Program documentation

# Overview

To control the robot there will be 3 programs running looking for data on the robot. They are robotCTRL.py, robotFDBK.py, and dashboard.py. A Program named main.py will start on bootup to call the other programs as subprocess.

# Main

Purpose of this is to call all other sub functions required to run the robot. Each process is called as a subprocess so they can all run at the same time. If one fails to run it will log the program, time and how many times it has failed in a log.txt file for historical tracking.

In Ubuntu this is called as a startup process. If changes are needed in the startup they are main in main.py instead of in Ubuntu startup process.

Process this program starts.

* Dashboard.py
* robotCTRL.py
* robotFDBK.py

In general commands are sent in the following form. In some cases a command does not have parameters. In this case parameters is not needed.

{

command : “Command Name”,

param : [different param sets. Specific to command]

}

After each command a feedback saying the command name and completed will be sent back to the server.

# robotCTRL

Purpose of this program is to handle the controls on the robot. This not only sends the commands to the arm and hand but all auxiliary equipment on the robot. This will talk with the hand program as well as the Arduino to make.

When sending a command to the robot a JSON with the following format should be sent. Each possible command has outlined what the command and parameters should be there are outlined under the “Call with” heading. 1 command is sent in each json message.

{“msg” : “robotCTRL”,

“data” : {

“command” : NAME of Command,

“param” : Parameters to send

}}

### Auxiliary Commands sent to the Robot

* Fan Power (RLY 1)
  + Used to turn on or off the fans inside the robot box.
  + Parameters
    - On – turns on the fans
    - Off – turns off the fans
    - Auto – turns on and off the fans based on internal temperature.
      * Will also need an array of temps first value will be fan off temp second value will be fan on temp.
      * First value must be less than 2nd value. If first value is greater than 2nd fan will turn on.
  + Call with
    - { command : FanPower,
    - param : [ auto/on/off, on temp, off temp]
* PC Reset (RLY 2)
  + Used to power off the pc. Will power back on 10 seconds after command is sent.
  + Should only be used in emergencies.
  + Params
    - Off – Turns off the PC. PC will automatically turn back on
  + Call with
    - {command : PCReset,
    - param : off}
* Network switch power (RLY 3)
  + Used to power on or off the network switch.
  + Parameters
    - On – turn on the network switched used to talk to arm and hand
    - Off – turn off the network switched used to talk to arm and hand
  + Call with
    - {command : SwitchPower,
    - param : on/off}
* Rover Power (RLY 4)
  + Used to turn on and off the power to the rover.
  + Parameters
    - On – Turns on power to the rover
    - Off – turns off power to the rover
  + Call with
    - {command : RoverPower,
    - param : on/off}
* Hand Controller (RLY5)
  + Used to turn on power to hand controller
  + Parameters
    - On - Turn on power
    - Off - turns off power
  + Call with
    - {command : HandPower,
    - param : on/off
* (RLY6)
  + Spare relay not currently used
  + Parameters
    - On – turns on power
    - Off – turns off power
  + Call with
    - {command : RLY6,
    - param : on/off
* Arm – ON (RLY7)
  + Turns on the UR controller. Internally this is a pulse signal will turn on for 10 seconds then turn back off
  + Parameters
    - On – turns on power
    - Off – turns off power
  + Call with
    - {command : ArmOn,
    - param : on/off
* Arm – Off (RLY8)
  + Turns off the UR controller. Internally this is a pulse signal will turn on for 10 seconds then turn back off
  + Parameters
    - On – turns on power
    - Off – turns off power
  + Call with
    - {command : ArmOff,
    - param : on/off
* Light
  + Used to control the robot light. A color value is sent based on status from the base station. In some cases internally the color will be over ridden to show a fault.
  + Parameters
    - Red – used to show a fault or estop condition
      * Internal to the robot this can be overtake to show an internal fault or estop
      * Turns on relay 9
    - Green – everything is ready no faults
      * Turns on relay 10
    - Blue – used to indicate robot is driving
      * Signal of driving will only be sent from the base station
        + Base station should send green when move is complete
      * Internally signal will flash
      * Turns on relay 11
    - Yellow – indicate arm is moving.
      * Signal is handled internally to the robot
      * Turns on relay 12
  + Call with
    - {command : Light,
    - param : red/green/yellow/blue}
* RLY 13
  + Spare relay not currently used
  + Parameters
    - On – turns on power
    - Off – turns off power
  + Call with
    - {command : RLY13,
    - param : on/off
* RLY 14
  + Spare relay not currently used
  + Parameters
    - On – turns on power
    - Off – turns off power
  + Call with
    - {command : RLY14,
    - param : on/off
* RLY 15
  + Spare relay not currently used
  + Parameters
    - On – turns on power
    - Off – turns off power
  + Call with
    - {command : RLY15,
    - param : on/off
* RLY 16
  + Spare relay not currently used
  + Parameters
    - On – turns on power
    - Off – turns off power
  + Call with
    - {command : RLY16,
    - param : on/off

