Remote Operation Guide

Table of Contents

1 Preface

The goal of this guide is to give users the tools to operate the robot without need for manual manipulation on the TeachPendant. Different interfaces will be covered to display the remote capabilities of the robot. Please note that this guide only covers PC-based interfacing. In addition, several snippets of code will be included as reference to some of the sections below. This is **sample code** and not meant to be a full interface. It is the responsibility of the user to develop proper GUIs/CLIs, integration tests, and error-handling based on the needs of their application.

1.1 Setup and Requirements

The following are requirements based on the time of writing this guide:

- An E-series robot running Polyscope 5.10+
    - Earlier software versions may not have all the functionality shown in the document (Interface Mode for example was introduced in 5.10)
- An ethernet connection to the robot
- Python 3.8.7
    - Similar to Polyscope, some functions may not behave as expected in earlier versions of Python
    - Users interested in other programming languages will need to translate the code over
- A TeachPendant (TP) for initial robot setup (which can be removed when moving to production)

It is recommended to have the robot on a static IP address. On the TP this can be found in the Hamburger menu -> Settings -> System -> Network (**Figure 1**):

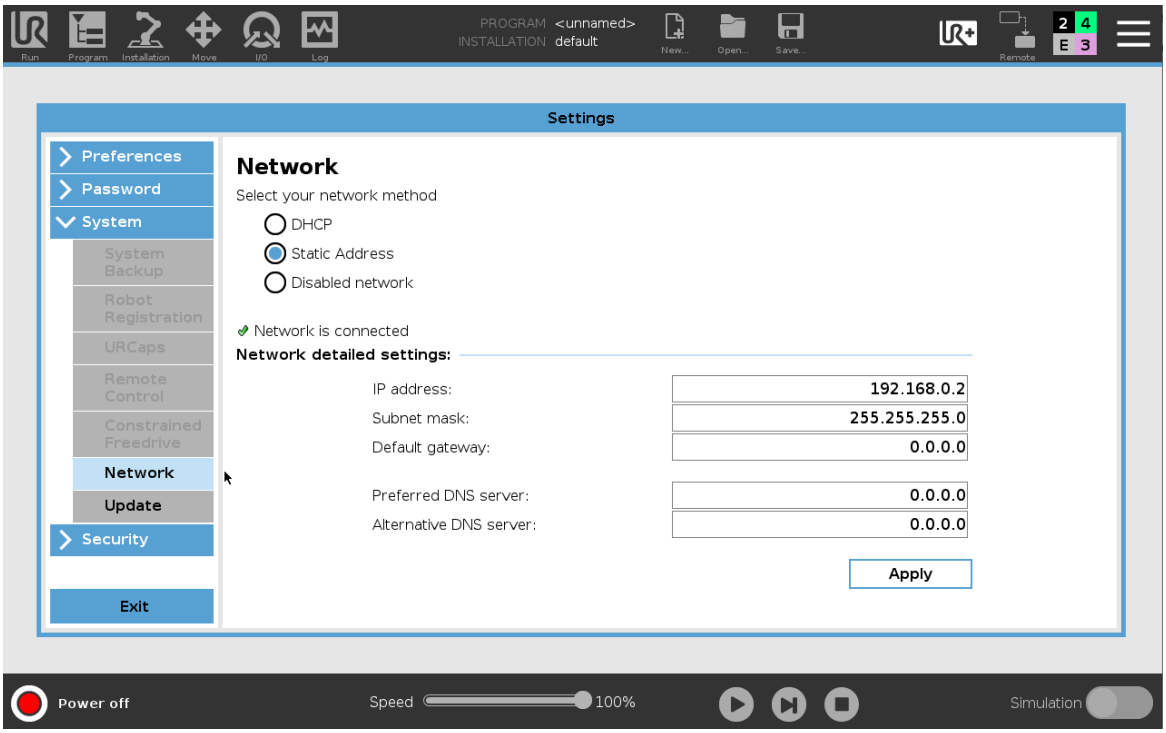


Figure 1: Polyscope network screen.

Make sure the robot is on the same network as the PC it's interfacing with under the 255.255.255.0 subnet mask. In general, issues connecting to the PC are related to either the Network settings of the robot/PC or an issue with the user's firewall or AntiVirus software. Try to disable both and then add an exception for the robot if communication starts functioning. In addition, several functions shared in this guide will require the robot to be in Remote Control mode. To enable, go to the Hamburger Menu -> Settings -> System -> Remote Control and hit "Enable." If successful, a new button will pop up next to the safety checksum in the top-right (see **Figure 2**).

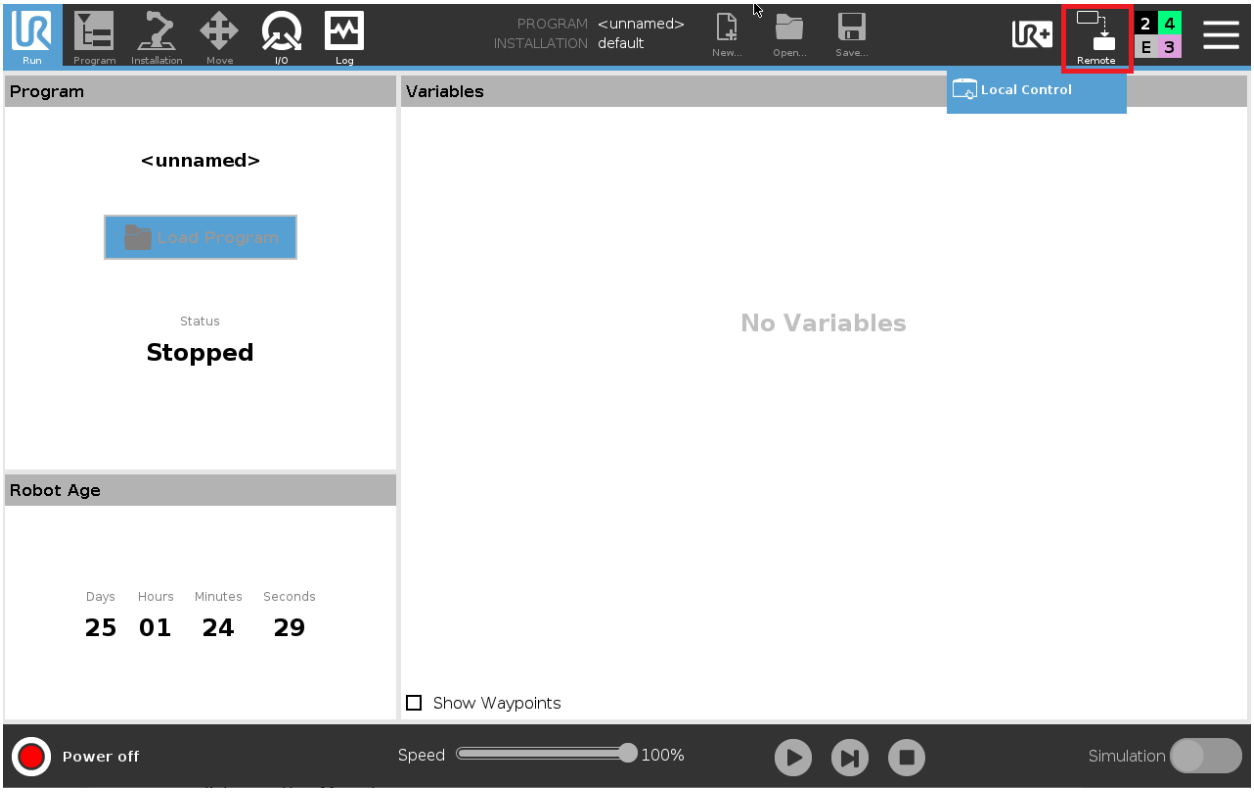


Figure 2: Control Mode Selection

2 Powering on The Robot

Use remote ON/OFF control to turn the Control Box on and off without using the Teach Pendant. It is typically used:

- When the Teach Pendant is inaccessible.
- When a PLC system must have full control.
- When several robots must be turned on or off at the same time.

The remote ON/OFF control provides an auxiliary 12V supply, kept active when the Control Box is turned off. The ON input is intended only for short time activation and works in the same way as the POWER button. The OFF input can be held down as desired. The electrical specifications are listed in the Polyscope User Manual.

Remote ON Button

Figure 3 illustrates connecting a remote **ON** button.

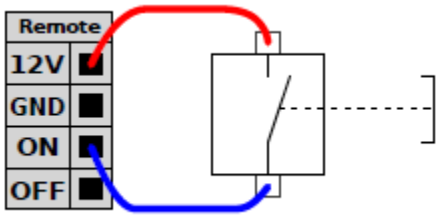
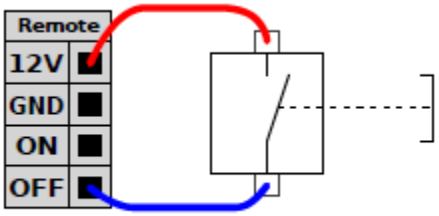


Figure 3: Wiring a Remote On switch.

Remote OFF Button

Figure 4 illustrates connecting a remote **OFF** button.



CAUTION

Do not press and hold the **ON** input or the **POWER** button as it switches off the Control Box without saving. You must use the **OFF** input for remote off control as this signal allows the Control Box to save open files and shut down correctly.

Figure 4: Wiring a Remote Off switch. Caution must be taken when activating these inputs.

