Using Universal Robots with LEonard

LEONARD

LECKY ENGINEERING

Software Version 22.11.1.0

LEonard Software by Lecky Engineering, LLC

Document Version	Date	Major Additions
21.11.4.0	11/04/2021	Initial user interface and device management system, Java interpreter
22.04.1.0	04/01/2022	Universal Robot interface and grinding system, LEScript support
22.08.1.0	08/15/2022	LMI Gocator interface and demonstration
22.11.1.0	11/14/2022	Python support, screen sizing and display management

CONTENTS

Overview

LEonard provides a custom interface for industrial cobots from Universal Robots (UR).

The interface is currently in use with UR-5e and UR-10e robots in many places.

For more information and documentation on the exciting UR product line, see www.universal-robots.com.

Basic Ethernet Connection

The UR and the computer running LEonard must have an Ethernet interface capable of communication. At Lecky Engineering, our test machine is on 192.168.0.252/24 and our UR-5e is on 192.168.0.2/24 (`robotIP=192.168.0.2` for us).

You should be able to use `ping robotIP` and `ssh root@robotIP` to verify communication. SSH uses `root` with default password `easybot` on a UR!

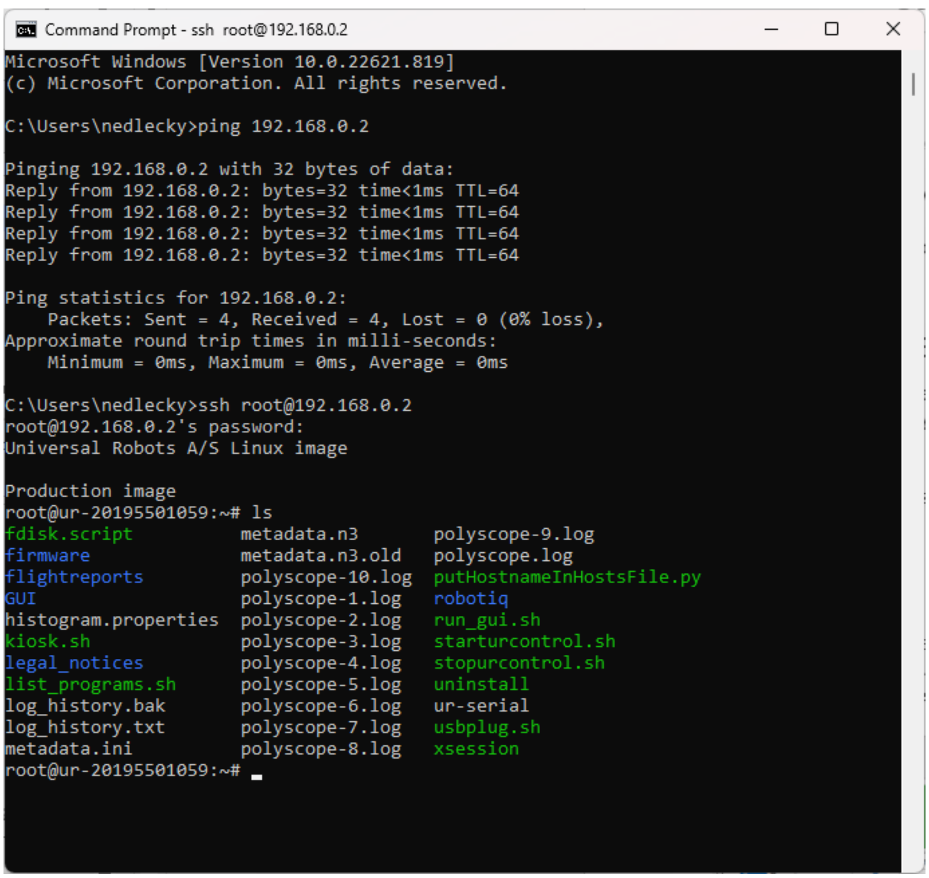

Figure 1 Pinging and SSH into a UR to Verify Communication

If you're like us, you may also want to be able to move files on and off your UR easily. We use WinSCP. Just connect your FTP client to `robotIP:22` and use that `root, easybot` login to get access.

