

# DRC Valve Turning Task Mission Manual

## version 0.1.0



Dr. Jim Mainprice, [jmainprice@wpi.edu](mailto:jmainprice@wpi.edu)  
Halit Bener Suay, [benersuay@wpi.edu](mailto:benersuay@wpi.edu)  
Calder Phillips-Grafflin, [cnphillipsgraffl@wpi.edu](mailto:cnphillipsgraffl@wpi.edu)  
Nicholas Alunni, [nalunni@wpi.edu](mailto:nalunni@wpi.edu)

Assist. Prof. Dmitry Berenson  
Assist. Prof. Sonia Chernova  
Assoc. Prof. Robert W. Lindeman

# I. Introduction

Our contributions to the DRC team fall into two main categories: the software we have developed directly for our task, and the infrastructure we have developed to support both our task and the team as a whole. Following this division, our mission manual has two major sections; first, an introduction to our task, the user interface, and operation, and second, an overview of the various infrastructure components we have developed. The objective of this manual is to provide both a user's guide to our specific task and a high-level user's guide to our software infrastructure - given the scale of the latter, the discussion is largely at the level of ROS packages with limited discussion of individual nodes when necessary.

For end users of our interface, this manual will serve as the canonical reference on the operation, performance, and limitations of our software. In keeping with this, it will be updated regularly, with the version number of this document matching the tags in the relevant software repositories.

For users developing software based on our infrastructure, this manual should serve as an introduction to what we have developed and as an overview to our development roadmap. Deliberately, however, this document does not provide extensive documentation of specific APIs or code, as doing so is outside the scope of the manual. For specific information or help using our software, the most efficient and effective method is to contact the maintainer or author(s) listed in the package.

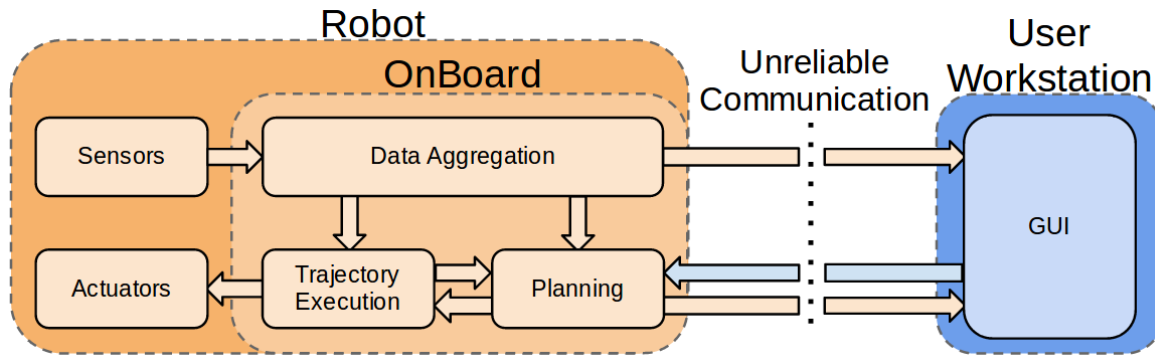
Relevant repositories:

[https://github.com/WPI-ARC/hubo\\_ros\\_core](https://github.com/WPI-ARC/hubo_ros_core)  
[https://github.com/WPI-ARC/hubo\\_ros\\_control](https://github.com/WPI-ARC/hubo_ros_control)  
[https://github.com/WPI-ARC/teleop\\_toolkit](https://github.com/WPI-ARC/teleop_toolkit) (Private)  
[https://github.com/WPI-ARC/localization\\_tools](https://github.com/WPI-ARC/localization_tools)  
[https://github.com/WPI-ARC/drc\\_hubo](https://github.com/WPI-ARC/drc_hubo) (Private)  
[https://github.com/calderpg/dynamixel\\_motor](https://github.com/calderpg/dynamixel_motor)

## II. Operation Manual

### Framework Overview

The system we have developed for valve turning consists of three main parts: (1) a user-guided perception interface which provides task-level commands to the robot, (2) a planning algorithm that autonomously generates robot motions while obeying balance, closed-kinematic and collision constraints, and (3) a trajectory execution and monitoring system. Our goal is that all three of these parts be usable on different robots in both the physical world and simulated environments.



In the first component (GUI), the operator aligns a model of the valve to a point cloud provided by the robot's sensors. Given a good guess for the valve location we make use of an autonomous perception algorithm namely the Iterative Closest Point (ICP) algorithm to reduce the error and “snap” the rough user-generated alignment into place. Once satisfied with the alignment, the operator commands the robot to perform the task.

The manipulation planning component of the system consists of the CBiRRT algorithm [Berenson11], which is capable of generating constrained quasi-static motion for high-DOF robots. Once a motion path is constructed by the planning component, it is executed by the execution monitoring component. We rely on lower level components for control, namely hubo-ach (Drexel) and hubo-motion-rt (GT) which operate with real time constraints.

To support operations in environments with limited bandwidth and unreliable networks as specified by DRC ruling, we have developed a set of tools specifically for communications in networks with low bandwidth, dropouts, and high latency. We also intend to have the operator control the data rates that are sent over the communication link through an interface in RViz (together with related interface components from U. Delaware).

