 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
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
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 off the new thread, and then goes off to execute the instruction following the `run` instruction.

To wait for a running thread to finish, use the `join` command:

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

```
thrd = run myThread()

join thrd
```

This halts the calling threads execution, until the thread is finished

executing. 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
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 are provided. A critical section ensures that the code it encloses is allow to finish, before another thread is allowed to run. It is therefore important that the critical section is kept as short as possible. The syntax is as follows:

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

1.7.1 Threads and scope

The scoping rules for threads are exactly the same, as those used for functions. See section 1.6 for a discussion of these rules.

1.7.2 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 robin[1] 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 doesn't control the robot, and therefor doesn't 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 frame[2]. 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 doesn't control the robot, it still uses "physical" time. The same is true for the `sync` instruction.

1.8 Program Label Messages

A special feature is added to the script code, to make it simple to keep track of which lines are executed by the runtime machine. An example *Program Label Message* in the script code looks as follows;

```
sleep(0.5)
$ 3 "AfterSleep"
digital_out[9] = True
```

After the the Runtime Machnie executes the sleep command, it will send a message of type `PROGRAM_LABEL` to the latest connected primary client. The message will hold the number *3* and the text *AfterSleep*. This way the connected client can keep track of which lines of codes are being executed by the Runtime Machine.

2 Module motion

This module contains functions and variables built into the URScript programming language.

URScript programs are executed in real-time in the URControl RuntimeMachine (RTMachine). The RuntimeMachine communicates with the robot with a frequency of 125hz.

Robot trajectories are generated online by calling the move functions `movej`, `movel` and the speed functions `speedj`, `speedl` and `speedj_init`.

Joint positions (q) and joint speeds (qd) are represented directly as lists of 6 Floats, one for each robot joint. Tool poses (x) are represented as poses also consisting of 6 Floats. In a pose, the first 3 coordinates is a position vector and the last 3 an axis-angle (*http://en.wikipedia.org/wiki/Axis_angle*).

Version: 1.5

Author: Universal Robots <esben@universal-robot.com>

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

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

2.1 Functions

movec(*pose_via*, *pose_to*, *a*=`1.2`, *v*=`0.3`, *r*=`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.

Parameters

- `pose_via:` path point (note: only position only is used).
- `pose_to:` target pose
- `a:` tool acceleration [m/s^2]
- `v:` tool speed [m/s]
- `r:` blend radius (of target pose) [m]

movej(*q*, *a*=`3`, *v*=`0.75`, *t*=`0`, *r*=`0`)

Move to position (linear in joint-space) When using this command, the robot must be at standstill or come from a movej og movel with a blend. The speed and acceleration parameters controls the trapezoid speed profile of the move. The t parameters can be used in stead to set the time for this move. Time setting has priority over speed and acceleration settings. The blend radius can be set with the r parameters, to avoid the robot stopping at the point. However, if he blend region of this mover overlaps with previous or following regions, this move will be skipped, and an 'Overlapping Blends' warning message will be generated.

Parameters

- `q:` joint positions
- `a:` joint acceleration of leading axis [rad/s^2]
- `v:` joint speed of leading axis [rad/s]
- `t:` time [S]
- `r:` blend radius [m]

movel(*pose*, *a*=`1.2`, *v*=`0.3`, *t*=`0`, *r*=`0`)

Move to position (linear in tool-space)

See movej.

Parameters

- `pose:` target pose
- `a:` tool acceleration [m/s^2]
- `v:` tool speed [m/s]
- `t:` time [S]
- `r:` blend radius [m]

movep(*pose*, *a*=`1.2`, *v*=`0.3`, *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
- `a:` tool acceleration [m/s^2]
- `v:` tool speed [m/s]
- `r:` blend radius [m]

servoc(*pose*, *a*=`1.2`, *v*=`0.3`, *r*=`0`)

Servo Circular

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

Parameters

- `pose:` target pose (position + orientation)
- `a:` tool acceleration [m/s^2]
- `v:` tool speed [m/s]
- `r:` blend radius (of target pose) [m]

servoj(*q*, *a*=`3`, *v*=`0.75`, *t*=`0`)