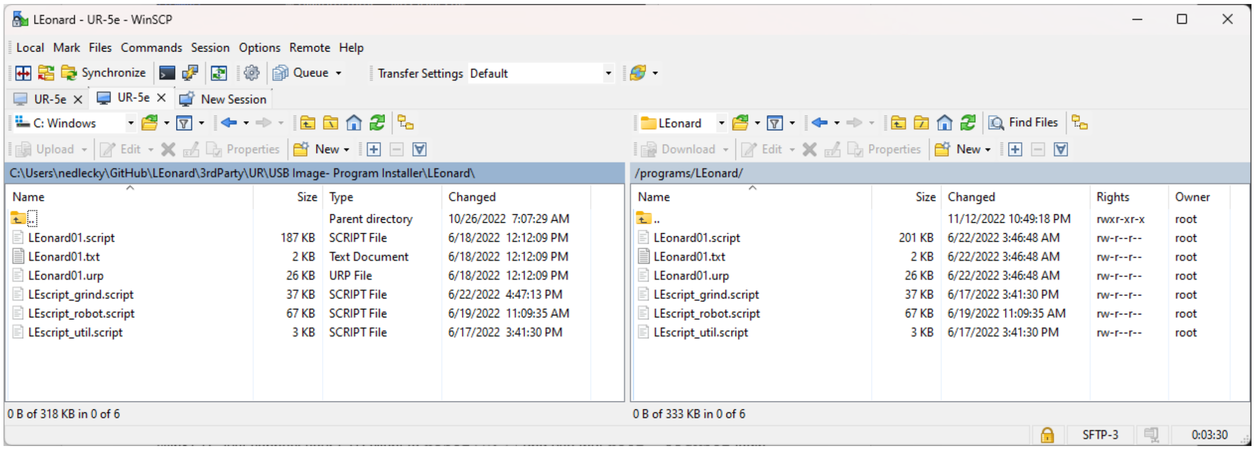

Figure 2 WinSCP FTP access to a UR

The LEonard Interface

To communicate with the UR, the **Devices** list in LEonard needs one or two entries for the robot:

1. If you just want to use the UR Dashboard interface and ask the robot to load/run existing PolyScope programs on the robot, you only need a **UrDashboard** device.
2. If you want to use the Lecky Engineering `LEonard01.urp` PolyScope program that allows commanding, driving, and sequencing the robot, as well as using the Lecky Engineering Grinding programs, you also need a **UrCommand** device.

Both devices connect to the same robot, just on different ports.

1. Port 29999 is a standard fixed port that provides dashboard control services on all UR robots. It connects to the robot as a **TcpClient** but has some special features that a basic **TcpClient** device does not.

2. Port 30000 is a custom port used in the Lecky Engineering `LEonard01.urp` PolyScope program. The program looks for a **TcpServer** on the machine running LEonard. The UrCommand device is a customized **TcpServer** device that has some special features to help with the UR interface.

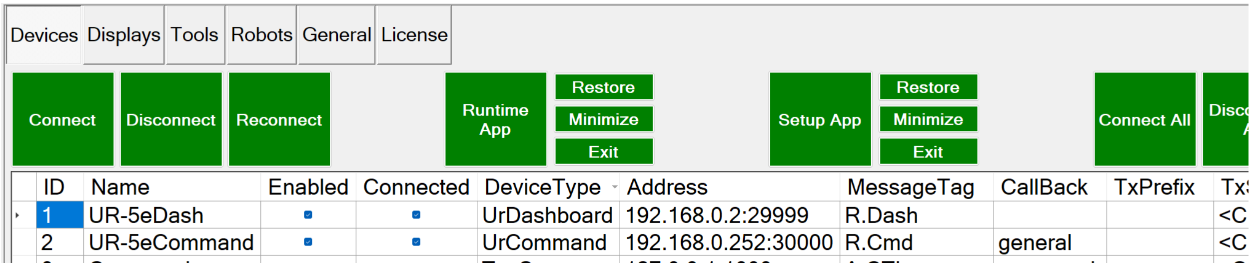

Figure 3 Device Entries for Universal Robot

It is important to use the `UrCommand` callback as well as the displayed `TxSuffix` and `RxTerminator`. Default devices that have everything setup just the way we need although you might need to edit the IP address!

The PolyScope job that you wish to be loaded by the UrDashboard can be included in the `Jobfile` field of the dashboard device entry. For our example, this is set to `LEonard/LEonard01.urp` since the `LEonard01.urp` program is stored on the robot in `programs/LEonard/LEonard01.urp`.

Installing Programs on the UR

As an aside, how do we get the programs on the UR? The program needs to wind up on the robot in `programs/LEonard`, and the PolyScope program also requires three somewhat complicated URScript programs to support robot and grinding applications.

Program Installation Method 1: FTP

We already discussed one method, FTP. Just copy the files from your PC on `LEonardRoot\3rdParty\UR\USB Image- Program Installer/LEonard` onto the robot in `programs/LEonard`.