### Commands that can be sent to the arm

* stop J
  + Stop (linear in joint space) - decelerate joint speeds to zero.
  + Parameters
    - a: joint acceleration [rad/s^2] (rate of deceleration of the leading axis).
    - asynchronous: a bool specifying if the stop command should be asynchronous. Stopping a fast move with a stopJ with a low deceleration may block the caller for some seconds. To avoid blocking set asynchronous = true
  + Call with
    - {command : stopJ,
    - params : [ a, asynchronous]}
* moveJ
  + Move to joint position (linear in joint-space)
  + Parameters
    - q: joint positions
    - speed: joint speed of leading axis [rad/s]
    - acceleration: joint acceleration of leading axis [rad/s^2]
    - asynchronous: a bool specifying if the move command should be asynchronous. If asynchronous is true it is possible to stop a move command using either the stopJ or stopL function. Default is false, this means the function will block until the movement has completed.
  + Call with
    - {command : moveJ,
    - params : [q, speed, acceleration, asynchronous]}
* speed J
  + Joint speed - Accelerate linearly in joint space and continue with constant joint speed.
  + Parameters
    - qd: joint speeds [rad/s]
    - acceleration: joint acceleration [rad/s^2] (of leading axis)
    - time: time [s] before the function returns (optional)
  + Call with
    - {command: speedJ,
    - Params : [qd, acceleration, time]}
* servoJ
  + Servo to position (linear in joint-space)
  + Parameters
    - q: joint positions [rad]
    - speed: NOT used in current version
    - acceleration: NOT used in current version
    - time: time where the command is controlling the robot. The function is blocking for time t [S]
    - lookahead\_time: time [S], range [0.03,0.2] smoothens the trajectory with this lookahead time
    - gain: proportional gain for following target position, range [100,2000]
  + call with
    - {command: servoJ,
    - Param: [q, speed, acceleration, time, lookahead\_time, gain]}
* servo Stop
  + Stop servo mode and decelerate the robot.
  + Parameters
    - a: rate of deceleration of the tool [m/s^2]
  + Call with
    - {command: servoStop,
    - Param: a}
* speed Stop
  + Stop speed mode and decelerate the robot.
  + Parameters
    - a: rate of deceleration of the tool [m/s^2] if using speedL, for speedJ its [rad/s^2] and rate of deceleration of leading axis.
  + Call with
    - {command: speedStop,
    - Param: a}
* Force Mode
  + 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
    - 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.
  + Call with
    - {command: forceMode,
    - Param : [task\_frame, selection\_vector, wrench, type, limits]
* Force Mode Stop
  + Resets the robot mode from force mode to normal operation.
  + Call with
    - {command: forceModeStop}
* Jog Start
  + Starts jogging with the given speed vector with respect to the given feature.
  + When jogging has started, it is possible to provide new speed vectors by calling the jogStart() function over and over again. This makes it possible to use a joystick or a 3D Space Navigator to provide new speed vectors if the user moves the joystick or the Space Navigator cap.
  + Parameters
    - speed: Speed vector for translation and rotation. Translation values are given in mm / s and rotation values in rad / s.
    - feature: Configures to move to move with respect to base frame (FEATURE\_BASE), tool frame (FEATURE\_TOOL) or custom frame (FEATURE\_CUSTOM) If the feature is FEATURE\_CUSTOM then the custom\_frame parameter needs to be a valid pose.
    - custom\_frame: The custom\_frame given as pose if the selected feature is FEATURE\_CUSTOM
  + call with
    - {command: jogStart,
    - param : [ speed, feature, frame]
* Jog Stop
  + Stops jogging that has been started start\_jog.
  + Returns
    - True when complete
  + Call with
    - {command: jogStop}
* Zero FT Sensor
  + Zeroes the TCP force/torque measurement from the builtin force/torque sensor by subtracting the current measurement from the subsequent.
  + Return
    - True after complete
  + Call with
    - {command : zeroFTSensor]
* Set Payload
  + Set payload.
  + Parameters
    - mass: Mass in kilograms
    - cog: Center of Gravity, a vector [CoGx, CoGy, CoGz] specifying the displacement (in meters) from the toolmount. If not specified the current CoG will be used.
  + Call with
    - {command: setPayload,
    - param : {mass, [0,0,0]}
* Teach Mode
  + 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.
  + Call with
    - {command : teachMode}
* End Teach Mode
  + Set robot back in normal position control mode after freedrive mode.
  + Call with
    - {command: endTeachMode}
  + Call with
    - {command: endTeachMode}
* Force Mode Set Damping
  + Sets the damping parameter in force mode.
  + A value of 1 is full damping, so the robot will decellerate 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. Call this function before force mode is entered (otherwise default value will be used).
  + Parameters
    - damping: Between 0 and 1, default value is 0.005
  + call with
    - {command: forceModesetdamping,
    - param : .5}
