Script Directory

PolyScope X

Original instructions (en)

PolyScope X

Contents

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.

2. Connecting to URControl

URControl is the low-level robot controller running on the Embedded PC in the Control Box cabinet. 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-<serial number> (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

Important:
It is recommended to always read data from the socket. At least 79 bytes have to be read from socket before closing to ensure that underlying TCP protocol closes socket orderly. Otherwise data sent from client may be discarded before script is executed.

It is especially important in cases when socket is opened just to send single script, and closed immediately.

It is recommended to keep sockets open instead of opening end closing for each and every request.

3. Numbers, Variables, and Types

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

```
1+2-3
4*5/6
(1+2)*3/(4-5)
2346.44 % 10
"Hello" + ", " + "World!"
```

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
"Hello" != "World" and "abc" == "abc"
```

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 vari- able. 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.

Note that strings are fundamentally byte arrays without knowledge of the encoding used for the characters it contains. Therefore some string functions that may appear to operate on characters (e.g. `str_len`), actually operates on bytes and the result may not correspond to the expected one in case of string containing sequences of multi-byte or variable-length characters. Refer to the description of the single function for more details.

4. Lists and Structs

4.1. Struct

A set of variables can be aggregated into structs, and thus transferred and stored as a single variable.

Structs can be obtained through multiple means:

- Using the struct function
- Doing an RPC call that returns a struct
- Receiving ROS2 message (available on Polyscope X only)

The struct function takes one or more named arguments, and each argument name becomes a member in the struct. All values must be initialized by value, and the type of the value cannot be changed subsequently.

4.1.1. Using a struct

Create a struct:

```
myStruct = struct(identifier1 = 1, identifier2 = 2, myMember = "Hello structs",
listMember = [1,2,3])
```

Reassign a member:

```
myStruct.myMember = "Goodbye structs"
```

Use a member:

```
myVar = myStruct.myMember
```

Use a nested list:

```
myListElement = myStruct.listMember[0]
```

Use the second member by index (identifier2):

```
myVar = myStruct[1]
```

A nested struct, stored by value:

```
myStruct = struct(myStructMember = struct(myMember = "Hi nested struct") )
```

Conversion of a struct to a list, if all the struct members are of same type and if the list has the same type. Value of myList will be [1.1, 2.2, 3.3, 4.4].

```
  myStruct = struct(m1 = 1.1, m2 = 2.2, m3 = 3.3, m4 = 4.4)
  myList = [0.0, 0.0, 0.0, 0.0]
  myList = myStruct
```

Structs can be passed to and returned from function. In this example we create a new struct extended with a boolean member.

```
  struct_1 = struct(m1 = 1.1, m2="a string", m3=[1,2,3])
  def ExtendStructWithBoolMember(struct_arg):
    struct_local = struct(m1 = 0.0, m2 = "n/a", m3 = make_list(0, 0, 10), extra_
member = False)
    struct_local.m1 = struct_arg.m1
    struct_local.m2 = struct_arg.m2
```

```
    struct_local.m3 = struct_arg.m3
    struct_local.extra_member = True
    return struct_local
  end

  new_extended_struct = ExtendStructWithBoolMember(struct_1)
```

4.2. List

A list is a set of variables with the same type aggregated into a single object.

A list object in URScript has two attributes: length and capacity. The length indicates how many elements the list currently holding. The capacity tells how many elements the list can hold maximum.

Once declared, the capacity of the list cannot be changed.

Fixed length lists can be created with square bracket operator:

```
aa = [11, 22, 33, 44, 55, 66, 77]
```

Both length and capacity of this list is equal to 7.

Variable length lists can be created:

```
bb = make_list(length = 7, initial_value = 11, capacity = 20)
```

Length of this list is 7, but capacity is 20. List can be extended and contracted between 0, and 20 numeric elements.

Lists can hold any type that URScript supports. This includes complex values created with struct() keyword:

```
  aa = [1, 2, 3.5, 4, 5.5]
  bb = make_list(10, struct(p1 = 1, p2 = "text"), 10)
  cc = ["a", "b", "c", "d"]
  dd = [struct(m1 = 10, m2 = "hello", m3 = make_list(length = 25, initial_value
= 0, capacity = 100)) , struct(m1 = 20, m2 = "hi", m3 = make_list(length = 50,
initial_value = 0, capacity = 100))]
```

List can be assigned only to existing list of greater or equal capacity to the length of source list:

```
  aa = [1, 2, 3, 4, 5, 6] # aa.length() == 6, aa.capacity() == 6
  bb = make_list(5, 0, 100) # bb.length() == 5, bb.capacity() == 100
  aa = bb # aa capacity will remain 6 aa length will be 5
  aa = [1, 2, 3, 4, 5, 6]
  bb = aa # bb capacity will remain 100 bb length will be 6
```

List can hold structs (aka complex data types). All structs in the list have to be exactly of the same type:

```
  aa = make_list(10, struct(p1 = 1, p2 = "text"), 10)
  a = aa[4].p1 # a = 1
  b = aa[4].p2 # b = "text"
  aa[3].p1 = 22.5
  aa[4] = struct(p1 = 99, p2 = "different text")
```

4.2.1. Limitations of lists

Lists can be passed to, and returned from functions only as copy by value.

List elements can't change type.

If list is returned from a function or list method, then the target list have to be earlier initialized with enough capacity.

List of lists is not supported as this is how matrices are implemented in URScript.

4.3. Methods in URScript

Starting from Polyscope 5.15, methods (member functions) are callables on list, matrix and structs (currently). The name of the list followed by a "." will invoke the function.

4.3.1. Methods on List

append(element)

Adds the element to the end of the list. Raises an error if at capacity.

Example: add the value 88 to a list.

```
l1 = make_list(0, 0, 10) # empty list of integers with capacity of 10
l1.append(88) # add element to the end of the list, length increases, exception thrown if capacity exceeded
```

capacity()

Returns the maximum capacity of the list (>=length).

Example: merge list 2 to list 1 until list 1 is full. result: [-1, -1, -1, -1, -1, 6, 7, 8, 9, 10]

```
l1 = make_list(5, -1, 10)
l2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
idx = l1.length()
while(idx < l1.capacity()):
  l1.append(l2[idx])
  idx = idx + 1
end
```

clear()

Clear the list by setting length to 0.

```
list_1.clear() # list_1 will be []
```

excess_capacity()

Returns the unused capacity (= capacity-length).

Example: add element if the list has free space. result: [9,9,9,9,9,1,2,3,4,5]; popup "no more space"

```
l1 = make_list(5, 9, 10)
l2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
idx = 0
while(idx < l2.length()):
      if(l1.excess_capacity() > 0):
        l1.append(l2[idx])
```

```
      else:
        popup("no more space")
        break
      end
  idx = idx + 1
end
```

extend(list of elements)

Adds all elements from the parameter list at the end. Raises an error if at capacity. The list in the input must be of the same type as the list.

Example: add list 2 to list 1. result = [0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```
l1 = make_list(2, 0, 100)
l2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
  if l1.excess_capacity() >= l2.length():
        l1.extend(l2)
  end
```

insert(index, element)

Inserts the element at the given index, shifting remaining list. Raises an error if at capacity.

length()

Returns the current length of the list.

Example: update elements of a list in a loop

```
l1 = [1, 2, 3, 4, 5, 6]
idx = 0
    while (idx < l1.length()):
      l1[idx] = 10 + idx
      idx = idx + 1
    end
```

pop()

Removes the last element from the list.

remove(index)

Removes the element at a given index.

Example: remove even numbers from a collection.

```
l2 = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 10, 10, 11]
idx = l2.length() - 1
     while(idx > 0):
      if(l2[idx] % 2 == 0):
      l2.remove(idx)
      end
           idx = idx - 1
```

```
end
```

slice(begin index, end index)

Returns a sub-list of elements [list[param1]... list[param2-1]]. Does not modify the original list.

to_string()

Returns a string representation of the list. has ...] if out of space.

4.3.2. Methods on Struct

length()

Returns the number of elements in the struct.

to_string()

Returns a string representation of the struct. has ...} if out of space.

4.3.3. Methods on Matrix

get_column(index)

Returns the column at the index by value.

get_row(index)

Returns the row at the index by value.

shape()

Returns the number of rows and columns in the matrix.

to_string()

Returns a string representation of the list. has ...] if out of space.

5. Matrix and Array Expressions

Matrix and array operations can be assigned to a variable to store and manipulate multiple numbers at the same time. It is also possible to get access and write to a single entry of the matrix. The matrix and array are 0-indexed.

```
array = [1,2,3,4,5]
a = array[0]
array[2] = 10


matrix = [[1,2],[3,4],[5,6]]
b = matrix[0,0]
matrix[2,1] = 20
```

Matrix and array can be manipulated by matrix-matrix, array-array,matrix-array,matrixscalar and array-scalar operations.

Matrix-matrix multiplication operations are supported if the first matrix's number of columns matches the second matrix's number of rows. The resulting matrix will have the dimensions of the first matrix number of rows and the second matrix number of columns.

```
C = [[1,2],[3,4],[5,6]] * [[10,20,30],[40,50,60]]
```

Matrix-array multiplication operations are supported if the matrix is the first operand and array is second. If the matrix's number of columns matches the arrays length, the resulting array will have the length as the matrix number of rows.

```
C = [[1,2],[3,4],[5,6]] * [10,20]
```

Array-array operations are possible if both arrays are of the same length and supports: addition, subtraction, multiplication, division and modulo operations. The operation is executed index by index on both arrays and thus results in an array of the same length. E.g. `a[i] b[i] = c[i].`

```
mul = [1,2,3] * [10,20,30]
div = [10,20,30] / [1,2,3]
add = [1,2,3] + [10,20,30]
sub = [10,20,30] - [1,2,3]
mod = [10,20,30] % [1,2,3]
```

Scalar operations on a matrix or an array are possible. They support addition, subtraction, multiplication, division and modulo operations. The scalar operations are done on all the entries of the matrix or the array. E.g. `a[i] + b = c[i]`

```
mul1 = [1,2,3] * 5
mul2 = 5 * [[1,2],[3,4],[5,6]]
div1 = [10,20,30] / 10
div2 = 10 / [10,20,30]
add1 = [1,2,3] + 10
add2 = 10 + [1,2,3]
sub1 = [10,20,30] - 5
```

```
sub2 = 5 - [[10,20],[30,40]]
mod1 = [11,22,33] % 5
mod2 = 121 % [[10,20],[30,40]]
```

6. 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.

6.1. Special keywords

- `halt` immediately stops program execution and terminates the program. Not advisible while robot is in motion.
- `return` returns from a function. When no value is returned, the keyword `None` must follow the keyword `return`.
- `pause` pauses program execution.

7. 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
```

If default values are given in the declaration, arguments can be either input or skipped as below:

```
result = add(0,0)
result = add()
```

When calling a function, it is important to comply with the declared order of the ar- guments. If the order is different from its definition, the function does not work as ex- pected.

Arguments can only be passed by value (including arrays). This means that any modi- fication 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.

8. 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()

camera.closeXMLRPCClientConnection()

...
```

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 as- signed the result. The `camera.initialize("RGB"), takeSnapshot() and getTarget()` functions are the responsibility of the RPC server.

The `closeXMLRPCClientConnection` is closing the XMLRPC connection created by the `rpc_factory`. It is recommended to close the connection periodically, or when it's not used for a longer time. Some server implementations by default have a limit of rpc requests or inactivity watchdog timers.

NOTE: The RPC handle does not automatically close the connecetion.

The technical support website: http://www.universal-robots.com/support contains more examples of XMLRPC servers.

8.1. closeXMLRPCClientConnection()

Closes a Remote Procedure Call (RPC) handle.

9. 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.

10. 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
```

10.1. 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.

10.2. 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 500 Hz, or in other words, it must be told what to do every 0.002 second (each 0.002 second period is called a frame). To achieve this, each thread is given a “physical” (or robot) time slice of 0.002 seconds to use, and all threads in a runnable state is then scheduled in a [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 do not control the robot, and therefore do not use any “physical” time. If a thread uses up its entire time slice, either by use of “physical” time or by computational heavy instructions (such as an infinite loop that do not control the robot) 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 [2]. The reason for this state switching can be a join instruction or simply because the thread terminates.

[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.

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. Inserting `sync` or `sleep` will allow time for other threads to be executed and is therefore recommended to use next to computational heavy instructions or inside infinite loops that do not control the robot, otherwise an exception like "Lost communication with Controller" can be raised with a consequent protective stop.

11. 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
```

12. Secondary Programs

Secondary program is executed by the robot controller concurrently and simultaneously with the primary script program. It could be used to handle the I/O signals, while the primary program moves the robot between waypoints. A secondary program could be sent to controller via primary or secondary TCP/IP socket, just like any other script program and must follow the same script syntax as regular robot programs.

Notable exception is that secondary program should not use any “physical” time. In particular, it cannot contain sleep, or move statements. Secondary program must be simple enough to be executed in a single controller step, without blocking the realtime thread.

Exceptions on secondary program do not stop execution of the primary program. Exceptions are reported in robot controller log files.

The secondary function must be defined using the keyword "`sec`" as follows:

```
sec secondaryProgram():
  set_digital_out(1,True)
end
```

12.1. Secondary Program Limitations

Secondary programs are typically run as support programs, they are designed to be executed concurrently with the primary program using real time control loop. Entire execution time is added to single real time cycle. Due to their nature they have some limitations in using specific commands.

Motion Commands: Using any motion commands in secondary program such as `movej`, `movel` and `servoj` will not work. They will cause an error of "Runtime is too much behind". The motion commands depend on the external factor (real robot motion) and are waiting for the result of the execution, which is not possible in secondary programs.

Interpreter Mode: Sending new commands during runtime essentially goes against the design of secondary programs. This will again give you the error of runtime too much behind.

Socket: Socket communication can be used within secondary programs, but this is limited to executing simple non-blocking commands like single `socket_send`. Slow remote server or attempt to execute multiple socket commands will cause "Runtime too much behind", or prevent controller from executing real-time tasks. Socket commands should be avoided in secondary programs.