Program Installation Method 2: The Supplied Magic File

Another method is to use UR magic files:

1. Make sure your existing robot programs are backed up! This process should not affect them, but it never hurts to be careful.
2. Take a fully erased USB thumb drive and insert it in your PC.
3. Copy all the files and subdirectories from `LEonardRoot\3rdParty\UR\USB Image- Program Installer` onto the thumb drive.
4. Plug the thumb drive into your robot USB port on the pendant- all of the necessary files will be installed on your robot in the `programs/LEonard` directory.

You will likely need to manually open `LEonard01.urp` on your robot to associate it with the installation file you are using in your installation. You'll be prompted to select an installation file. Once you do, resave the program and everything should be automatic from there.

There is also a set of three hard-coded IP address that the robot looks at to find LEonard. This allows a robot to be directly tied to only one LEonard. You will have to adjust this if the PolyScope program if your address is something different. Lecky Engineering can help if you need some assistance!

The LEonard01.urp PolyScope program currently looks for LEonard on:
169.254.254.200:30000
169.254.254.210:3000
192.168.0.252:30000

Connection is initiated by selecting the desired row and pressing **Connect**. In addition, if you have selected **Auto Connect On Load** for your device file, the connection will be started automatically when LEonard starts.

LEonard always starts a UrDashboard connection with a set of commands to initialize the robot, load any specified program, and start it.

```
close safety popup
is in remote control (response must be true)
if JobFile <> “”
  load <Jobfile> (as in Device… default LEonard/LEonard01.urp
  get loaded program (make sure response matches above)
  play
```

Upon successful connection, the `Connected` field should check itself and the UR Status annunciators should appear on the Run tab.

The Run tab in LEonard uses `robotmode` to determine whether the robot has booted, and regularly sends `robotmode`, `safetystatus`, and `programstate` to keep an eye on the robot.

