armstron

Clark B. Teeple, Harvard Microrobotics Lab

May 18, 2023

DOCUMENTATION

The Armstron package allows you to This ROS package and associated GUI make use of the 6-axis force/torque sensor on the Universal Robot e-series to apply complex loads to things. We can essentially perform tests similar to an Instron Uniaxial Testing machine, but in all axes.

Table of Contents

CHAPTER ONE

QUICKSTART GUIDE

(from “ README.md *” in github repo)*

1.1 Armstron

This ROS package and associated GUI make use of the 6-axis force/torque sensor on the Universal Robot e-series to apply complex loads to things. We can essentially perform tests similar to an Instron Uniaxial Testing machine, but in all axes.

1.1.1 Installation

1. Set up your robot arm for use with the Universal_Robots_ROS_Driver.
   - Clone the package to the **src** folder of your catkin workspace
   - Follow instructions in the ur_robot_driver/doc folder.
   - Be sure to create a calibration package for your robot per these instructions. In this example, we have created a package called “_ur_user*calibration*” with a dedicated launch file “_bringup*armando.launch*” where the IP address of the robot is set.
2. Set up this package
   1. Clone the simple_ur_move package to the **src** folder of your catkin workspace
   2. Clone this package (armstron) to the **src** folder of your catkin workspace
   3. In the root folder of your workspace, install dependencies:
      - `rosdep install --from-paths src --ignore-src -r -y`
   4. Navigate to the armstron package folder and install a few extra non-ROS python requirements:
      - `cd src/armstron/armstron`
      - `pip install -r requirements.txt`
   5. Navigate back to the workspace folder, and build your workspace (`catkin_make`)
3. Set up the runfile (so you can run Armstron as though it is a regular application)
   1. Navigate to the armstron top level folder:
      - `cd src/armstron`
   2. Generate the run files

- `sudo bash armstron/bash/generate_desktop.sh [ROBOT PACKAGE] [ROBOT LAUNCH FILE]`
- For example: `sudo bash armstron/bash/generate_desktop.sh ur_user_calibration bringup_armando.launch`

3. Now you can run Armstron with one click by going to your application menu (Super + A) and choosing "Armstron".

1.1.2 Usage

Bringup the robot

1. *(Teach Pendant)* Turn on the robot, and get into *manual* mode.
2. *(Teach Pendant)* Start the robot (tap the small red dot on the bottom left corner)

Start Armstron

1. *(Host Computer)* Choose "Armstron" from your application menu (Super + A).
   - This starts communication with the robot arm, starts the Armstron test server, and starts the Armstron GUI.
2. Use the GUI to load/build test profiles, set data save locations.
3. Run tests
   1. *(Teach Pendant)* Move the arm around manually to set things up.
   2. *(Teach Pendant)* Once you are ready to test, run the "EXTERNAL_CONTROL.urp" program. (press "play" in the bottom bar)
   3. *(Host Computer)* Run a test using the "Run Test" button.

1.1.3 Advanced Usage

Since everything is modular, you can run each part of the Armstron software stack as independent ROS processes. This is useful for debugging purposes

Bringup the robot

1. *(Teach Pendant)* Turn on the robot, get into *manual* mode, then load the "EXTERNAL_CONTROL.urp" program.
2. *(Teach Pendant)* Start the robot (tap the small red dot on the bottom left corner)
3. *(Host Computer)* (new terminal): `roslaunch ur_user_calibration bringup_armando.launch`

Use the Armstron test server

1. *(Host Computer)* Start the test server (new terminal): `roslaunch armstron bringup_testing.launch`
2. Setup the robot

1. *(Teach Pendant)* Move the arm around manually to set things up.
2. *(Teach Pendant)* Once you are ready to test, run the "EXTERNAL_CONTROL.urp" program. (press "play" in the bottom bar)