## The Chain of Command

In the actual state of the framework, due to the complexity of operating the valve turning task, it necessitates ideally a 4 person team:

- Primary operator: who makes decisions and does the operation
- Mission Specialist 1: communication and sensors
- Mission Specialist 2: robot hardware and execution
- Mission Specialist 3: --
- Mission Specialist 4: --

The primary operator interacts directly with the main UI and provided the task is executed as planned, their task is simply to localizing the valve and send high level commands to the robot to plan and execute. In the interests of efficiency, they should have experience using the interface to “snap” the valve and should have simulation and physical world experience with the system so that they can avoid limitations of the robot and planning software.

However, in unexpected cases, task-specific expertise is necessary to quickly respond to errors in the blocks that make up our task: motion planning, control, sensing and communication. Although the primary operator can be an expert on one of these domains, given the complexity of the components used in our task, it is both unlikely and unadvisable for a single operator to be an expert on all of these domains. As such, we believe having multiple backup operators with specific experience in these domains reduces risk and provides for more reliable operation.

One important aspect of our design is the ability for the operator to pre-visualize the robot motion before sending it for execution. However we are still debating whether to perform motion planning on the workstation. This will depend on point cloud data will be treated by the motion planner. Currently motion planning is performed on the workstation, in which case the robot operator should be familiar with the motion planner as they can tune parameters online to select the most stable and efficient valve turning motions if necessary

In spite of relying on one primary operator, for the backups to intervene quickly it is ideal to provide several interconnected workstations in order to identify data rates in the system, restart software with proper parameters or even fix bugs online. As GT has already discussed, it makes sense for one of the backup operators to be from their team, as they are most familiar with the low-level behavior of the robot and are best prepared to address hardware problems as they appear.

Finally, the complete framework will require to tele-operate the robot walk from the starting point to the valve using software yet operational. We thus lack information to estimate our needs concerning that aspect, but a specific operator may be needed at that point.

# System Startup

Before executing commands described in this section, turn on the robot's on-board computer, the backpack computer, and SSH into both to run the relevant software. The valve turning software framework has 3 major components:

## a) Processes that run on the robot's backpack computer

```
$ ssh <username>@<backpack_ip>
```

### 1. Start the backpack software (this includes TF and head controllers):

```
$ roslaunch hubo_launch backpack.launch
```

## b) Processes that run on the robot's on-board computer:

```
$ ssh hubo@<robot_ip>
```

### 1. Set ROS environment variable

```
$ export ROS_MASTER_URI=http://<backpack_ip>:11311/
```

### 2. Start hubo-motion based controller @jim

### 3. Start the hubo-ach to ROS interface node:

```
$ roslaunch hubo_launch full_body_feedback_node.launch
```

### 4. Start JointTrajectory ActionServer @jim

## c) Processes that run on the workstation (i.e. a remote computer)

For every terminal you start, you, should execute the following two bash commands:

```
$ export ROS_MASTER_URI=http://<backpack_ip>:11311/  
$ export ROS_HOSTNAME=<workstation_ip>
```

For ease of use, you could add the two lines mentioned above at the end of the `.bashrc` file in `/home/<your_user_name>` directory on the workstation.

After exporting the environment variables, ROS processes that you run on the workstation can now communicate with the robot and the backpack computer.

### 1. Start the OpenRAVE Based Motion Planner

```
$ roslaunch hubo_planner valve_planner.launch
```

## 2. Start the Interactive Marker Server for Valve Localization

```
$ rosrun valve_localization valve_localizer.py
```

### 3. a. Start RViz User Interface

```
$ rosrun rviz rviz
```

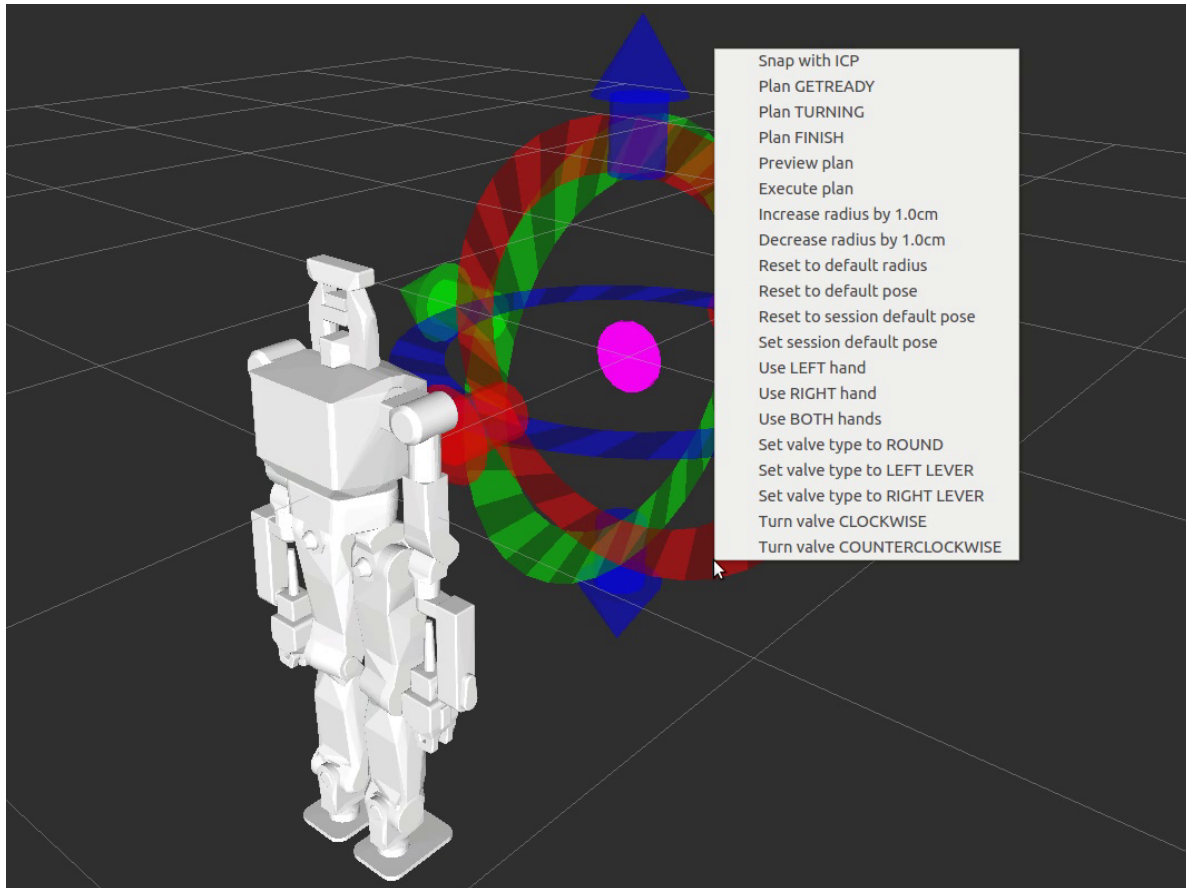
### 3. b. Add necessary visualizations from a configuration file