Servo to position (linear in joint-space)

Parameters

- `q:` joint positions
- `a:` NOT used in current version
- `v:` NOT used in current version
- `t:` time [S]

set_pos(*q*)

Set joint positions of simulated robot

Parameters

`q:` joint positions

speedj(*qd*, *a*, *t_min*)

Joint speed

Accelerate to and move with constant joint speed

Parameters

`qd:` joint speeds [rad/s]

`a:` joint acceleration [rad/s^2] (of leading axis)

`t_min:` minimal time before function returns

speedj_init(*qd*, *a*, *t_min*)

Joint speed (when robot is in ROBOT_INITIALIZING_MODE)

Accelerate to and move with constant joint speed

Parameters

`qd:` joint speeds [rad/s]

`a:` joint acceleration [rad/s^2] (of leading axis)

`t_min:` minimal time before function returns

speedl(*xd*, *a*, *t_min*)

Tool speed

Accelerate to and move with constant tool speed

http://axiom.anu.edu.au/~roy/spatial/index.html

Parameters

`xd:` tool speed [m/s] (spatial vector)

`a:` tool acceleration [/s^2]

`t_min:` minimal time before function returns

stopj(*a*)

Stop (linear in joint space)

Decellerate joint speeds to zero

Parameters

`a:` joint acceleration [rad/s^2] (of leading axis)

stopl(a)

Stop (linear in tool space)

Decellerate tool speed to zero

Parameters

`a`: tool accleration [m/s^2]

2.2 Variables

Name	Description
__package__	**Value:** `'Motion'`
a_joint_default	**Value:** `3`
a_tool_default	**Value:** `1.2`
v_joint_default	**Value:** `0.75`
v_tool_default	**Value:** `0.3`

3 Module internals

3.1 Functions

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()

Forward kinematics

Forward kinematic transformation (joint space -> tool space) of current joint positions

Return Value

tool pose (spatial vector)

get_inverse_kin(*x*, *q_near*, *maxPositionError*, *maxOrientationError*)

Inverse kinematics based on numerical algorithm

Inverse kinematic transformation (tool space -> joint space). Solution closest to q_near joint position is returned. The algorithm tries to find a solution with an cartesian error less than maxPostionError and maxOrientationError. If a solution cannot be found, a runtime exception is raised.

Parameters

`x`: tool pose (spatial vector)

Return Value

joint positions

get_joint_positions()

Returns the angular position of all joints

The position of all the joints in radians, returned as a vector of length 6.

Return Value

The joint vector; ([float])

get_joint_speeds()

Returns the angular speed of all joints

The speed of all the joints in radians/second, returned as a vector of length 6.

Return Value

The joint speed vector; ([float])

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 of the joints, compensated by the torque neccesary to move the robot itself, returned as a vector of length 6.

Return Value

The joint torque vector; ([float])

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

powerdown()

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

set_gravity(*d*)

Set the direction of the gravity

Parameters

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

set_payload(*m*)

Set payload mass

Parameters

`m:` mass [kg]

set_tcp(*pose*)

"Set the Tool Center Point

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

Parameters

`pose:` A pose describing the transformation.

sleep(*t*)

Sleep for an amount of time

Parameters

`t:` time [s]

sync()

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

textmsg(*s*)

Send text message

Send message to be shown on the GUI log-tab

Parameters

`s`: message string

3.2 Variables

Name	Description
__package__	**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.

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.

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.

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.

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

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.

d2r(*d*)

Returns degrees-to-radians of d

Returns the radian value of 'd' degrees. Actually: (d/180)*MATH_PI

Parameters

`d`: The angle in degrees

Return Value

The 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

force()

Returns the force exceted at the TCP

Return the current externally excerted force at the TCP. The force is the lengt of the force vector calculated using get_tcp_force().

Return Value

The force in newtons (float)

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

get_tcp_force()

Returns the force twist at the TCP

The force twist is computet baed on the error between the joint torques required to stay on the trajectory, and the expected joint torques. In Newtons and Newtons/rad.