3 Loading Programs

This section will cover different approaches to program loading. Sending script commands or files to the robot will be covered in the section "Commanding the Robot".

3.1 Choosing a Default Program on Startup

The user has an option of choosing a default program when the robot boots up. Please note, the "default" installation will *always* load up by default and this cannot be changed. It is therefore recommended to modify and save any changes into the "default.installation" if this feature will be used.

Navigate to the “Startup” menu inside the Installation tab and check “Load default program”:

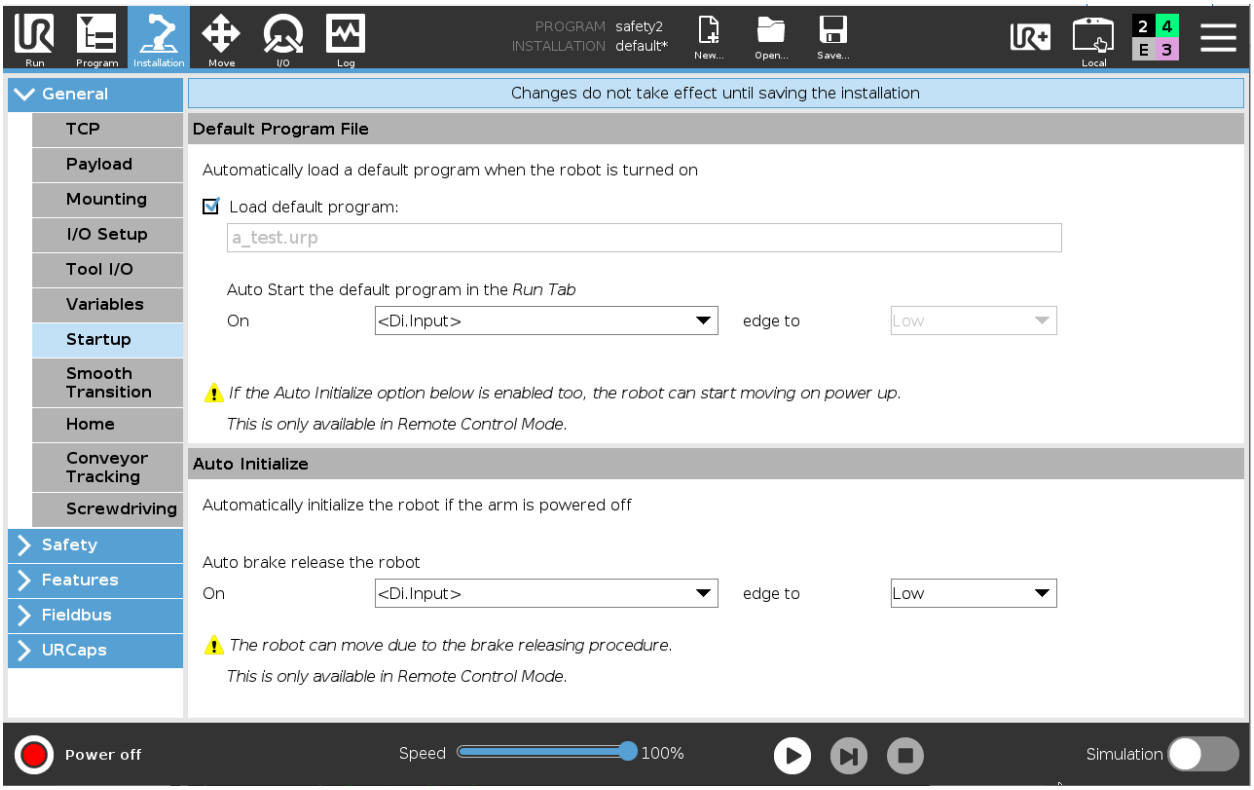


Figure 5: Startup options in Polyscope.

There are two additional options available on this screen: Auto Start and Auto Initialize (both of which require the robot to be in Remote Control mode). Auto Start will allow a digital input (DI) to automatically start the default-loaded program when the robot first boots up. Please note that this only works for the first boot, meaning if the program pauses or stops the Auto Start input *will no longer function until the robot is rebooted*. A normal “Play” function is available as an action tied to a DI; see the user manual for more information on this topic. Auto Initialize allows the user to have a DI which will automatically power up and release its brakes, thus putting it in a state to start running a program.

3.2 Loading Programs via Dashboard

The Dashboard server is an interface created to allow simple commands to the GUI over a TCP/IP socket. This server runs on port 29999 on the robot’s IP address and requires all commands sent to be terminated by a “\n” newline character. For more information, including a list of all available commands, please go to our Support site.

The sample code “Dashboard.py” or a program like SocketTest can be used to send commands and receive responses from the server. Loading a program from the robot SD card can be done by typing the command “load xxx.urp” as seen in **Figure 6** below:

```
Enter command: load safety1.urp
Loading program: /programs/safety1.urp, /programs/new folder_1/default_7.installation
```

Figure 6: Loading a program via Dashboard.

The robot will load the program and look for the associated installation file (the two do not have to be in the root program directory, but the program *does* have to be saved with its installation file). This command will fail if the robot is not able to locate the installation file (whether it was moved or in the case of this example – deleted). The command will also fail if the installation file's safety settings had changed without the program being saved with it (see **Figure 7** and **Figure 8**).

Python View:

```
Enter command: load safety1.urp
Error while loading program: /programs/safety1.urp:unknown failure
```

Figure 7: Error when trying to load a program with a different safety configuration.

Polyscope View:

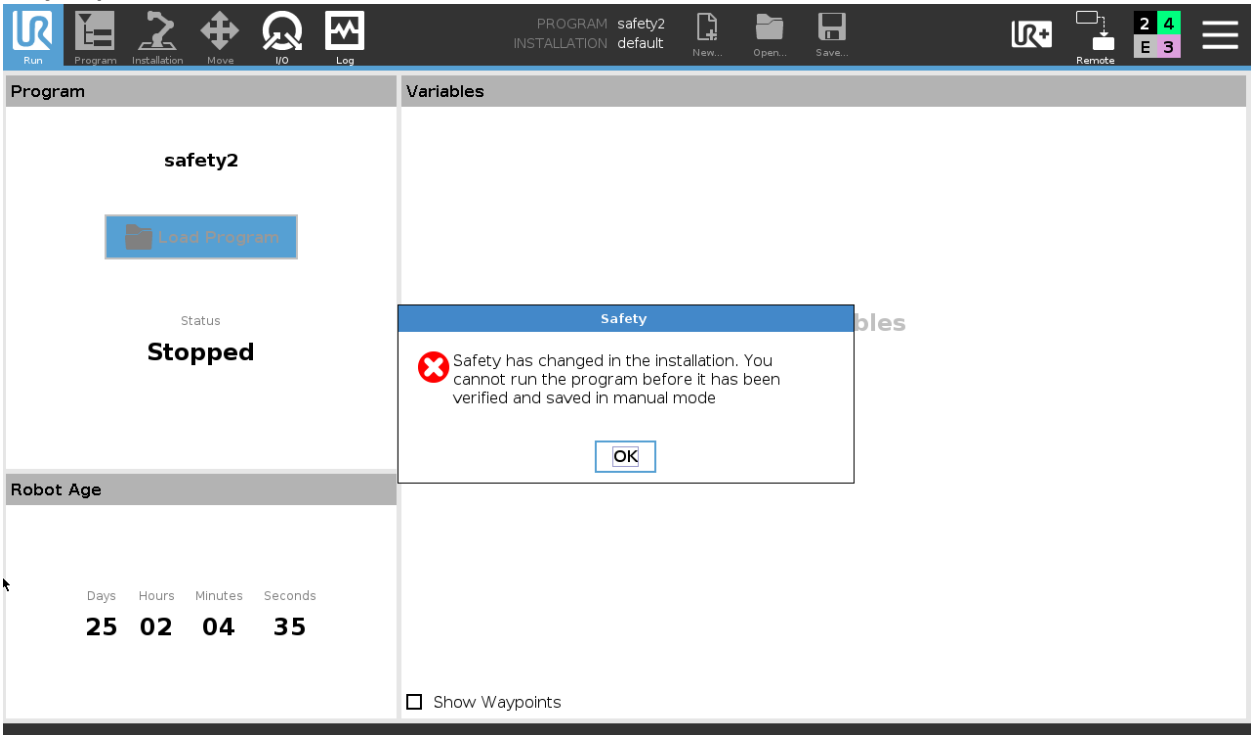


Figure 8: Polyscope error when attempting to load a program with a different safety configuration than what was saved last.