After RViz starts, press Ctrl+O to open a visualization configuration from file. When the open file box appears, it will ask you to find where the configuration file is stored. The configuration file you are looking for is stored under

```
/your_path_to/hubo_planner/rvizconfig/valve_turning_rviz_config.rviz
```

You can press Ctrl+S if you'd like to save the configuration on the workstation for the next session, in which case you will not have to open the rviz configuration file when you start RViz.

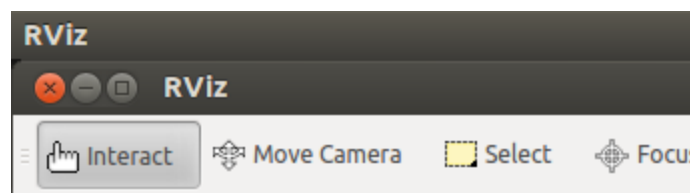
## Supervised Autonomy



*RViz Interface*

In our framework, the user controls the progress of the task using a graphical interface. We use the robot visualization tool, RViz, that comes with ROS out of the box. After running all executables, and finally starting user interface, you should see an interactive marker. By default the marker is going to be a disc. See figure above. The operator then can click and drag the interactive marker on each axis (XYZ axes are shown in RGB colors respectively), or rotate the marker around each axis by clicking and dragging, using the rings shown in same colors.

If you see the marker in magenta without the RGB arrows and rings (controls), make sure that the interaction mode is selected. On the upper left corner of RViz find the buttons shown below, and select interact. When clicked on Move Camera, controls disappear to give the user a clear view of the scene.



*Interact shows controls, whereas Move Camera hides controls*

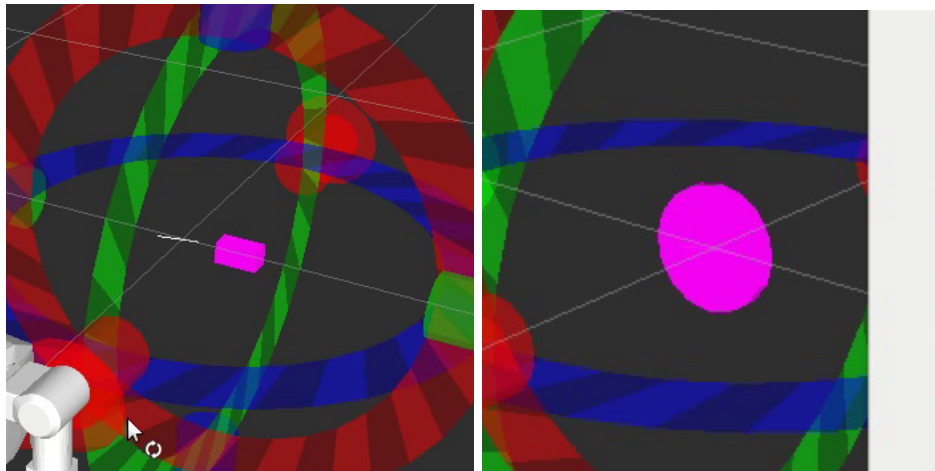
When you right click on the interactive marker, you will see a dropdown menu where you can select different options, and actions you can execute for the task. When operating, you should follow the steps below:

1. Choose the type of the valve:

Round type,

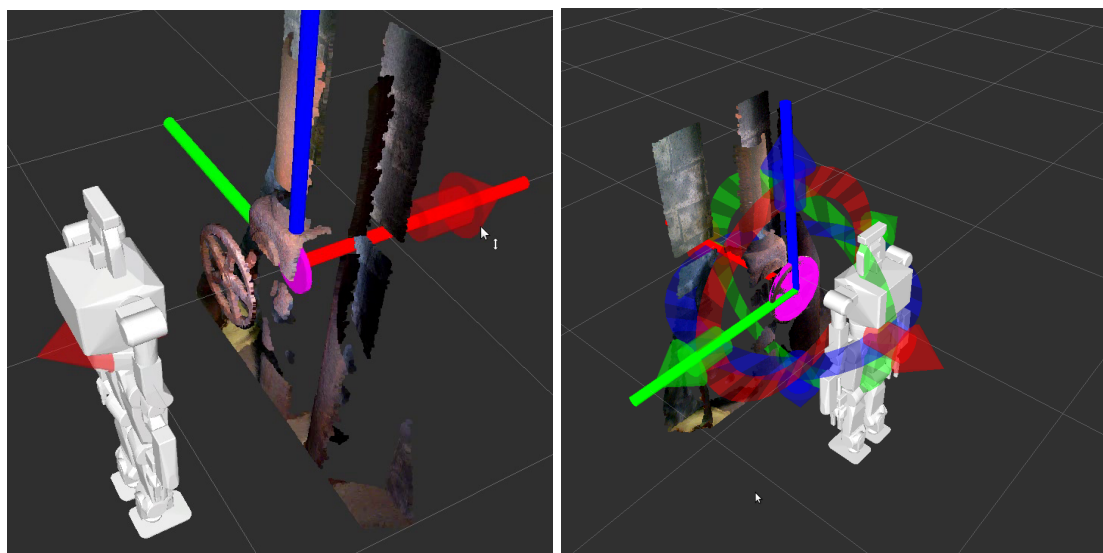
Left lever type (the rotation axis is on the left of the valve),

Right lever type (the rotation axis is on the right of the valve).



*Interactive marker for the lever type valve (left), and round wheel type valve.*

2. Locate the valve in the pointcloud, and using controls (see aligned / non-aligned valve figure below) align the marker over the valve you see in the point cloud.





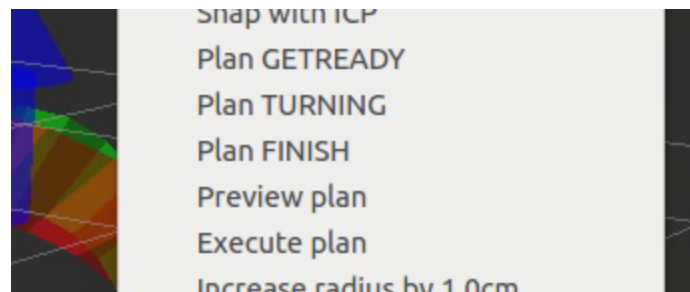
*Aligning interactive marker with the point cloud is the first thing the operator should do.  
Left: Marker is not aligned with the valve. Right: Marker is aligned and sized appropriately.*

3. Rotate the marker, and match the orientation of the marker to the orientation of the valve (pitch and yaw angles).
4. Adjust the size of the valve by increasing or decreasing by a default value. If required, adjust the orientation or position further.
6. Choose the direction of rotation for the valve (clockwise, counterclockwise)
7. Choose which manipulators (right, left, or both hands) to use for task execution
8. Choose which task stage to execute (see details below).
9. Plan the task stage.
10. Preview the plan in OpenRAVE and make sure that it looks safe.
11. Finally, choose execute the plan.

### **Task Stages for Valve Turning**

Valve turning task consists of three stages. At every stage you can:

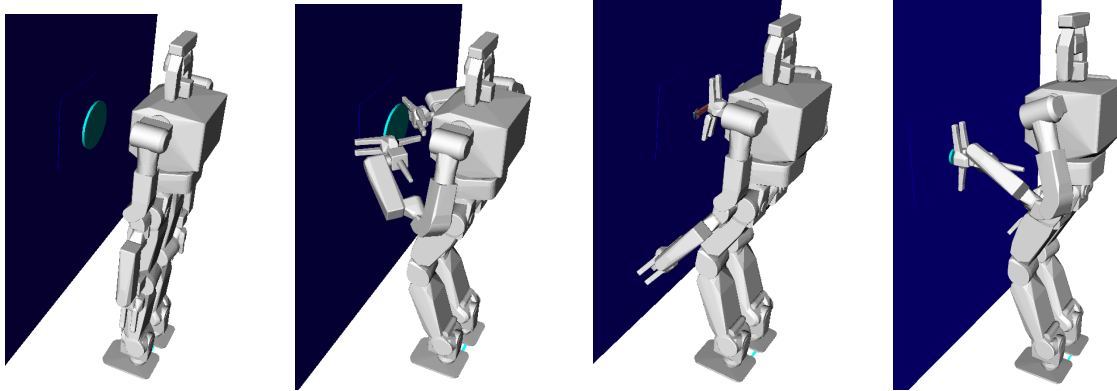
- a) trigger the trajectory planner using “Plan <task\_stage>” command,
- b) preview the planner trajectory the planning is done (and this is strictly advised and necessary for safety purposes),
- c) and finally, if you think that the trajectory is safe, you can command to execute the trajectory on the real robot.