Return Value

A force twist (pose)

interpolate_pose(*x_from*, *x_to*, *alpha*)

Linear interpolation of tool position and orientation.

When alhpa is 0, returns x_from. When alpha is 1, returns x_to. As alpha goes from 0 to 1, returns a pose going in a straigt line (and geodaetic orientation change) from x_from to x_to. If alpha is less than 0, returns a point before x_from on the line. If alpha is greater than 1, returns a pose after x_to on the line.

Parameters

`x_from`: tool pose (pose)

`x_to`: tool pose (pose)

`alpha`: Floating point number

Return Value

interpolated pose (pose)

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 are negative, or if b i 1 an runtime error is raised.

Parameters

`b`: floating point value

`f`: floating point value

Return Value

the logarithm of f to the base of b.

norm(*a*)

Returns the norm of the argument

The argument can be one of three diffrent 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.
```

Parameters

`a`: Pose, float or int

Return Value

norm of a

pose_add(*x_1*, *x_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:

x_3.P = x_1.P + x_2.P

x_3.R = x_1.R * x_2.R

Parameters

`x_1:` tool pose 1(pose)

`x_2:` tool pose 2 (pose)

Return Value

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

pose_dist(*x_from*, *x_to*)

Pose distance

Parameters

`x_from:` tool pose (pose)

`x_to:` tool pose (pose)

Return Value

distance

pose_inv(*x_from*)

Get the invers of a pose

Parameters

`x_from:` tool pose (spatial vector)

Return Value

inverse tool pose transformation (spatial vector)

pose_sub(*x_to*, *x_from*)

Pose subtraction

Parameters

`x_to:` tool pose (spatial vector)

`x_from:` tool pose (spatial vector)

Return Value

tool pose transformation (spatial vector)

pose_trans(*x_from*, *x_from_to*)

Pose transformation

The first argument, x_from, is used to transform the second argument, x_from_to, and the result is then returned. This means that the result is the resulting pose, when starting at the coordinate system of x_from, and then in that coordinate system moving x_from_to.

This function can be seen in two different views. Either the function transforms, that is translates and rotates, x_from_to by the parameters of x_from. Or the function is used to get the resulting pose, when first making a move of x_from and then from there, a move of x_from_to.

Parameters

`x_from:` starting pose (spatial vector)

`x_from_to:` pose change relative to starting pose (spatial vector)

Return Value

resulting pose (spatial vector)

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

random()

Random Number

Return Value

peseudo-random number between 0 and 1 (float)

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.

sqrt(*f*)

Returns the square root of f

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

Parameters

`f`: floating point value

Return Value

the square root of f.

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.

4.2 Variables

Name	Description
__package__	**Value:** `None`

5 Module interfaces

5.1 Functions

get_analog_in(*n*)

Get analog input level

Parameters

`n`: The number (id) of the input. (int) @return float, The signal level [0,1]

get_analog_out(*n*)

Get analog output level

Parameters

`n`: The number (id) of the input. (int)

Return Value

float, The signal level [0;1]

get_digital_in(*n*)

Get digital input signal level

Parameters

`n`: The number (id) of the input. (int)

Return Value

boolean, The signal level.

get_digital_out(*n*)

Get digital output signal level

Parameters

`n`: The number (id) of the output. (int)

Return Value

boolean, The signal level.

get_euromap_input(*port_number*)

Reads the current value of a specific Euromap67 input signal. See http://support.universal-robots.com/Manuals/Euromap67 for signal specifications.

```
>>> get_euromap_input(3):
```

Parameters

`port_number`: An integer specifying one of the available Euromap67 input signals.

Return Value

A boolean, either True or False

get_euromap_output(*port_number*)

Reads the current value of a specific Euromap67 output signal. This means the value that is sent from the robot to the injection moulding machine. See http://support.universal-robots.com/Manuals/Euromap67 for signal specifications.

