# Baxter Gazebo Simulator

* Run the baxter shell script with sim specified:

**./baxter.sh**

* Start simulation with controllers：

**roslaunch baxter\_gazebo baxter\_world.launch**

* Wait for the following three lines before the simulator is truly running:

[ INFO] [1400513321.531488283, 34.216000000]: Simulator is loaded and started successfully

[ INFO] [1400513321.535040726, 34.219000000]: Robot is disabled

[ INFO] [1400513321.535125386, 34.220000000]: Gravity compensation was turned off

**By default the robot would be in disabled state.**

* Enable Robot

1. Robot State:  
   **rosrun baxer\_tools enable\_robot.py -s**
2. Enable Robot

**rosrun baxter\_tools enable\_robot.py -e**

1. Disable Robot

**rosrun baxter\_tools enable\_robot.py -d**

1. Reset

**rosrun baxter\_tools enable\_robot.py -r**

1. Stop

**rosrun baxter\_tools enable\_robot.py -S**

* Tuck Arms Example

The tuck arms example program will allow you to tuck/untuck Baxter’s arms to/from ‘shipping pose’.



The pose that Baxter is shipping in is reffered to as ‘shipping pose’. This tool is particularly useful for the initial unpaxking of Baxter. This tools is also useful for subsequent movements of Baxter(i.e. moving Baxter through doorways etc.) as this is the most compact form for Baxter’s transportation.

1. To untuck the robot’s arms:

**rosrun baxter\_tools tuck\_arms.py -u**

1. To tuck the robot’s arms:

**rosrun baxter\_tools tuck\_arms.py -t**

Jonits

Baxter has 7 joints(DoF) in each of its two arms, and two more joints in its head (side-to-side panning, ad binary, up-down nodding). The control for the head is done separately from the arms; however, you can read the current joint states (position, velocity, and effort) for all the joints on both arms and head by subscribing to one topic:

**\robot\joint\_states (sensor\_msgs-JointState)**

where the units for the position of a joint are in (rad), the units of velocity are in (rad/s) and the units of effort in each joint is in (Nm)

1. Arm Joints

Component IDs:

left\_e0, left\_e1, left\_s0, left\_s1, left\_w0, left\_w1, left\_w2, right\_e0, right\_e1, right\_s0, right\_s1, right\_w0, right\_w1, right\_w2

1. Arm Joint States   
   name[i]: '<component\_id>' of i-th joint in message value arrays.

position[i]: position of joint i rad

velocity[i]: velocity of joint i in rad/s

effort[i]: effort applied in joint i in Nm

Joint states are published by the robot\_state\_publisher and are updated with information from the sensors for every cycle.

• Set joint State Publishing Rate

/robot/joint\_state\_publish\_rate

The rate at which the joint are published can be controlled by publishing a frequency on this topic. Default rate is **100Hz;** Maximun is **1000Hz**.

1. Arm Joint Control

There are three modes for controlling the arms: **Position Mode, Velocity Mode, and Torque Mode**. Alternatively, a joint trajectory action server has been created in support of timestamped joint position trajectories using the ROS standard joint trajectory action.

Each arm can be in independent control modes by publishing the desired control mode to the topic for the appropriate arm.

* + Position Mode

Users publish target joint angles for given joints and the internal controller drives the arm to the published angles.

* + Velocity Mode

**Warning: Advanced Control Mode.**

Users publish velocities for given joints and the joints will move at the specified velocity.

* + Torque Mode

**Warning: Advanced Control Mode.**

Users publish torques for given joints and the joints will move at the specified torque.

\* Switching Modes

Mode is set implicity by specifying the mode in the command message. Publish a **JointCommand** message to the **joint\_command** topic for a given arm to set the arm into the desired control mode.

* + Joint Trajectory Action

The Joint Trajectory Action provides an **ROS action interface** for tracking trajectory execution.

Usage:

*# Verify that the robot is enabled:*  
**$ rosrun baxter\_tools enable\_robot.py***# Start the joint trajectory action server:*  
**$ rosrun baxter\_interface joint\_trajectory\_action\_server.py**

* + Parameters:

<joint\_name>\_kp, <joint\_name>\_kd, <joint\_name>\_ki

goal\_time, \_goal, trajectory

**Dynamic Reconfigure GUI** is suggest for use with ROS Distributions >=Groovy for setting these parameters.

*# Start the dynamic reconfigure GUI:*

**$ rosrun rqt\_reconfigure rqt\_reconfigure**

Alternatively, these parameters can be set via a YAML file, command line, or programmatically (rospy, roscpp).

* Head Joint

1. Head Position and State

•Component IDs: head\_nod, head\_pan

•Head State

\***pan** field gives you the current angle (radians) of the head. 0 is forward, -pi/2 to Baxter’s right, and +pi/2 to Baxter’s left.