3. *(Host Computer)* Start a test (new terminal):

```
roslaunch armstron run_test.launch config:="ceti_pull_test.yaml" save:="~/armstron_
↪data/testing_launch.csv"
roslaunch armstron run_test.launch config:="ceti_force_hold.yaml" save:="~/armstron_
↪data/testing_launch.csv"
```

Start the GUI

1. *(Host Computer)* Start the test server (new terminal): `roslaunch armstron bringup_testing.launch`
2. *(Host Computer)* Start the Armstron GUI (new terminal): `rosrun armstron gui.py`

Useful commands for debugging

- Show the controller manager: `rosrun rqt_controller_manager rqt_controller_manager`
- Enable sending of single messages
  - `rqt`
  - Go to *Plugins* >> *Topics* >> *Message Publisher*

CHAPTER TWO

EXAMPLES

Some basic examples of how armstron can be used will be include here. You can find them in the armstron/launch folder in the github repo.

2.1 Run a Test

Lets walk through the easy way to build and run tests.

Contents:

2.1.1 Startup

1. **Bringup the robot**

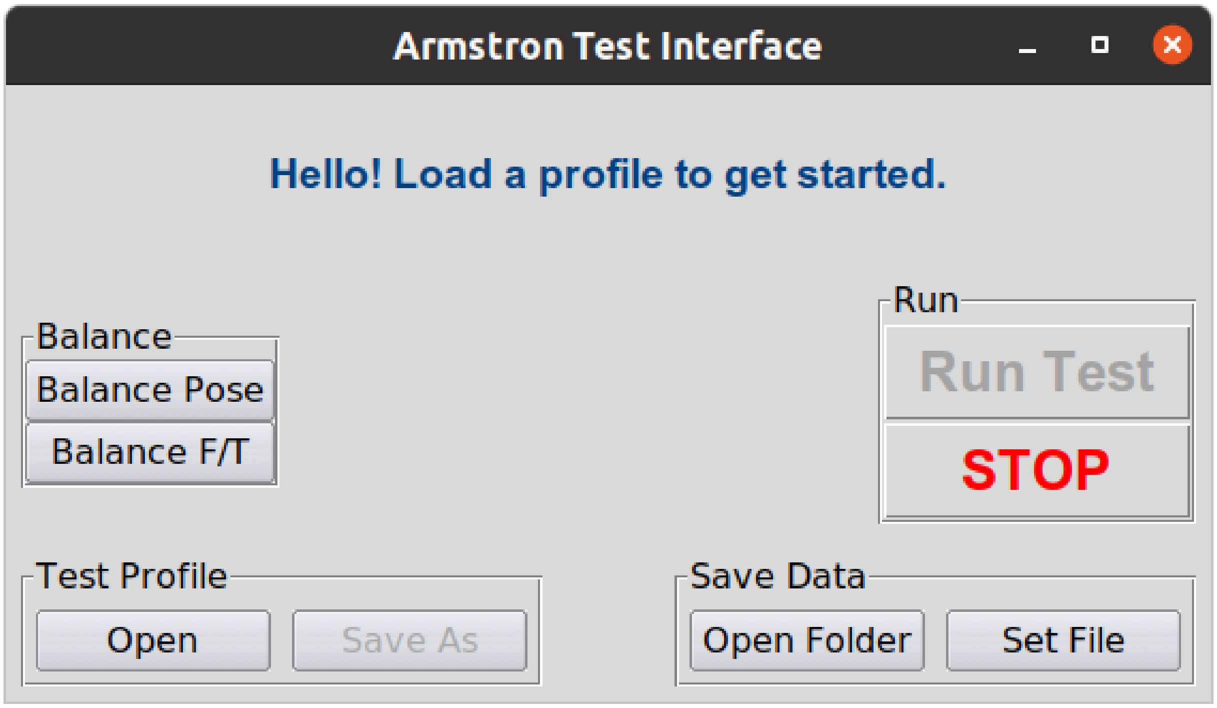
    a. *(Teach Pendant)* Turn on the robot, get into _manual_ mode, then load the "EXTERNAL_CONTROL.urp" program.