*You can choose to plan end effector trajectories for the three stages of the task, preview the planned trajectory, and executing the plan on the robot using the options shown above in the dropdown menu.*

## Task Stage 1: Get Ready

This is the very first stage of the valve turning task. At this stage, when the planner triggered, it plans a path from the robot's current configuration (which is read from the encoder values of the robot), to a known configuration, and then to a suitable configuration for getting ready to grasp the valve. At the end of this stage the hands of the robot should be open and ready to grasp the valve as shown in figures below.



*OpenRAVE screenshots during motion planning. From left to right: Home position, get ready for turning a round valve with both hands, get ready for turning a left lever type valve with the right hand, and a round valve with the left hand.*

## Task Stage 2: Turn Valve

At this stage, the robot starts from its current configuration, plans a path to turn the valve, and if the valve turning task is executed with both hands, it goes back to a suitable configuration for turning the valve once again if necessary. In the case of round valve turning, the operator may trigger the planner once for the very first time, and execute the same trajectory more than once in order to keep the valve turning until the valve is observed to be fully on / off. In the case of lever type valve turning (for example a small ball valve), the robot's hand does **not** go back to where it starts from. In order to perform another turn the operator has to finish executing the task fully, and start a second task cycle.

## Task Stage 3: End Task

After turning the valve, the operator should trigger the planner for ending the task. At this stage, the robot plans a path from its current configuration to a known, safe return configuration, closes its hands, and goes back to its home position.

After executing this last stage, the operator may cycle through the stages once again.

# System Shutdown and Troubleshooting

## System Shutdown Procedure

TODO

## Troubleshooting

**Scenario:** Planner failed (startik is in collision, goalik is in collision, planner timeout).

TODO

**Scenario:** Real robot is in collision -- need to restart the planner.

TODO

**Scenario:** Real robot is in home position -- need to restart the planner.

TODO

**Scenario:** Can't see the pointcloud.

Make sure that RViz is subscribed to `/rgb_d_longrange/depth/points_xyzrgb` topic. Then make sure that the topic is actually being published. To check whether the topic is being published or not, open a terminal and type in the following command:

```
$ rostopic echo /rgb_d_longrange/depth/points_xyzrgb  
or  
$ rostopic echo /rgb_d_shortrange/depth/points_xyzrgb
```

If you see the data being printed out in the terminal, it means that the topic is being published. If you still can't see the pointcloud although it is being published, try removing the pointcloud visualization in RViz, and adding a new PointCloud2 visualization, and try subscribing to the topic you are having trouble with again.

**Scenario:** Pointcloud does not have color.

TODO

**Scenario:** The real robot moves but the robot mode in RViz does not move.

TODO

**Scenario:** Can't move the interactive marker, or interactive marker jumps back to its previous

location.  
TODO

**Scenario:** Right clicking on the interactive marker does not pop up the menu.  
TODO

# III. Software Description

## 1. hubo\_ros\_core

This stack forms the core of all ROS support for the hubo robot, both Hubo2+ and DRCHubo. It provides message definitions, the core interface to hubo-ach, publishers for joint state data, publishers for the robot's TF tree, and launch files to startup the robot and onboard components.

### 1. 1. Components

The components of **hubo\_ros\_core** fall into 3 major categories:

1. Action, message, and service definitions.

These are provided in **hubo\_robot\_msgs**, and **hubo\_sensor\_msgs**.

These provide definitions for all core ROS communications on the robot, such as robot joint states, commands for joint and joint trajectories, and commands for pointing the sensor head and assembling laser pointclouds.

2. Base interface from hubo-ach to ROS

This is provided in **hubo\_ach\_ros\_bridge**.

The “feedback” node provided in this package reads the state of the robot out of hubo-ach at approximately 200Hz and publishes it out in ROS. Above this level, it reprocesses the verbose 200Hz raw states into 20Hz compressed sensor\_msgs/JointState messages used by RViz and all other non-control nodes. An additional node “feedforward” is provided to take ROS commands and write them directly into hubo-ach's ref-filter or ref channel, however, this usage is not advised and has not been used extensively on DRCHubo.

3. Launch and configuration files

This is provided in **hubo\_launch**.

This package provides a central place for all launch files for the Hubo2+ and DRCHubo robots, including software not in hubo\_ros\_core. Once a new piece of ROS software for the robot itself (i.e. sensors, controllers) has been stabilized, launch files for it will be added here. Our goal is to have a single launch file for **all** ROS software that runs on both the chest and backpack computers.

### 1. 2. Development Status

All software in **hubo\_ros\_core** is stable. New launch files are added periodically, along with compatibility fixes to match drchubo, however, no major development is currently planned. These packages form the base building block for *all* ROS software running on Hubo2+ and

DRCHubo, so the addition of new features is contingent on them being stable (and compatible with Hubo2+ if possible).

### 1. 3. Dependencies

To compile, these packages require a full installation of ROS Groovy “desktop full”, the ACH IPC (install from debian package), and a copy of hubo-ach. Additional dependencies including `openni2_camera`, `openni2_launch`, `dynamixel_motor`, `drchubo`, and `hubo_ros_control` are required to run some of the launch files, but they are not required to compile.

### 1. 4. Build Instructions

Follow the instructions on the DASL DRC wiki and relevant github page to build this software. In particular, the DASL wiki provides instructions on how to install **hubo\_ros\_core** on the actual chest computers, as there are minor changes required for running on the actual robot.