3.3 FTP

Users may want the capability to move programs, installation files, and scripts between the robot and their PC. This could allow them to maintain better version control than natively on the robot where this control is limited to good naming convention. To do so, an FTP program like Filezilla needs to be used. See the PDF "FTP Filezilla" for instructions on how to install it and connect to the robot. By default the user will connect to the root folder of the robot. Navigate up a level to find all programs and installations shown in **Figure 9**:

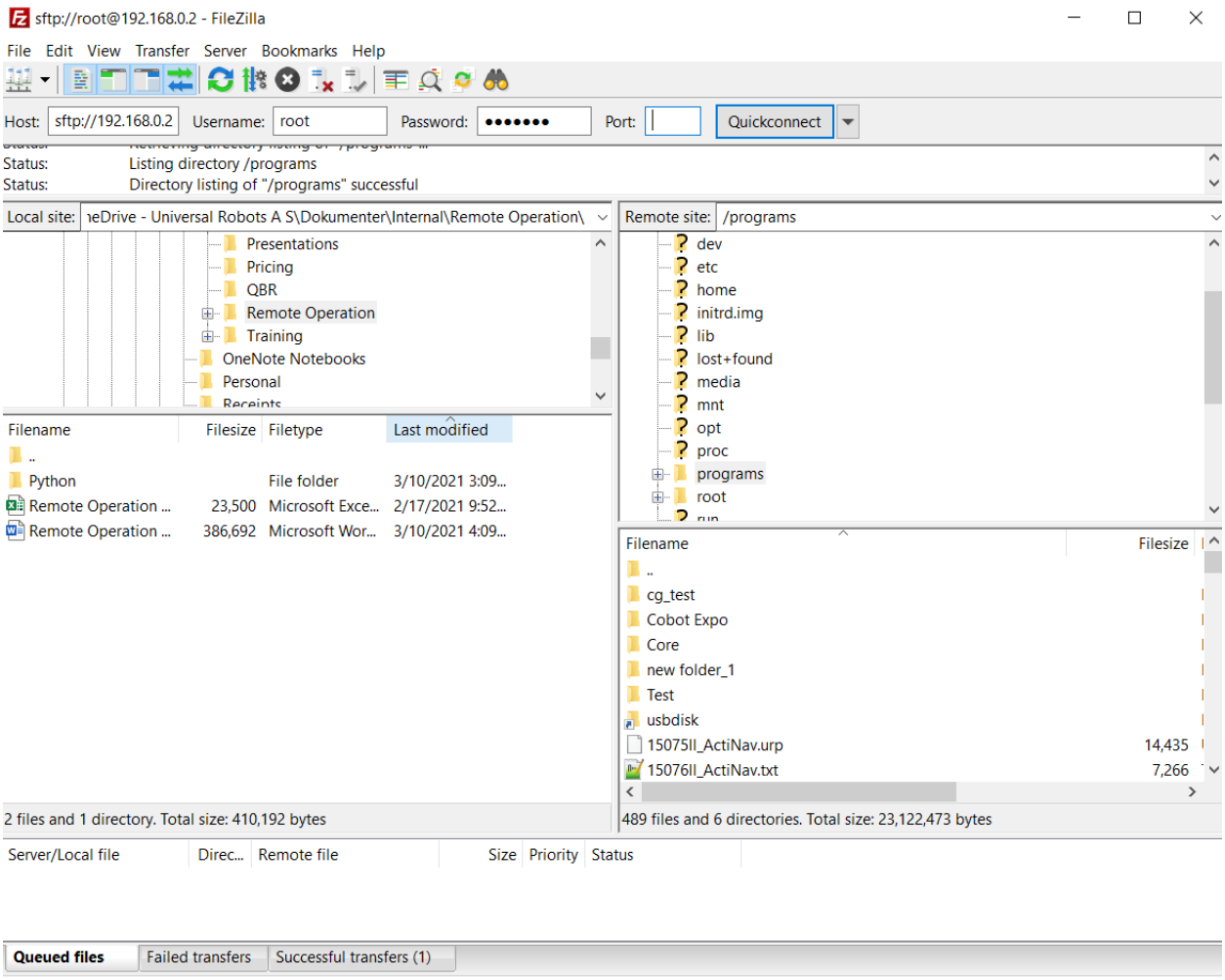


Figure 9: FTP Filezilla client after connecting to the robot.

From here programs, installations, and scripts can be dragged and dropped between the robot and PC.

***WARNING:** Take EXTREME caution when moving files between the robot and PC. Moving files not in "Programs" out of their folders can cause Polyscope to crash and require a brand-new SD card.*

4 Robot State & System Commands

There are several different interfaces available for gathering the robot's state (whether it is powered on, running a program, protective stopped, etc.). This section will cover those interfaces and their differences.

4.1 Dashboard

On top of loading programs, the Dashboard server can also get the robot's status and apply some system-level commands to it.

Figure 10, **Figure 11**, and Figure 12 show different system-level commands which can be remotely sent to the robot.

Python View:

```
Enter command: robotmode
Robotmode: RUNNING

Enter command: safetystatus
Safetystatus: NORMAL
```

Figure 10: Dashboard commands (shown in green) can be typed out in Python to query robot states.

Polyscope View:

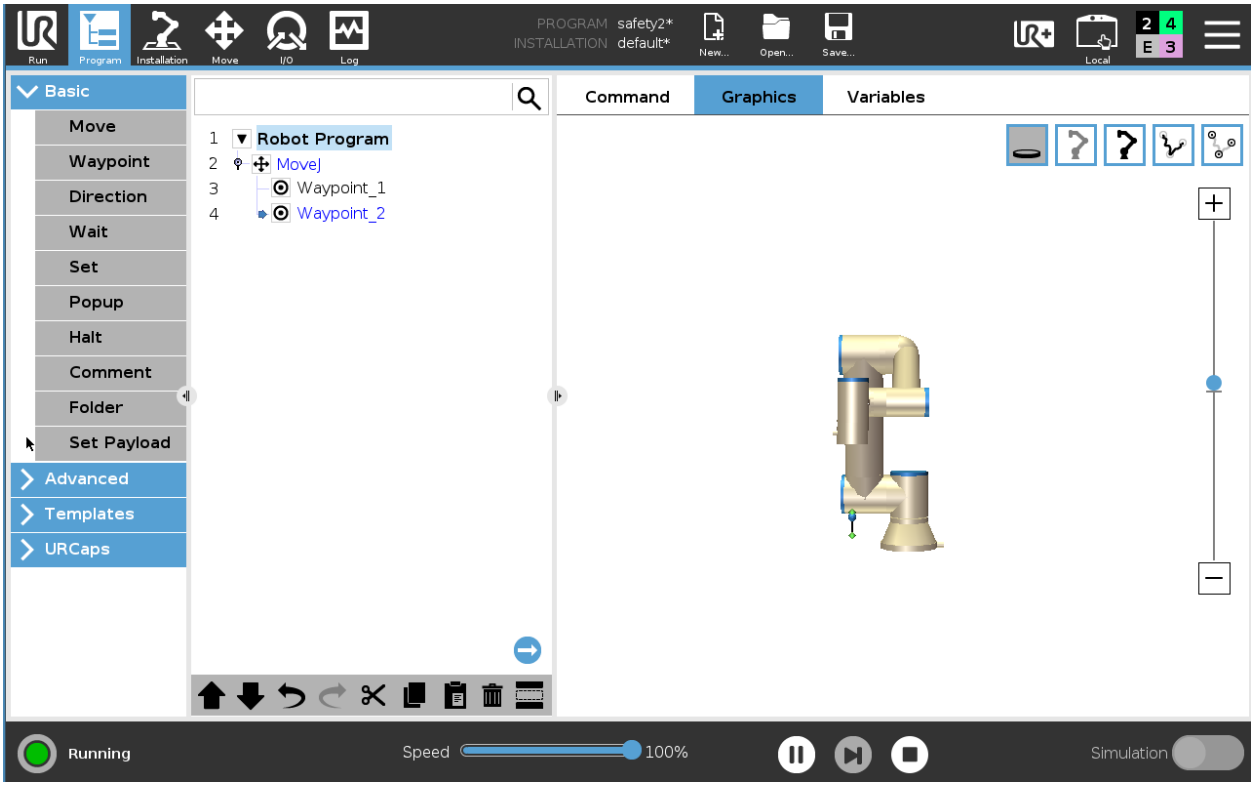


Figure 11: Robot mode in the bottom-left corner of Polyscope shown to be "RUNNING".

```
Enter command: power on
Powering on

Enter command: brake release
Brake releasing
```

Figure 12: Python view powering on and unlocking the robot remotely.

Note: if the controller has booted the user can simply type “brake release” to both power on the arm AND release its brakes, thus not needing to issue a “power on” command (however there may be circumstances where users want the robot in an Idle state for debugging). Other useful system commands include: “power off”, which will remove power from the arm but leave the controller on, “shutdown”, which will safely power off the arm AND turn off the controller, and “generate support file” which can be used to put the Support file onto a USB drive for robot debugging.

4.2 RTDE

The Real-Time Data Exchange (RTDE) interface provides a way to synchronize external applications with the UR controller over a standard TCP/IP connection, without breaking any real-time properties of the UR controller. This functionality is, among others, useful for interacting with fieldbus drivers (e.g. Ethernet/IP), manipulating robot I/O and plotting robot status (e.g. robot trajectories). The RTDE interface is by default available when the UR controller is running.

RTDE is split into two stages: the setup procedure and the synchronization loop. The setup procedure is accomplished through an XML recipe, or *data package*, that creates the combination of controller inputs and outputs necessary for operation. A sample XML used for the commanding the robot via RTDE (covered later in this guide) is shown in **Figure 13** below:

```xml
<?xml version="1.0"?>
<rtde_config>
    <recipe key="state">
        <field name="actual_TCP_pose" type="VECTOR6D"/>
        <field name="robot_mode" type="INT32"/>
        <field name="output_int_register_0" type="INT32"/>
        <field name="runtime_state" type="UINT32"/>
        <field name="safety_status" type="INT32"/>
    </recipe>

    <recipe key="set_q">
        <field name="input_double_register_0" type="DOUBLE"/>
        <field name="input_double_register_1" type="DOUBLE"/>
        <field name="input_double_register_2" type="DOUBLE"/>
        <field name="input_double_register_3" type="DOUBLE"/>
        <field name="input_double_register_4" type="DOUBLE"/>
        <field name="input_double_register_5" type="DOUBLE"/>
    </recipe>
```

Figure 13: RTDE recipe file in XML format.