\***isPanning and inNodding** are Boolean fields that will switch to True while the robot is executing a command.

•Head (Joint) State

The position of the head may also be determined from the joint\_state message. Note: The ‘nod’ joint will never update, as it is only a binary state.

1. Head Movement Control

•Pan Head

\* **target** sets the target angle. 0.0 is straight ahead.

\* **speed** is an integer from [0, 100]

\* Setting an angle in the command\_head\_pan topic does not gurantee the head will get to that position. There is a small deband around the reference angle around the order of +/-0.12 radians. **Note:** Speed of the head movement cannot be controlled at this point of time.

•Nod head: Send True to nod.

1. Example

*# Check head position/state:*

**$ rostopic echo /robot/head/head\_state**

*# Move (pan) head side-to-side:*

**$ rostopic pub /robot/head/command\_head\_pan**

**baxter\_core\_msgs/HeadPanCommand -- 10.0 100**

*# Make head nod up-down:*

**$ rostopic pub /robot/head/command\_head\_nod**

**std\_msgs/Nool True**

1. Caresian (Endpoint State)

Published at 100Hz, the endpoint state topic provides the current Cartesian Position, Velocity and Effort at the endpoint for either limb.

\*Endpoint State

The endpoint state message provides the current **position/orientation pose, linear/angular velocity, and force/torque effort** of the robot end-effector at 100Hz. **Note:** The wrench and the twist values would be 0 for the simulated baxter since it is not implemented.

Sensors

* Accelerometers

Each hand has a 3-axis accelerometer located inside the cuff, in the same plane as the gripper electrical connection header. The positive z-axis points back ‘up’ the arm (towards the previous wrist joint, w0). The positive x-axis points towards the camera, and the y-axis points towards the cuff buttons, using standard Right-Hand-rule notation.

\*Component IDs: **left\_accelerometer, right\_accelerometer**

\*Read Linear Acceleration: /robot/accelerometer/<component\_id>/state

\*Acceleration **values** (in m/s^2) are published under **linear\_acceleration** for the x, y, and z axes. The force of **gravity is NOT ccompensated for.**

\*Note: **linear\_acceleration** is currently the only valid data in the message.

* IR Range

There are two IR Range snesors on each hand. The sensor data is published on the th frame **<side>\_hand\_range**, and points down the +x axis, as is convention for **rviz**

\*Component IDs: **left\_hand\_range, right\_hand\_range**

\*Read Range Sensor Measurements: /robot/range/<component\_id>

\*Range is published in meters in the **range** field

\*When the sensor goes beyond its max range, the invalid value of 65.5350036621(m) is published instead. Users should check for valid measurements using the**min\_range** and **max\_range** fields.

* Sonar

Mounted in a ring around the head are 12 sonar distance sensors.

\*Component IDs: **head\_sonar**

• Read Sonar 3D Distance (Point Cloud):/robot/sonar/head\_sonar/state

The sonar range measurements are published as a set of 3D coordinate points in space around Baxter, called a **PointCloud.** This gives a “mapping” of detections in the workspace and is well-suited for occupancy-oriented reasoning in the robot’s world.

\*The points for all incoming, valid, current readings are collected each time and published as an **array** in the PointCloud message.

\*Note: the result of this is that each published message contains measurements for only a subset of the sensors, not reading for all 12 sensors every time.

\*The **points** field contains the array of the actual 3D Points - one for each current measurement - in terms of the (x,y,z) coordinates for each point, relative to Baxter’s **base** tf frame.

\*The **channels** field contains arrays with additional information about each point. The values in each array **map 1:1** to the values at the same indices in the **points** arrays.

\*The array with the name **SensorId** lishs which physical sensor each point came from.

\*The array with the name **Distance** lists the original range measurement from that sensor.

Inputs and Outputs

* Navigators

There are four Navigators on that are simulated as a QT plugin in the simulated baxter: two for each side of the torso and one for each arm. Each Navigator is comprised of three push buttons, and an indexing scroll wheel.

\*Component IDs: **left\_navigator, right\_navigator, torso\_left\_navigator, torso\_right\_navigator**

• Navigator Input Buttons

\*Read Button States: /robot/navigators/<component\_id>/state

\*The states of the three push buttons are the first three values in the Boolean **buttons[]** array. A value of ‘True’ indicates the button is currently pushed down. The order is as follows:

\*OK\_BUTTON (**buttons[0]**): The small rectangular button labelled as ‘OK’.

\*CANCEL\_BUTTON (**button[1]**): The button ‘above’ the OK\_BUTTON.

\*SHOW\_BUTTON (**buttons[2]**): The SHOW\_BUTTON, or “Rethink button”, is ‘below’ the OK\_BUTTON.