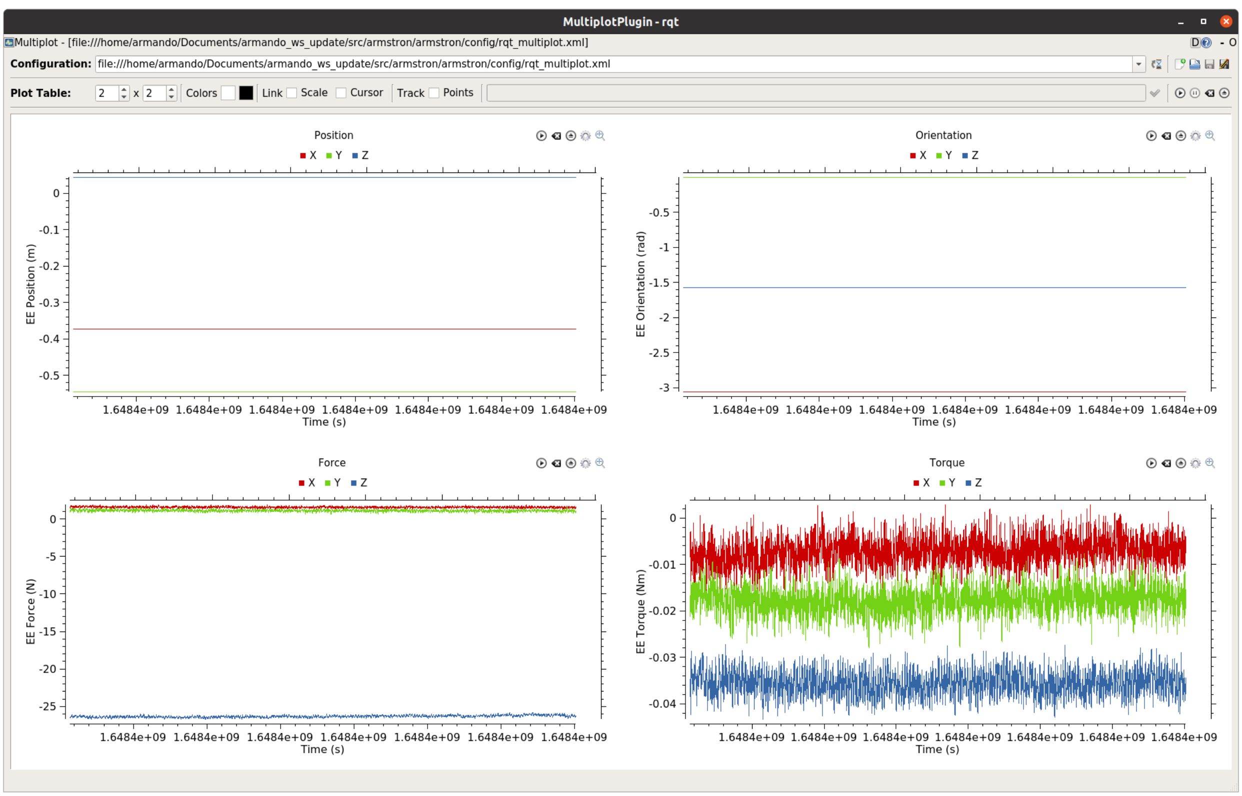
    b. *(Teach Pendant)* Start the robot (tap the small red dot on the bottom left corner)

2. **Start Armstron**

    a. *(Host Computer)* Choose "Armstron" from your application menu (Super + A).