XMLRPC: Like socket, XMLRPC calls can be used in secondary programs although they need to be executed very quickly by remote server. XMLRPC calls should be avoided in secondary programs.

Sleep: Sleep, similar to motion commands is a blocking call by nature. It requires multiple real-time cycles to complete. Using `Sleep` will cause an error of "Runtime is too much behind"

Threads: Threads in secondary programs are not supported. Thread code execution is not guaranteed, and leads to undefined behaviour.

13. Interpreter Mode

The interpreter mode enables the programmer to send and execute any valid script statement at runtime, except declaration of new globals.

An internal interpreter thread is created at the start of execution of each primary program. The interpreter socket(30020) accepts complete and valid UR-script statements when a program is running. When Interpreter mode is active it compiles and links the received statements into the running program and executes them in the scope of the interpreter thread. These statements become a part of the running program.

The scope of statements in the interpreter mode is inherited from the scope from which interpreter mode was called. Be aware that declaring new global variables in interpreter mode is not supported.
When statements are sent at a faster rate than what the interpreter can handle, they are queued in an internal buffer before they can be appended to the running program.

When the last statement received is executed, the interpreter thread will be idle until new statements are received.

Interpreter mode can be stopped by calling `end_interpreter()` over the interpreter mode socket, or by calling it from a thread in the main program. Interpreter mode also stops when the program is stopped.

Each statement should be sent in a single line, so the following statement:

```
def myFun():
 mystring = "Hello Interpreter"
 textmsg(mystring)
end
```

Should be formatted like below:

```
def myFun(): mystring = "Hello Interpreter" textmsg(mystring) end
```

With a newline character to end the statement. Multiple statements can be sent on a single line, and will only be accepted if all can be compiled.

13.1. Interpreter Mode replies

Received valid commands and statements are always acknowledge with a reply on the socket in form of:

```
ack: <id>: <statements>
```

Where <id> is a consecutively unique id for the received <statement>.

If a program is not running or the statement results in a compilation or linker error, the interpreter will reply with a discard message:

```
discard: <reason>: <statement>.
```

Note: There exists an upper limit to the size of the interpreted statements per interpreter mode. To avoid reaching that limit `clear_interpreter()` can be called to clear up everything interpreted into the current interpreter mode.

Important: It is important to remember that every time a statement is interpreted the size and the complexity of the program will grow if interpreter mode is not cleared either on exit or with `clear_interpreter()`. Too large programs should be avoided.

13.2. Entering Interpreter Mode

`interpreter_mode(clearQueueOnEnter = True, clearOnEnd = True)`

This is a blocking function that puts the controller in interpreter mode.

This function can only be called in the main thread, and nested interpreter modes are not supported. This means that one can't send an interpreter_mode() call to the interpreter socket.

Parameters

`clearQueueOnEnter`: If `True` queued statements will be cleared before interpreter mode is started.

It is possible to send statements to the interpreter socket before interpreter mode is run. These are put into a queue, which by default is cleared when `interpreter_mode()` is called. However, setting the `clearQueueOnEnter` argument to `False` will cause interpreter mode to start executing the statements already in the queue when entering interpreter mode.

`clearOnEnd`: If `True` interpreted statements will be cleared when `end_interpreter()` is called.

Interpreted statements become part of the running program in the scope from which `interpreter_mode()` was called, but are by default removed from the program again when `end_interpreter()` is called. However, when entering interpreter mode it is possible to choose to have a persistent behavior by calling `interpreter_mode(clearOnEnd = False)`. The interpreted statements will in that case be a part of the program until the end of program execution, thus they can be called/accessed in subsequent calls to `interpreter_mode()`.

Example statement:

- `interpreter_mode()`
  - Starts interpreter mode with default behavior.
- `interpreter_mode(clearQueueOnEnter = False)`
  - Starts interpreter mode by interpreting and executing the statements already in the interpreter queue.

Important: It is the programmers responsibility to implement the synchronization necessary to ensure that the program is in the wanted interpreter mode, and that it is ready to receive the statements. It is insufficient to detect if interpreter mode is running, as multiple interpreter mode can exist in the same program.

13.3. End Interpreter Mode

`end_interpreter()`

Ends the interpreter mode, and causes the `interpreter_mode()` function to return. This function can be compiled into the program by sending it to the interpreter socket(30020) as any other statement, or can be called from anywhere else in the program.

By default everything interpreted will be cleared when ending, though the state of the robot, the modifications to local variables from the enclosing scope, and the global variables will remain affected by any changes made. The interpreter thread will be idle after this call.

13.4. Clear Interpreter Mode

`clear_interpreter()`

Clears all interpreted statements, objects, functions, threads, etc. generated in the current interpreter mode. Threads started in current interpreter session will be stopped, and deleted. Variables defined outside of the current interpreter mode will not be affected by a call to this function.

Only statements interpreted before the `clear_interpreter()` function will be cleared. Statements sent after `clear_interpreter()` will be queued. When cleaning is done, any statements queued are interpreted and responded to. Note that commands such as abort, skipbuffer and state commands are executed as soon as they are received.

Note: This function can only be called from an interpreter mode.

Tip: To expedite the clean, `skipbuffer` can be sent right before `clear_interpreter()`.

13.5. Aborting Current Function

`abort`

The interpreter mode offers a mechanism to abort limited number of script functions, even if they are called from the main program. Currently only `movej` and `movel` can be aborted.

Aborting a movement will result in a controlled stop if no blend radius is defined.

If a blend radius is defined then a blend with the next movement will be initiated right away if not already in an initial blend, otherwise the command is ignored.

Return value should be ignored

Note: `abort` must be sent in a line by itself, and thus cannot be combined with other commands or statements.

13.6. Skip Non Executed Statements

`skipbuffer`

The interpreter mode furthermore supports the opportunity to skip already sent but not executed statements. The interpreter thread will then (after finishing the currently executing statement) skip all received but not executed statements.

After the skip, the interpreter thread will idle until new statements are received. `skipbuffer` will only skip already received statements, new statements can therefore be send right away.

Return value should be ignored

Note: `skipbuffer` must be sent in a line by itself, and thus cannot be combined with other commands or statements.

13.7. State Commands for Interpreter Mode

The state commands that can be used to get a status from interpreter mode are listed here. When the state commands below responds with an id, it is the id given in the ackownlegde message at the time the statement was recevied by the interpreter.

Note that these ids start at 1 and might wrap around to 1 in very long running programs. A 0 represents an uninitialized or undefined value, such as the last executed statement if none has been executed yet.

statelastexecuted

Replies with the largest id of a statement that has started being executed.

`state: <id>: statelastexecuted`

statelastinterpreted

Replies with the latest interpreted id, i.e. the highest number of interpreted statement so far.

`state: <id>: statelastinterpreted`

statelastcleared

Replies with the id for the latest statement to be cleared from the interpreter mode. This clear can happen when ending interpreter mode, or by calls to `clear_interpreter()`

`state: <id>: statelastcleared`

stateunexecuted

Replies with the number of non executed statements, i.e. the number of statements that would have be skipped if `skipbuffer` was called instead.

`state: <#unexecuted>: stateunexecuted`

13.8. Interpreter mode log files

Statements acknowledged by the interpreter mode are logged to the file `/tmp/log/urcontrol/interpreter.log`. The file contains basic information on when the log was started, and for each statement a line:

```
e: <id_e> c: <id_a> : <statement>
```

Where <id_e> is the last statement for which execution has started when the statement <statement> with id <id_a> was compiled into the program. To avoid filling the memory of the robot, the logfile is cleaned up according to the following rules:

1. If the interpreter mode was entered with the parameter clearOnEnd=False, all statements in the interpreter mode are appended to the file `/tmp/log/urcontrol/interpreter.saved.log` when end_interpreter() is called.
2. Any other call to end_interpreter() or clear_interpreter() will cause the log to be moved to `/tmp/log/urcontrol/interpreter.0.log` overwriting any data previously stored there.

All interpreter mode log files are included in failure report files.

14. 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 (RTMa- chine). The RuntimeMachine communicates with the robot with a frequency of 500hz.

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

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).

14.1. conveyor_pulse_decode(type, A, B)

Deprecated: 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.

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.

Deprecated: This function is replaced by `encoder_enable_pulse_decode` and it should therefore not be used moving forward.

```
>>> 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).

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

14.2. encoder_enable_pulse_decode(encoder_index, decoder_type, A, B)

Sets up an encoder hooked up to the pulse decoder of the controller.

```
>>> encoder_enable_pulse_decode(0,0,1,8,9)
```

This example shows how to set up encoder 0 for decoding a quadrature signal connected to pin 8 and 9.

Parameters

`encoder_index`: Index of the encoder to define. Must be either 0 or 1.

`decoder_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 pin. Must be 8-11.

`B`:

Encoder input B pin. Must be 8-11.

14.3. encoder_enable_set_tick_count(encoder_index, range_id)

Sets up an encoder expecting to be updated with tick counts via the function `encoder_set_tick_count`.

```
>>> encoder_enable_set_tick_count(0,0)
```

This example shows how to set up encoder 0 to expect counts in the range of [-2147483648 ; 2147483647].

Parameters

`encoder_index:`

Index of the encoder to define. Must be either 0 or 1.

`range_id:`

decoder_index: Range of the encoder

(integer). Needed to handle wrapping nicely.

0 is a 32 bit signed encoder, range [-2147483648 ; 2147483647]

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]

14.4. encoder_get_tick_count(encoder_index, opt="")

Returns the filtered tick count of the designated encoder.

```
>>> encoder_get_tick_count(0)
```

This example returns the current filtered tick count of encoder 0.

Parameters

`encoder_index`: Index of the encoder to query. Must be either 0 or 1.

`opt`: Optional option parameter. Default is `opt=""`. Get the raw unfiltered integer value by `opt="raw"`.

Return Value

The filtered conveyor encoder tick count (float) or the raw value (int32)

Example command 1: `encoder_get_tick_count(0)`

This example returns the current filtered tick count of encoder 0.
Use caution when subtracting encoder tick counts as it wraps around when reaching the maximum count value. The range of the filtered encoder value is [0; 65536[.
Please see the function `encoder_unwind_delta_tick_count`.

Example command 2: `encoder_get_tick_count(1, opt="raw")`

This example returns the current raw tick count of encoder 1.
Use caution when subtracting encoder tick counts.
The range of the raw encoder value is [-2^{31}, 2^{31}[.

14.5. encoder_set_tick_count(encoder_index, count)

Tells the robot controller the tick count of the encoder. This function is useful for absolute encoders (e.g. MODBUS).