\*Read Wheel Index: /robot/navigators/<component\_id>/state

\*The wheel field returns an integer between **[0,255].** Each physical ‘click’ of the wheel corresponds to a +/-1 increment. The value will loop when it goes above or below the buounds.

• Control Navigator Lights

There are two sets of lights on each avigator: the ‘linner’ light ring around the circular OK Button, and the ‘outer’ oval light ring around the entire Navigator. Each light is identified by the ‘Component ID’ of its Navigator followed by **\_light\_inner or \_lighter\_outer.**

\*Component IDs:

Inner Lights: **left\_inner\_light, right\_innrt\_light, torso\_left\_inner\_light, torso\_right\_inner\_light**

Outer Lights: **left\_outer\_light, right\_outer\_light, torso\_left\_outer\_light, torso\_right\_outer\_light**

\*Turn LEDs On/Off

name:<navigator\_component\_id>\_light\_inner/outer

value:{True, False}

Publish a DigitalOutputCommand message with the component id of the light as the **name** and a **value** of True or False to turn the LEDs On or Off, respectively.

\*Check State of LEDs: The state field will give you the current setting of the **LED, ON(1) or OFF(0).** *Note: The Navigator LED are not simulated in gazebo and hence viewing the lights go on and off is not possible.*

•Shoulder Buttons

There are two shoulder buttons on the back of the torso, one on each side. The state of each button is published in a DigitalIOState message under its own topic (DigitalIOState constants: PRESSED==1, UNPRESSED==0).

\*Component IDs: **left\_shoulder\_button, right\_shoulder\_button**

•Cuff Buttons

There are two buttons and one touch sensor in the cuff of each hand.

\*Cuff (Squeeze) Sensor

Component IDs: **left\_lower\_cuff, right\_lower\_cuff**

\*Cuff OK Button: This is the circular button on the cuff.

Component IDs: **left\_lower\_button, right\_lower button**

\*Cuff Grasp Button: This is the long, this button on the cuff.

Component IDs: **left\_upper\_button, right\_upper\_button**

Lights

•Head Halo (Red, Green)

•Sonar Rings (Yellow)

•Navigator LEDs (inner, outer)

Digital IO

•Read Digital Input State

\*state: field will be 0 (for Trie, Pressed, On) or 1 (for False, Released, Off). If the component is an output, then the state field will be the current setting of the outpu.

\*isInputOnly: field tells you if the component is an input (sensor) only, and cannot output (not a light, actuator, nor indicator).

•Control Digital Output:

\* Publish a DigitalOutputCommand message with the component id of the Output as the name and a value of True or False to turn the Output On or off, respectively.

•All Digital Component IDs:

***Outputs:***Inner\_Lights: **left\_inner\_light, right\_inner\_light, torso\_left\_inner\_light, torso\_right\_inner\_light**Outer\_Lights: **left\_outer\_light, right\_outer\_light, torso\_left\_outer\_light, torso\_right\_outer\_light**

***Inputs:***Back Shoulder Buttons:**left\_shoulder\_button, right\_shoulder\_button**Cuff (Squeeze) Sensor: **left\_lower\_cuff, right\_lower\_cuff**Cuff OK Button: **left\_lower\_button, right\_lower\_button**Cuff Grasp Button: **left\_upper\_button, right\_upper\_button**

Cameras

You can access Baxter’s two hand cameras and the head camera using the standard ROS image types and image\_transport mechanism listed below. These cameras would show rendered images of the gazebo environment. Useful tools for using cameras in ROS include **rviz and the image\_view program**. IMPORTANT: There is no limitation on the number of cameras used simultaneously in simulation as in the real baxter.

Component IDs**:** **left\_hand\_camera, right\_hand\_camera, head\_camera**

**•**Camera Published Topics

\*Raw Image: /cameras/<component\_id>/image

\*Camera Intrinsics: /cameras/<component\_id>/camera\_info

*# View now open camera topics*

$ **rostopic list /cameras**

/cameras/head\_camera/camera\_info

/cameras/head\_camera/image

/cameras/right\_hand\_camera/camera\_info

/cameras/right\_hand\_camera/image

/cameras/left\_hand\_camera/camera\_info

/cameras/left\_hand\_camera/image

Screen (xdisplay)

Images can be displayed on Baxterr’s LCD screen by publishing the image data as a ROS **sensor\_msgs/Image.**

Display Image: **/robot/xdisplay**

\*Publish image data as a Ros Image message to update the display.

\*The screen resolution is 1024\*600. Images smaller than this will appear in the top-left corner.

\*There are dedicated ROS packages for working with and sending ROS image messages, including **image\_transport and image pipeline.**

\*Useful tools for working with images in ROS include **Image\_view and republish.** Also see **camera\_drivers** for assistance working with your own cameras.