```
>>> get_euromap_output(3):
```

Parameters

`port_number`: An integer specifying one of the available Euromap67 output signals.

Return Value

A boolean, either True or False

get_flag(*n*)

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

Parameters

`n`: The number (id) of the flag [0;32]. (int)

Return Value

Boolean, The stored bit.

modbus_add_signal(*IP*, *slave_number*, *signal_address*, *signal_type*, *signal_name*)

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.

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 interal use only. Must be set to False.

Return Value

An integer. For digital signals: 1 for True, 0 for False. For register signals: The register value expressed as an unsigned integer. For all signals: -1 for inactive signal, check then the signal name, addresses and connections.

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.

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

`signal_name`:	A string identifying an output register signal that in advance has been added.
`register_value`:	An integer which must be a valid word (0-65535) value.
`is_secondary_program`:	A boolean for interal use only. 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

`signal_name`:	A string identifying an output digital signal that in advance has been added.
`digital_value`:	A boolean to which value the signal will be set.
`is_secondary_program`:	A boolean for interal use only. 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.

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

Parameters

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

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 1-125 specifying the update frequency in Hz.

set_analog_inputrange(*port*, *range*)

Set range of analog inputs

Port 0 and 1 is in the controller box, 2 and 3 is in the tool connector For the ports in the tool connector, range code 2 is current input.

Parameters

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

`range`: analog input range

set_analog_out(*n*, *f*)

Set analog output level

Parameters

`n`: The number (id) of the input. (int)

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

set_analog_outputdomain(*port*, *domain*)

Set domain of analog outputs

Parameters

`port`: analog output port number

`domain`: analog output domain

set_digital_out(*n*, *b*)

Set digital output signal level

Parameters

`n`: The number (id) of the output. (int)

`b`: The signal level. (boolean)

set_euromap_output(*port_number*, *signal_value*)

Sets the value of a specific Euromap67 output signal. This means the value that is sent from the robot to the injection moulding machine. See http://support.universal-robots.com/Manuals/Euromap67 for signal specifications.

```
>>> set_euromap_output(3,True):
```

Parameters

`port_number`: An integer specifying one of the available Euromap67 output signals.

`signal_value`: A boolean, either True or False

set_euromap_runstate_dependent_choice(*port_number*, *runstate_choice*)

Sets whether an Euromap67 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://support.universal-robots.com/Manuals/Euromap67 for signal specifications.

```
>>> set_runstate_dependent_choice(3,0)
```

Parameters

`port_number`: An integer specifying a Euromap67 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.

set_flag(*n*, *b*)

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

Parameters

`n`: The number (id) of the flag [0;32]. (int)

`b`: The stored bit. (boolean)

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 votage can be 0, 12 or 24 volts.

Parameters

`voltage`: The voltage (as an integer) at the tool connector

socket_close()

Closes ethernet communication

Closes down the socket connection to the server.

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

socket_get_var(*name*)

Reads an integer from the server

Sends the message "get <name> " through the socket. Expects the response "<name> <int> " within 2 seconds.

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

Parameters

`name`: Variable name (string)

Return Value

an integer from the server (int)

socket_open(*server*, *port*)

Open ethernet communication

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

Parameters

`server`: Server name (string)

`port`: Port number (int)

Return Value

False if failed, True if connection succesfully established

socket_read_ascii_float(*number*)

Reads a number of ascii float from the TCP/IP connected. A maximum of 15 values can be read in one command.

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

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

The returned list would first have 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 will return the list [0].

Parameters

`number`: The number of variables to read (int)

Return Value

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

socket_read_binary_integer(*number*)

Reads a number of ascii float from the TCP/IP connected. Bytes are in network byte order. A maximum of 16 values can be read in one command.

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

Returns (for example) [3,100,2000,30000]

Parameters

`number`: The number of variables to read (int)

Return Value

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

socket_read_byte_list(*number*)

Reads a number of ascii float from the TCP/IP connected. Bytes are in network byte order. A maximum of 16 values can be read in one command.

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

Returns (for example) [3,100,200,44]

Parameters

`number`: The number of variables to read (int)

Return Value

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

socket_send_byte(*value*)

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_send_int(*value*)

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_send_string(*str*)

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_set_var(*name, value*)

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)

5.2 Variables

Name	Description
__package__	**Value:** `None`