### 1. 5. Run / Programming Guide

To start the core ROS software on the robot itself:

```
$ roslaunch hubo_launch full_body_feedback_node.launch
```

To start the software that runs on the backpack (if using it):

```
$ roslaunch hubo_launch backpack.launch
```

If you are not using the backpack, you will need to launch a similar file on your computer:

```
$ roslaunch hubo_launch display_drchubo_state.launch
```

To show the robot state in RViz, you will need to add a “Robot Model” display type and set the `robot_description` parameter to “/drchubo\_fullbody\_interface/robot\_description”. You should use “/world\_orientation\_frame” as your fixed frame in RViz, as this will show you the current orientation of the robot as measured by the robot’s inertial sensors.

To get meaningful data from these sensors, you will need to enable them in hubo-ach, either from the console or by using `hubo_init`.

## 2. hubo\_ros\_control

This package contains packages for controllers operated through ROS. They are robot-specific for DRCHubo with limited support for Hubo+.

### 2. 1. Components

This “stack” provides two high-level interface for controlling the sensor head and for executing full body motion trajectories:

1. **hubo\_head\_controller** : This package provides sensor head control through two Python nodes:
  - a. `point_head_controller.py` - This implements the `hubo_robot_msgs/PointHead` action interface. To control the head, you must specify a point in a frame of the robot and a frame on the head of the robot to look at it. The controller will then compute pan and tilt angles for the head, and drive the head to that orientation, provided the target orientation does not violate safety constraints.
  - b. `laser_scan_controller.py` - This implements the `hubo_sensor_msgs/LaserScan` action interface. The user specifies a minimum and maximum angle, a rate (in radians/s) to tilt the LIDAR, and the controller actuates the laser and assembles a pointcloud from the laser scans taken.
2. **hubo\_trajectory\_interface** : This package provides the hubo counterpart of the PR2 joint trajectory action, [http://ros.org/wiki/joint\\_trajectory\\_action](http://ros.org/wiki/joint_trajectory_action) with a limited number of feature implemented. It consists of an action server and three implementations of the backend trajectory executor. The action server provides a ROS interface that monitors the execution of the trajectory performed by one of the executor which send periodically configurations or chunks of the trajectory to hubo-motion-rt.

### 2. 2. Development Status

1. **hubo\_head\_controller** :
  - a. `point_head_controller` - Complete and fairly well tested, with support for both prebeta and beta sensor heads. Care should be taken when using the controller as it is not aware of the robot’s safety harness, and the head may get tangled in the safety lines.
  - b. `laser_scan_controller` Control code is complete and tested, however, the pointcloud assembly code is not complete. This code will be completed and tested following our return to WPI and access to similar LIDAR scanners as will be used on DRCHubo.
2. **hubo\_trajectory\_interface** : The action interface is stable, it is configurable using the parameters provided in the parameter server. The currently supported controller

interface is the `hubo_motion_interface`. It implements a daemon using `Hubo_Control` object (`hubo-motion-rt`) and has to be run with realtime privileges on the robot. Once a trajectory is received, configurations are sent to the control-daemon at 50Hz in trajectory mode which itself synchronizes with `hubo-daemon` 200Hz. It is implemented in a class which can easily be derivable to implement more complicated controllers such as operational space controllers. This frequency is adjustable.

## 2. 3. Dependencies

The `hubo_head_controller` package depends on the `dynamixel_motor` package to provide driver support for the servos used to control the head. To ensure reliability and a stable codebase, we have forked the official driver stack and will be maintaining our own variant of the `dynamixel` drivers tuned for our use. As a result, the standard ROS `dynamixel_motor` package is not suitable!

## 2. 4 Build Instructions

The `hubo_head_controller` requires no special build operations - `catkin_make` will build both the necessary definitions and the C++ nodes.

## 2. 5 Run / Programming Guide

A launch file to start the head controllers is provided in `hubo_launch`. Individual nodes for the head controllers should not be run by hand unless otherwise necessary. When the head starts, the sensor head will come up the vertical, forward-facing “zero” position, while the LIDAR mount will self-test to minimum and maximum angles before returning to the zero position. Both controllers will repeatedly attempt to start if their startup/self-test procedure fails, any simple errors like tangled cables or temporary obstructions should be recoverable with (and often without) human help.

# 3. localization\_tools

This set of packages provides the interactive marker-based user interface we use for valve localization and the service API used to call the planner. The interface node is designed to serve as an example for using asynchronous user-guided localization to provide a target for a motion planner - with fairly limited work, the skeleton of our interface can be reused for any task that can be broken down into “sense->align->command->execute” pipeline where processed sensor data is delivered to the user, the user performs perception operations to identify objects in the environment, the user sends a command to the robot, and then the robot autonomously plans and executes the task.



### 3. 1. Components

1. **icp\_server & icp\_server\_msgs** - Message/service API and server to compute Iterative-Closest-Point alignment between pointclouds. The server is intended to be used to “snap” objects against the pointclouds produced by the robot’s sensors, either when commanded by the user or used online in an autonomous application such as visual servoing.
2. **valve\_planner\_msgs** - The message/service API used to communicate between our user interface and our planner.
3. **valve\_localization** - Our interactive marker-based user interface.