Now the Armstron GUI and a live graph will open:

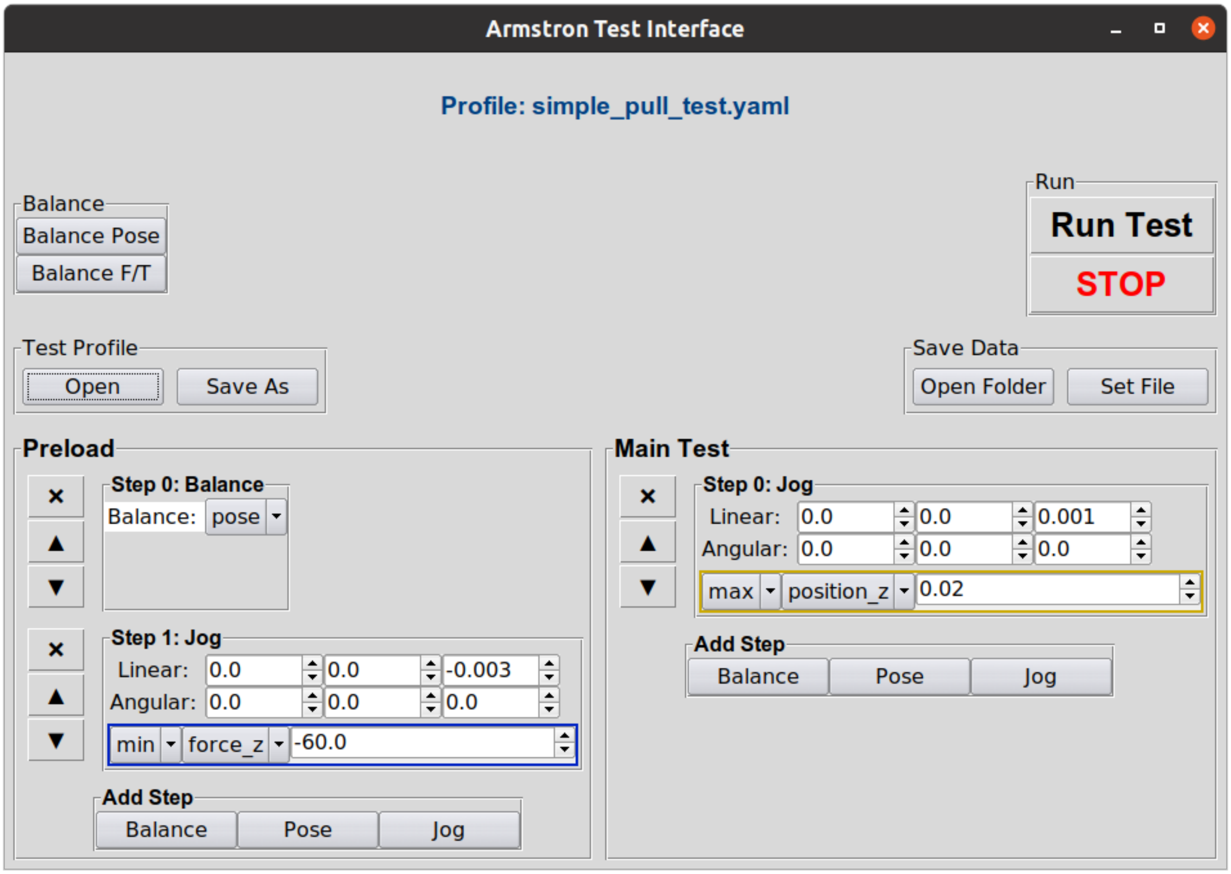




2.1.2 Build a Test Profile

1. Click the “Load” button and choose an existing profile

2. Use the and buttons to move a step up or down
3. Use the button to remove a step
4. Add a step with the "add step" buttons.



2.1.3 Run a Test

1. *(Teach Pendant)* Move the arm around manually to set things up.
2. *(Teach Pendant)* Once you are ready to test, run the "EXTERNAL_CONTROL.urp" program. (press "play" in the bottom bar)
3. *(Host Computer)* Choose where to save data with the "Set File" button.
4. *(Host Computer)* Run a test using the "Run Test" button.
5. *(Host Computer)* Abort a test using the "STOP" button. *Note: Data is saved continuously during tests, so your data is safe even if a test gets aborted.*

Note: Savefile names are auto-incremented to prevent overwriting of data, so you only need to set the filename once.