* Force Mode Set Gain Scaling
  + Scales the gain in force mode.
  + 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. Call this function before force mode is entered (otherwise default value will be used)
  + Parameters
    - scaling: scaling parameter between 0 and 2, default is 1.
  + Call with
    - {command: forceModeSetGainScaling,
    - param: 1}
* Tool Contact
  + Detects when a contact between the tool and an object happens.
  + Returns
    - 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.
  + Parameters
    - direction: 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.
  + Call with
    - {command: toolContact}
* Trigger Protective Stop
  + Triggers a protective stop on the robot
  + Can be used for testing and debugging
  + Call with
    - {command: triggerProtectiveStop}
* Program running
  + Returns true if a program is running on the controller, otherwise it returns false This is just a shortcut for getRobotStatus() & RobotStatus::ROBOT\_STATUS\_PROGRAM\_RUNNING
  + Call with
    - {command: isProgramRunning}
* Robot Status
  + Returns
    - Robot status Bits 0-3: Is power on | Is program running | Is teach button pressed | Is power button pressed There is a synchronization gap between the three interfaces RTDE Control RTDE Receive and Dashboard Client. RTDE Control and RTDE Receive open its own RTDE connection and so the internal state is not in sync. That means, if RTDE Control reports, that program is running, RTDE Receive may still return that program is not running. The update of the Dashboard Client even needs more time. That means, the dashboard client still returns program not running after some milliseconds have passed after RTDE Control already reports program running.
  + Note
    - If you work with RTDE control and receive interface and you need to read the robot status or program running state, then you should always use the getRobotStatus() function from RTDE Control if you need a status that is in sync with the program uploading or reuploading of this object.
  + Call with
    - {command: getRobotStatus}
* Async Operation in Progress
  + Reads progress information for asynchronous operations that supports progress feedback (such as movePath).
  + Returns
    - <0: Indicates that no async operation is running or that an async operation has finished. The returned values of two consecutive async operations is never equal. Normally the returned values are toggled between -1 and -2. This allows the application to clearly detect the end of an operation even if it is too short to see its start. That means, if the value returned by this function is less than 0 and is different from that last value returned by this function, then a new async operation has finished.
    - 0: Indicates that an async operation has started - progress 0
    - >=: 0 Indicates the progress of an async operation. For example, if an operation has 3 steps, the progress ranges from 0 - 2. The progress value is updated, before a step is executed. When the last step has been executed, the value will change to -1 to indicate the end of the async operation.
  + Call with
    - {command: getAsyncOperationProgress}
* Joint Within Safety Limits
  + Called automatically when running MoveJ
  + Checks if the given joint position is reachable and within the current safety limits of the robot.
  + This check considers joint limits (if the target pose is specified as joint positions), safety planes limits, TCP orientation deviation limits and range of the robot. If a solution is found when applying the inverse kinematics to the given target TCP pose, this pose is considered reachable
  + Return
    - a bool indicating if the joint positions are within the safety limits.
  + Parameters
    - q: joint positions
  + Call with
    - {command: isJointWithinSafetyLimits,
    - Params: [base, shoulder, elbow, wrist 1, wrist 2, wrist 3]}
* 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
    - The joint torque vector in Nm: [Base, Shoulder, Elbow, Wrist1, Wrist2, Wrist3]
  + Call with
    - {command: getJointTorques}
* Steady
  + Checks if robot is fully at rest.
  + True when the robot is fully at rest, and ready to accept higher external forces and torques, such as from industrial screwdrivers.
  + Note: This function will always return false in modes other than the standard position mode, e.g. false in force and teach mode.
  + Return
    - True when the robot is fully at rest. Returns False otherwise.
  + Call with
    - {command: isSteady}
* Move Until contact
  + Move the robot until contact, with specified speed and contact detection direction.
  + The robot will automatically retract to the initial point of contact.
  + Return
    - True once the robot is in contact.
  + Parameters
    - xd: tool speed [m/s] (spatial vector)
    - 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. You can also set direction=get\_target\_tcp\_speed() in which case it will detect contacts in the direction of the TCP movement.
    - acceleration: tool position acceleration [m/s^2]
  + Call with
    - {command: moveUntilContact,
    - Parmas: [2, [1,0,0,0,0,0], 2]}
* UR Robotic Arm
  + Position
* QB Hand