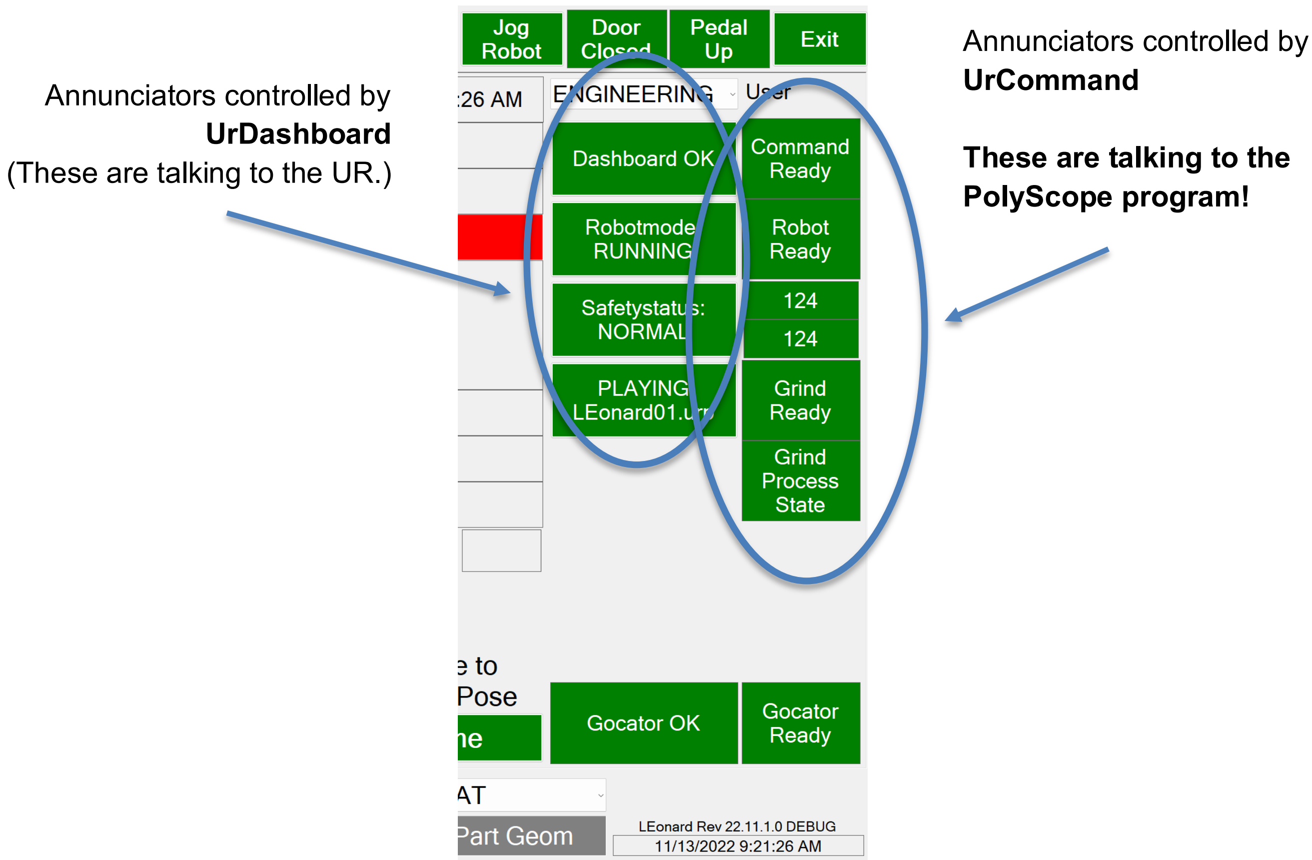

Figure 4 Gocator Status Annunciators

LEonard.urp Communications

The LeckyEngineering PolyScope program supplied for robot sequencing, control, and the grind functions is complex.

Our simple PolyScope program uses large set of underlying URScript code to perform most functions.

That said, UR-savvy users are able to add their own functions to the PolyScope program with relative ease. All communications between LEonard and the PolyScope program happen over a single socket using a message ID, a simple checksum, and the parameters which are all assumed to be numeric.

All return messages from the robot are asynchronous **LEonardMessages** and can set variables or execute LEScript, Java, or Python functions within LEonard. The whole architecture is quite powerful.

If you need to customize `LEonard01.urp` or the underlying URScript code for your own needs, don't hesitate to contact Lecky Engineering for some start-up assistance!

Controlling the Robot

The UR may be commanded from LEScript, Java, or Python with equal ease.

All three languages provide a set of functions that cover most point-to-point, grinding, and inspection tasks that you might want to do with your cell.

LElib.UR Library for Universal Robots

These functions work with Universal Robots robotic systems. The commands fall into three categories

1. Dashboard
2. Command Interface to Lecky Engineering's `LEonard01.urp` PolyScope program
3. Grinding Package for force-controlled surface following

LElib.UR.dashboard: Commands to control the UR dashboard

The UR robot provides a dashboard interface that allows controlling the robot operation.

`string ur_dashboard(string message, int timeout_ms)`

Sends the command `message` to the currently selected Universal Robot dashboard connection and waits for up to `timeout_ms` milliseconds for a response.

Response:
LEScript: Any response received is placed in the variable `ur_dashboard_response`
Java, Python: Function returns any string received or and empty string.

The UR dashboard system provides many commands that are useful in loading, starting, and stopping the robot. The Run tab in LEonard uses `robotmode` to determine whether the robot has booted, and regularly sends `robotmode`, `safetystatus`, and `programstate` to keep an eye on the robot.

When you press the **Robot Mode** button, LEonard cycles through the robot modes as appropriate- RUNNING initiates sending `power off`. IDLE initiates sending `brake release`. And POWER_OFF initiates sending `power on`. This allows you to cycle through UR operating modes.
The **Safety Status** button sends `unlock protective stop` and `close safety popup` when the robot is in safety stop but not in E-Stop.

The **Program State** button toggles between sending `play` and `stop` to start and stop the loaded PolyScope program. The UR Dashboard device sends a `load JobFile` command when the UR connects with the dashboard to get your default PolyScope program loaded.

A comprehensive discussion of the dashboard interface is available on the UR website:

https://www.universal-robots.com/articles/ur/dashboard-server-e-series-port-29999/

Here are the handiest ones that are used internally by LEonard!

Useful UR Dashboard Commands

`get robot model`	Returns robot model number, as in "UR5"
`get serial number`	Returns robot serial number, for example "20195501xxxx"
`PolyscopeVersion`	Returns PolyScope version installed on robot
`power on`	Power system up
`brake release`	Release from IDLE to READY
`load LEonard/LEonard01.urp`	Load a PolyScope program (default shown)
`play`	Start it
`close popup`	Close a popup prompt on the pendant
`close safety popup`	Close a safety popup prompt on the pendant
`unlock protective stop`	Recover from E-stop or safety stop
`stop`	Stop execution of the program
`robotmode`	Get mode `POWER_OFF POWER_ON BOOTING IDLE RUNNING`
`programstate`	Return program state `STOPPED file PLAYING file`
`power off`	Power servos down (and put brakes on)

LElib.UR.robot: The UR Robot PolyScope Interface

Lecky Engineering supplies an extensive PolyScope program that supports robot control and grinding functions. This code is supplied with the LEonard installation and must be installed on the UR robot.