### 3. 2. Development Status

1. **icp\_server & icp\_server\_msgs** - Message/service API is stable, however, the ICP server is not yet tested.
2. **valve\_planner\_msgs** - Message/service API is largely stable.
3. **valve\_localization** - User interface is currently semi-stable. We intend to move from a purely interactive-marker based approach to a combined marker and RViz panel UI where our entire interface can be launch from the GUI.

### 3. 3. Dependencies

There are no dependencies beyond those that ship in ROS Groovy “desktop full”.

### 3. 4. Build Instructions

No special build steps are required. Run `catkin_make` to build all packages.

### 3. 5. Run / Programming Guide

To run the user interface:

```
$ rosrun valve_localization valve_localizer.py
```

Note that the `valve_localizer.py` script will wait the planner to be up and running. For information on how to run the planner see Section `hubo_planner` package details below.

## 4. teleop\_toolkit

As DARPA has repeatedly noted, the datalink between the robot and user workstation will be constrained. This will either take place along the the lines described at kickoff (severe and variable latency, jitter, and packet loss) or along the lines used at the VRC (tiers of total bits used, the fewer the better) and used as a component of team scoring. If the VRC tiers are any indication of what will be available in the actual challenge, they will be extremely restrictive (the lowest robot->user tier is ~30Kb/s, the lowest user->robot tier is 64*bits/s*). Therefore, the **teleop\_toolkit** packages exists to provide a platform for testing and experimenting with datalink software implemented both using higher-level ROS transport mechanisms and lower-level transport mechanisms (i.e. sockets) with the explicit goal of reducing the data transmitted from robot to workstation (and the reverse).

The objective of the tools in this package is to enable supervised autonomous control on DRCHubo in poor network conditions and suitable for the extremely low bandwidth we expect to encounter in December **in a manner that is transparent to client software**. We believe that supervised autonomy currently gives us the best chance of successfully accomplishing the tasks with the least reasonable intervention from the user, and that as a result, **teleop\_toolkit** provides key enabling support for this strategy. As we believe that trying to implement support for constrained networking at the level of individual nodes would result in excessive development time and wasted processing power, teleop\_toolkit serves to provide a common backbone for all client code that needs to use the datalink between the robot and workstation.

### 4. 1. Components

1. **opportunistic\_link** - This package provides datalink endpoints for generic ROS data types, and dedicated ones to support compressed image transfer. These link endpoints support automatic switching to start/stop data flow over the link and rate control to support data forwarding at any arbitrary rate ("native" to zero). Additionally, the rate controllers support single-message requests in which data will only be forwarded as a response to an explicit request.
2. **teleop\_msgs** - This package provides the message and service API implemented by the various nodes in the teleop\_toolkit package. The package's message definitions support transmission of serialized, compressed, and aggregated message data. Complementing these messages, the package's service definitions provide for link flow control, link rate control, and direct message data requests.
3. **teleop\_mux\_demux** - This package provides experimental support for aggregating multiple topics together for transmissions combined with automatic dis-aggregation on the recipient side. This functionality is intended mainly for use with general-purpose

compression algorithms (as yet unimplemented) which we believe will be far more effective on larger blocks of messages than single messages.

4. **teleop\_launch** - This package hosts all teleoperation/supervised autonomy-related launch files, including those used for standalone testing of datalink components and for testing datalink components with the real DRCHubo.

## 4. 2. Development Status

The entire teleop\_toolkit repository is undergoing active testing and development. The message and service APIs should be expected to be stable in their current states - we do not plan to change these unless our testing identifies serious problems. Depending on the specific *observed behavior* of the code currently in this repository (rather than the behavior defined in the message/service API) is potentially risky and we advise against it. See the Programming Guide in section 5.5 for further details on these APIs and integration of them into other software.

## 4. 3. Dependencies

Currently, this package depends only on the components of a standard ROS Groovy installation. In the coming weeks, as we add support for better image/video encoding using H.264/x264, this may add additional system dependencies to support image transfer.

## 4. 4. Build Instructions

This package requires no special build configuration. Simply clone it inside an existing Catkin workspace and run `catkin_make` to build all packages.

## 4. 5. Run / Programming Guide

There are currently four nodes in active use. These have all been experimentally incorporated into a set of launch files for using the limited datalink with DRCHubo. Additionally, there are two services that make up the core of the ROS API for the package.

1. Launch files:

- a. `drchubo_robot`

```
[$ roslaunch teleop_launch drchubo_robot.launch]
```

This is run on the backpack computer and it starts all core ROS software for the robot, including the hubo-ach to ROS bridge node on the robot itself, the `joint_state` publisher, the controllers for the sensor head, the drivers for the RGBD cameras, and all robot-side TF broadcasters. **Note that this launch file remaps all use of `/tf` to `/robot_tf` on the robot. You must include a `<remap from="/tf" to="$(arg remapped_tf)" />` tag for each node in your launch files to support this remapping.**

b. drchubo\_workstation

```
[$ roslaunch teleop_launch drchubo_workstation.launch]
```

This is run on the workstation computer, and it starts client-side TF broadcasters and link endpoint software to support the datalink. **Note that this launch file remaps all use of /tf to /workstation\_tf on the workstation. You must include a <remap from="/tf" to="\$(arg remapped\_tf)" /> tag for each node in your launch files to support this remapping.**