# robotFDBK

Purpose of this program to handle the feedback from the robot. This not only gets feedback from the hand and arm but all auxiliary equipment on the robot. This will take to the Arduino to get feedback from all systems. Messages will be sent back to the server every .1 second and contain all the information below.

### Auxiliary feedback

* Fan Power
  + Indicates the power to the fan is on
* NS Power
  + Indicates power to the network switch is on
* UR connected
  + Indicates UR control is connected over the network. Power must be turned on before this will get a signal
* Hand connected
  + Indicates the Hand is connected over the network. Power must be turned on before this will get a signal
* Rover Power
  + Indicates power is turned on to the rover
* Arduino Connected
  + Indicates the serial connection to the Arduino has been established
* Hand Power
  + Indicates the hand power has been turned on.
* Battery Voltage
  + Sends back the actual voltage currently on the batter.
* Temp Sensor
  + Measurement of temperature outside the box
* Humidity Sensor
  + Measurement of humidity external to the box.
* Front Distance Sensor
  + Measurement of distance to object directly in front of the robot. Current sensor is 10cm to 80cm
* Prox FL
  + Front left prox sensor. Indicates when something is within a set distance
* Prox FR
  + Front right prox sensor. Indicates when something is within a set distance
* Prox RL
  + Rear left prox sensor. Indicates when something is within a set distance
* Prox RR
  + Rear right prox sensor. Indicates when something is within a set distance
* Internal Front Temp
  + Temperature next to the fans at the front of the rover inside the rover
* Battery Temp
  + Temperature sensor right next to the battery
* UR Controller Temp
  + Temerature at outlet of UR controller
* PodCom Temp
  + Temperature on the podcom

