The URScript Programming Language

For version 1.3

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 builtin

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

[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.

each robot joint. Tool poses (x) are also represented as 6 Floats. The first 3 coordinates is a position vector and the last 3 an axis-angle (*http://en.wikipedia.org/wiki/Axis_angle*).

Version: 1.2

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

Copyright: (C) 2008 Universal Robots Aps

2.1 Functions

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

Move to position (linear in joint-space)

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)

Parameters

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

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]

set_pos(*q*)

Set joint positions of simulated robot

Parameters

- `q:` joint positions

sleep(t)

Sleep for an amount of time

Parameters

`t`: time [s]

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.

set_digital_out(n, b)

Set digital output signal level

Parameters

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

`b`: The signal level. (boolean)

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 float, The signal level [0;1]

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)

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 Boolean, The stored bit.

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)

textmsg(*s*)

Send text message
Send message to be shown on the GUI log-tab

Parameters
- `s:` message string

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?

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_outputdomain(*port*, *domain*)

Set domain of analog outputs

Parameters

`port:` analog output port number
`domain:` analog output domain

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

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.

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.

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*)

Inverse kinematics
Inverse kinematic transformation (tool space -> joint space). Solution closest to current joint positions is returned

Parameters

`x`: tool pose (spatial vector)

Return Value

joint positions

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)

pose_dist(*x_from*, *x_to*)

Pose distance

Parameters

`x_from`: tool pose (pose)
`x_to`: tool pose (pose)

Return Value

distance

pose_add(*x_from*, *x_from_to*)

Pose addition

Parameters

`x_from`: tool pose (pose)
`x_from_to`: tool pose transformation (pose)

Return Value

transformed tool pose (pose)

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_to*, *x_from*)

Pose transformation

Parameters

`x_to`: tool pose (spatial vector)
`x_from`: tool pose (spatial vector)

Return Value

tool pose transformation (spatial vector)

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)

random()

Random Number

Return Value

peseudo-random number between 0 and 1 (float)

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_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_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)

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_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_binary_integer(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_close()

Closes ethernet communication

Closes down the socket connection to the server.

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

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.

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)
```

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)
```

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.

get_tcp_force()

Return 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)

force()

Return 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)

floor(*f*)

Return largest integer not greater than f
Rounds floating point number to the largest integer no greater than f.

Parameters
`f`: floation point value

Return Value
rounded integer

get_joint_temp(*j*)

Return 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_controller_temp()

Return 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_joint_positions()

Return 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()

Return 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_torques()

Return 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])

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

sync()

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

powerdown()

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

2.2 Variables

Name	Description
v_joint_default	joint speed - default parameter in movej() **Value:** `0.75`
a_joint_default	joint acceleration - default parameter in move() **Value:** `3`
v_tool_default	"tool speed - default parameter in movel() **Value:** `0.3`
a_tool_default	tool acceleration - default parameter in movel() **Value:** `1.2`