Important: Don't forget to run the "EXTERNAL_CONTROL.urp" program before running tests! It's easy to forget, so let this note serve as a reminder.

2.1.4 Happy Testing!

This is all you really need to know to use the Armstron. If you are interested in doing more-advanced things, keep reading in *Manual Testing* section

2.2 Run a Test Manually

Lets walk through the manual way to build and run a test.

Contents:

2.2.1 Build a Test Profile

Test profiles are defined in YAML files inside the armstron/config/test_profiles folder.

Profile Structure

Test profiles must be structured in a specific way:

1. **type** (`str`): The type of profile. For now, the only valid value is "*sequence*"
2. **params** (`dict`): A set of parameters
   a. **preload** (`list`): A list of actions to perform during the "preload" phase
   b. **test** (`list`): A list of actions to perform during the "test" phase

Testing Actions

Actions can be either "jog", "pose" or "balance".

Jog steps have motion parameters (how the arm moves) and stop conditions (when the arm should stop moving):

```
jog: # jog motions about the end effector (TCP of the robot)
    linear: [X, Y, Z]           # [mm/sec]
    angular: [X, Y, Z]          # [rad/sec]
stop_conditions:
    max_time: [TIME]            # [sec]
    max_force_x: [FORCE]        # [N], options: min/max, and x/y/z
    max_torque_x: [TORQUE]      # [Nm], options: min/max, and x/y/z
    max_position_x: [POSITION] # [m], options: min/max, and x/y/z
    max_orientation_x: [ORI]    # [rad], options: min/max, and x/y/z
```

Pose steps move the arm to a specific pose over a given time:

```
pose: # pose of the end effector (TCP of the robot)
    position: [X, Y, Z]          # [m]
    orientation: [X, Y, Z]         # [degrees (euler angles)]
stop_conditions:
    max_time: [TIME]             # [sec]
    max_force_x: [FORCE]         # [N], options: min/max, and x/y/z
    max_torque_x: [TORQUE]       # [Nm], options: min/max, and x/y/z
    max_position_x: [POSITION] # [m], options: min/max, and x/y/z
    max_orientation_x: [ORI]    # [rad], options: min/max, and x/y/z
```

Balancing steps can balance (zero) either the pose or the F/T sensor readings:

```
balance: [TYPE]   # options: 'pose' and 'ft'
```

Note: Omit (or comment out) `stop_conditions` if you do not want to use them.

Examples

Here is an example of a simple testing profile:

ceti_pull_test.yaml

```
type: 'sequence'
params:
  preload:
    # Step 0: Balance the pose
    - balance: 'pose'

    # Step 1: Preload
    - jog:
        linear: [0, 0, -0.003] # [mm/sec]
        angular: [0.0, 0, 0]     # [rad/sec]
      stop_conditions:
        min_force_z: -60 # [N]

  test:
    # Step 1: Preload
    - jog:
        linear: [0.00, 0.0, 0.001] # [mm/sec]
        angular: [0.0, 0.0, 0.0]    # [rad/sec]
      stop_conditions:
        max_position_z: 0.02 # [m]
```

Here is an example of a more complex, multi-step testing profile:

ceti_force_hold.yaml

```
type: 'sequence'
params:
  preload:
    # Step 0: Balance the pose
    - balance: 'pose'

    # Step 1: Move slowly down until 60 N are applied
    - jog:
        linear: [0, 0, -0.001] # [mm/sec]
        angular: [0.0, 0, 0]     # [rad/sec]
```

(continues on next page)