### Feedback from the Arm:

* Actual Currents
  + Returns
    - Actual joint currents
  + Call with
    - getActualCurrent()
  + Message Tag
    - ActualCurrent
* Actual Digital Input buts
  + Returns
    - Current state of the digital inputs. 0-7: Standard, 8-15: Configurable, 16-17: Tool
  + Call with
    - getActualDigitalInputbits()
  + Message Tag
    - ActualDigitalInputbits
* Actual Digital Output Bits
  + Returns
    - Current state of the digital outputs. 0-7: Standard, 8-15: Configurable, 16-17: Tool
  + Call with
    - ActualDigitalOutputBits()
  + Message Tag
    - ActualDigitalOutputBits
* Actual Execution Time
  + Returns
    - Controller real-time thread execution time
  + Call with
    - getActualExecutionTime()
  + Message Tag
    - ActualExecutionTime
* Actual Joint Voltage
  + Returns
    - Actual joint voltages
  + Call with
    - getActualJointVoltage()
  + Message Tag
    - ActualJointVoltage
* Actual Main Voltage
  + Returns
    - Safety Control Board: Main voltage
  + Call with
    - getActualMainVoltage()
  + Message Tag
    - ActualMainVoltage
* Actual Momentum
  + Returns
    - Norm of Cartesian linear momentum
  + Call with
    - getActualMomentum()
  + Message Tag
    - ActualMomentum
* Actual Q
  + Returns
    - Actual joint positions
  + Call with
    - getActualQ()
  + Message Tag
    - ActualQ
* Actual Qd
  + Returns
    - Actual joint velocities
  + Call with
    - getActualQd
  + Message Tag
    - ActualQd
* Actual Robot current
  + Returns
    - Safety Control Board: Robot current
  + Call with
    - getActualRobotcurrent()
  + Message Tag
    - ActualRobotCurrent
* Actual Robot Voltage
  + Returns
    - Safety Control Board: Robot voltage (48V)
  + Call With
    - getActualRobotVoltage()
  + Message Tag
    - ActualRobotVoltage
* Actual TCP Force
  + Returns
    - Generalized forces in the TCP
  + Call with
    - getActualTCPForce()
  + Message Tag
    - ActualTCPForce
* Actual TCP Pose
  + Returns
    - Actual Cartesian coordinates of the tool: (x,y,z,rx,ry,rz), where rx, ry and rz is a rotation vector representation of the tool orientation
  + Call with
    - getActualTCPPose()
  + Message Tag
    - ActualTCPPose
* Actual TCP Speed
  + Returns
    - Actual speed of the tool given in Cartesian coordinates
  + Call with
    - getActualTCPSpeed()
  + Message Tag
    - ActualTCPSpeed
* Actual Tool Accelerometer
  + Returns
    - Tool x, y and z accelerometer values
  + Call with
    - getActualToolAccelerometer()
  + Message Tag
    - ActualToolAccelerometer
* Ft Raw Wrench
  + Get the raw force and torque measurement, not compensated for forces and torques caused by the payload.
  + Returns
    - the raw force and torque measurement
  + Call with
    - getFtRawWrench()
  + Message Tag
    - FtRawWrench
* Joint Control Output
  + Returns
    - Joint control currents
  + Call with
    - getJointControlOutput()
  + Message Tag
    - JointControlOutput
* Joint Mode
  + Returns
    - Joint control modes
  + Call with
    - getJointMode()
  + Message Tag
    - JointMode
* Joint Temperatures
  + Returns
    - Temperature of each joint in degrees Celsius
  + Call with
    - getJointTemperatures()
  + Message Tag
    - JointTemperatures
* Payload
  + Get the payload of the robot in [kg].
  + Returns
    - the payload in [kg]
  + Call with
    - getPayload()
  + Message Tag
    - Payload
* Payload Cog
  + Get the payload Center of Gravity (CoGx, CoGy, CoGz)
  + Returns
    - the payload Center of Gravity (CoGx, CoGy, CoGz) in [m]
  + Call with
    - getPayloadCog()
  + Message Tag
    - PayloadCog
* Payload Inertia
  + Get the payload inertia matrix elements (Ixx,Iyy,Izz,Ixy,Ixz,Iyz) expressed in kg\*m^2.
  + Returns
    - the payload inertia matrix elements (Ixx,Iyy,Izz,Ixy,Ixz,Iyz) expressed in kg\*m^2
  + Call with
    - getPayloadInertia()
  + Message Tag
    - PayloadInertia
* Robot Mode
  + Returns
    - -1 = ROBOT\_MODE\_NO\_CONTROLLER
    - 0 = ROBOT\_MODE\_DISCONNECTED
    - 1 = ROBOT\_MODE\_CONFIRM\_SAFETY
    - 2 = ROBOT\_MODE\_BOOTING
    - 3 = ROBOT\_MODE\_POWER\_OFF
    - 4 = ROBOT\_MODE\_POWER\_ON
    - 5 = ROBOT\_MODE\_IDLE
    - 6 = ROBOT\_MODE\_BACKDRIVE
    - 7 = ROBOT\_MODE\_RUNNING
    - 8 = ROBOT\_MODE\_UPDATING\_FIRMWARE
  + Call with
    - getRobotMode()
  + Message Tag
    - RobotMode
* Robot Status
  + Returns