In this case the recipe key "state" represents all of the chosen controller outputs to monitor for the application, while "set_q" represent all the inputs. The user can split up the inputs and outputs into as many keys as they desire, but inputs and outputs should remain under separate keys. Note that all the fields within each input key are sent when the "send" command is issued in Python (i.e. all the input double register values would be sent to the robot when "send(set_q)" would be executed).

Once the user sets up and sends the recipe to the robot, the synchronization loop can begin. This typically begins with receiving a packet of data (via the "receive()" command through the RTDE Python module) and then sending inputs to the robot which can be joint positions, cartesian points, register data etc. For more information, please see the article on RTDE which also includes an example control loop to run on the robot and PC.

4.2.1 RTDE Runtime Monitor Example

This example utilizes RTDE to show the robot's runtime state while executing a program. The Python file to run this is called "rtdeState.py" and its accompanying configuration file (or recipe) is called "rtdeState.xml". The first thing the main loop does is create an instance of "RtdeState"; this is a class created so that the user can either run the Python file by itself or utilize it in other pieces of code. As it uses a lot of RTDE's core methods, it may be beneficial to modify this code to inherit from the RTDE class. The main loop is shown below:

```python
if __name__ == "__main__":
    state_monitor = RtdeState(ROBOT_HOST, config_filename)
    state_monitor.initialize()
    runtime_old = 0
    while state_monitor.keep_running:
```

```
        state = state_monitor.receive()

        if state is None:
            break

        if state.runtime_state != runtime_old:
            logging.info(f'Robot program is
{state_monitor.programState.get(state.runtime_state)}')
            runtime_old = state.runtime_state

    state_monitor.con.send_pause()
    state_monitor.con.disconnect()
```

RtdeState requires a robot IP address and the path to the recipe XML file (which for this example are preset at the top of the Python file). The code will then run an initialization method which will try to connect to RTDE and set up all the inputs and outputs requested by the recipe. Note that the default output frequency set in the "RtdeState" class is set to 500Hz but it can be optionally changed if necessary. Once started, the program enters its main loop governed by a "keep_running" flag. Users will want to use this flag in their interface to break out of the main loop and shut down their connection to RTDE. The main loop will receive a packet of data (based on the list of outputs in the XML recipe) and then check to see if the program state has changed from the last iteration. If for some reason no packets of data are received (usually due to a lost connection with the robot), the loop breaks and RTDE safely shuts down. The program is set up to log these messages and output them to the terminal. **Figure 14** illustrates the Python output when a robot program is started, paused, resumed, and then stopped again:

Python View:

```
INFO:rtde:RTDE synchronization started
INFO:root:Robot program is Stopped
INFO:root:Robot program is Playing
INFO:root:Robot program is Pausing
INFO:root:Robot program is Paused
INFO:root:Robot program is Resuming
INFO:root:Robot program is Playing
```

Figure 14: Python output from running the RTDE monitor example.

4.3 Primary/Secondary Interface

The Primary and Secondary interfaces provide servers to send robot data and receive URScript commands directly. For more information, please see the "Overview of Client Interfaces" on the UR Support site.

A caution to those looking to utilize these interfaces to pull robot state data: all messages are output in a constant datastream and therefore must be manually parsed by a custom, user-made program. Additionally, the order of those bits are subject to change at any time during a software update (however this is generally mentioned in release notes). It is therefore encouraged to utilize a different interface that presents this data in a more organized manner (such as the RTDE interface shown above).

5 Commanding the Robot

In a typical application, the robot will be programmed by a user on the Teachpendant utilizing Polyscope's nodes as well as any nodes coming from installed URCaps. When the "play" button is pressed on a new or unsaved program, URScript is generated based on the nodes in the Robot program along with the current Installation settings and any additional URCap contributions.

Instead of utilizing Polyscope directly users looking to operate the robot remotely may want to send the robot positional data, joint targets, or simply send full URScript files over from their external system. The following sections outline different strategies to command the robot remotely.

NOTE: Users choosing to utilize the official Universal Robots ROS driver should seek the Github page linked in the Reference section as it is outside the scope of this guide.

Here we will review different ways to send the robot data via RTDE, socket, and XML-RPC. We will also review how to stream waypoints and send script commands. I'll include some sample Python and Polyscope programs to showcase the different interfaces.

5.1 RTDE

Controlling the robot via RTDE is done by sending TCP or joint position data to the robot's general purpose registers. From there the robot can utilize the data either via variable waypoints or servo/speed commands. This guide will cover two examples for controlling the robot via RTDE: a "setp" case where the robot alternates between two waypoints being fed by the PC and a "servo" case where the robot will follow a path being streamed to it at 500Hz.

5.1.1 SetP Waypoint Example

This example was taken directly from the RTDE Support page article and utilizes the files inside the "rtde_examples" folder. The active robot program needed to execute this is called "rtde_control_loop.urp" while the active Python program is called "example_control_loop.py". This example also uses the "control_loop_configuration.xml" file as the recipe for RTDE.

The Python code begins similarly to the initialization of the RtdeState class in that it requires a robot IP address, a port (set to 30004 which is what RTDE uses), and a recipe file. From there it sends the requested inputs and outputs (from the recipe XML file) to RTDE. Inputs include 6 input double (float) registers for holding each component of the requested robot pose as well as a single integer register for RTDE's watchdog (covered in the Polyscope section below). The output side includes the targeted "q" (joint positions) and "qd" (joint velocities) as well as an output integer register used for syncing up the robot and PC (note that the "q" and "qd" outputs are not actually used in this example). The main control loop is shown below:

```
# control loop
while keep_running:
    # receive the current state
    state = con.receive()

    if state is None:
        break;

    # do something...
    if state.output_int_register_0 != 0:
```

```
        new_setp = setp1 if setp_to_list(setp) == setp2 else setp2
        list_to_setp(setp, new_setp)
        # send new setpoint
        con.send(setp)


    # kick watchdog
    con.send(watchdog)
```

The first thing done is receiving a data packet from the robot (called “state”) followed by a check to break out of the loop if no data is returned. The next section of the loop only runs if the output integer register is not zero. This register is changed on the robot side at the beginning of its program and acts to sync the execution of the PC code with the robot code. Once changed from zero, the Python code swaps between the two setpoints (setp1 and setp2) using two helper functions called “setp_to_list” and “list_to_setp”. After sending the setpoint to the robot the code also sends the watchdog to confirm communication for the robot and the loop starts over again. Once the loop is broken, the program will send a pause signal to RTDE, followed by a safe disconnect to end the connection between the two machines.

Polyscope View:

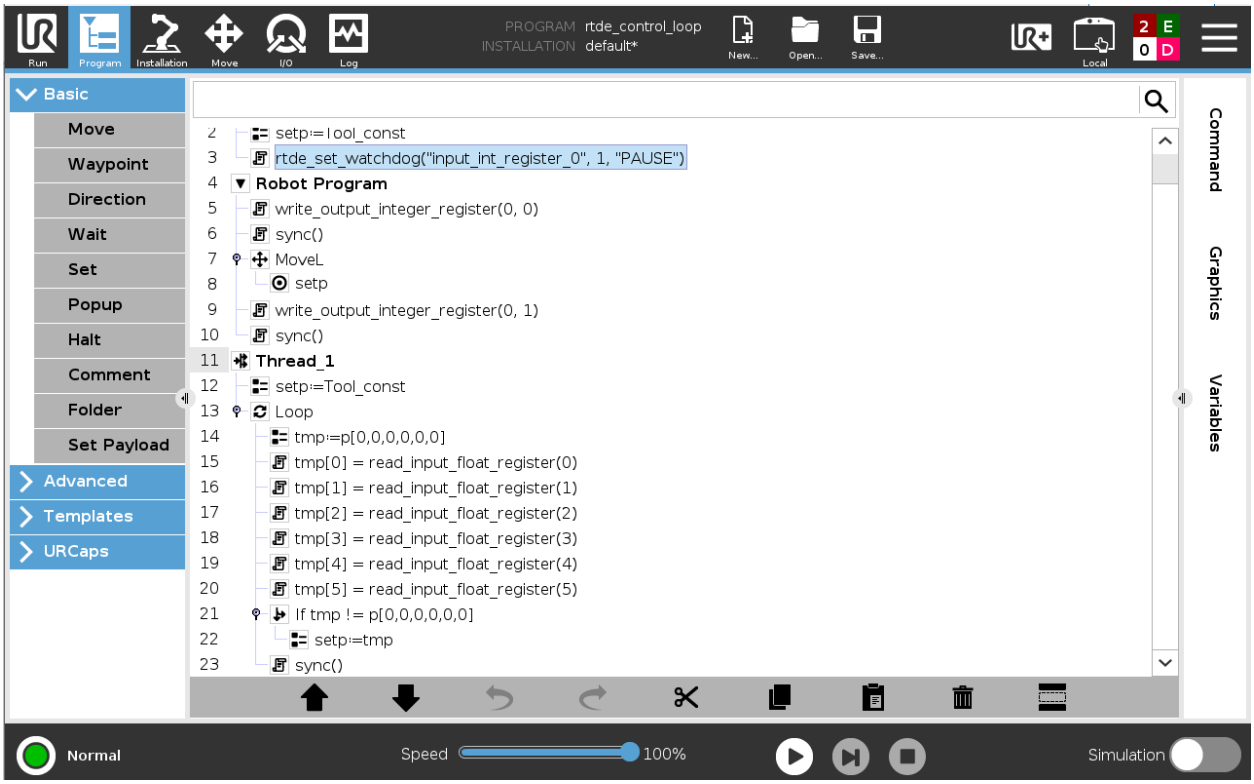


Figure 15: Polyscope code for the SetP example.