```
>>> encoder_set_tick_count(0, 1234)
```

This example sets the tick count of encoder 0 to 1234. Assumes that the encoder is enabled using `encoder_enable_set_tick_count` first.

Parameters

`encoder_index`: Index of the encoder to define. Must be either 0 or 1.

`count`: The tick count to set. Must be within the range of the encoder.

14.6. encoder_unwind_delta_tick_count(encoder_index, delta_tick_count)

Returns the delta_tick_count. Unwinds in case encoder wraps around the range. If no wrapping has happened the given delta_tick_count is returned without any modification.

Consider the following situation: You are using an encoder with a UINT16 range, meaning the tick count is always in the [0; 65536[range. When the encoder is ticking, it may cross either end of the range, which causes the tick count to wrap around to the other end. During your program, the current tick count is assigned to a variable (start:=encoder_get_tick_count(...)). Later, the tick count is assigned to another variable (current:=encoder_get_tick_count(...)). To calculate the distance the conveyor has traveled between the two sample points, the two tick counts are subtracted from each other.

For example, the first sample point is near the end of the range (e.g., start:=65530). When the conveyor arrives at the second point, the encoder may have crossed the end of its range, wrapped around, and reached a value near the beginning of the range (e.g., current:=864). Subtracting the two samples to calculate the motion of the conveyor is not robust, and may result in an incorrect result

(delta=current-start=-64666).

Conveyor tracking applications rely on these kinds of encoder calculations. Unless special care is taken to compensate the encoder wrapping around, the application will not be robust and may produce weird behaviors (e.g., singularities or exceeded speed limits) which are difficult to explain and to reproduce.

This heuristic function checks that a given delta_tick_count value is reasonable. If the encoder wrapped around the end of the range, it compensates (i.e., unwinds) and returns the adjusted result. If a delta_tick_count is larger than half the range of the encoder, wrapping is assumed and is compensated. As a consequence, this function only works when the range of the encoder is explicitly known, and therefore the designated encoder must be enabled. If not, this function will always return nil.

Parameters

`encoder_index`: Index of the encoder to query. Must be either 0 or 1.

`delta_tick_count`: The delta (difference between two) tick count to unwind (float)

Return Value

The unwound delta_tick_count (float)

14.7. end_force_mode()

Resets the robot mode from force mode to normal operation.

This is also done when a program stops.

14.8. end_freedrive_mode()

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

14.9. end_screw_driving()

Exit screw driving mode and return to normal operation.

14.10. end_teach_mode()

Deprecated:

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

This function is replaced by `end_freedrive_mode` and it should therefore not be used moving forward.

14.11. force_mode(task_frame, selection_vector, wrench, type, limits)

Set robot to be controlled in force mode

Parameters

`task_frame`: A pose vector that defines the force frame relative to the base frame.

`selection_vector`: A 6d vector of 0s and 1s. 1 means that the robot will be compliant in the corresponding axis of the task frame.

`wrench`: The forces/torques the robot will apply to its environment. The robot adjusts its position along/about compliant axis in order to achieve the specified force/torque. Values have no effect for non-compliant axes.

Actual wrench applied may be lower than requested due to joint safety limits. Actual forces and torques can be read using get_tcp_force function in a separate thread.

`type`:

An integer [1;3] specifying how the robot interprets the force frame.

1: The force frame is transformed in a way such that its y-axis is aligned with a vector pointing from the robot tcp towards the origin of the force frame.

2: The force frame is not transformed.

3: The force frame is transformed in a way such that its x-axis is the projection of the robot tcp velocity vector onto the x-y plane of the force frame.

`limits`: (Float) 6d vector. For compliant axes, these values are the maximum allowed tcp speed along/about the axis. For non-compliant axes, these values are the maximum allowed deviation along/about an axis between the actual tcp position and the one set by the program.

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.

14.12. force_mode_example()

This is an example of the above `force_mode()` function

Example command: `force_mode(p[0.1,0,0,0,0.785], [1,0,0,0,0,0], [20,0,40,0,0,0], 2, [.2,.1,.1,.785,.785,1.57])`

Example Parameters:

- Task frame = p[0.1,0,0,0,0.785] ! This frame is offset from the base frame 100 mm in the x direction and rotated 45 degrees
- in the rz direction
- Selection Vector = [1,0,0,0,0,0] ! The robot is compliant in the x direction of the Task frame above.
- Wrench = [20,0,40,0,0,0] ! The robot apples 20N in the x direction. It also accounts for a 40N external force in the z direction.
- Type = 2 ! The force frame is not transformed.
- Limits = [.1,.1,.1,.785,.785,1.57] ! max x velocity is 100 mm/s, max y deviation is 100 mm, max z deviation is 100 mm, max rx deviation is 45 deg, max ry deviation is 45 deg, max rz deviation is 90 deg.

14.13. force_mode_set_damping(damping)

Sets the damping parameter in force mode.

Parameters

`damping`:

Between 0 and 1, default value is 0.005

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).

14.14. force_mode_set_gain_scaling(scaling)

Scales the gain in force mode.

Parameters

`scaling`:

scaling parameter between 0 and 2, default is 1.

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

The value is stored until this function is called again. Add this to the beginning of your program to ensure it is called before force mode is entered (otherwise default value will be used).

14.15. freedrive_mode (freeAxes=[1, 1, 1, 1, 1, 1], feature=p[0, 0, 0, 0, 0, 0])

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

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

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

Parameters

`freeAxes`: A 6 dimensional vector that contains 0's and 1's, these indicates in which axes movement is allowed. The first three values represents the cartesian directions along x, y, z, and the last three defines the rotation axis, rx, ry, rz. All relative to the selected feature.

`feature`: A pose vector that defines a freedrive frame relative to the base frame. For base and tool reference frames predefined constants "base", and "tool" can be used in place of pose vectors.

Example commands:

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

- Example Parameters:
  - freeAxes = `[0,1,0,0,0,0]` -> The robot is compliant in the y direction relative to the "tool" feature.
  - feature = `"tool"` -> The "tool" feature is located in the active TCP.

Note: Immediately before entering freedrive mode, avoid:

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

High acceleration and deceleration can both decrease the control accuracy and cause protective stops.

14.16. freedrive_mode_no_incorrect_payload_check()

This method, like teach_mode() and freedrive_mode(), changes the robot mode to teach mode, but this function does not check for an incorrect payload during the initial state change, nor if the payload is updated during freedrive. For this reason, it is exceedingly important for users to be certain the payload is correct.

It is possible for the user to exit teach mode/freedrive in the usual manner, using: end_teach_mode() or end_freedrive_mode()

14.17. get_conveyor_tick_count()

*Deprecated:*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

Deprecated: This function is replaced by `encoder_get_tick_count` and it should therefore not be used moving forward.

14.18. get_freedrive_status()

Returns status of freedrive mode for current robot pose.

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

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

Return Value

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

14.19. get_target_tcp_pose_along_path()

Query the target TCP pose as given by the trajectory being followed.

This script function is useful in conjunction with conveyor tracking to know what the target pose of the TCP would be if no offset was applied.

Return Value

Target TCP pose

14.20. get_target_tcp_speed_along_path()

Query the target TCP speed as given by the trajectory being followed.

This script function is useful in conjunction with conveyor tracking to know what the target speed of the TCP would be if no offset was applied.

Return Value

Target TCP speed as a vector

14.21. 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/s^2]

`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

14.22. 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/s^2]

`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.

14.23. optimovej(goal, a=0.3, v=0.3, r=0)

Move to the goal position (linear in joint-space)

OptiMove dynamically adapts speed and acceleration to perform smooth motions using jerk-limited speed profiles. The speed and acceleration parameters control the speed profile of the move. Setting speed and acceleration parameters to 1 causes the fastest cycle time the robot is capable of. This command is similar to movej() but with smoother motions with less vibration.

Parameters

goal: `(q, pose, struct{pose, frame}, string)` - the target for the TCP motion can be defined in different ways:

- (q) as robot joint positions.
- (pose) as a pose in robot base coordinate frame. The target joint positions will be calculated by inverse kinematics.
- (struct{pose, frame}) as a pose and the name of a reference coordinate frame. The goal will be set to this pose in this reference coordinate frame.
- (string) as the name of a world model object. The goal will be set to the object's pose.

`a (optional)`: Joint acceleration as a fraction of what the joints are able to perform - a∈ (0.0,1.0]

`v (optional)`: Joint speed as a fraction of how fast the joints can move during the motion - v∈ (0.0,1.0]

`r (optional)`: Blend radius [m]

If a blend radius is set, the robot arm trajectory will be modified within the blend radius of the destination position. 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 in the log screen.

Example command: `optimovej([0, 1.57, -1.57, 3.14, -1.57, 1.57], a=0.4, v=0.6, r=0.0)`

- Example Parameters:
  - goal = [0, 1.57, -1.57, 3.14, -1.57, 1.57] → joint positions with base at 0 deg rotation, shoulder at 90 deg rotation, elbow at -90 deg rotation, wrist 1 at 180 deg rotation, wrist 2 at -90 deg rotation, wrist 3 at 90 deg rotation.

- a = 0.4 → acceleration at either end of the motion is 40% of the acceleration the robot is capable of producing in the specific joint configuration.
- v = 0.6 → velocity during motion cruise phase is 60% of the velocity the joints can move at.
- r = 0.0 → the blend radius is zero meters, meaning the robot will stop at the waypoint.

Notes:

- The absolute speed and acceleration of the robot depends on the joint configuration during the move. A value of e.g. 0.4 might therefore produce a faster speed/acceleration in one area of the robot's workspace and a slower speed/acceleration in another area of the robot's workspace. Values of 1.0 will always give the highest speed and acceleration that are possible for a given robot path.
- To avoid high accelerations that can cause dropped items in e.g. suction cup grippers, consider using the command `tool_wrench_limit_set()` to limit the acceleration of the items held by the gripper.
- It is possible to blend into this move type from movej/l and optimovej/l. When coming from other movement types the robot should be at standstill when starting the move.

14.24. movel(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/s^2]

`v`: tool speed [m/s]

`t`: time [S]

`r`: blend radius [m]

Example command: `movel(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/s^2
  - 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.

14.25. optimovel(goal, a=0.3, v=0.3, r=0)

Move to the goal position (linear in Cartesian space).

OptiMove dynamically adapts speed and acceleration to perform smooth motions using jerk-limited speed profiles. The speed and acceleration parameters control the speed profile of the move. Setting speed and acceleration parameters to 1 causes the fastest cycle time the robot is capable of.

This command is similar to movel() but with smoother motions with less vibration.

Parameters

goal: `(q, pose, struct{pose, frame}, string)` target for the TCP motion can be defined in different ways:

- (q) as robot joint positions. The target pose will be calculated by forward kinematics.
- (pose) as a pose in robot base coordinate frame. The target joint positions will be calculated by inverse kinematics.
- (struct{pose, frame}) as a pose and the name of a reference coordinate frame. The goal will be set to this pose in this reference coordinate frame.
- (string) as the name of a world model object. The goal will be set to the object's pose.

a (optional): Tool acceleration as a fraction of what the robot is able to perform - a∈ (0.0,1.0]

v (optional): Tool speed as a fraction of the maximum Cartesian velocity the robot can travel at during the trajectory, given the maximum joint speeds - v∈ (0.0,1.0]

r (optional): Blend radius [m]

If a blend radius is set, the robot arm trajectory will be modified within the blend radius of the destination position. 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: `optimovel(pose, a=0.4, v=0.6, r=0.0)`

- Example Parameters:
  - goal = 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 deg, ry = 0 deg, rz = 180 deg.
  - a = 0.4 -> acceleration at either end of the motion is 40% of the acceleration the robot is capable of producing in the specific joint configuration.
  - v = 0.6 -> velocity during motion cruise phase is 60% of the velocity the joints can move at.
  - r = 0.0 -> the blend radius is zero meters, meaning the robot will stop at the waypoint.

Notes:

- The absolute speed and acceleration of the robot depends on the joint configuration during the move. A value of e.g. 0.4 might therefore produce a faster speed/acceleration in one area of the robot's workspace and a slower speed/acceleration in another area of the robot's workspace (typically close to singularities). Values of 1.0 will always give the highest speed and acceleration that are possible for a given robot path.
- To avoid high accelerations that can cause dropped items in e.g. suction cup grippers, consider using the command `tool_wrench_limit_set()` to limit the acceleration of the items held by the

gripper.

- It is possible to blend into this move type from movej/l and optimovej/l. When coming from other movement types the robot should be at standstill when starting the move.

14.26. 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 mm, y = 300 mm, z = 500 mm, rx = 0, ry = 0, rz = 180 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.

14.27. path_offset_disable(a=20)

Disable the path offsetting and decelerate all joints to zero speed.

Uses the `stopj` functionality to bring all joints to a rest. Therefore, all joints will decelerate at different rates but reach stand-still at the same time.

Use the script function `path_offset_enable` to enable path offsetting

Parameters

`a`: joint acceleration [rad/s^2] (optional)

14.28. path_offset_enable()

Enable path offsetting.

Path offsetting is used to superimpose a Cartesian offset onto the robot motion as it follows a trajectory. This is useful for instance for imposing a weaving motion onto a welding task, or to compensate for the effect of moving the base of the robot while following a trajectory.

Path offsets can be applied in various frames of reference and in various ways. Please refer to the script function path_offset_set for further explanation.

Enabling path offsetting doesn't cancel the effects of previous calls to the script functions `path_offset_set_max_offset` and `path_offset_set_alpha_filter`. Path offset configuration will persist through cycles of enable and disable.

Using Path offset at the same time as Conveyor Tracking and/or Force can lead to program conflict. Do not use this function togther with Conveyor Tracking and/or Force.

14.29. path_offset_get(type)

Query the offset currently applied.

Parameters

`type`: Specifies the frame of reference of the returned offset. Please refer to the `path_offset_set` script function for a definition of the possible values and their meaning.

Return Value

Pose specifying the translational and rotational offset. Units are meters and radians.

14.30. path_offset_set(offset, type)

path_offset_set(offset, type)

Specify the Cartesian path offset to be applied.

Use the script function path_offset_enable beforehand to enable offsetting. The calculated offset is applied during each cycle at 500Hz.

Discontinuous or jerky offsets are likely to cause protective stops. If offsets are not smooth the function path_offset_set_alpha_filter can be used to engage a simple filter.

The following example uses a harmonic wave (cosine) to offset the position of the TCP along the Z-axis of the robot base:

```
>>> thread OffsetThread():
>>> while(True):
>>> # 2Hz cosine wave with an amplitude of 5mm
>>> global x = 0.005*(cos(p) - 1)
>>> global p = p + 4*3.14159/500
>>> path_offset_set([0,0,x,0,0,0], 1)
>>> sync()
```

>>> end

>>> end

Parameters

offset: Pose specifying the translational and rotational offset.

type: Specifies how to apply the given offset. Options are:

1: (BASE) Use robot base coordinates when applying.

2: (TCP) Use robot TCP coordinates when applying.

3: (MOTION) Use a coordinate system following the un-offset trajectory when applying. This coordinate system is defined as follows. X-axis along the tangent of the translational part of the un-offset trajectory (rotation not relevant here). Y-axis perpendicular to the X-axis above and the Z-axis of the tool (X cross Z). Z-axis given from the X and Y axes by observing the right-hand rule. This is useful for instance for superimposing a weaving pattern onto the trajectory when welding.

4: (WORLD) This can be used to follow a trajectory in world (inertial) space, while the base coordinate system of the robot is being translated and/or rotated by something external, e.g. a mobile robot or another robot arm. The offset is thus the pose of the robot base relative to the world coordinate system, and it is also the world coordinate system which the commanded trajectory should be understood relative to.

14.31. path_offset_set_alpha_filter(alpha)

Engage offset filtering using a simple alpha filter (EWMA) and set the filter coefficient.

When applying an offset, it must have a smooth velocity profile in order for the robot to be able to follow the offset trajectory. This can potentially be cumbersome to obtain, not least as offset application starts, unless filtering is applied.

The alpha filter is a very simple 1st order IIR filter using a weighted sum of the commanded offset and the previously applied offset: filtered_offset= alpha*offset+ (1-alpha)*filtered_offset.

See more details and examples in the UR Support Site: Modify Robot Trajectory

Parameters

alpha: The filter coefficient to be used - must be between 0 and 1.
A value of 1 is equivalent to no filtering.

For welding; experiments have shown that a value around 0.1 is a good compromise between robustness and offsetting accuracy.

The necessary alpha value will depend on robot calibration, robot mounting, payload mass, payload center of gravity, TCP offset, robot position in workspace, path offset rate of change and underlying motion.

14.32. path_offset_set_max_offset(transLimit, rotLimit)

Set limits for the maximum allowed offset.

Due to safety and due to the finite reach of the robot, path offsetting limits the magnitude of the offset to be applied. Use this function to adjust these limits. Per default limits of 0.1 meters and 30 degrees (0.52 radians) are used.

Parameters

`transLimit`: The maximum allowed translational offset distance along any axis in meters.

`rotLimit`: The maximum allowed rotational offset around any axis in radians.

14.33. pause_on_error_code(code, argument)

Makes the robot pause if the specified error code occurs. The robot will only pause during program execution.

This setting is reset when the program is stopped. Call the command again before/during program execution to re-enable it.

```
>>> pause_on_error_code(173, 3)
```

In the above example, the robot will pause on errors with code 173 if its argument equals 3 (corresponding to 'C173A3' in the log).