    - Robot status Bits 0-3: Is power on | Is program running | Is teach button pressed | Is power button pressed There is a synchronization gap between the three interfaces RTDE Control RTDE Receive and Dashboard Client. RTDE Control and RTDE Receive open its own RTDE connection and so the internal state is not in sync. That means, if RTDE Control reports, that program is running, RTDE Receive may still return that program is not running. The update of the Dashboard Client even needs more time. That means, the dashboard client still returns program not running after some milliseconds have passed after RTDE Control already reports program running.
  + Call with
    - getRobotStatus()
  + Message Tag
    - RobotStatus
* Runtime State
  + Returns
    - Program Running State
  + Call by
    - getRuntimeState()
  + Message Tag
    - RuntimeState
* Safety Mode
  + Returns
    - Safety Mode
  + Call with
    - getSafetyMode()
  + Message Tag
    - SafetyMode
* Safety Status Bits
  + Returns
    - Safety status bits Bits 0-10: Is normal mode | Is reduced mode | Is protective stopped | Is recovery mode | Is safeguard stopped | Is system emergency stopped | Is robot emergency stopped | Is emergency stopped | Is violation | Is fault | Is stopped due to safety
  + Call with
    - getSafetyStatusBits()
  + Message Tag
    - SafetyStatusBits
* Speed Scaling
  + Returns
    - Speed scaling of the trajectory limiter
  + Call with
    - getSpeedScaling()
  + Message Tag
    - SpeedScaling
* Speed Scaling Combined
  + Get the combined speed scaling The combined speed scaling is the speed scaling resulting from multiplying the speed scaling with the target speed fraction.
  + The combined speed scaling takes the runtime\_state of the controller into account. If eg. a motion is paused on the teach pendant, and later continued, the speed scaling will be ramped up from zero and return to speed\_scaling \* target\_speed\_fraction when the runtime\_state is RUNNING again.
  + This is useful for scaling trajectories with the slider speed scaling currently set on the teach pendant.
  + Returns
    - the actual combined speed scaling
  + Call with
    - getSpeedScalingCombined()
  + Message Tag
    - SpeedScalingCombined
* Standard Analog Input 0
  + Returns
    - Standard analog input 0 [A or V]
  + Call with
    - getStandardAnalogInput0()
  + Message Tag
    - StandardAnalogInput0
* Standard Analog Input 1
  + Returns
    - Standard analog input 1 [A or V]
  + Call with
    - getStandardAnalogInput1()
  + Message Tag
    - StandardAnalogInput1
* Standard Analog Output0
  + Returns
    - Standard analog output 0 [A or V]
  + Call with
    - getStandardAnalogOutput0()
  + Message Tag
    - StandardAnalogOutput0
* Standard Analog Output1
  + Returns
    - Standard analog output 1 [A or V]
  + Call with
    - getStandardAnalogOutput1()
  + Message Tag
    - StandardAnalogOutput1
* Target Current
  + Returns
    - Target joint currents
  + Call with
    - getTargetCurrent()
  + Message Tag
    - TargetCurrent
* Target Moment
  + Returns
    - Target joint moments (torques)
  + Call with
    - getTargetMoment()
  + Message Tag
    - TargetMoment
* Target Q
  + Returns
    - Target joint positions
  + Call with
    - getTargetQ()
  + Message Tag
    - TargetQ
* Target Qd
  + Returns
    - Target joint velocities
  + Call with
    - getTargetQd()
  + Message Tag
    - TargetQd
* Target Qdd
  + Returns
    - Target joint accelerations
  + Call with
    - getTargetQdd()
  + Message Tag
    - TargetQdd
* Target Speed Fraction
  + Returns
    - Target speed fraction
  + Call with
    - getTargetSpeedFraction()
  + Message Tag
    - TargetSpeedFraction
* Target TCP Pose
  + Returns
    - Target Cartesian coordinates of the tool: (x,y,z,rx,ry,rz), where rx, ry and rz is a rotation vector representation of the tool orientation
  + Call with
    - getTargetTCPPose()
  + Message Tag
    - TargetTCPPose
* Target TCP Speed
  + Returns
    - Target speed of the tool given in Cartesian coordinates
  + Call with
    - getTargetTCPSpeed()
  + Message Tag
    - TargetTCPSpeed
* Timestamp
  + Returns
    - Time elapsed since the controller was started [s]
  + Call with
    - getTimestamp()
  + Message Tag
    - Timestamp
* init Period
  + Returns
    - This function is used in combination with waitPeriod() and is used to get the start of a control period / cycle.
  + Call with
    - initPeriod()
  + Message Tag
    - initPeriod
* Connected
  + Returns
    - Connection status for RTDE, useful for checking for lost connection.
  + Call with
    - isConnected()
  + Message Tag
    - Connected
* Emergency Stopped
  + Returns
    - a bool indicating if the robot is in ‘Emergency stop’
  + Call with
    - isEmergencyStopped()
  + Message
    - EmergencyStopped
* Protective Stopped
  + Returns
    - a bool indicating if the robot is in ‘Protective stop’
  + Call with
    - isProtectiveStopped()
  + Message Tag
    - ProtectiveStopped