Getting robot communications working is discussed in Basic Ethernet Connection and The LEonard Interface.

Installation of the code on the robot is discussed in Installing Programs on the UR

`select_tool(string tool_name)`

Setup all the necessary environment to be able to use `tool_name`. No motion is performed. Future tool moves, position moves, and grinds will assume this tool is attached.

`set_part_geometry(string FLAT|CYLINDER|SPHERE, double part_diam_mm)`

Future tool moves and grinds will assume the specified geometry.

`save_position(position_name)`

The current robot position is stored in the Positions Table as position_name.

`move_linear(position_name)`

The robot moves along a linear path to Position position_name.

`move_joint(position_name)`

The robot performs a joint move to Position position_name.

`move_relative(dx_mm, dy_mm)`

Move (dx_mm, dy_mm) relative to current tool position. If the part geometry selected is CYLINDER or SPHERE, robot moves along the part.

`move_tool_home()`

Perform a joint move to the home position associated with the current tool.

`move_tool_mount()`

Perform a joint move to the mounting position associated with the current tool.

`free_drive(0=OFF|1=ON)`

Turn robot free drive mode on or off.

The commands below provide a programmatic way to set the default motion parameters.

`set_linear_speed(speed_mm/s)`

Sets default linear speed used for robot linear moves.

`set_linear_accel(accel_mm/s^2)`

Sets default linear acceleration used for robot linear moves.

`set_joint_speed(speed_deg/s)`

Sets default joint speed used for robot joint moves.

`set_joint_accel(double accel_deg/s^2)`

Sets default joint acceleration used for robot joint moves.

`set_blend_radius(double blend_radius_mm)`

Sets default blend radius used in all robot moves.

`get_actual_tcp_pose()`

Ask the current robot to perform `get_actual_tcp_pose()` and return the value in the LEonard variable `actual_tcp_pose`.

`get_target_tcp_pose()`

Ask the current robot to perform `get_target_tcp_pose()` and return the value in the LEonard variable `target_tcp_pose`.

`get_actual_joint_positions()`

Ask the current robot to perform `get_actual_joint_positions()` and return the value in the LEonard variable `actual_joint_positions`.

`get_target_joint_positions()`

Ask the current robot to perform `get_target_joint_positions()` and return the value in the LEonard variable `target_joint_positions`.

`get_actual_both()`

Performs both `get_actual_joint_positions()` and `get_actual_tcp_pose()` on the current robot and return the values to the LEonard variables `actual_joint_positions` and `actual_tcp_pose`.

`get_target_both()`

Performs both `get_target_joint_positions()` and `get_target_tcp_pose()` on the current robot and return the values to the LEonard variables `target_joint_positions` and `target_tcp_pose`.

`movej(double j1, j2, j3, j4, j5, j6)`

Performs a `movej` to **joint positions** on the current robot as follows:

```
q = [j1, j2, j3, j4, j5, j6]
movej(q, a=robot_joint_accel_rpss, v=robot_joint_speed_rps)
```

`movel(double x, y, z, rx, ry, rz)`

Performs a `movel` to a **pose** on the current robot as follows:

```
p = p[x, y, z, rx, ry, rz]
movej(q, a=robot_joint_accel, v=robot_joint_speed)
```

`get_tcp_offset()`

Ask the current robot to perform `get_tcp_offset()` and return the value in the LEonard variable `tcp_offset`.

`movel_incr_base(double x,y,z,rx,ry,rz)`

Ask the current robot to move incrementally from the current position in base coordinates as in URScript:

```
local p0 = get_target_tcp_pose()
local p1 = p[x,y,z,dx,dy,dz]
local p2 = pose_add(p0, p1)
if p1[0] == 0 and p1[1] == 0 and p1[2] == 0:  # Rotational move
  movel(p2, robot_joint_accel_rpss, robot_joint_speed_rps)
else:
  movel(p2, robot_linear_accel_mpss, robot_linear_speed_mps)
end
```

`movel_incr_tool(double x,y,z,rx,ry,rz)`

Ask the current robot to move incrementally from the current position in TCP coordinates as in URScript:

```
local p1 = p[x,y,z,rx,ry,rz]
local p2 = pose_trans(get_target_tcp_pose(), p1)
if p1[0] == 0 and p1[1] == 0 and p1[2] == 0:  # Rotational move
  movel(p2, robot_joint_accel_rpss, robot_joint_speed_rps)
else:
  movel(p2, robot_linear_accel_mpss, robot_linear_speed_mps)
end
```

`movel_incr_part(x,y,z,rx,ry,rz)`

Ask the current robot to move incrementally from the current position in PART coordinates. X and Y are interpreted based on `set_part_geometry(…)`. For cylinders, X is along the axis of the cylinder and Y is interpreted as a fixed-distance rotation about the cylinder.