The Polyscope program is shown **Figure 15** above. A “Before Start” sequence assigns the variable “setp” to the “Tool_const” pose which is the current value of the TCP relative to the robot Base Feature (the same value which would be returned by “get_actual_tcp_pose()”). The next line sets up the watchdog by calling the function “rtde_set_watchdog”. This function requires a register, in this case input integer register 0, to be updated at a specified frequency which this example is set to 1Hz. There are three default

actions if the register is not updated: pause, stop, or ignore (with “pause” being the default action). In the case of pause and stop, the robot will throw the following error in **Figure 16** if the watchdog is not updated:

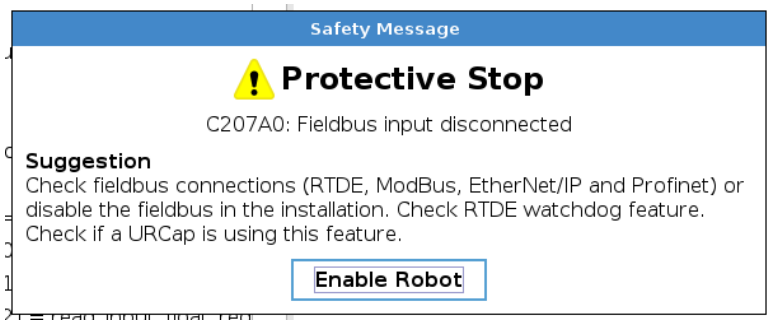


Figure 16: Protective stop which will pop up if the robot does not detect input from RTDE.

The only difference between the two actions are whether the robot program pauses or completely stops after this error is triggered. As this is a protective stop, the user needs to unlock and re-enable the robot via the Teachpendant or the Dashboard server if the condition is triggered. The final condition “ignore” will simply ignore the watchdog not being updated and continue the program as normal.

The main robot program has the following logic:

1. Set the output register used by the Python code to zero to prevent it from sending any new setpoints.
2. Move to the current value of “setp”.
3. Set the output register to one to allow the Python code to send a new setpoint.

It is good practice to add a sync() or wait(0.01) statement after each input register write in order to keep the controller synced up.

A Thread is also added which executes in parallel to the robot program. This thread simply reads the six input registers that the PC is sending the decomposed pose variable to and composes them into a readable pose by the robot. A quick check is done before that pose is assigned to the “setp” variable the main program uses. This is to make sure the values received aren’t all zeros in the case that no new data is sent in the frame (or timeslice) the thread grabs the register values.

Note: Because a watchdog is set up in this example, users will want to start their Python code *before* their robot program or they will most likely trigger a protective stop due to that watchdog not being updated within a second.

5.1.2 Servo Example

This example will walk through one strategy for streaming joint positions to the robot. Users who want to see the path the robot will take beforehand can load the “sendPath.urp” program on the robot and play it. The path was recorded using a helper script called “rtdePathRecorder.py” that grabs the joint positions of the robot as it traverses its path and then appends them to an Excel file called “path500.csv”. The robot program used for this example is called “rtdeCommanding.urp” and the associated Python file is called “commanding_RTDE.py”. It’s very important users have a good connection with the robot or the execution

of the given path will become jittery. For best results, start the robot program *before* starting the Python program.[1]

This example is based on the "servoj" command in URScript. "Servoing" is a method of online realtime control whereby the user provides joint positions for the robot to try to achieve within a given timeslice. It is different from typical programming as the user becomes responsible for accelerating and decelerating the robot by nature of providing different joint angles for the robot to reach in those fixed timeslices (or "frames"). The user can similarly instruct the robot accelerate in cartesian or joint space via the "speedl" or "speedj" commands. For more information, please see the script manual which can be found on the UR Support site.

The Python code begins similarly to the previous "setp" example in that it asks for a robot IP, port, and recipe file location. However it also asks for a file location for the joint positions that will be streamed to the robot, in this case called "path500.csv". In addition, a helper function is taken from the "setp" example that will convert a list of values to joint positions that can be sent to the robot. Before initializing RTDE, the program reads the list of joint positions from the CSV file and decomposes them into individual joint values that will be read and sent to the robot. After this is established, RTDE is initialized via the RtdeState class and input double registers are set up that will hold the joint positions on the robot side. At this point, a handshake takes place where the Python code waits to receive a value of "1" from the robot's output register and then it sends a "1" back to the robot's input register for the robot to enter its servo loop. This is simply one strategy for syncing the robot and PC; users may need a more robust methodology for their applications.

The main Python loop is shown below:

```
# Main control loop. Receive an output packet from the robot and then send the next
joint positions.
for i in range(len(q1)):
    rtde.receive()
    list_to_set_q(rtde.set_q, [q1[i], q2[i], q3[i], q4[i], q5[i], q6[i]])
    rtde.con.send(rtde.set_q)
```

The main loop will iterate through the range of joint positions taken from the CSV file. Each iteration works by receiving an output packet from the robot, moving the list of joint positions at that iteration to the robot's input registers, and then sending that data to the robot.

It's important to note that the robot expects the PC to time the sending of joint positions at the frequency specified in the "servoj" command (up to 500Hz or every 0.002 seconds for E-series robots). In other words, if the user sent 100 joint positions in one frame, the robot would only execute the 100th position and throw out the rest. The challenge here is that general operating systems like Windows (which this guide was developed in) cannot reliably execute commands within the given frame. They are designed to run a variety of different applications simultaneously, making sure all tasks get some amount of processing time. This example gets around that by simply receiving a packet of data from the robot before sending a new joint position. The robot can only output packets at a max rate of 500Hz which then limits how fast

[1] *If for some reason the robot program is stopped before the output integer can be reset to zero on line 16 of the robot program, users should add in a "wait" statement between lines 10 and 11 that allows them time to start the Python program. Actual applications should develop a more robust handshaking between the robot and PC.*

the PC can send the next position back. This approach has one possibly major drawback: processor load. On the robot side, output packages can be skipped if the controller is loaded too heavily as the real-time loop inside the controller takes priority. If the controller doesn't output a package in the given frame, the PC will not update the robot with a new position and the robot motion may become jittery. On the PC side, it's possible that other tasks may get scheduled when the robot is expecting new data and the same jitter would be seen. Many users employ realtime operating systems, such as the PREEMPT-RT kernel on Linux, in order avoid this issue. These systems enforce task scheduling more strictly and allow for higher degrees of precision when it comes to timing.

Once all the joint positions have been sent to the robot, the input register is reset back to zero to tell the robot to stop its servo loop and the Python program disconnects from RTDE. **Figure 17** below outlines the Polyscope program.

Polyscope View:

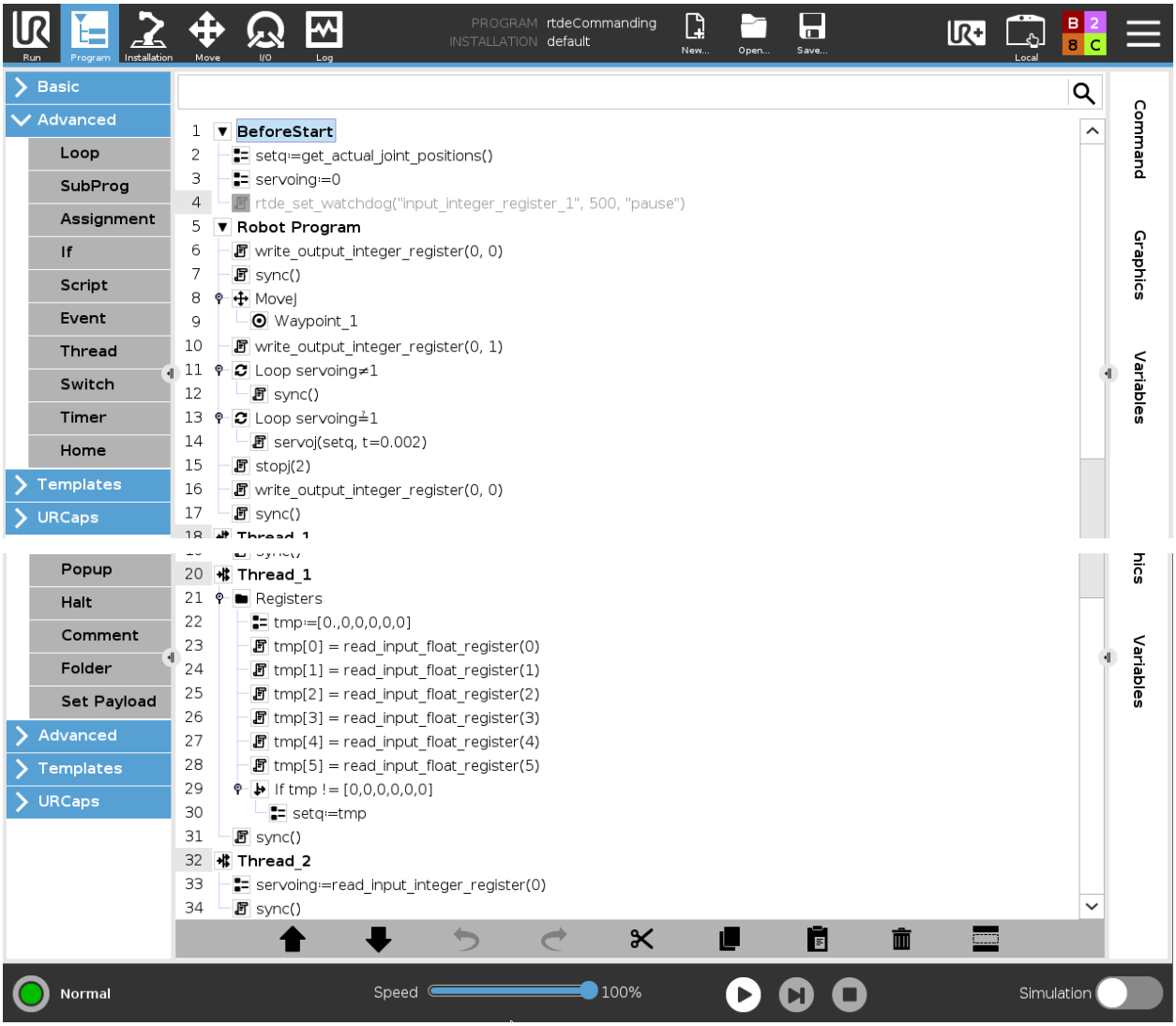


Figure 17: Polyscope code for the Servo example.

The main robot program begins by setting its output register to zero and moving to the start position of the servo path. When that's complete it sets the output register to one to let the PC know it's ready to begin servoing. It then loops and waits for the servoing variable to switch to one (checked through a Thread). Once the handshake it complete, the robot enters the servo loop which is simply a "servoj" command. The "servoj" command has optional "gain" and lookahead_time" values which dictate how aggressively and quickly the robot reacts to the commands it receives; in this case the default values were used. One thing to note is that the default timeslice is set to 8ms (the clock speed for CB-series) so users may want to reduce that to get finer control over the robot as is done here. It's also recommended to put a "stopj" or "stopl" command directly after the servo loop as that will force the robot to safely decelerate in joint or cartesian space (respectively) to a complete stop. Otherwise users may experience very hard stops from the robot that could potentially trigger a protective stop notification. There are additionally two threads added to the program. The first one mimics the earlier "setp" example in that it reads its input float registers and assigns them to a joint position array called "setq" which the robot uses for the servoj command. The second thread simply reads its input integer register which is used for the handshaking between the robot and PC.

5.2 Interpreter Mode

Interpreter mode enables the user to send and execute any valid URScript statements at runtime. When entering Interpreter mode the controller opens an interpreter socket (port 30020) where it accepts these statements, appends them to the running program and executes them in the scope of the dedicated interpreter thread. There are several things user should be aware of when using this mode:

1. Statements in Interpreter mode are inherited from the scope in which it was called. Declaring new global variables inside Interpreter mode is not supported.
2. All statements and functions need to be sent as a single line (the effect of this will be shown in the example below).
3. Multiple statements can be sent on a single line but will only execute if they can be compiled.
4. If interpreted commands are not cleared periodically then a "runtime too far behind" error can pop up. It's recommended to clear the buffer after 500 commands have been executed. Otherwise, commands will be cleared upon exiting of the "interpreter_mode" node (unless the user specifically decides to prevent this via an optional argument).

5.2.1 Interpret Example

This example will go over one method to send a full URScript program over to the robot from the PC. Previously the only way to do this was to send scripts line by line or wrapped inside a function via the Primary or Secondary interfaces (ports 30001 and 30002). The issue with this resides in the fact that those interface calls are not blocking, meaning the user is responsible for knowing exactly when the robot has completed its requested action. It is very easy to send a move command to the robot before it has completed the one previous, resulting in hard stops and unintended paths.

The example utilizes the "interpret.urp" program on the robot side as well as the "commanding_Interp.py" program on the PC side. The script being sent has the robot complete a cyclic pick and place where gripper commands to open and close are replaced by 1 second waits. Users can preview the robot path by loading and playing the "pickandplace.urp" file on the robot. The following will be executed by running the Python file:

1. The robot arm will attempt to power on if not already.
2. The robot will try to find the “interpret.urp” program, load it, and start playing it (this requires the user to enter the path to the program at the top of the Python file).
3. The URScript program will be sent over to the robot which contains the pick and place instructions.
4. The robot executes the pick and place loop once and then the program stops.
5. If during the execution of the program the robot encounters a protective stop, the Python console will prompt the user to decide if it is safe to continue running the program. If they type “y” or “yes”, it will log the decision and wait 5 seconds, unlock the protective stop, and continue the program. If they type “n” or “no”, it will stop program execution.
   NOTE: In actual applications the robot should not be regularly protective stopping. Users need to conduct a risk assessment to confirm this remote clearing of the stop is safe.

The Python code begins in a similar fashion to previous examples, requiring the robot’s IP and RTDE config file. It additionally asks for the URScript text file and the path to the “interpret.urp” program in the robot. Note that the “root” where programs are stored (i.e. when selecting “Open-> Program” at the top of Polyscope) is actually in “/programs/” so if the “interpret.urp” is stored there the path would become “/programs/interpret.urp”. The next couple sections will break up the Python code in order to talk through the general logic of each.

5.2.1.1 Robot Boot

```
def robot_boot():
    # Check to make sure robot is in remote control.
    remoteCheck = dash.sendAndReceive('is in remote control')
    if 'false' in remoteCheck:
        logging.error('Robot is in local mode. Cannot issue system commands.
Exiting...')
        shutdown()
        sys.exit()
    # Check robot mode and boot if necessary.
    powermode = dash.sendAndReceive('robotmode')
    if 'POWER_OFF' in powermode:
        logging.info('Attempting to power robot and release brakes.')
        logging.info(dash.sendAndReceive('brake release'))
        temp_state = rState.receive()
        bootStatus = temp_state.robot_mode
        # Grab robot mode from RTDE and loop until robot fully boots. Monitor safety
state
        # and exit if safety state enters a non-normal mode.
        while bootStatus != 7:
            temp_state = rState.receive()
            bootStatus = temp_state.robot_mode
            safetyStatus = temp_state.safety_status
            if safetyStatus != 1:
                logging.error('Robot could not boot successfully. Exiting...')
                shutdown()
                sys.exit()
        logging.info('Robot booted succesfully. Ready to run.')
```

This function attempts to boot the robot from a “Power Off” to “Normal” state. The first thing done is a check to make sure the robot is in Remote Control mode as most of the commands require this to work.

If unsuccessful, the program disconnects from the robot interfaces and the Python code stops. Following this, the program will check the robot's mode and issue a "brake release" to power it into a "Normal" mode. As an error-catch, the safety status of the robot is monitored over RTDE such that an error will get written and the program will exit if it the robot goes into a safeguard stop/violation/fault etc. Some users may want to handle something like a safeguard stop, reduced mode, or protective stop differently. If this is the case, the "if safetyStatus != 1" block should be re-written to reflect the change in behavior.

5.2.1.2 Starting the Robot Program

```
def startRobotProgram():
    # Check to see if currently loaded program is Interpret.urp
    currentProgram = dash.sendAndReceive('get loaded program')
    # Users should add error handling in case program load fails.
    if not currentProgram.endswith('interpret.urp'):
        dash.sendAndReceive(f'load {pathToRobotProgram}')
        logging.info('Found and loaded Interpret.urp')
    else:
        logging.info('Interpret.urp already loaded.')
    dash.sendAndReceive('play')
    # Check program status via RTDE and wait until it returns a "Playing" state.
    temp_state = rState.receive()
    startStatus = temp_state.runtime_state
    while startStatus != 2:
        temp_state = rState.receive()
        startStatus = temp_state.runtime_state
    logging.info('Playing program')
```

This function checks for the currently loaded program and if it does not end with "interpret.urp" it sends a command to load it based on the path given at the top of the Python code. As stated in the Dashboard section of this guide, the command will fail if "interpret.urp" was saved with a different safety configuration than what is currently loaded on the robot. For this reason, users should first manually load and save the program or simply re-create it themselves as it only contains one script line (shown at the end of this example). This function will then check the robot's program state and loop until it receives the signal that the program is playing.

5.2.1.3 Protective Stop Recovery

```
def pStopRecover():
    response = input('Is the robot safe to continue running? (y/n): ')
    if response == str.lower('y') or str.lower('yes'):
        # Log operator confirming safety.
        now = dt.datetime.now()
        logging.warning(f'[{now:%Y-%m-%d %H:%M}]Operator confirmed cell safety.')
        logging.warning('Robot restarting in 5 seconds.')
        # Dashboard requires 5 second wait before unlocking a pstop.
        time.sleep(5)
        logging.warning('Restarting...')
        dash.sendAndReceive('unlock protective stop')
        temp_state = rState.receive()
        safetyStatus = temp_state.safety_status
        runStatus = temp_state.runtime_state
        # Wait until robot fully restarts from protective stop.
        while safetyStatus != 1 or runStatus != 4:
            temp_state = rState.receive()
            safetyStatus = temp_state.safety_status
```

```
            runStatus = temp_state.runtime_state
        time.sleep(1)
        dash.sendAndReceive('play')

    else:
        logging.warning('Unable to verify cell safety. Please manually clear issue.
Exiting...')
        shutdown()
```

This function is triggered in the main loop of the Python program if it detects the safety status of the robot entering a "Protective Stop" mode. It first will prompt the operator on the console to decide whether the robot is in a safe state to unlock its protective stop and continue working. Selecting "y" or "yes" logs the date and time the operator confirmed cell safety. Dashboard requires users to wait five (5) seconds before unlocking a protective remotely after which the Python program issues the unlock. When the stop is cleared the function automatically continues playing the robot program and URScript.