# Dashboard

Used to collect data from the UR controller. You are also able to start, stop, reset safety, unlock protective, close connection, power on, power off, and release the break

Uses socket to read data from controller. Access controller by IP address on port 29999. Sends the data over websocket to ‘ws://localhost:4007’. Host IP address and server IP address need to be sent for this program to work correctly.

Under normal conditions. Status will be sent to the user interface. On startup of the rover the command “power on” will be sent. After mode changes to idle “brake release” should be sent. After this arm is controllable. When not using the battery to save power a power off command can be sent. This reduces the power consumption from 100W to around 10W. (need to verify usage)

## Data Produced

Produced data is sent once per second. Will send to the server with the message “dashboardData”

Most data is pulled at that rate. A few items are pulled only when a command is sent.

Produced message will be in the following format

data={

"msg": "dashboardData",

"data":

{"Connected": true,

"robotmode": "Robotmode: POWER\_OFF",

"Polyscope Version": "URSoftware 5.12.4.1101661 (Aug 31 2022)",

"SafetyStatus": "Safetystatus: NORMAL",

"In Remote": "true",

"Serial": "20195399999",

"model": "UR3",

"Unlocking Protective": "Unknown",

"Quit": "unknown"

}}

Data pulled once a second is:

* Connected state – Returns true or false depending on if UR controller is connected
  + Possibilities are True or False
* Robot mode – sends the current mode of the robot
  + Possibilities are no\_controller, disconnected, confirm\_safety, booting, Power\_off, Power\_on, Idle, Backdrive, running
* Polyscope Version – Returns the software version installed
* Safety Status – Returns the current status of the safety system
  + Possibilities are – Normal, reduced, protective\_stop, recovery, safetgaurd\_stop, system\_emergency\_stop, Robot\_emergency\_stop, Violation, Fault, Automatic\_mode\_safegaurd\_stop, System\_three\_position\_enablin\_stop
* In Remote – Returns if the remote is in remote control local control mode.
  + For commands to be send robot must be in remote mode. This change must be done from the teach pendant
  + Returns True or False
* Serial – Returns the serial number of the robot
* Model – Returns the model of the robot arm connected
* Unlocking Protective – Returns the state of unlocking the protective stop (updated on command)
  + Will be unknown until the unlock protective command is sent to the controller.
  + On success message will be “Protective stop releasing”
  + On Failure message will be “Cannot unlock protective stop until 5s after occurrence. Always inspect cause of protective stop before unlocking”
* Quit – Used to disconnect the robot will show disconnected if robot is disconnected (updated on command)

## Data Consumed

Is pull as needed. Looks for a message called “dashboardCMD”.

Accepts the following messages:

* “shutdown”
  + Used to turn off the power to the controller.
  + Note: this may fault the program reading from the controller. Only do this under controlled circumstances
* “power off”
  + Turns the controller to the off state. Polyscope is still running but no power is applied to the arm
  + Can be run at any time
* “power on”
  + Turns the controller to the idle state. Arm will not have power after powering on
  + Can be run when mode is power off
* “brake release”
  + Can be run from power off or idle state will apply power to robot arm.
  + State will be running after running this command
* “unlock protective stop”
  + If a protective stop triggered this will reset the fault.
  + Note: must wait at least 5 seconds after failure to start robot again
  + Must power on and release brake after running
* “restart safety”
  + If safety is triggered you can run this to reset the safety
  + You will need to run power on and brake release after running this command
* “quit”
  + Used to close the connection to the rover.
  + Note: the way the program is written after running this the program will stop and start again resetting the connection.

# Connecting to Casual

Because Casual is a web based application there are two ways to connect. If you are running casual from the same PC as the Node server you can use ws://localhost:4007 to connect to the nodejs server. If you are trying to connect to a remote server currently you have to use a tunnel. Using localtunnel.me you will generate a randome HTTPS:// address. You will change the HTTPS to wss with the randomly generated URL to access the data.

The simplest form of connected and reading the data from the socket in causal is the following.

//let socket = new WebSocket("ws://localhost:4007");

let socket = new WebSocket("wss://sad-mice-read-162-191-47-209.loca.lt");

socket.onopen = function(e) {

  os.toast("[open] Connection established");

  os.toast("Sending to server");

  socket.send('{"msg": "dashboardCMD", "data": "shutdoswn"}');

  socket.onmessage

};

socket.onmessage = function(event) {

  console.log(`[message] Data received from server: ${event.data}`)

  return event.data

}