```
>>> pause_on_error_code(173)
```

In the above example, the robot will pause on error code 173 for any argument value.

Parameters

`code`: The code of the error for which the robot should pause (int)

`argument`: The argument of the error. If this parameter is omitted the robot will pause on any argument for the specified error code (int)

Notes:

- Error codes appear in the log as CxAy where 'x' is the code and 'y' is the argument.

14.34. 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.

14.35. 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 limitted 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.

14.36. screw_driving(f, v_limit)

Enter screw driving mode. The robot will exert a force in the TCP Z-axis direction at limited speed. This allows the robot to follow the screw during tightening/loosening operations.

Parameters

`f`: The amount of force the robot will exert along the TCP Z-axis (Newtons).

`v_limit`: Maximum TCP velocity along the Z axis (m/s).

Notes:

Zero the F/T sensor without the screw driver pushing against the screw.

Call `end_screw_driving` when the screw driving operation has completed.

>>> def testScrewDriver():

>>> # Zero F/T sensor

```
>>> zero_ftsensor()
>>> sleep(0.02)
>>>
>>> # Move the robot to the tightening position
>>> # (i.e. just before contact with the screw)
>>> ...
>>>
>>> # Start following the screw while tightening
>>> screw_driving(5.0, 0.1)
>>>
>>> # Wait until screw driver reports OK or NOK
>>> ...
>>>
>>> # Exit screw driving mode
>>> end_screw_driving()
>>> end
```

14.37. servoj(q, a, v, t=0.002, lookahead_time=0.1, gain=300)

Servoj can be used for online realtime control of joint positions.

The gain parameter works the same way as the P-term of a PID controller, where it adjusts the current position towards the desired (q). The higher the gain, the faster reaction the robot will have.

The parameter lookahead_time is used to project the current position forward in time with the current velocity. A low value gives fast reaction, a high value prevents overshoot.

Note: A high gain or a short lookahead time may cause instability and vibrations. Especially if the target positions are noisy or updated at a low frequency

It is preferred to call this function with a new setpoint (q) in each time step (thus the default t=0.002)

You can combine with the script command get_inverse_kin() to perform servoing based on cartesian positions:

```
>>> q = get_inverse_kin(x)
>>> servoj(q, lookahead_time=0.05, gain=500)
```

Here x is a pose variable with target cartesian positions, received over a socket or RTDE registers.

Example command: `servoj([0.0,1.57,-1.57,0,0,3.14], 0, 0, 0.002, 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 = 0.002 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] smoothens the trajectory with this lookahead time
  - gain = 300 proportional gain for following target position, range [100,2000]

14.38. set_conveyor_tick_count(tick_count, absolute_encoder_resolution=0)

Deprecated: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

`tick_count:`

Tick count of the conveyor (Integer)

`absolute_encoder_resolution:`

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]

Deprecated: This function is replaced by encoder_set_tick_count and it should therefore not be used moving forward.

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)

14.39. 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

14.40. 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.

14.41. 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/s^2] (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/s^2 -> acceleration of the leading axis (shoulder in this case)
  - t = 0.5 s -> time before the function returns

14.42. speedl(xd, a, t, aRot='a')

Cartecian velocity control

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 positional acceleration [m/s^2]

`t`: time [s] before function returns (optional)

`aRot`: tool rotational acceleration [rad/s^2] (optional). If not defined, position acceleration value in [rad/s^2] will be used

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

- Example Parameters:
  - xd -> Tool speeds of: x=500 mm/s, y=400 mm/s, rx=90 deg/s, ry and rz=0 deg/s
  - a = 0.5 m/s^2 -> acceleration of the tool
  - t = 0.5 s -> time before the function returns

14.43. stop_conveyor_tracking(a=20)

Stop tracking the conveyor, started by track_conveyor_linear() or track_conveyor_circular(), and decelerate all joint speeds to zero.

Parameters

`a`: joint acceleration [rad/s^2] (optional)

Example command: `stop_conveyor_tracking(a=15)`

- Example Parameters:
  - a = 15 rad/s^2 -> acceleration of the joints

14.44. stopj(a)

Stop (linear in joint space)

Decelerate joint speeds to zero

Parameters

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

Example command: `stopj(2)`

- Example Parameters:
  - a = 2 rad/s^2 -> rate of deceleration of the leading axis.

14.45. stopl(a, aRot='a')

Stop (linear in tool space)

Decelerate tool speed to zero

Parameters

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

`aRot`: tool acceleration [rad/s^2] (optional), if not defined a, position acceleration, is used

Example command: `stopl(20)`

- Example Parameters:
  - a = 20 m/s^2 -> rate of deceleration of the tool
  - aRot -> tool deceleration [rad/s^2] (optional), if not defined, position acceleration, is used. i.e. it supersedes the "a" deceleration.

14.46. tool_wrench_limit_set(frame_offset, Fx, Fy, Fz, Mx, My, Mz)

Limit the wrench (forces and torques) caused by motion of the robot in a frame given relative to the tool flange. The wrench is limited in normal and reduced mode operation, as well as during protective stops, safeguard stops, 3PE stops and emergency stops. For this reason, it can affect robot motion speed to ensure adherence to safety limits. Usage can help prevent dropping items by limiting accelerations as well as reducing wrench applied to the attached tool.

This limitation does not affect the forces and torques that can be applied in force control.

Parameters:

`frame_offset`: Pose specifying frame relative to the tool flange similarly to how the TCP offset is specified. The first three coordinates specify translational offset along the x- y- and z-axis in meters. The last three specify the rotational offset using the axis-angle representation in radians.

`Fx (optional)`: Float, setting maximum acceleration force along the X-axis in the specified frame.

`Fy (optional)`: Float, setting maximum acceleration force along the Y-axis in the specified frame.

`Fz (optional)`: Float, setting maximum acceleration force along the Z-axis in the specified frame.

`Mx (optional)`: Float, setting maximum acceleration torque around the X-axis in the specified frame.

`My (optional)`: Float, setting maximum acceleration torque around the Y-axis in the specified frame.

`Mz (optional)`: Float, setting maximum acceleration torque around the Z-axis in the specified frame.

Any optional parameter not specified means the axis is only limited by standard robot limitations.

Example command: tool_wrench_limit_set `(p[0, 0, 0.1, 0, 0, 1.57], Mx=10, My=15)`

Example Parameters:

- frame_offset = p[0, 0, 0.1, 0, 0, 1.57] → limitation will be applied in a frame offset 10 cm in front of the tool flange rotated by 45 degrees around the axis of displacement.
- Mx = 10 → acceleration torque will be limited to 10 Nm around the X-axis in the specified frame.
- My = 15 → acceleration torque will be limited to 15 Nm around the Y-axis in the specified frame.

Remaining forces and torques will not be limited by this algorithm.

Example command: tool_wrench_limit_set(get_tcp_offset(), Fz=50)

Example Parameters:

- frame_offset is equal to the active TCP offset.
- Fz = 50 → acceleration force is limited such that force does not exceed 50 N along the Z-axis in the TCP frame.

NOTICE

The set limit is persisted until shutdown of the controller or until explicitly disabled by executing tool_wrench_limit_disable().

14.47. tool_wrench_limit_disable()

Disable tool wrench limitation set by tool_wrench_limit_set.

14.48. teach_mode()

Deprecated:

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.

Deprecated:

This function is replaced by `freedrive_mode` and it should therefore not be used moving forward.

14.49. track_conveyor_circular(center, ticks_per_revolution, rotate_tool='False', encoder_index=0)

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 coneyor or stay in the orientation specified by the trajectory (movel() etc.).

`encoder_index`: The index of the encoder to associate with the conveyor tracking. Must be either 0 or 1. This is an optional argument, and please note the default of 0. The ability to omit this argument will allow existing programs to keep working. Also, in use cases where there is just one conveyor to track consider leaving this argument out.

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 (movel() etc.).

14.50. track_conveyor_linear(direction, ticks_per_meter, encoder_index=0)

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

`encoder_index`: The index of the encoder to associate with the conveyor tracking. Must be either 0 or 1. This is an optional argument, and please note the default of 0. The ability to omit this argument will allow existing programs to keep working. Also, in use cases where there is just one conveyor to track consider leaving this argument out.

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.

15. Module internals

15.1. 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 Newton (float)

Note: Refer to force_mode() for taring the sensor.

15.2. estimate_payload(poses, wrenches)

Parameters

- poses - A list of at least four TCP poses. The orientation of the poses should be as varied as possible, in order to get a good estimate.

  If the rotational distance between any two poses is less than Pi / (2*n) radians, where n is the number of poses in poses, an exception will be thrown.
- TCP poses can be recorded with get_actual_tcp_pose().

  wrenches - A list of wrenches resulting from gravity acting on the payload. Must have the same length as poses. Each wrench in wrenches should be measured at the corresponding pose in poses. The wrenches must be given at the tool flange but in robot base orientation. Wrenches in the required orientation can be recorded with get_tcp_force() when in the desired pose.

Return value:

struct[mass, cog]

mass is a double representing the weight of the payload in kg.

cog is a 3d vector representing the offset from the tool flange to the payload center of gravity in tool frame in meters.

Example usage (question)

```
repeat n times
    movej(*some distinct pose*)
    sleep(*long enough for the arm to stabilize*)
    pose_list.append(get_tcp_force)
    wrench_list.append(get_actual_tcp_pose)
payload = estimate_payload(pose_list, wrench_list)
set_payload(payload.mass, payload.cog)
```

15.3. get_actual_joint_positions()

Returns the actual angular positions read by the joint encoders

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]

15.4. get_actual_joint_positions_history(steps=0)

Returns the actual past angular positions of all joints

This function returns the angular positions as reported by the function "`get_actual_joint_positions()`" which indicates the number of controller time steps occurring before the current time step.

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

Parameters

`steps`: The number of controller time steps required to go back. 0 corresponds to "`get_actual_joint_positions()`"

Return Value

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

15.5. 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]

15.6. 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]

15.7. get_actual_tcp_speed()

Returns the current measured TCP speed

The speed of the TCP retuned 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]

15.8. 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.

15.9. get_base_acceleration()

Returns the robot base acceleration vector (see set_base_acceleration) currently active in the controller and SCB kinematics and dynamics models.

Return Value

User specified robot base acceleration vector in m/s^2 as a 3D vector ([float, float, float])

15.10. 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)

15.11. 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.

15.12. get_gravity()

Returns the gravity acceleration vector (see set_gravity) currently active in the controller and SCB kinematics and dynamics models.

Return Value

User specified gravity acceleration vector in m/s^2 as a 3D vector ([float, float, float])

15.13. 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 is used.

Parameters

`x`: tool pose

`qnear`: list of joint positions (Optional)

`maxPositionError`: the maximum allowed position error (Optional)

`maxOrientationError`: the maximum allowed orientation error (Optional)

`tcp`: tcp 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=100mm, y=200mm, z=200mm and rotation vector of rx=0 deg., ry=180 deg, rz=0 deg.
  - qnear = [0.,3.14,1.57,.785,0,0] -> solution should be near to joint angles of j0=0 deg, j1=180 deg, j2=90 deg, j3=45 deg, j4=0 deg, j5=0 deg.
  - maxPositionError is by default 1e-10 m
  - maxOrientationError is by default 1e-10 rad

15.14. get_inverse_kin_has_solution(pose, qnear, maxPositionError=1E-10, maxOrientationError=1e-10, tcp="active_tcp")

Check if `get_inverse_kin` has a solution and return boolean (True) or (False).

This can be used to avoid the runtime exception of `get_inverse_kin` when no solution exists.

Parameters

`pose`: tool pose

`qnear`: list of joint positions (Optional)

`maxPositionError`: the maximum allowed position error (Optional)

`maxOrientationError`: the maximum allowed orientation error (Optional)

`tcp`: tcp offset pose (Optional)

Return Value

True if `get_inverse_kin` has a solution, False otherwise (bool)

15.15. 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)

15.16. 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]

15.17. get_steptime()

Returns the duration of the robot time step in seconds.

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

Return Value

duration of the robot step in seconds

15.18. get_target_joint_positions()

Returns the desired angular positions that are sent to all the joints at each time step

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]

15.19. 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]

15.20. get_target_payload()

Returns the weight of the active payload

Return Value

The weight of the current payload in kilograms

15.21. get_target_payload_cog()

Retrieve the Center Of Gravity (COG) coordinates of the active payload.

This scripts returns the COG coordinates of the active payload, with respect to the tool flange

Return Value

The 3d coordinates of the COG [CoGx, CoGy, CoGz] in meters

15.22. get_target_payload_inertia()

Returns the most recently set payload inertia matrix.

This script function returns the inertia matrix of the active payload in tool flange coordinates, with origin at the CoG.

Return Value

The six dimensional coordinates of the payload inertia matrix [Ixx, Iyy, Izz, Ixy, Ixz, Iyz] expressed in kg*m^2.

15.23. 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]

15.24. 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 |rz,ry,rz| defines the angular velocity in radians/s.

Return Value

The TCP speed (pose)

15.25. get_target_waypoint()

Returns the target waypoint of the active move

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

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

Return Value

The desired waypoint TCP vector [X, Y, Z, Rx, Ry, Rz]

15.26. get_tcp_force()

Returns the force/torque vector at the tool flange.

The function returns p[Fx(N), Fy(N), Fz(N), TRx(Nm), TRy(Nm), TRz(Nm)] where the forces: Fx, Fy, and Fz in Newtons and the torques: TRx, TRy and TRz in Newtonmeters are all measured at the tool flange with the orientation of the robot base coordinate system.

Return Value

The force/torque vector

Note:

Refer to zero_ftsensor() for taring sensor.

Example:

```
def get_wrench_at_tool_flange():
     ft = get_tcp_force()
     t_flange_in_base
 = pose_trans(get_target_tcp_pose(), pose_inv(get_tcp_offset())) flange_rot
= pose_inv(p[0, 0, 0, t_flange_in_base[3], t_flange_in_base[4], t_flange_in_
base[5]])
     f = pose_trans(flange_rot, p[ft[0], ft[1], ft[2], 0, 0, 0])
     t = pose_trans(flange_rot, p[ft[3], ft[4], ft[5], 0, 0, 0])
     return [f[0], f[1], f[2], t[0], t[1], t[2]]
end
def get_wrench_at_tcp():
     return wrench_trans(get_tcp_offset(), get_wrench_at_tool_flange())
end
```

15.27. 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

15.28. 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 composant of the measured acceleration in SI-units (m/s^2).

15.29. get_tool_current()

Returns the tool current

The tool current consumption measured in ampere.

Return Value

The tool current in ampere.

15.30. get_tool_temp()

Returns the most recently measured temperature of the tool.

Return Value

Measured tool temperature in degrees Celcius (float)

15.31. high_holding_torque_disable()

Disables automatically applying high hold torque when the robot is stationary, which is the default behavior. The UR controller automatically applies high holding torque when the following is true:

- The program state is PROGRAM_STATE_RUNNING
- All actual joint movement <= 0.01 rad/s
- All target joint velocities == 0

Parameters:

None

Example command:

```
high_holding_torque_disable()
```

This function script disables the high holding torque behavior. Note that the default is restored to enabled after restarting the controller.

See also:

```
high_holding_torque_enable()
```

Applications: Disable high holding torque if you want a stationary robot to issue a protective stop when colling with an object. For example, if the robot is being transported on a linear rail or vertical lift. However, if just the base of the robot collides with an object while moving, the protective stop may not be issued. Adequate safety precautions should be put in place to guard against this situation.

15.32. high_holding_torque_enable()

Enables high hold torque when the robot is stationary. This function is used to reverse the behavior of the `high_holding_torque_disable()` command.

Parameters:

None

Example command:

```
high_holding_torque_enable()
```

See also

```
high_holding_torque_disable()
```

15.33. is_steady()

The function will return true when the robot has been standing still with zero target velocity for 500ms

When the function returns true, the robot is able to adapt to large external forces and torques, e.g. from screwdrivers, without issuing a protective stop.

Return Value

True when the robot able to adapt to external forces, false otherwise (bool)

15.34. is_within_safety_limits(position, qNear=current joint configuration)

Checks if the given pose or joint positions are reachable and within the currently active safety limits of the robot.

This check considers:

- Joint position limits
- Safety planes
- Tool orientation limit
- Physical range of the robot

Parameters

`position`: Pose or joint positions. When a pose is provided, it is recommended to also supply qNear to ensure that the correct inverse kinematics solution is checked.

`qNear`: List of joint angles (optional). Only used for calculating inverse kinematics when position is a pose. If not specified, the current joint positions are used.

Return Value

True if within limits, false otherwise (bool).

NOTICE

In order to simply check if a pose is physically reachable by the robot, use get_inverse_kin_has_solution instead.

15.35. 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.

15.36. powerdown()

Shut down the robot, and power off the robot and controller.

15.37. protective_stop()

Trigger a protective stop, pausing the program and stopping motion on the planned trajectory.

Notes:

This function is not intended for use for simply pausing the running program (see Special keywords: `pause`).

15.38. set_base_acceleration(a)

Sets the acceleration of the robot base. This function is used when the robot is attached to a moving base such a linear rail or vertical lift. Specifying the base acceleration is used to prevent premature protective stops by informing the control system that forces are being exerted on the robot through acceleration of the base.

Parameters

a: the linear acceleration of the base in x, y, z directions

Example command:

set_base_acceleration([0.10 0.0 0.0])

Example Parameters:

a = [0.10 0 0] specifies acceleration in the linear X direction of 0.10 m/s²

15.39. set_baselight_off()

NOTICE
Only applies to UR20 / UR30

Turns the baselight completely off.

15.40. set_baselight_iec()

NOTICE
Only applies to UR20 / UR30

Make the baselight comply to the IEC 60204-1 standard and indicate whether the robot is in a safety stop, in freedrive as well as the operational mode. Further details of the colors can be seen in the UR20 manual.

15.41. set_baselight_solid(r,g,b)

NOTICE

Only applies to UR20 / UR30

Set a color on the entire baselight ring as specified by the given RGB values in the range 0-255.

15.42. set_gravity(d)

Set the direction of the acceleration experienced by the robot. When the robot mounting is fixed, this corresponds to an accleration 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/s^2 (1g).

15.43. set_payload(m, cog)

Parameters

`m`: mass in kilograms

`cog`: Center of Gravity, a vector [CoGx, CoGy, CoGz] specifying the displacement (in meters) from the toolmount.

Deprecated: See set_target_payload to set mass, CoG and payload inertia matrix at the same time.

Set payload mass and center of gravity while resetting payload inertia matrix

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

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

Note: The force torque measurements are automatically zeroed when setting the payload. That ensures the readings are compensated for the payload. This is similar to the behavior of `zero_ftsensor()`

Warnings:

- This script is **deprecated** since SW 5.10.0 because of the risk of inconsistent payload parameters. Use the `set_target_payload` instead to set mass, CoG and inertia matrix.
- Omitting the cog parameter is **not recommended**. The Tool Center Point (TCP) will be used if the cog parameter is missing with the side effect that later calls to `set_tcp` will change also the CoG to the new TCP. Use the `set_payload_mass` function to change only the mass or use the `get_target_payload_cog` as second argument to not change the CoG.
- Using this script function to modify payload parameters will reset the payload inertia matrix.

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=300 mm from the center of the tool mount in tool coordinates
- `set_payload(2.5, get_target_payload_cog())`
  - Example Parameters:
    - m = 2.5 → mass is set to 2.5 kg payload
    - cog = use the current COG setting

15.44. set_payload_cog(CoG)

Deprecated: See `set_target_payload` to set mass, CoG and payload inertia matrix at the same time.

Set the Center of Gravity (CoG) and reset payload inertia matrix

Warning: Using this script function to modify payload parameters will reset the payload inertia matrix.

Note: The force torque measurements are automatically zeroed when setting the payload. That ensures the readings are compensated for the payload. This is similar to the behavior of `zero_ftsensor()`

15.45. set_payload_mass(m)

Parameters

`m`: mass in kilograms

Deprecated: See `set_target_payload` to set mass, CoG and payload inertia matrix at the same time.

Set payload mass and reset payload inertia matrix

See also `set_payload`.

Sets the mass of the payload and leaves the center of gravity (CoG) unchanged.

Note: The force torque measurements are automatically zeroed when setting the payload. That ensures the readings are compensated for the payload. This is similar to the behavior of `zero_ftsensor()`

Warnings:

- This script is **deprecated** since SW 5.10.0 because of the risk of inconsistent payload parameters. Use the `set_target_payload` instead to set mass, CoG and inertia matrix.
- Using this script function to modify payload parameters will reset the payload inertia matrix.

15.46. set_target_payload(m, cog, inertia=[0, 0, 0, 0, 0, 0], transition_time=0)

Sets the mass, CoG (center of gravity), the inertia matrix of the active payload and the transition time for applying new settings.

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

Parameters

`m`: mass in kilograms.

`cog`: Center of Gravity, a vector with three elements [CoGx, CoGy, CoGz] specifying the offset (in meters) from the tool mount.

`inertia`: payload inertia matrix (in kg*m^2), as a vector with six elements [Ixx, Iyy, Izz, Ixy, Ixz, Iyz] with origin in the CoG and the axes aligned with the tool flange axes.

`transition_time`: the duration of the payload property changes in seconds

Notes:

- This script should be used instead of the deprecated `set_payload`, `set_payload_mass`, and `set_payload_cog`.
- The payload mass and CoG are required, the inertia matrix and transition time are optional.
  - When inertia matrix is left out, a zero matrix will be used.
  - When transition time is left out, the change will be applied instantaneously, identically to how the deprecated `set_payload`, `set_payload_mass` and `set_payload_cog` work.
- The maximum value allowed for each of the components of the inertia matrix is +/- 133 kg*m^2. An exception is thrown if limits are exceeded.
- The first three elements of the inertia matrix (i.e. Ixx, Iyy, Izz) cannot be negative. An exception is thrown if either value is negative.
- The force/torque measurements are automatically zeroed when setting the payload. That ensures the readings are compensated for the payload. This is similar to the behaviour of `zero_ftsensor()`.
- Setting a transition time larger than zero avoids the robot doing a small "jump" when payload changes. This is useful when picking up or releasing heavy objects.
- The internal force/torque sensor in the robot tool is reset each time the payload is updated. This means that the final reset will be performed at the end of the payload transition time. If the payload is being accelerated at the time of the final reset, the force/torque measurement will be affected. It's always recommended to call `zero_ftsensor()` to reset the force/torque sensor before using it, e.g. in Force Mode.

15.47. set_tcp(pose, tcp_name="")

Sets the active tcp offset, i.e., the transformation from the output flange coordinate system to the TCP as a pose, and assigns a name to the TCP. If no name is provided, the default name is an empty string.

Parameters

- `pose`: A pose describing the transformation.
- `tcp_name (optional, default="")`: A string that assigns a name to the TCP.

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

- 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.
  - tcp_name = "custom_tcp_name" -> the name assigned to the TCP.

15.48. sleep(t)

Sleep for an amount of time

Parameters

`t`: time [s]

Example command: `sleep(3.)`

- Example Parameters:
  - t = 3. -> time to sleep

15.49. time(mode=0)

Get current time from selected source.

Parameters

`mode`: integer, one of:

- 0: Controller execution time. Time counted since low level controller start. Guaranteed monotonic. Reset on robot restart.
- 1: Reserved.
- 2: System time in GMT time zone. Not guaranteed to be monotonic - can go backwards when system time is adjusted.

Return

Function retruns structure in format struct(sec, nanosec)

Example 1: Get seconds part of current time counted since low level controller start.

```
current_time_s = time().sec
```

Example 2: Get current system time in seconds including fraction of second.

```
t = time() global current_time_s = t.sec + t.nanosec / 1000000000
```

Example 3: Get current date derived from system clock. NOTE: time(2) function returns GMT time.

```
# Converts seconds since 1970.01.01 to date
# Based on http://howardhinnant.github.io/date_algorithms.html
# Returns:
# struct(year, month, day)
def seconds_to_date(z):
      local d = struct(year = 0, month = 0, day = 0)
      z = floor(z / 86400)
      z = z + 719468
      local era = floor(z/146097)
      local doe = z - era * 146097
      local yoe = floor((doe - floor(doe/1460) + floor(doe/36524) - floor
(doe/146096)) / 365)
      d.year = yoe + era * 400
      local doy = doe - (365*yoe + floor(yoe/4) - floor(yoe/100))
      local mp = floor((5*doy + 2)/153)
      d.day = doy - floor((153*mp + 2)/5) + 1
      if(mp < 10):
            d.month = mp + 3
      else:
            d.month = mp - 9
      end
      if(d.month <= 2):
            d.year = d.year + 1
end
return d
end

date_gmt = seconds_to_date(time(2).sec)
```

Example 4: Get current GMT time derived from system clock.

```
Converts seconds since 1970.01.01 to time of day
# Returns:
# struct(hour, minute, second)
def seconds_to_time(z):
```