`movel_single_axis(axis,value)`

Ask the current robot to move to its current pose with the coordinate `axis` changed to `value`.

`movel_rot_only(rx,ry,rz)`

Ask the current robot to move to its current pose with the new rotations `rx`, `ry`, and `rz`.

`movel_rel_set_tool_origin(double x,y,z,rx,ry,rz)`

`movel_rel_set_tool_origin_here()`

Sets a tool-coordinate origin for the current robot either to a specified pose or to the current robot position. Subsequent calls to `movel_rel_tool()` will move in tool coordinates relative to this origin.

`movel_rel_tool(x,y,z,rx,ry,rz)`

Move to a tool coordinate position that is relative to the `movel_rel_set_tool_origin`.

`movel_rel_set_part_origin(x,y,z,rx,ry,rz)`

`movel_rel_set_part_origin_here()`

Sets a part-coordinate (FLAT, CYLINDER, or SPHERE) origin for the current robot either to a specified pose or to the current robot position. Subsequent calls to `movel_rel_part()` will move in part coordinates relative to this origin.

`movel_rel_part(x,y,z,rx,ry,rz)`

Move to a part coordinate position that is relative to the `movel_rel_set_part_origin`.

`send_robot(string message)`

Sends any command to the Lecky Engineering PolyScope program. All communications with the Lecky Engineering PolyScope program is handled by this command.

1. Commands are sent with a message ID and a checksum as follows:
    a. `(ID, checksum, message)`
2. ID can be any integer. LEonard sends an incrementing number between 100 and 999.
3. Checksum is expected to be 1000 – ID.
4. `message` is typically 1 or more comma-separated numeric values.

5. The command is non-blocking.
6. The PolyScope program is expected to send a start message:
```
robot_starting = ID
robot_ready = False
```
7. After the command is complete, the PolyScope program is expected to send back the following:
```
robot_response = response_message
robot_ready=True
robot_completed = ID (as it was received)
```

In addition, the UR Command device runs the “general” callback, so the UR robot can return **LeonardMessages** to set variables or trigger other actions in LEonard at any time.

`set_output(int port, bool value)`

Set UR digital output `port` to `value`.

`robot_socket_reset()`

Commands the Lecky Engineering UR PolyScope program to reset (bounce) its socket connection to LEonard. Program must be running on the UR!

`robot_program_exit()`

Commands the Lecky Engineering UR PolyScope program to terminate. Program must be running on the UR!

Low-level Setup Calls

These are all called automatically by `select_tool()`, `set_part_geometry()`,and the `grind_xxx()` functions. They should be used through that high level interface except during testing or for special purposes!

`set_tcp(x,y,x,rx,ry,rz)`

Executes `set_tcp(p[x,y,z,rx,ry,rz])` on the current robot only if `x > 10`. Always returns the current `get_tcp_offset()` in the LEonard variable `robot_tcp`.

`set_payload(mass_kg,cog_x_m, cog_y_m, cog_z_m)`

Executes `set_paylod(mass_kg, [cog_x_m, cog_y_m, cog_z_m])` on the current robot only if `mass_kg > 0`. Always returns the current `robot_payload_mass_kg` and `robot_paylod_cog_m` in corresponding LEonard variables.

`set_door_closed_input(int dig_in, int state)`

Specifies what digital input and state is expected to signify that the door is closed to the current robot.

`set_footswitch_pressed_input(int dig_in, int state)`

Specifies what digital input and state is expected to signify that the footswitch is pressed on the current robot.

`set_tool_on_outputs(int dig_out, int state, …)`

Sets a set of digital output,state pairs (1 – 4) to specify what outputs should be controlled when `tool_on()` is executed on the current robot.

`set_tool_off_outputs(int dig_out, int state, …)`

Sets a set of digital output,state pairs (1 – 4) to specify what outputs should be controlled when `tool_off()` is executed on the current robot.

`set_coolant_on_outputs(int dig_out, int state, …)`

Sets a set of digital output,state pairs (1 – 4) to specify what outputs should be controlled when `coolant_on()` is executed on the current robot.

`set_coolant_off_outputs(int dig_out, int state, …)`

Sets a set of digital output,state pairs (1 – 4) to specify what outputs should be controlled when `coolant_off()` is executed on the current robot.