5.2.1.4 Main Program (setup)

```
# Set up Dashboard monitor.
dash = Dashboard(ROBOT_HOST)
dash.connect()
# Set up RTDE monitor.
rState = RtdeState(ROBOT_HOST, rtde_config, frequency=500)
rState.initialize()
robot_boot()
startRobotProgram()
# Set up Interpreter.
interpreter = InterpreterHelper(ROBOT_HOST)
interpreter.connect()
# Attempt to call an interpreter function. Exit if robot program is stopped.
try:
    sendFile.send_cmd_interpreter_mode_file(interpreter, robotCommands)
except Exception as e:
    # Look for "invalid state" in the interpreter error message. Raise exception
otherwise.
    if 'invalid state' in e.args[1]:
        logging.warning('Robot program state invalid. Exiting...')
        shutdown()
    else:
        raise

runtime_old = 0
```

The main program starts by setting up connections to Dashboard and RTDE. Following this, the robot will power on and begin playing its program. Interpreter mode is then initialized, and the program attempts to send the URScript file over to the robot. This is encapsulated in a "try-except" block as there is a possibility the robot program has been stopped remotely via Dashboard or other sources. If this happens, Interpreter mode throws an exception which says the robot is in an invalid state. Users may want to add error catching in their Interpreter module to prevent the need for this check in their main program.

5.2.1.5 Main Program (loop)

```
while rState.keep_running:
    state = rState.receive()

    if state is None:
        logging.error('No RTDE data received. Exiting...')
        break

    if state.runtime_state != runtime_old:
        logging.info(f'Robot program is {rState.programState[state.runtime_state]}')
        runtime_old = state.runtime_state
        if state.runtime_state == 1 or state.runtime_state == 0:
            logging.info('Robot program was stopped. Exiting...')
            break

    if state.safety_status == 3:
        logging.error('Robot went into Protective Stop.')
        pStopRecover()
```

The main loop in the program is similar to the RTDE section in that it receives a packet of data and breaks the loop if nothing was returned. The next section checks the runtime state of the robot, exiting the program if it detects the robot was stopping or stopped. If the robot went into a protective stop, the program will log it and enter the protective stop recovery function covered previously.

```
    # Safely exit interpreter mode once commands have been executed. Handle edge case
where the robot
    # is already stopped by the time this part of the main loop executes.
    try:
        linesleft = interpreter.get_unexecuted_count()
        if linesleft == 0:
            logging.info('All commands completed. Exiting...')
            interpreter.end_interpreter()
            time.sleep(2)
            dash.sendAndReceive('close popup')
            break
    except Exception as e:
        if 'invalid state' in e.args[1]:
            logging.warning('Robot program state invalid. Exiting...')
            break
        else:
            raise
```

An advantage of Interpreter mode over the Primary/Secondary interface is its built-in state commands. As all valid commands get a unique ID and provide an acknowledgement by the system, several functions were introduced to provide information on the current state of the running program. In this example the Python code tracks the amount of lines left and if the response is zero, attempts to end the interpreter and break the main loop. Users looking to loop their programs could consider adding the "send_cmd_interpreter_mode_file()" function call to the above "if" statement but should also clear the interpreter buffer to prevent runtime issues. Additionally, users should be aware that command ID's will eventually reset to one if the value of the ID gets too high.

Polyscope View:

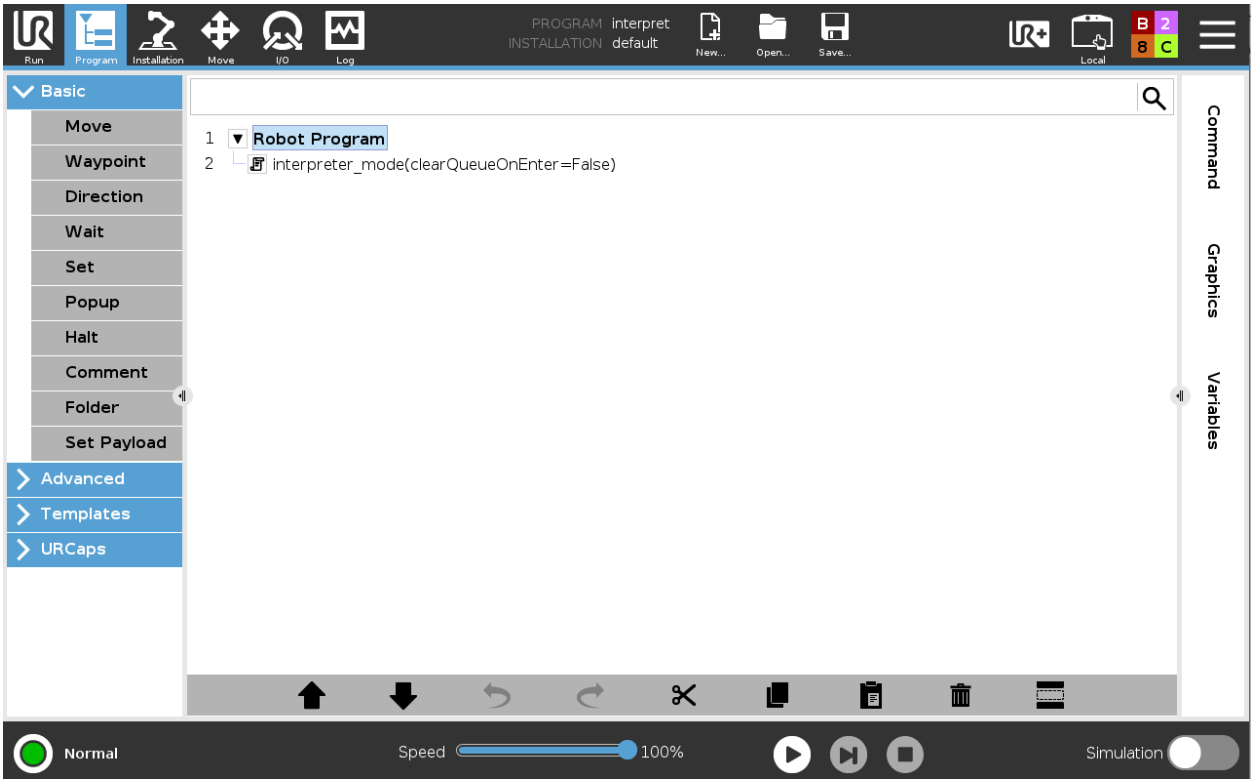


Figure 18: Polyscope code for the Interpreter example.

The Polyscope program shown in **Figure 18** simply contains a script function calling the interpreter mode with the option for clearing the buffer on entry set to False. This was done for the robot to execute the incoming commands instead of clearing them (when set to True the robot would do nothing as it had nothing to execute).

5.3 Socket TCP/IP

Sockets are a common way for messages to be sent across a network. UR robots commonly communicate with outside equipment like cameras and PC's using this protocol. Users should consider this method for passing part counts, job numbers, positional offsets, or any information that does not require realtime feedback. This example creates a simple socket server in Python which sends the robot a position to move to upon request (very similar to the way a camera provides offsets after identifying a part). The robot program being run is called "Socket_example.urp" while the Python program is called "socket_server.py."

Python View

```
import socket

HOST = '0.0.0.0'
PORT = 50000
pose = '(-0.15,-0.5,0.3,0,-3.14,0)'

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
```

```
    s.listen()
    conn, address = s.accept()
    with conn:
        print('Connected by', address)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            if "get pose" in str(data):
                conn.sendall(pose.encode())
                print('Data sent.')
```

This example starts off with a host and port for the socket server to bind to. Note that the IP address “0.0.0.0” allows the server to listen on every available network interface compared to “127.0.0.1” or “localhost” which is a local-only interface. The port in this case was chosen to be 50000, but users may select any port that is not already in use. The main loop binds the socket and issues a blocking command to accept an incoming connection. Once the robot connects to the server, the Python program waits until the string “get pose” is issued at which point it will send back the pose defined at the top of the program. Once the socket is closed by the client (robot), the server automatically closes the socket on its side and the program terminates.