```
      local t = struct(hour = 0, minute = 0, second = 0)
      local sod = z % 86400
      t.hour = floor(sod / 3600)
      t.minute = floor((sod - t.hour * 3600) / 60)
      t.second = sod % 60
      return t
end
time_gmt = seconds_to_time(time(2).sec)
```

15.50. str_at(src, index)

Provides direct access to the bytes of a string.

This script returns a string containing the byte in the source string at the position corresponding to the specified index. It may not correspond to an actual character in case of strings with special encoded character (i.e. multi-byte or variable-length encoding)

The string is zero-indexed.

Parameters

`src`: source string.

`index`: integer specifying the position inside the source string.

Return Value

String containing the byte at position `index` in the source string. An exception is raised if the index is not valid.

Example command:

- `str_at("Hello", 0)`
  - returns "H"
- `str_at("Hello", 1)`
  - returns "e"
- `str_at("Hello", 10)`
  - error (index out of bound)
- `str_at("", 0)`
  - error (source string is empty)

15.51. str_cat(op1, op2)

String concatenation

This script returns a string that is the concatenation of the two operands given as input. Both operands can be one of the following types: String, Boolean, Integer, Float, Pose, List of Boolean / Integer / Float / Pose. Any other type will raise an exception.

The resulting string cannot exceed 1023 characters, an exception is thrown otherwise.

Float numbers will be formatted with 6 decimals, and trailing zeros will be removed.

The function can be nested to create complex strings (see last example).

Parameters

`op1`: first operand

`op2`: second operand

Return Value

String concatenation of op1 and op2

Example command:

- `str_cat("Hello", " World!")`
  - returns "Hello World!"
- `str_cat("Integer ", 1)`
  - returns "Integer 1"
- `str_cat("", p[1.0, 2.0, 3.0, 4.0, 5.0, 6.0])`
  - returns "p[1, 2, 3, 4, 5, 6]"
- `str_cat([True, False, True], [1, 0, 1])`
  - returns "[True, False, True][1, 0, 1]"
- `str_cat(str_cat("", str_cat("One", "Two")),str_cat(3, 4))`
  - returns "OneTwo34"

15.52. str_empty(str)

Returns true when `str` is empty, false otherwise.

Parameters

`str`: source string.

Return Value

True if the string is empty, false otherwise

Example command:

- `str_empty("")`
  - returns True
- `str_empty("Hello")`
  - returns False

15.53. str_find(src, target, start_from=0)

Finds the first occurrence of the substring target in `src`.

This script returns the index (i.e. byte) of the the first occurrence of substring `target` in `str`, starting from the given (optional) position.

The result may not correspond to the actual position of the first character of target in case `src` contains multi-byte or variable-length encoded characters.

The string is zero-indexed.

Parameters

`src`: source string.

`target`: substring to search.

`start_from`: optional starting position (default 0).

Return Value

The index of the first occurrence of target in `src`, -1 if `target` is not found in `src`.

Example command:

- `str_find("Hello World!", "o")`
  - returns 4
- `str_find("Hello World!", "lo")`
  - returns 3
- `str_find("Hello World!", "o", 5)`
  - returns 7
- `str_find("abc", "z")`
  - returns -1

15.54. str_len(str)

Returns the number of bytes in a string.

Please not that the value returned may not correspond to the actual number of characters in sequences of multi-byte or variable-length encoded characters.

The string is zero-indexed.

Parameters

`str`: source string.

Return Value

The number of bytes in the input string.

Example command:

- `str_len("Hello")`
  - returns 5
- `str_len("")`
  - returns 0

15.55. str_sub(src, index, len)

Returns a substring of `src`.

The result is the substring of `src` that starts at the byte specified by index with length of at most `len` bytes. If the requested substring extends past the end of the original string (i.e. `index + len > src length`), the length of the resulting substring is limited to the size of `src`.

An exception is thrown in case `index` and/or `len` are out of bounds. The string is zero-indexed.

Parameters

`src`: source string.

`index`: integer value specifying the initial byte in the range [0, src length]

`len`: (optional) length of the substring in the range [0, MAX_INT]. If len is not specified, the string in the range [index, src length].

Return Value

the portion of src that starts at byte index and spans len characters.

Example command:

- `str_sub("0123456789abcdefghij", 5, 3)`
  - returns "567"
- `str_sub("0123456789abcdefghij", 10)`
  - returns "abcdefghij"
- `str_sub("0123456789abcdefghij", 2, 0)`
  - returns "" (len is 0)
- `str_sub("abcde", 2, 50)`
  - returns "cde"
- `str_sub("abcde", -5, 50)`
  - error: index is out of bounds

15.56. sync()

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

15.57. textmsg(s1, s2='')

Send text message to log

Send message with s1 and s2 concatenated to be shown on the PolyScope log-tab.

The PolyScope log-tab is intended for general application status.

It is not recommended to add many messages at a high rate.

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"

15.58. to_num(str)

Converts a string to a number.

`to_num` returns an integer or a float depending on the presence of a decimal point in the input string. Only '.' is recognized as decimal point independent of locale settings.

Valid strings can contains optional leading white space(s) followed by an optional plus ('+') or minus sign ('-') and then one of the following:

(i) A decimal number consisting of a sequence of decimal digits (e.g. 10, -5), an optional '.' to indicate a float number (e.g. 1.5234, -2.0, .36) and a optional decimal exponent that indicates multiplication by a power of 10 (e.g. 10e3, 2.5E-5, -5e-4).

(ii) A hexadecimal number consisting of "0x" or "0X" followed by a nonempty sequence of hexadecimal digits (e.g. "0X3A", "0xb5").

(iii) An infinity (either "INF" or "INFINITY", case insensitive)

(iv) A Not-a-Number ("NAN", case insensitive)

Runtime exceptions are raised if the source string doesn't contain a valid number or the result is out of range for the resulting type.

Parameters

`str`: string to convert

Return Value

Integer or float number according to the input string.

Example command:

- `to_num("10")`
  - returns 10 //integer
- `to_num("3.14")`
  - returns 3.14 //float
- `to_num("-3.0e5")`
  - returns -3.0e5 //float due to '.' in the input string
- `to_num("+5.")`
  - returns 5.0 //float due to '.' in the input string
- `to_num("123abc")`
  - error string doesn't contain a valid number

15.59. to_str(val)

Gets string representation of a value.

This script converts a value of type Boolean, Integer, Float, Pose (or a list of those types) to a string.

The resulting string cannot exceed 1023 characters.

Float numbers will be formatted with 6 decimals, and trailing zeros will be removed.

Parameters

`val`: value to convert

Return Value

The string representation of the given value.

Example command:

- `to_str(10)`
  - returns "10"
- `to_str(2.123456123456)`
  - returns "2.123456"
- `to_str(p[1.0, 2.0, 3.0, 4.0, 5.0, 6.0])`
  - returns "p[1, 2, 3, 4, 5, 6]"
- `to_str([True, False, True])`
  - returns "[True, False, True]"

15.60. tool_contact(direction)

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

Parameters

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

Return Value

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

15.61. tool_contact_examples()

Example of usage in conjunction with the "get_actual_joint_positions_history()" function to allow the robot to retract to the initial point of contact:

```
>>> def testToolContact():
>>> while True:
>>> step_back = tool_contact()
>>> if step_back <= 0:
>>> # Continue moving with 100mm/s
>>> speedl([0,0,-0.100,0,0,0], 0.5, t=get_steptime())
>>> else:
>>> # Contact detected!
>>> # Get q for when the contact was first seen
>>> q = get_actual_joint_positions_history(step_back)
>>> # Stop the movement
>>> stopl(3)
>>> # Move to the initial contact point
>>> movel(q)
>>> break
>>> end
>>> end
>>> end
```

Example command: `tool_contact(direction = get_target_tcp_speed())`

- Example Parameters:
  - direction=get_target_tcp_speed() will detect contacts in the direction of TCP movement

`tool_contact(direction = [1,0,0,0,0,0])`

- Example Parameters:
  - direction=[1,0,0,0,0,0] will detect contacts in the direction robot base X

16. Module urmath

16.1. 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

16.2. 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

16.3. 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

16.4. 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 atan(.5/.5) = .785

16.5. 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:
  - I represents the binary values 1001
    - Returns 9

16.6. 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

16.7. 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

16.8. 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

Example command: `d2r(90)`

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

16.9. 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

16.10. make_list(length, initial_value, capacity=length)

Create a new list of length "length" with the initial value of each element given by "initial_value" and assign it to a variable.

The "initial_value" sets the type of the list. It can be a complex type like struct. If not provided, the "capacity" will be defaulted to "length".

Creation list of list with this function is not supported (they are matrices in URScript).

Parameters

`length`: Number of elements which will be initialized

`initial_value`: Initial value of the elements

`capacity`: Maximum number of elements. List can be extended and contracted between 0, and capacity (Optional default value equals to length)

Example command 1: `list_1 = make_list(5, "a")`

Equivalent to ["a", "a", "a", "a", "a"]

- Example Parameters:
  - length = 5
  - initial_value = "a"
  - capacity = 5

Example command 2: `list_2 = make_list(10, 0, 100)`

- Example Parameters:
  - length = 10
  - initial_value = 0
  - capacity = 100

Example command 3: `list_3 = make_list(0, 0, 100)`

Create an initially empty list with the potential to hold 100 elements of integers.

- Example Parameters:
  - length = 0
  - initial_value = 0
  - capacity = 100

16.11. 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

16.12. 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

16.13. 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 geodetic 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]

16.14. inv(m)

Get the inverse of a matrix or pose

The matrix must be square and non singular.

Parameters

`m`: matrix or pose (spatial vector)

Return Value

inverse matrix or pose transformation (spatial vector)

Example command:

- `inv([[0,1,0],[0,0,1],[1,0,0]])` -> Returns [[0,0,1],[1,0,0],[0,1,0]]
- `inv(p[.2,.5,.1,1.57,0,3.14])` -> Returns p[0.19324,0.41794,-0.29662,1.23993,0.0,2.47985]

16.15. 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

16.16. 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

16.17. 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

16.18. normalize(v)

Returns the normalized form of a list of floats

Except for the case of all zeroes, the normalized form corresponds to the unit vector in the direction of v.

Throws an exception if the sum of all squared elements is zero.

Parameters

`v`: List of floats

Return Value

normalized form of v

Example command:

- `normalize([1, 0, 0])` -> Returns [1, 0, 0]
- `normalize([0, 5, 0])` -> Returns [0, 1, 0]
- `normalize([0, 1, 1])` -> Returns [0, 0.707, 0.707]

16.19. 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

16.20. 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]

16.21. 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

16.22. 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]

16.23. 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]

16.24. 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_world->to = T_world->from * T_from->to T_x->to = T_x->from * T_from->to

Parameters

`p_from`: starting pose (spatial vector)

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

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,-0.0996,0.60048,3.14,0.0,0.0]

16.25. 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.

16.26. 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

16.27. random()

Random Number

Return Value

pseudo-random number between 0 and 1 (float)

16.28. 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 Rrpy = Rz(yaw)Ry(pitch)Rx(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, -0.16202, 0.9] -> roll=-2.80856, pitch=-0.16202, yaw=0.9

16.29. 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 Rrpy = Rz(yaw)Ry(pitch)Rx(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

16.30. 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

16.31. size(v)

Returns the size of a matrix variable, the length of a list or string variable

Parameters

`v`: A matrix, list or string variable

Return Value

Given a list or a string the length is returned as an integer. Given a matrix the size is returned as a list of two numbers representing the number of rows and columns, respectively.

Example command:

- `size("here I am")` -> Returns 9
- `size([1,2,3,4,5])` -> Returns 5
- `size([[1,2],[3,4],[5,6]])` -> Returns [3,2]

16.32. 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

16.33. 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

16.34. transpose(m)

Get the transpose of a matrix

Parameters

`m`: matrix or an array

Return Value

transposed matrix or array

Example command:

`transpose([[1,2],[3,4],[5,6])` -> Returns [[1,3,5],[2,4,6]]

`transpose([1,2,3])` -> Returns [[1],[2],[3]]

`transpose([[1],[2],[3]])` -> Returns [1,2,3]

16.35. wrench_trans(T_from_to, w_from)

Wrench transformation

Move the point of view of a wrench.

Note: Transforming wrenches is not as trivial as transforming poses as the torque scales with the length of the translation.

w_to = T_from->to * w_from

Parameters

`T_from_to`: The transformation to the new point of view (Pose)

`w_from`: wrench to transform in list format [F_x, F_y, F_z, M_x, M_y, M_z]

Return Value

resulting wrench, w_to in list format [F_x, F_y, F_z, M_x, M_y, M_z]

17. Module interfaces

17.1. enable_external_ft_sensor(enable, sensor_mass=0.0, sensor_measuring_offset=[0.0, 0.0, 0.0], sensor_cog=[0.0, 0.0, 0.0])

Deprecated:

This function is used for enabling and disabling the use of external F/T measurements in the controller. Be aware that the following function is impacted:

- `force_mode`
- `screw_driving`
- `freedrive_mode`

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

Parameters

`enable`: enable or disable feature (bool)

`sensor_mass`: mass of the sensor in kilograms (float)

`sensor_measuring_offset`: [x, y, z] measuring offset of the sensor in meters relative to the tool flange frame

`sensor_cog`: [x, y, z] center of gravity of the sensor in meters relative to the tool flange frame

Deprecated

When using this function, the sensor position is applied such that the resulting torques are computed with opposite sign. New programs should use ft_rtde_input_enable in place of this.

Notes:

- The TCP Configuration in the installation must also include the weight and offset contribution of the sensor.
- Only the enable parameter is required, sensor mass, offset and center of gravity are optional (zero if not provided).

Example command: Please refer to `ft_rtde_input_enable` for some examples of usage

17.2. ft_rtde_input_enable(enable, sensor_mass=0.0, sensor_measuring_offset=[0.0, 0.0, 0.0], sensor_cog=[0.0, 0.0, 0.0])

This function is used for enabling and disabling the use of external F/T measurements in the controller. Be aware that the following function is impacted:

- `force_mode`
- `screw_driving`
- `freedrive_mode`

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

Parameters

`enable`: enable or disable feature (bool)

`sensor_mass`: mass of the sensor in kilograms (float)

`sensor_measuring_offset`: [x, y, z] measuring offset of the sensor in meters relative to the tool flange frame

`sensor_cog`: [x, y, z] center of gravity of the sensor in meters relative to the tool flange frame

Notes:

This function **replaces** the deprecated `enable_external_ft_sensor.`

The TCP Configuration in the installation must also include the weight and offset contribution of the sensor.

Only the `enable` parameter is required; `sensor mass`, `offset` and `center of gravity` are optional (zero if not provided).

Example command: `ft_rtde_input_enable(True, 1.0, [0.1, 0.0, 0.0], [0.2, 0.1, 0.5])`

- Example Parameters:
  - enable -> Enabling the feed of an external F/T measurements in the controller.
    - sensor_mass -> mass of F/T sensor is set to 1.0 Kg.
    - sensor_measuring_offset -> sensor measuring offset is set to [0.1, 0.0, 0.0] m from the tool flange in tool flange frame coordinates.
    - sensor_cog -> Center of Gravity of the sensor is set to x=200 mm, y=100 mm, z=500 mm from the center of the tool flange in tool flange frame coordinates.

- `ft_rtde_input_enable(True, 0.5)`
  - Example Parameters:
    - enable S{rarr} Enabling the feed of an external F/T measurements in the controller.
    - sensor_mass S{rarr} mass of F/T sensor is set to 0.5 Kg.
    - Both sensor measuring offset and sensor's center of gravity are zero.
  - @example:
    - `C{ft_rtde_input_enable(False)}`
    - Disable the feed of external F/T measurements in the controller (no other parameters required)

17.3. 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

17.4. 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

17.5. 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

17.6. 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
    - R'eturns True or False

17.7. 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

17.8. 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

17.9. get_flag(n)

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

Parameters

`n`: The number (id) of the flag, integer: [0:31]

Return Value

Boolean, The stored bit.

Example command: `get_flag(1)`

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

17.10. get_rtde_value(key)

Returns the corresponding value of the supplied RTDE output field key.

This function retrieves an RTDE value from the RTDE output buffer, which also can be collected through a client. The RTDE value is one time step behind, therefore it is suggested to make a sync() call, before calling get_rtde_value(key).

Parameters

`key`: RTDE output field key of value to retrieve. See the complete list of available fields and their corresponding field types:

www.universal-robots.com/articles/ur/interface-communication/real-time-data-exchange-rtde-guide/

Return Value

The type of the return value depends on the `key` parameter and can be one of the following:

- bool
  - For RTDE field type: BOOL
- number - either int or float
  - For RTDE field types: UINT8, UINT32, UINT64, INT32, DOUBLE
- fixed length list of numbers either int or float
  - For RTDE field types: VECTOR3D, VECTOR6D, VECTOR6INT32, VECTOR6UINT32

Example command: `get_rtde_value("target_qd")`

- Example Parameters:
  - key = "target_qd" → retrieve target joint velocities.

17.11. 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

17.12. 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

17.13. 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

17.14. 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

17.15. 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

17.16. 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

17.17. 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

17.18. modbus_add_signal(IP, slave_number, signal_address, signal_type, signal_name, sequential_mode=False, register_count=1)

Adds a new modbus signal for the controller to supervise. Expects no response. If the signal is an output type, then until the first set_output_signal/register() command the function code will be 2/3, after the call it will switch to 15/16.

Matrix of function codes used for accessing coils or discrete inputs:

	Read		Write	
Signal type	**Single Coil**	**Multiple Coils**	**Single Coil**	**Multiple Coils**
0 = Digital input	2	2	-	-
1 = Digital output	1	1	15	15
15 = Multiple digital outputs	1	1	15	15

Matrix of function codes used for accessing coils or discrete inputs:

	Read		Write	
Signal type	**Single register**	**Multiple registers**	**Single register**	**Multiple registers**
2 = Register input	4	4	-	-
3 = Register output	3	3	6	16

16 = Multiple register outputs	3	3	16	16

```
>>> 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. The IP can not be empty.