`tool_on()`

Performs the `tool_on` output list set in `set_tool_on_outputs()` on the current robot.

`tool_off()`

Performs the `tool_off` output list set in `set_tool_off_outputs()` on the current robot.

`coolant_on()`

Performs the `coolant_on` output list set in `set_coolant_on_outputs()` on the current robot.

`coolant_off()`

Performs the `coolant_off` output list set in `set_coolant_off_outputs()` on the current robot.

LElib.UR.grind: The UR grinding package

The grinding commands use a set of common parameters described below:

> `dx_mm, dy_mm, diam_mm`: dimensions of the patterns in mm
> `n_cycles`: times to repeat the pattern (ignored if test grinding)

`speed_mm/s`: speed to grind at (ignored if test grinding
`force_N`: force in Newtons to apply
`stay_in_contact`: 0 to retract at end of grind, 1 to stay in contact

```
grind_line(dx_mm, dy_mm, n_cycles, speed_mm/s, force_N,
stay_in_contact)
```

```
grind_line_deg(length_mm, angle_deg, n_cycles, speed_mm/s,
force_N, stay_in_contact)
```

Grind in a straight line centered on the current position, defined either by endpoints or angle.

```
grind_rect(dx_mm, dy_mm, n_cycles, speed_mm/s, force_N,
stay_in_contact)
```

Grind along a rectangle centered on the current position at the current RZ angle of the tool.

```
grind_serp(dx_mm, dy_mm, n_xsteps, n_ysteps, n_cycles,
speed_mm/s, force_N, stay_in_contact)
```

Grind a serpentine pattern within a rectangle centered on the current position. N_xsteps and n_ysteps is the number of moves needed to span the rectangle. One or the other of these must be equal to 1.

```
grind_poly(circle_diam_mm, n_sides, n_cycles, speed_mm/s,
force_N, stay_in_contact)
```

Grind along a polygon of n_sides inscribed in circle_diam_mm centered on the current position.

```
grind_circle(circle_diam_mm, n_cycles, speed_mm/s, force_N,
stay_in_contact)
```

Grind along a circle centered on the current position.

```
grind_spiral(circle1_diam_mm, grind_circle2_diam_mm, n_spirals,
n_cycles, speed_mm/s, force_N, stay_in_contact)
```

Grind along a variable diameter circle centered on the current position. The circle goes from the first diameter to the second in n_spirals full revolutions.

```
grind_retract()
```

Ensure not in contact with the part. Happens automatically if a non-grind command is sent, if stop or pause is selected, or if grind_max_wait timer expires.

`grind_contact_enable(0=Touch OFF,Grind OFF|1=Touch ON,Grind OFF|2=Touch ON,Grind ON)`

Set the grinding mode programmatically as shown.

The commands below provide a programmatic way to set the grinding parameters.

`grind_touch_retract(touch_retract_mm)`

Set grind retract speed used after touchoff.

`grind_touch_speed(touch_speed_mm/s)`

Set speed used to go in for touchoff in Z.

`grind_force_dwell(dwell_time_ms)`

A dwell time performed when force mode is turned on to allow the robot to settle against the grind surface.

`grind_max_wait(max_time_before_retract_ms)`

If the tool is left in contact with the surface awaiting the next grind command, it will retract if this timeout is exceeded.

`grind_max_blend_radius(grind_blend_radius_mm)`

Sets the maximum blend radius that will be used in any pattern. This will be reduces for small geometries.

`grind_trial_speed(trial_speed_mm/s)`

Sets the speed used for “air grinding” when not in Touch + Grind mode.

`grind_linear_accel(accel_mm/s^2)`

Sets the linear acceleration used for grinding operations.

`grind_point_frequency(point_frequency_hz)`

Sets a point interpolation frequency used for complex figures. Obsolete.

`grind_jog_speed(trial_speed_mm/s)`

Sets the speed used when the grinding requires a robot move while not in contact with the part.

`grind_jog_accel(accel_mm/s^2)`

Sets the acceleration used for grinding m,oves not in contact with the part.

`grind_force_mode_damping(damping: 0.0 – 1.0)`

Sets the UR force_mode_damping parameter to assist in stabilizing force-mode performance.

`grind_force_mode_gain_scaling(scaling: 0.0 – 2.0)`

Sets the force_mode_gain_scaling parameter to assist in stabilizing force-mode performance.

Grind User Timers: Internal Use

Enabling these will time each grind operation and place it in a circular buffer of user_timers that can be returned to the variable list with `return_user_timers()`. Used primarily for internal testing.