2. Nodes:

- a. limited\_link\_startpoint.py - This node provides a generic datalink startpoint that supports all ROS message types. It controls the flow of republished data via LinkControl and RateControl service calls. This node is suitable for all topics *except* for images, as image data requires special compression. This node is run on the sending side on the datalink.
- b. link\_endpoint.py - This complements the startpoint, providing automatic LinkControl switching when subscribers connect and disconnect from the endpoint node. This node is run on the receiving end of the datalink, and publishes the topics that your software will subscribe to. Additionally, it provides automatic link reconnection if the datalink between s
- c. limited\_image\_link\_startpoint - This node is an image-specific variant that supports transparent image compression using the ROS image\_transport library. Note that while it supports a range of compression algorithms, we have found the existing theora implementation to be extremely unreliable in poor network environments, and we currently recommend that you use jpeg/png compression instead (specified with the "compressed" argument).
- d. image\_link\_endpoint - Matching client-side node for image data transfer.

3. Services:

If you intend to implement the API defined in these packages with your own code, such as a special sensor driver, or an interface tool to request sensor data, you should implement the following service API.

- a. LinkControl - Required on the sending side. Service calls to this interface can turn the flow of data on and off regardless of the currently set publishing rate. This service is intended to be used internally between the start and endpoint node pair, and should not usually be used by client nodes directly.
- b. RateControl - Required on the server side, and used directly on the client side to control data flow. Service calls to this interface can set the republishing rate for data over the link. There are three reserved values: 0.0, meaning "stop"; INF, meaning "native rate", and -1.0, meaning "request". Stop simply means that data flow should be stopped. Native rate means that fixed-rate republishing should be stopped and data should be republished as fast as it is available. Request means that **only** the currently stored message should be republished. Note that using

Request when the rate is set to INF will result in an error, as there will never be a currently stored message. To use this option, you need to stop the flow first, then request individual messages.

## 5. hubo\_planner

This package keeps the motion planning script that the operator interacts from the user interface.

### 5. 1. Components

1. **drchubo\_planner\_interface.py**: ROS interface for the motion planner
2. **drchubo\_v2\_wheel\_turning.py**: OpenRAVE and CoMPS based motion planner
3. **base\_wheel\_turning.py**: Base class for wheel turning class
4. **valve\_planner.launch**: ROS Launch file for running the motion planner

EXPLAIN USING IKFAST / NOT USING IKFAST HERE!!!

### 5. 2. Development Status

The code has passed its alpha stage where we experimented with balance and trajectory constraints. As of this version, the planner still uses drchubo-v2.robot.xml. We are actively improving a) error handling and reporting in case of a failure, b) task autonomy, c) safety and c) obstacle avoidance.

### 5. 3. Dependencies

OpenRAVE 0.8.2 [http://www.openrave.org/docs/latest\\_stable/install/](http://www.openrave.org/docs/latest_stable/install/)  
CoMPS <http://sourceforge.net/projects/comps/>  
valve\_localization [https://github.com/WPI-ARC/localization\\_tools](https://github.com/WPI-ARC/localization_tools)  
valve\_planner\_msgs [https://github.com/WPI-ARC/localization\\_tools](https://github.com/WPI-ARC/localization_tools)  
RViz <http://www.ros.org/wiki/rviz>

### 5. 4. Build Instructions

All scripts are written in Python, however since there are message types that the package depends on, the user should run catkin\_make after cloning all the dependencies in his/her catkin workspace.

### 5. 5. Run / Programming Guide

When used with the real robot, the planner should be launched using:

```
$ roslaunch hubo_planner valve_planner.launch
```

In your terminal, you should see the following message to know that the planner is up and running:

```
Robot ready to plan, waiting for a valve pose update...
```

However for training or testing purposes, the planner can be run without the real robot. For this, the user has to run on a workstation:

```
$ roscore
$ roslaunch hubo_launch display_drchubo_state.launch
$ roslaunch hubo_planner valve_planner_sim.launch
$ rosrun valve_localization valve_localizer.py
$ rosrun rviz rviz
```

The last five commands above run the RViz (interface) to OpenRAVE (planner) pipeline without the necessity of a real robot. This way the user can test the planner under different circumstances, for example, using different manipulators, or valve sizes for planning.

Back-up Trajectories are generated under

```
/your_path_to/hubo_planner/trajectories/
```

Following is the list of the trajectory files generated. Files with .traj extension are readable by Dan Lafaro's hubo\_trajectory\_reader. Files with .txt extension are OpenRAVE trajectory files.

**Get ready** outputs:

```
movetraj0
openhands_after_movetraj0
movetraj1
```

**Turn valve with both hands** outputs:

```
movetraj2
closehands_before_movetraj3
movetraj3
openhands_after_movetraj3
movetraj4
movetraj5
movetraj6
```

**Turn valve with left hand** outputs:

movetraj2  
closehands\_before\_movetraj3  
movetraj3  
openhands\_after\_movetraj3  
movetraj4 -- only if it's a round wheel

**Turn valve with right hand** outputs:

movetraj2  
openhands\_before\_movetraj3  
movetraj3  
closehands\_after\_movetraj3  
movetraj4 -- only if it's a round wheel

**End task** outputs:

openhands\_before\_movetraj7  
movetraj7  
closehands\_after\_movetraj7  
movetraj8