Note: Numerical IP addresses are recommended. DNS name resolution may lead to unexpected program stops.

`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. The value must be greater or equal to 0.

`signal_type`: An integer specifying the type of signal to add. 0 = digital input, 1 = digital output, 2 = register input, 3 = register output, 15 = multiple digital output, 16 = multiple register output. *Note: this function does not accept 23 = multiple read-write signal type.*

`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. The length of the string can not exceed 20 characters. The signal name cannot be empty.

`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).

`register_count`: Number of registers/coils accessed by the signal [1-123] (Optional, the default value is 1).

Example command 1: `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

Example command 2: `modbus_add_signal("172.140.17.11", 255, 5, 16, "output2", False, 10)`

- Example Parameters:
  - IP address = 172.140.17.11
  - Slave number = 255
  - Signal address = 5
  - Signal type = 16 multiple register output
  - Signal name = output 2
  - sequential_mode = False
  - register_count = 10

17.19. modbus_add_rw_signal(IP, slave_number, read_address, read_register_count, write_address, write_register_count, signal_name, sequential_mode=False)

Adds a new modbus signal for the controller to supervise. This function will use the function code 23. The read and write addresses can overlap. Until the first set_output_register() command the function code will be 3, after the call it will switch to 23.

```
>>> modbus_add_rw_signal("172.140.17.11", 255, 5, 10, 15, 10, "output1")
```

Parameters

`IP`: A string specifying the IP address of the modbus unit to which the modbus signal is connected. The IP can not be empty.

`slave_number`: An integer normally not used and set to 255, but is a free choice between 0 and 255.

`read_address`: An integer specifying the address of the first register that this new signal should read from.

`read_register_count`: Number of registers to read [1-123].

`write_address`: An integer specifying the address of the first register that this new signal should write to.

`write_register_count`: Number of registers to write [1-123].

`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. The length of the string can not exceed 20 characters. The signal name cannot be empty.

`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_rw_signal("172.140.17.11", 255, 5, 10, 15, 10 "output1")`

This example will create a signal that cyclically reads 10 registers from addresses 5-14, an writes 10 registers to addresses 10-19 Signal will use Function Code 3 to read register until first time modbus_set_register_output(...) is called. Afterwards Function Code 23 will be used to both read and write registers in remote device.

- Example Parameters:
  - IP address = 172.140.17.11
  - Slave number = 255
  - Signal read address = 5
  - Signal read register count = 10
  - Signal write address = 15
  - Signal write register count = 10
  - Signal name = output 1

17.20. 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. The signal name can not be empty.

Example command: `modbus_delete_signal("output1")`

- Example Parameters:
  - Signal name = output1

17.21. modbus_get_signal_status(signal_name, is_secondary_program=False)

Reads the current value(s) of a specific signal. If the modbus watchdog is active, this will return the last valid value(s). No error will be thrown within the watchdog time. If needed, the modus_get_error() or the modbus_get_time_since_signal_invalid() can be used to detected that the signal is in an error state before the watchdog expires.

```
>>> 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. Can not be empty.

`is_secondary_program`: A boolean for internal use only. Must be set to False. (Optional, default is 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. If the signal was declared to have access for multiple registers/coild, then the return value will be an array of unsigned integers / booleans. The length of the returned array is equal to the register_count of the signal.

Example command: `modbus_get_signal_status("output1")`

Example Parameters:

- Signal name = output 1
  - Is_secondary_program = False by default

17.22. 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. Can not be empty.

`slave_number`: An integer specifying the slave number to use for the custom command. Must be between [0-255].

`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

17.23. modbus_set_digital_input_action(signal_name, action)

Sets the selected digital input signal to either a "default" or "freedrive" action.

```
>>> modbus_set_digital_input_action("input1", "freedrive")
```

Parameters

`signal_name`: A string identifying a digital input signal that was previously added. Can not be empty.

`action`: The type of action. The action can either be "default" or "freedrive". Can not be empty. (string)

Example command: `modbus_set_digital_input_action("input1", "freedrive")`

- Example Parameters:
  - Signal name = "input1"
  - Action = "freedrive"

17.24. modbus_set_output_register(signal_name, register_value, is_secondary_program=False)

Sets the output register(s) 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. Can not be empty.

`register_value`: An integer which must be a valid word (0-65535) value or a list of integer values. The list can not be empty. The size of the list must be less than 123 and must be equal or less to the signal's declared register count. *Note: if a shorter list is given as an input, then registers with greater indexes will keep previous value*

`is_secondary_program`: A boolean for internal use only. Must be set to False. (Optional, false by default)

Example command 1: `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)

Example command 2:

```
modbus_add_signal("127.0.0.1", 255, 0, 16, "output2", False, 10)
list_var:=[10,9,8,7,6,5,4,3,2,1]
modbus_set_output_register("output2", list_var)
```

- Example Parameters:
  - Signal name = output2
  - Register values = 10,9,8,7,6,5,4,3,2,1
  - Is_secondary_program = False by default

17.25. modbus_set_output_signal(signal_name, digital_value, is_secondary_program, False)

Sets the output digital signal(s) 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. Can not be empty.

`digital_value`: A boolean to which value the signal will be set or a list of boolean values. The list can not be empty. The size of the list must be less than 123 and must be equal or less to the signal's declared register count. *Note: if a shorter list is given as an input, then coils with greater indexes will keep previous value*

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

Example command 1: `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)

Example command 2:

```
modbus_add_signal("127.0.0.1", 255, 0, 15, "output2", False, 5)
list_var:=[True, False, True, False, True]
modbus_set_output_signal("output2", list_var)
```

- Example Parameters:
  - Signal name = output2
  - Digital values = True, False, True, False, True
  - Is_secondary_program = False by default.

17.26. 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. Can not be empty.

`update_frequency`: An integer in the range 0-500 specifying the update frequency in Hz.

Note: The function accepts -1 and 0 as a valid input as a special value to create acyclic signals.

Note: If the input is 0 the signal will be acyclic.

Example command: `modbus_set_signal_update_frequency("output2", 20)`

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

17.27. modbus_get_error(signal_name)

Returns the current error state of the signal.

```
>>> modbus_get_error("output1")
```

Parameters

`signal_name`: A string equal to the name of the signal. The signal name can not be empty.

`error_source`: Selects the type of the returned error. Integer, 0 = combined, 1 = device error, 2 = connection error (Optional, default is 0 = combined)

Return Value

32 bit integer

0 = No error, Device errors

1 = Device error - illegal function code

2 = Device error - illegal data access

3 = Device error - illegal data value

4 = Device error - server failure

5 = Device error - acknowledge exception

6 = Device error - server busy

10 = Device error - gateway problem

11 = Device error - gateway target failure

Connection errors

Values from 1-113, see details in https://en.wikipedia.org/wiki/Errno.h and https://www.ibm.com/docs/en/db2/11.5?topic=message-tcpip-errors

NOTICE

If combined errors are requested, then the returned value's upper 2 bytes will store the connection error, the lower 2 bytes will store the device errors. Example returned value: 0xFFFE0004 (dec:-131068) -> upper 0xFFFE = -2 Disconnected ; lower 0x0004 = Device error - server failure

Example command: `modbus_get_error("output1")`

- Example Parameters:
  - Signal name = output1

17.28. modbus_get_time_since_signal_invalid(signal_name)

Returns the time in seconds since the signal is invalid (has communication or device error).For input signals function tells how long ago was last time when signal value was successfully read from remote device. For output signals function tells how long ago was last time when signal value was successfully written to remote device.

```
>>> modbus_get_time_since_signal_invalid("output1")
```

Parameters

`signal_name`: A string equal to the name of the signal. The signal name can not be empty.

Return Value

A float number representing the time in seconds since the signal is in a communication or device error.

Example command: `modbus_get_time_since_signal_invalid("output1")`

- Example Parameters:
  - Signal name = output1

17.29. modbus_request_update_signal_value(signal_name)

Request an update of the signal regardless of the used frequency.

```
>>> modbus_request_update_signal_value("output1")
```

Parameters

`signal_name`: A string equal to the name of the signal. The signal name can not be empty.

Example command: `modbus_request_update_signal_value("output1")`

- Example Parameters:
  - Signal name = output1

17.30. modbus_reset_connection(connection_id, is_blocking=True)

Tear down, and reconnect all signals to remote device. By default it will block as long as there are errors in the connection.

```
>>> modbus_reset_connection("172.140.17.11")
```

Parameters

`connection_id`: A string specifying the IP address of the modbus unit to which the modbus signal is connected. The IP can not be empty.

Example command: `modbus_reset_connection("172.140.17.11")`

Example Parameters:

- connection_id = 172.140.17.11
    - is_blocking = True

17.31. modbus_set_signal_watchdog(signal_name, new_timeout_in_sec)

Set tolerance for modbus errors (both communication, and device exceptions) as minimum time between valid device responses. No error will be thrown within the watchdog time. If needed, the modus_get_error () or the modbus_get_time_since_signal_invalid() can be used to detected that the signal is in an error state before the watchdog expires.Default signal timeout is 2 seconds.

```
>>> modbus_set_signal_watchdog("signal1", 5)
```

Parameters

`signal_name`: A string identifying an output digital signal that in advance has been added. Can not be empty.

`new_timeout_in_sec`: A number in range 0-300 representing seconds.

Example command: `modbus_set_signal_watchdog("signal1", 10.5)`

Example Parameters:

- signal_name = signal1
    - new_timeout_in_sec = 10.5 seconds

17.32. 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.

Parameters

`address`: Address of the register (0:127)

Return Value

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

Note: The lower range of the boolean input registers [0:63] is reserved for FieldBus/PLC interface usage. The upper range [64:127] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

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

Example command: `read_input_boolean_register(3)`

- Example Parameters:
  - Address = input boolean register 3

17.33. 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.

Parameters

`address`: Address of the register (0:47)

Return Value

The value held by the register (float)

Note: The lower range of the float input registers [0:23] is reserved for FieldBus/PLC interface usage. The upper range [24:47] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

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

Example command: `read_input_float_register(3)`

- Example Parameters:
  - Address = input float register 3

17.34. 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.

Parameters

`address`: Address of the register (0:47)

Return Value

The value held by the register [-2,147,483,648 : 2,147,483,647]

Note: The lower range of the integer input registers [0:23] is reserved for FieldBus/PLC interface usage. The upper range [24:47] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

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

Example command: `read_input_integer_register(3)`

- Example Parameters:
  - Address = input integer register 3

17.35. 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.

Parameters

`address`: Address of the register (0:127)

Return Value

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

Note: The lower range of the boolean output registers [0:63] is reserved for FieldBus/PLC interface usage. The upper range [64:127] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

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

Example command: `read_output_boolean_register(3)`

- Example Parameters:
  - Address = output boolean register 3

17.36. 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.

Parameters

address: Address of the register (0:47)

Return Value

The value held by the register (float)

Note: The lower range of the float output registers [0:23] is reserved for FieldBus/PLC interface usage. The upper range [24:47] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

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

Example command: `read_output_float_register(3)`

- Example Parameters:
  - Address = output float register 3

17.37. 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.

Parameters

`address`: Address of the register (0:47)

Return Value

The int value held by the register [-2,147,483,648 : 2,147,483,647]

Note: The lower range of the integer output registers [0:23] is reserved for FieldBus/PLC interface usage. The upper range [24:47] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

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

Example command: `read_output_integer_register(3)`

- Example Parameters:
  - Address = output integer register 3

17.38. 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 port map on Support site, page "Modbus Server")

Return Value

The value held by the port (True, False)

Example command: `read_port_bit(3)`

- Example Parameters:
  - Address = port bit 3

17.39. 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 port map on Support site, page "Modbus Server")

Return Value

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

Example command: read_port_register(3)

Example Parameters:

Address = port register 3

17.40. 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

17.41. 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

17.42. 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.

17.43. 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

17.44. 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

17.45. 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

17.46. 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: [0:31]

`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

17.47. 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

17.48. 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

17.49. 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

17.50. set_tool_communication(enabled, baud_rate, parity, stop_bits,

This function will activate or deactivate the 'Tool Communication Interface' (TCI). The TCI will enable communication with a external tool via the robots analog inputs hereby avoiding external wiring.