`enable_user_timers(integer 0=off, 1=on)`

Turn the UR-internal user timers on or off.

`zero_user_timers()`

Zero all UR-internal user timers.

`return_user_timers()`

Return an array of timers. Each timer represents one grinding operation. Repeating the same grinding operations on different surface geometries can be used to validate Lecky Engineering's internal speed calibration system.

LElib.UR.grind Grinding Examples

Here are a few sequences that show the kinds of things that can be done in a recipe. The Examples subdirectory in the Code folder has many more complicated examples that you can examine (and run!).

These examples are shown in LEScript and require slight edits in Java or Python sequences.

`Remove Current Tool`

Just remove the current tool from the robot. As long as the one actually mounted is selected, this goes to the tool home followed by the mount/demount position and prompts the operator when it is time to remove.

```
# Remove Current Tool
# Go through demount procedure
# Assumes you have selected whatever tool is actually mounted!

prompt(Please confirm: you wish to demount {robot_tool}?)
```

```
move_tool_home()
move_tool_mount()
prompt(Please demount tool {robot_tool})

select_tool(none)
```

Install A Tool

This goes through prompting to mount a specific tool.

```
# Install 2F85
# Example to install a tool when none is currently installed
# We just select the new tool, move to the mount position, prompt
the operator, and move to tool_home

# Change to whatever tool you like
tool=2F85

# Operator confirmation
prompt(About to mount {tool})

# Mounting process
select_tool({tool})  # This only informs the robot what is
mounted

# This does the physical swap
move_tool_mount()
prompt(Please mount tool {tool})
move_tool_home()
```

Integrated Example

Here we start with the 2F85 tool ready to grind and swap tools and continue from the same location mid-recipe.

```
# Integrated Example
# Assumes we're where we want to grind initially but need to do a
tool swap mid-way

tool1=2F85
tool2=vertest

# Program assumes we are starting with tool1- verify internally
and with operator!
assert(robot_tool,{tool1})
prompt(Confirming tool {tool1} is currently mounted and you are
grinding on {robot_geometry})
```

```
# This will always be our grind_start position
save_position(grind_start)

# Do some grinding with tool1
move_linear(grind_start)
grind_rect(30,30,3,10,10,1)
grind_rect(20,20,3,10,10,1)

prompt(Ready to swap {tool1} to {tool2}?)
# Remove {tool1}
move_tool_home()
move_tool_mount()
prompt(Please remove {tool1})

# Install {tool2}
select_tool({tool2})
move_tool_mount()
prompt(Please install {tool2})
move_tool_home()

# Do some grinding with tool2
move_linear(grind_start) # Returns us to the starting position
grind_rect(30,30,3,10,10,1)
grind_rect(20,20,3,10,10,1)
```

Computed Concentric Circles

Here’s a test recipe that grinds 3 concentric circles explicitly and in a loop, not lifting until the final one.

```
# 26 Concentric Circle Test

# Old school
grind_circle(30,2,0.9,10,1)
grind_circle(20,2,0.9,10,1)
grind_circle(10,2,0.9,10,0)

# Do it with a loop
size = 30
count = 2
speed = 0.9
force = 10

repeat:
```

```
grind_circle({size},{count},{speed},{force},1)
size -= 10
jump_gt_zero(size,repeat)
```

Lots of Grinds

By pre-teaching points and swapping geometries, a whole day's work could be done (other than tool swaps!)

```
# Test all the patterns on all the geometries

size1=40
size2=10
count=3
speed=5
force=10

select_tool(2F85)
cycleCount=0

redo:
move_linear(demo_flat)

set_part_geometry(FLAT,0)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(CYLINDER,400.1)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)
```

```
set_part_geometry(CYLINDER,600.1)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(CYLINDER,800.1)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(CYLINDER,1000.1)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(SPHERE,400.2)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(SPHERE,600.2)
```

```
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(SPHERE,800.2)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

set_part_geometry(SPHERE,1000.2)
grind_line({size1},{size2},{count},{speed},{force},1)
grind_line(-{size2},{size1},{count},{speed},{force},1)
grind_rect({size1},{size2},{count},{speed},{force},1)
grind_rect({size2},{size1},{count},{speed},{force},1)
grind_serp({size1},{size1},1,3,{count},{speed},{force},1)
grind_serp({size1},{size1},3,1,{count},{speed},{force},1)
grind_circle({size1},{count},{speed},{force},1)
grind_circle({size2},{count},{speed},{force},1)
grind_spiral({size1},{size2},3,{count},{speed},{force},1)

cycleCount++
jump(redo)
```

INDEX