Polyscope View:

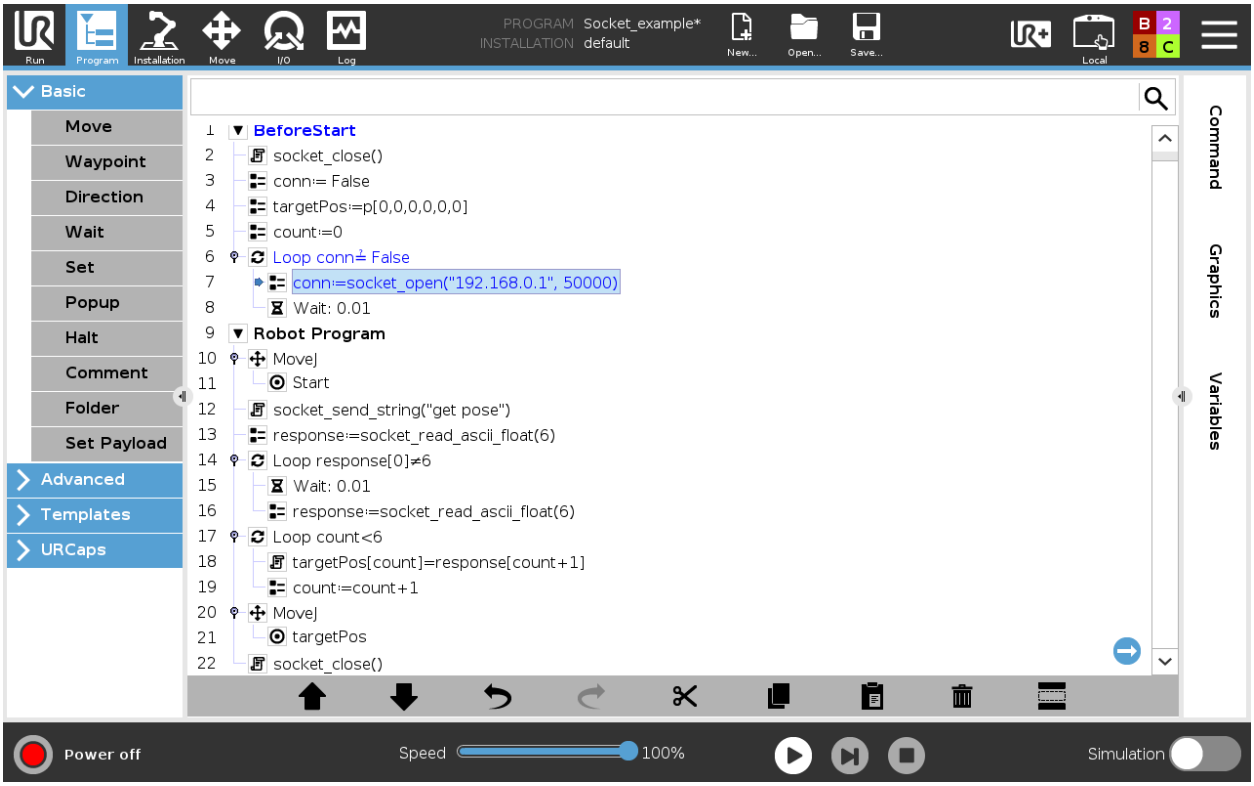


Figure 19: Polyscope code for the Socket TCP/IP example.

The “Before Start” sequence contains several steps to successfully receive data from the PC. While not completely necessary, the first line closes the socket named inside the parentheses (by default called “socket_0”) in case it was not successfully closed on a previous run. Several variables are then created

called “conn”, “targetPos”, and “count”. Next, the robot attempts to open a socket at the specified IP address and port of the server. The command “socket_open” will return True once a connection has been successfully made, so the loop does not exit until this completes. Note that opening and closing sockets does require resource allocation so users should keep sockets open until all the data required has been exchanged.

The main program moves to the starting position and then sends the string “get pose” over to the PC, triggering a response that it then collects. Note that the command “socket_read_ascii_float(*n*)” asks for an array in the form “(x,y,z,…n)” and then returns an array “[n,x,y,z,…]” where the 0th element is always the number of floats received not including itself. In this example, “response” would return “[6, -0.15,-0.5,0.3,0,-3.14,0]” which is then used in the following loop to add the values of the given pose into the variable “targetPos”. Finally, the robot moves to the variable position given by the PC and then closes the socket connection.

5.4 RTDE Connector: An Alternative Approach

The RTDE Connector was developed as a way to speed up and automate the setup procedure of RTDE, allowing users to debug and develop their interfaces more rapidly. The following sections will review the code itself, how to utilize the new input and output functions, and finish off with an example comparing the original method for accessing RTDE to this one.

5.4.1 Connector Overview

Users will remember from Section 4.2 that RTDE is split up into two stages: a setup phase and a main loop. The Connector aims to automate and remove the setup stage almost completely. The Connector has the following inputs which are modifiable by the user:

1. ROBOT_HOST = The IP address of the robot.
2. Config_filename = The XML containing the inputs and outputs the user wishes to use with RTDE (under their respective keys)
3. RTDE_inputs = The list of all RTDE controller inputs taken from the website (manually copied and turned into a CSV).
4. RTDE_outputs = The list of all RTDE controller outputs taken from the website (manually copied and turned into a CSV).

The connector takes parameters for the robot’s IP and the XML file, defaulting to the values specified at the top of the program. It can optionally take a frequency to request outputs from RTDE which by default is set to 500Hz for E-series. On startup it will connect to RTDE and check the controller version. The connector will stop program execution if it detects a non-E-series robot or one whose software version is pre-5.10. From there it will iterate through all the keys in the XML and compare the chosen fields against the master list pulled from the RTDE Support article. If the user has incorrectly specified a field, associated the field with an incorrect type, or has a mix of inputs and outputs within one key, the connector will print an error and exit program execution[2]. Users should have all their outputs inside of one key as receiving a

[2] There is a known interaction where if all the fields of a key can be RTDE inputs OR outputs, the Connector will assume they are inputs and set them up as such (this mainly applies to the input bit, integer, and double registers that can be set up as either). If the user wants to track the status of them as outputs, a workaround would be to

packet of data will provide an update to all outputs, regardless of how they are split up in the XML file. Inputs, on the other hand, should be split such that the user can ALWAYS send updated values for all fields within a key as the program will fail otherwise.

5.4.2 Connector Commands

The following is a list of commands the user can utilize inside of their main RTDE loops.

5.4.2.1 receive()

Receive a packet of data from RTDE at the frequency specified in instantiation of the Connector.

5.4.2.2 send(key, field, value)

Send RTDE inputs to the robot with a given list of fields and values. All fields within a key must be sent with an associated value.

Parameters:

key (str): the key to pull the corresponding inputs from.
field (list): A list of fields (RTDE inputs) to send updated values to.
value (list): A list of updated input values to send to RTDE. Must match the order and type of the field parameter.

5.4.2.3 sendall(key, value)

Send RTDE inputs to the robot with a given list of values. The order of values matches the recipe XML file for a given key.

Parameters:

key (str): the key to pull the corresponding inputs from.
value (list): A list of updated input values to send to RTDE

5.4.3 Connector Example and Comparison

The connector example utilizes the "control_loop_connector.py" on the PC side and the "example_control_loop.urp" on the robot side. The goal of the example is to mimic the original "control_loop" taken from the RTDE guide and highlight the reduction of code necessary to run it. Both programs start off the same way (typing in the robot IP address, the XML file to use, etc.), but then start to differ when it comes to the helper functions:

```
def setp_to_list(output):
    setp = [output.input_double_register_0, output.input_double_register_1,
output.input_double_register_2,
            output.input_double_register_3, output.input_double_register_4,
output.input_double_register_5]
    set_list = [format(elem, '.2f') for elem in setp]

    return [float(x) for x in set_list]
    # Users running 5.11.5 or later can simply return "setp" instead of
set_list.
    # return setp
```

simply add an additional field within that key that can ONLY be an output (like the timestamp, robot_mode, joint_temperatures, etc.).

This function simply converts the setpoints taken from the RTDE output packet into a list that the code can then compare against. Note the comment given: users running 5.11.5 or later do not necessarily need all the formatting done and can instead just return the variable "setp" after it is turned into a list. There is, however, no harm in keeping the function "as-is" and the example will run just the same. Users will also note that there is no need for the other helper function from the original example as the conversion from a list to a "setp" to be fed back to the robot is handled by the Connector module. The next portion is the control loop:

```
# control loop
monitor = RTDEConnect(ROBOT_HOST, config_filename)
setp1 = [-0.12, -0.43, 0.14, 0, 3.11, 0.04]
setp2 = [-0.12, -0.51, 0.21, 0, 3.11, 0.04]
while keep_running:
    # receive the current state
    state = monitor.receive()

    if state is None:
        break

    # do something...
    if state.output_int_register_0 != 0:
        new_setp = setp1 if setp_to_list(state) == setp2 else setp2
        monitor.sendall("setp", new_setp)

    # kick watchdog
    monitor.send("watchdog", "input_int_register_0", 0)

monitor.shutdown()
```

This section also looks similar to the original control loop example. The program creates a variable called "monitor" which will initialize RTDE and set up all the inputs and outputs based on the XML file. The two setpoints for the robot to move between are also included before the main loop. Inside, the user receives a packet of data containing their selected outputs which the program assigns to the variable "state". If no data is received, the program will break out of its main loop. Next the program checks to see if the output integer register is not zero, which is the handshake from the robot program that tells the PC it's ready to receive its next setpoint/waypoint. If it's ready, the robot checks to see which of the two setpoints the robot is currently at and sends the opposite one for it to move to. This is done using the "sendall" command which sends all of the fields within a key (in this case "setp") in the order that they are listed within the XML file. Once that is done, the PC also sends a value to the watchdog, this time using the "send" command from the Connector. Users could alternatively use the "sendall" command which would look like so:

```
# kick watchdog
monitor.sendall("watchdog", 0)
```

Once the main loop breaks the "monitor" variable issues a shutdown command that closes the connection to RTDE and exits program execution. Users will want to have some sort of condition which sets "keep_running" to False when they are ready to exit as currently this program runs until it is manually stopped.

6 Changelog

Version	Date	Notes
1.0	May 27, 2021	-Initial release
1.1	December 17, 2021	-Added RTDE Connector info and examples