```
>>> set_tool_communication(True, 115200, 1, 2, 1.0, 3.5)
```

Parameters

`enabled`: Boolean to enable or disable the TCI (string). Valid values: True (enable), False (disable)

`baud_rate`: The used baud rate (int). Valid values: 9600, 19200, 38400, 57600, 115200, 1000000, 2000000, 5000000.

`parity`: The used parity (int). Valid values: 0 (none), 1 (odd), 2 (even).

`stop_bits`: The number of stop bits (int). Valid values: 1, 2.

`rx_idle_chars`: Amount of chars the RX unit in the tool should wait before marking a message as over / sending it to the PC (float). Valid values: min=1.0 max=40.0.

`tx_idle_chars`: Amount of chars the TX unit in the tool should wait before starting a new transmission since last activity on bus (float). Valid values: min=0.0 max=40.0.

Return Value

None

Note:

Enabling this feature will disable the robot tool analog inputs.

Example command:

set_tool_communication(True, 115200, 1, 2, 1.0, 3.5)

- Example Parameters:
  - enabled = True
  - baud rate = 115200
  - parity = ODD
  - stop bits = 2
  - rx idle time = 1.0
  - tx idle time = 3.5

17.51. set_tool_digital_output_mode (n, mode)

Sets the output mode of the tool output pin.

Parameters

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

`Mode`: The pin mode.

`Integer`: [1:3]

- 1=Sinking/NPN
- 2=Sourcing/PNP
- 3=Push-Pull

Example command: `set_tool_digital_output_mode(0,2)`

- Example Parameters:
  - 0 is digital output pin 0.
  - 2 is pin mode sourcing/PNP.

    The pin sources current when it is set to 1.

    The pin is high impedance when it is set to 0.

17.52. set_tool_output_mode (mode)

Sets the tool digital output mode.

Parameters

`Mode`: 1=power (dual pin) mode.

Example command: `set_tool_output_mode(1)`

- Example Parameters:
  - 1 is the power (dual pin) mode.

    The digital outputs are used as extra supply

17.53. 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, integer: 0, 12 or 24.

Example command: `set_tool_voltage(24)`

- Example Parameters:
  - voltage = 24 volts

17.54. 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

17.55. socket_get_var(name, socket_name='socket_0')

Reads an integer from the server

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

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: `x_pos = socket_get_var("POS_X")`

Sends: GET POS_X\n to socket_0, and expects response within 2s

- Example Parameters:
- name = POS_X -> name of variable
- socket_name = default: socket_0

17.56. 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 succesfully 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
  - port = 50000
  - socket_name = socket_10

17.57. 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.

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 (length=number+1, list of floats)

- **Example command:** `list_of_four_floats = socket_read_ascii_float(4,"socket_10")`
  - Example Parameters:
  - number = 4 → Number of floats to read
  - socket_name = socket_10
  - returns list

17.58. 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.

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

`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 (length=number+1, list of ints)

Example command: `list_of_ints = socket_read_binary_integer(4,"socket_10")`

- Example Parameters:
  - number = 4 -> Number of integers to read
  - socket_name = socket_10

17.59. socket_read_byte_list(number, socket_name='socket_0', timeout=2)

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

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 bytes 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 (length=number+1, list of ints)

Example command: `list_of_bytes = socket_read_byte_list(4,"socket_10")`

- Example Parameters:
  - number = 4 -> Number of byte variables to read
  - socket_name = socket_10

17.60. 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.

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: `line_from_server = socket_read_line("socket_10")`

- Example Parameters:
  - socket_name = socket_10

17.61. 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.

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.

Maxium length of received string including termination characters is limited to 1024 characters.

```
>>> 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.

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<" and calling this script function with the prefix=">" and suffix="<".

Note that leading spaces in the prefix and suffix strings are ignored in the current software and may cause communication errors in future releases.

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 command: `string_from_server = socket_read_string("socket_10",prefix=">",suffix="<")`

17.62. 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,"socket_10")`

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

17.63. 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,"socket_10")`

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

17.64. socket_send_line(str, socket_name='socket_0')

Sends a string with a newline character to the server - useful for communicating 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","socket_10")`

Sends: hello\n to socket_10

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

17.65. 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","socket_10")`

Sends: hello to socket_10

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

17.66. socket_set_var(name, value, socket_name='socket_0')

Sends an integer to the server

Sends the message "SET <name> <value>\n" through the socket. Expects no response.

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,"socket_10")`

Sends string: SET POS_Y 2200\n to socket_10

- Example Parameters:
  - name = POS_Y -> name of variable
  - value = 2200
  - socket_name = socket_10

17.67. 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.

Parameters

`address`: Address of the register (0:127)

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

Note: The lower range of the boolean output registers [0:63] is reserved for FieldBus/PLC interface usage. The upper range [64:127] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

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

Example command: `write_output_boolean_register(3,True)`

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

17.68. 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.

Parameters

`address`: Address of the register (0:47)

`value`: Value to set in the register (float)

Note: The lower part of the float output registers [0:23] is reserved for FieldBus/PLC interface usage. The upper range [24:47] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

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

Example command: `write_output_float_register(3,37.68)`

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

17.69. 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.

Parameters

`address`: Address of the register (0:47)

`value`: Value to set in the register [-2,147,483,648 : 2,147,483,647]

Note: The lower range of the integer output registers [0:23] is reserved for FieldBus/PLC interface usage. The upper range [24:47] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

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

Example command: `write_output_integer_register(3,12)`

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

17.70. 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 port map on Support site, page "Modbus Server")

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

Example command: `write_port_bit(3,True)`

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

17.71. 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 port map on Support site, page "Modbus Server")

`value`: Value to be set in the port (0 : 65536) or (-32768 : 32767)

Example command: `write_port_register(3,100)`

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

17.72. zero_ftsensor()

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

17.73. request_boolean_from_primary_client(message)

Request input from operator. Polyscope shows dialog box with "yes" and "no" buttons.

Function blocks until operator selects option on the Polyscope screen.

NOTE: Operator can also stop program by pressing "Cancel" button

Parameters

`message`: A string with a message shown on Polyscope dialog box. Can not be empty.

Return Value

True or False: Value selected by the operator.

Example command: `mark_part = request_boolean_from_primary_client("Should part be marked?")`

Show message to the operator, and save reply to mark_part variable

17.74. request_float_from_primary_client(message)

Request input from operator. Polyscope shows dialog box with decimal number entry field.

Function blocks until operator enters value on the Polyscope screen.

NOTE: Operator can also stop program by pressing "Cancel" button

Parameters

`message`: A string with a message shown on Polyscope dialog box. Can not be empty.

Return Value

Float: Value entered by the operator.

Example command: `offset_mm = request_float_from_primary_client("Enter gripping offset [mm]")`

Show message to the operator, and save reply to offset_mm variable

17.75. request_integer_from_primary_client(message)

Request input from operator. Polyscope shows dialog box with integer number entry field.

Function blocks until operator enters value on the Polyscope screen.

NOTE: Operator can also stop program by pressing "Cancel" button

Parameters

`message`: A string with a message shown on Polyscope dialog box. Can not be empty.

Return Value

Integer: Value entered by the operator.

Example command: `number_of_parts = request_integer_from_primary_client("Enter number of parts")`

Show message to the operator, and save reply to number_of_parts variable

17.76. request_string_from_primary_client(message)

Request input from operator. Polyscope shows dialog box with string entry field.

Function blocks until operator enters value on the Polyscope screen.

NOTE: Operator can also stop program by pressing "Cancel" button

Parameters

`message`: A string with a message shown on Polyscope dialog box. Can not be empty.

Return Value

String: Value entered by the operator.

Example command: `part_name = request_string_from_primary_client("Enter name of the part")`

Show message to the operator, and save reply to part_name variable

18. Module ioconfiguration

18.1. modbus_set_runstate_dependent_choice(signal_name, runstate_choice)

Sets the output signal levels depending on the state of the program.

Parameters

`signal_name`:

A string identifying a digital or register output signal that in advance has been added. Can not be empty.

`state`:

0: Preserve signal state,

1: Set signal Low when program is not running,

2: Set signal High when program is not running,

3: Set signal High when program is running and low when it is stopped,

4: Set signal Low when program terminates unscheduled,

5: Set signal High from the moment a program is started and Low when a program terminates unscheduled.

Note: An unscheduled program termination is caused when a Protective stop, Fault, Violation or Runtime exception occurs.

Example command:

```
modbus_set_runstate_dependent_choice("output2", 3)
```

- Example Parameters:
  - Signal name = output2
  - Runstate dependent choice = 3 ! set Low when a program is stopped and High when a program is running

18.2. 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)

18.3. set_configurable_digital_input_action(port, action)

Using this method sets the selected configurable digital input register to either a "default" or "freedrive" action.

See also:

- `set_input_actions_to_default`
- `set_standard_digital_input_action`
- `set_tool_digital_input_action`
- `set_gp_boolean_input_action`

Parameters

`port`: The configurable digital input port number. (integer)

`action`: The type of action. The action can either be "default" or "freedrive". (string)

Example command: `set_configurable_digital_input_action(0, "freedrive")`

- Example Parameters:
  - n is the configurable digital input register 0
  - f is set to "freedrive" action

18.4. set_gp_boolean_input_action(port, action)

Using this method sets the selected gp boolean input register to either a "default" or "freedrive" action.

Parameters

`port`: The gp boolean input port number. integer: [0:127]

`action`: The type of action. The action can either be "default" or "freedrive". (string)

Note: The lower range of the boolean input registers [0:63] is reserved for FieldBus/PLC interface usage. The upper range [64:127] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.

See also:

`set_input_actions_to_default`

`set_standard_digital_input_action`

`set_configurable_digital_input_action`

`set_tool_digital_input_action`

Example command: `set_gp_boolean_input_action(64, "freedrive")`

- Example Parameters:
  - n is the gp boolean input register 0
  - f is set to "freedrive" action

18.5. set_input_actions_to_default()

Using this method sets the input actions of all standard, configurable, tool, and gp_boolean input registers to "default" action.

See also:

`set_standard_digital_input_action`

`set_configurable_digital_input_action`

`set_tool_digital_input_action`

`set_gp_boolean_input_action`

Example command: `set_input_actions_to_default()`

18.6. set_runstate_configurable_digital_output_to_value (outputId, state)

Using this method assigns the output to one of the states. This will set the output signal level depending on the state.

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:`

0: Preserve signal state,

1: Set signal Low when program is not running,

2: Set signal High when program is not running,

3: Set signal High when program is running and low when it is stopped,

4: Set signal Low when program terminates unscheduled,

5: Set signal High from the moment a program is started and Low when a program terminates unscheduled,

6: Set signal High when the robot has drive power,

7: Set signal Low when the robot has drive power.

Note: An unscheduled program termination is caused when a Protective stop, Fault, Violation or Runtime exception occurs.

Example command:

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

- Example Parameters:
  - outputid = configurable digital output on port 5
  - Runstate choice = 4 ! configurable digital output on port 5 goes low when a program is terminated unscheduled.

18.7. set_runstate_gp_boolean_output_to_value (outputId, state)

Using this method assigns the output to one of the states. This will set the output value depending on the state.

Parameters

`outputId`: The output signal number (id), integer: [0:127]

`state`:

0: Preserve signal state,

1: Set signal to False when program is not running,

2: Set signal to True when program is not running,

3: Set signal to True when program is running and False when it is stopped,

4: Set signal to False when program terminates unscheduled,

5: Set signal to True from the moment a program is started and False when a program terminates unscheduled,

6: Set signal to True when the robot has drive power,

7: Set signal to False when the robot has drive power.

Notes:

- The lower range of the boolean output registers [0:63] is reserved for FieldBus/PLC interface usage. The upper range [64:127] cannot be accessed by FieldBus/PLC interfaces, since it is reserved for external RTDE clients.
- An unscheduled program termination is caused when a Protective stop, Fault, Violation or Runtime exception occurs.

Example command:

```
set_runstate_gp_boolean_output_to_value(64, 2)
```

- Example Parameters:
  - outputid = output on port 64
  - Runstate choice = 2 ! sets signal on port 64 to True when program is not running

18.8. set_runstate_standard_analog_output_to_value (outputId, state)

Using this method assigns the output to one of the states. This will set the output signal level depending on the state.

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`:

0: Preserve signal state,

1: Set signal Low when program is not running,

2: Set signal High when program is not running,

3: Set signal High when program is running and low when it is stopped,

4: Set signal Low when program terminates unscheduled,

5: Set signal High from the moment a program is started and Low when a program terminates unscheduled,

6: Set signal High when the robot has drive power,

7: Set signal Low when the robot has drive power.

Note: An unscheduled program termination is caused when a Protective stop, Fault, Violation or Runtime exception occurs.

Example command:

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

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

18.9. set_runstate_standard_digital_output_to_value (outputId, state)

Using this method assigns the output to one of the states. This will set the output signal level depending on the state.

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`:

0: Preserve signal state,

1: Set signal Low when program is not running,

2: Set signal High when program is not running,

3: Set signal High when program is running and low when it is stopped,

4: Set signal Low when program terminates unscheduled,

5: Set signal High from the moment a program is started and Low when a program terminates unscheduled,

6: Set signal High when the robot has drive power,

7: Set signal Low when the robot has drive power.

Note: An unscheduled program termination is caused when a Protective stop, Fault, Violation or Runtime exception occurs.

Example command:

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

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

18.10. 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`:

0: Preserve signal state,

1: Set signal Low when program is not running,

2: Set signal High when program is not running,

3: Set signal High when program is running and low when it is stopped,

4: Set signal Low when program terminates unscheduled,

5: Set signal High from the moment a program is started and Low when a program terminates unscheduled.

Note: An unscheduled program termination is caused when a Protective stop, Fault, Violation or Runtime exception occurs.

Example command:

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

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

18.11. 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)

18.12. set_standard_digital_input_action(port, action)

Using this method sets the selected standard digital input register to either a "default" or "freedrive" action.

See also:

- `set_input_actions_to_default`
- `set_configurable_digital_input_action`
- `set_tool_digital_input_action`
- `set_gp_boolean_input_action`

Parameters

`port`: The standard digital input port number. (integer)

`action`: The type of action. The action can either be "default" or "freedrive". (string)

Example command: `set_standard_digital_input_action(0, "freedrive")`

- Example Parameters:
  - n is the standard digital input register 0
  - f is set to "freedrive" action

18.13. 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)

18.14. set_tool_digital_input_action(port, action)

Using this method sets the selected tool digital input register to either a "default" or "freedrive" action.

See also:

- `set_input_actions_to_default`
- `set_standard_digital_input_action`
- `set_configurable_digital_input_action`
- `set_gp_boolean_input_action`

Parameters

`port`: The tool digital input port number. (integer)

`action`: The type of action. The action can either be "default" or "freedrive". (string)

Example command: `set_tool_digital_input_action(0, "freedrive")`

- Example Parameters:
  - n is the tool digital input register 0
  - f is set to "freedrive" action

Software Name: PolyScope X
Software Version: 10.8
Document Version: 10.11.11