(continued from previous page)

```
      stop_conditions:
          min_force_z: -60 # [N]

    # Step 2: Hold for 5 seconds
    - jog:
        linear: [0.00, 0.0, 0.000] # [mm/sec]
        angular: [0.0, 0.0, 0.0]   # [rad/sec]
      stop_conditions:
        max_time: 5 #[sec]

  test:
    # Step 1: Move slowly in Z until 10 N are applied or we've moved 50mm
    - jog:
        linear: [0.00, 0.0, 0.0005] # [mm/sec]
        angular: [0.0, 0.0, 0.0]   # [rad/sec]
      stop_conditions:
        max_force_x: 20 # [N]
        max_force_y: 20 # [N]
        max_force_z: 10 # [N]
        #max_position_z: 0.020 # [m]

    # Step 2: Hold for 10 seconds
    - jog:
        linear: [0.00, 0.0, 0.000] # [mm/sec]
        angular: [0.0, 0.0, 0.0]   # [rad/sec]
      stop_conditions:
        max_time: 10 #[sec]

    # Step 3: Pull suction cup off
    - jog:
        linear: [0.00, 0.0, 0.0005] # [mm/sec]
        angular: [0.0, 0.0, 0.0]   # [rad/sec]
      stop_conditions:
        max_position_z: 0.05 #[m]
```

2.2.2 Run a Test

Running a test is just a matter of running one terminal command (after you start the rest of the system up) with the profile you want to use, and the filename to save data.

Testing Procedure

1. Bringup the robot

   a. *(Teach Pendant)* Turn on the robot, get into _manual_ mode, then load the "EXTERNAL_CONTROL.urp" program.

   b. *(Host Computer)* In a new terminal: `roslaunch ur_user_calibration bringup_armando.launch`

2. Start the Armstron test server (this waits for tests to be started, and handles balancing and estop commands)

   a. In a new terminal, start the test server: `roslaunch armstron bringup_testing.launch`

3. Start a test (for example, *ceti_pull_test.yaml*), and save data in the Documents folder (*~/Documents/armstron_data/test.csv*)

a. *(Teach Pendant)* Move the arm around manually to set things up.

b. *(Teach Pendant)* Once you are ready to test, run the "EXTERNAL_CONTROL.urp" program. (press "play" in the bottom bar)

c. *(Host Computer)* In a new terminal, run:

```
roslaunch armstron run_test.launch config:="ceti_pull_test.yaml" save:="~/Documents/
↪armstron_data/test.csv"
```

Note: Savefile names are auto-incremented to prevent overwriting of data, so you can keep sending the same filename (and thus the same terminal command) over and over to keep repeating the same test procedure

Important: **Don't forget to run the "EXTERNAL_CONTROL.urp" program** before running tests! It's easy to forget, so let this note serve as a reminder.

More Details

When running the test, the launch file you are calling takes care of routing parameters to the correct script:

run_test.launch

```
<?xml version="1.0"?>
<launch>
        <!-- Get arguements  -->
        <arg name="config" doc="Filename of the test config" />
        <arg name="save" doc="Filename of data to save" />
        <arg name="debug" default="False" doc="Whether debug is on" />
        <arg name="action_name" default="armstron" doc="Name of the action server" />


        <!-- start the run node and pass it all of the parameters -->
    <node name="armstron_runner_node" pkg="armstron" type="run_single_test.py"␣
↪respawn="false"
            output="screen">
                <param name="debug" type="bool" value="$(arg debug)"/>
                <param name="action_name" type="str" value="$(arg action_name)" />
                <param name="config_file" type="str" value="$(arg config)" />
                <param name="save_file" type="str" value="$(arg save)" />

    </node>
</launch>
```

This launch file invokes a ros node that creates a *TestRunner* object:

run_single_test.py

```
#!/usr/bin/env python
import rospy
import rospkg
import os

from armstron.test_interface import TestRunner
```

(continues on next page)

(continued from previous page)

```
filepath_config = os.path.join(rospkg.RosPack().get_path('armstron'), 'config')

if __name__ == '__main__':
    try:
        rospy.init_node('v_inst_test_runner', disable_signals=True)
        print("V_INST TEST RUNNER: Node Initiatilized (%s)"%(rospy.get_name()))

        debug = rospy.get_param(rospy.get_name()+"/DEBUG",False)
        action_name = rospy.get_param(rospy.get_name()+"/action_name",None)
        config_file = rospy.get_param(rospy.get_name()+"/config_file",None)
        save_file = rospy.get_param(rospy.get_name()+"/save_file",None)

        config_path = os.path.join(filepath_config,'test_profiles',config_file)

        # Set settings
        sender = TestRunner(action_name, debug=debug)
        sender.load_profile(config_path)
        sender.set_savefile(save_file)

        print("V_INST TEST RUNNER: Running Test")
        sender.run_test(wait_for_finish=True)
        print("V_INST TEST RUNNER: Finished!")
        sender.shutdown()

    except KeyboardInterrupt:
        print("V_INST TEST RUNNER: Shutting Down")
        sender.estop()
        sender.shutdown()

    except rospy.ROSInterruptException:
        print("V_INST TEST RUNNER: Shutting Down")
        sender.estop()
        sender.shutdown()
```

CHAPTER

THREE

API REFERENCE

Each page contains details and full API reference for all the classes that can be accessed outside the armstron package. These are located in the `src/armstron` folder.

For an explanation of how to use all of it together, see *Quickstart Guide*.

3.1 Test Interface

Run and manage tests. This class can be imported into other ROS packages to enable control of tests. This interface is used in the backend of the GUI.

class test_interface.**TestRunner**(*name*, *debug=False*)

A ROS Action server to run single tests.

Parameters **name** (*str*) – Name of the Action Server

balance(*type*)

Balance either the load or F/T sensor

Parameters **type** (*str*) – The type of balance to perform. Must be either `pose` or `ft`

estop()

Perform an Emergency Stop

get_test_status()

Get the current state of the test

Returns **state** – The current state of the test

Return type ActionFeedback

load_profile(*filename*)

Load the config profile from a file

Parameters **filename** (*str*) – Filename to load

run_test(*wait_for_finish=True*)

Run a test and wait for the result

Parameters **wait_for_finish** (*bool*) – Wait for the test to finish

Returns **result** – The result of the test.

Return type ActionResult

set_profile(*profile*)

Set the config profile

Parameters **profile** (*dict*) – Testing profile to use

set_savefile(*save_file*)
Set the filename to save

Parameters **save_file** (*str*) – Filename to save data to

shutdown()
Shut down the node gracefully

3.2 Hardware Interface

3.2.1 Robot Controller

Control the robot's motion and get data. This interface relies on the Universal_Robots_ROS_Driver package to control UR robots using the builtin ROS controllers.

class hardware_interface.**RobotController**(*robot_name=''*, *debug=False*)
Hardware interface that can control a robot via ROS controllers

Parameters

- **robot_name** (*str*) – Name of the robot. This name must match the prefix of the robot's controller topics
- **debug** (*bool*) – Turn on debugging print statements

balance_ft()
Zero the internal F/T offsets

balance_pose()
Zero the internal pose offsets

get_offsets()
Get the internal F/T and Pose offsets

Returns **offsets** – Offset dict with "force", "torque", "position", and "orientation" keys

Return type dict

get_twist(*linear*, *angular*)
Build a twist message from vectors

Parameters

- **linear** (*list*) – The linear twist components [x,y,z]
- **angular** (*list*) – The angular twist components [x,y,z]

Returns **twist** – The resulting twist message

Return type geometry_msgs/Twist

load_controller(*controller*)
Load a ROS controller

Parameters **controller** (*str*) – Name of the controller to load

Returns **response** – Service response from the controller manager

Return type str

play_program()
Start the program on the teach pendant. This ony works if you are in remote control mode

Returns **result** – The result of the service call

Return type srv

set_controller(*controller*)
Set which ROS controller is started, and stop all others

Parameters **controller** (*str*) – Name of the controller to start

Returns **response** – Service response from the controller manager

Return type str

set_jog(*linear*, *angular*)
Set the Jog speed

Parameters

- **linear** (*list*) – The linear twist components [x,y,z]
- **angular** (*list*) – The angular twist components [x,y,z]

set_offsets(*offsets*)
Set the internal F/T and Pose offsets

Parameters **offsets** (*dict*) – Offset dict with "force", "torque", "position", and "orientation" keys

set_pose(*pose*, *time=5.0*)
Set the pose of the robot

Parameters

- **pose** (*dict*) – The pose dictionary with position and orientation components
- **time** (*float*) – Time to take (in seconds)

set_speed_slider(*fraction*)
Set the speed slider fraction

Parameters **fraction** (*float*) – Slider fraction to set (0.02 to 1.00)

Returns **result** – The result of the service call

Return type srv

shutdown()
Shutdown gracefully

stop_program()
Stop the program on the teach pendant. This ony works if you are in remote control mode

Returns **result** – The result of the service call

Return type srv

switch_controller(*start_controllers*, *stop_controllers*, *strictness=1*, *start_asap=False*, *timeout=0*)
Switch ROS controllers

Parameters

- **start_controllers** (*list*) – Names of the controllers to start
- **stop_controllers** (*list*) – Names of the controllers to stop
- **strictness** (*int*) – Strictness of controller switching
- **start_asap** (*bool*) – Decide whether controllers should be started immediately

- **timeout** (*int*) – Timeout (in seconds)

Returns **response** – Service response from the controller manager

Return type str

unload_controller(*controller*)
Unload a ROS controller

Parameters **controller** (*str*) – Name of the controller to unload

Returns **response** – Service response from the controller manager

Return type str

update_tool_pose(*data*)
Update the internal value of the tool pose. Saves a copy of the cartesian position (in m) and euler angle orientation (in rad). The pose is then balanced via offsets. The balanced wrench is published in the `/tf_balanced` topic.

Parameters **data** (*tf2_msgs/TFMessage*) – Wrench message

update_wrench(*data*)
Update the internal value of the wrench. The wrench is converted from the tool frame to the world frame, then balanced via offsets. The balanced wrench is published in the `/wrench_balanced` topic.

Parameters **data** (*geometry_msgs/Wrench*) – Wrench message

3.2.2 Data Logger

Log data to syncronized files. Given a filename and a map of topics (see armstron/config/data_to_save.yaml), log data from ROS topics to CSV files.

class hardware_interface.**DataLogger**(*filename*, *config*, *overwrite=False*)
Log data to csv files.

Parameters

- **filename** (*str*) – Filename to save to. *Note: files will be saved with suffixes corresponding to the name of the data being saved*
- **config** (*dict*) – Configuration, including the topic map

Raises **ValueError** – If the topic_map is invalid

pause()
Pause logging.

resume()
Resume logging.

shutdown()
Shutdown gracefully.

start()
Start logging data.

stop()
Stop logging data.

CHAPTER

FOUR

CONTRIBUTING

Contributing Checklist

- only through a new branch and reviewed PR (no pushes to master!)
- always bump the version of your branch by increasing the version number listed in _version.txt

CHAPTER FIVE

INDEX

CHAPTER

SIX

INSTALL

Follow instructions in the *Quickstart Guide*.

CHAPTER SEVEN

EXPLORE THE EXAMPLES

Check out the *Examples*, or run any of the launch files in the `armstron/launch` folder.

CHAPTER

EIGHT

LINKS

- **Documentation:** Read the Docs
- **Source code:** Github

CHAPTER NINE

CONTACT

If you have questions, or if you've done something interesting with this package, get in touch with Clark Teeple, or the Harvard Microrobotics Lab!

If you find a problem or want something added to the library, open an issue on Github.

CHAPTER

TEN

USED IN...

Armstron will enable many future works!

10.1 References

PYTHON MODULE INDEX

INDEX