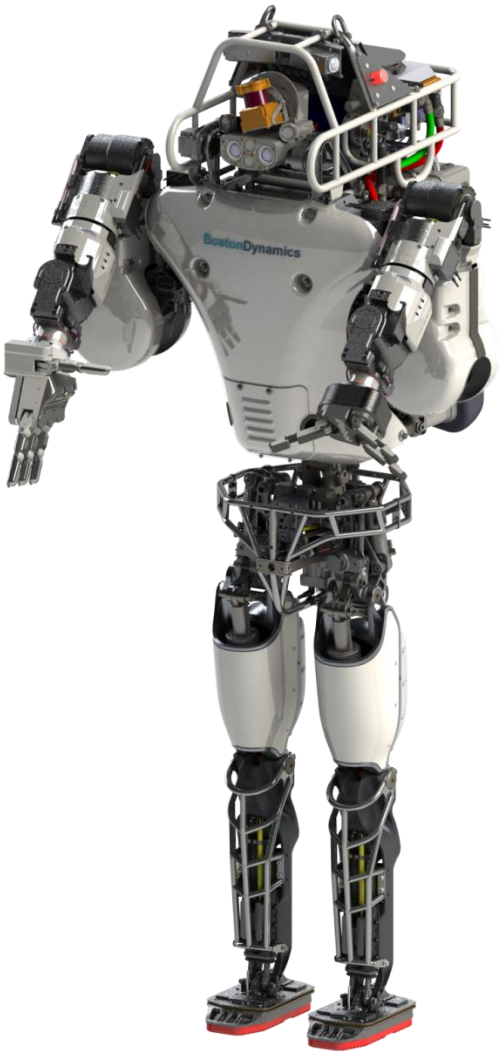
Boston Dynamics

Boston Dynamics Atlas Robot

Software and Control Manual



Document Number ATLAS-01-0019–v1.7

For AtlasRobotInterface Version 3.0

Document Revision History:

Revision	Date	Description of Change	Changed By
0.1	6/24/13	Initial Draft	Adam Crane
0.2	6/26/13	Updates for Pre Training Packet	Adam Crane
0.3	6/28/12	Updates for Training	Adam Crane
0.4	7/3/13	More updates for Training	Adam Crane
0.5	7/11/13	Edits for initial release	Adam Crane
0.6	7/23/13	Edits to servo valve joint control	Neil Neville
1.0	8/8/13	Update version to 1.0 after official release	Adam Crane
1.1	9/26/13	Updates to behavior documentation	Adam Crane
1.4	10/20/13	Updates to behavior documentation	Joel Chestnutt
1.5	11/25/13	Added details related to robot software installation	Neil Neville
1.6	2/07/14	Updates to API version 2.11.0	Yeuhi Abe
1.7	01/08/15	Updates for unplugged robot design, API / robot software versions 3.0	Neil Neville
2.0	02/05/15	Added Soft Stop, overall edit of behavior section.	Bill Blank

Table of Contents

List of Figures and Tables

List of Acronyms

F/T	Force/Torque
Gbps	Gigabit per second
GUI	Graphic User Interface
I/O	Input/Output
IMU	Inertial Measurement Unit
PID	Proportional Integral Derivative
PSI	Pounds per Square Inch
CIS	Cubic Inches per Second
SA	Situational Awareness
TBD	To Be Determined

Preface

Welcome to the Boston Dynamics *ATLAS Robot Software and Control Manual*. This manual will give you an overview of the Atlas robot software, guide you through the use of the Atlas Robot Interface, the API used for communicating with and controlling the Alas robot. If you have any questions, please feel free to contact Boston Dynamics.

Intended Audience and Use

This guide is intended for use by operators of the Boston Dynamics Atlas Robot who will operate and program the robot.

Contacting Boston Dynamics

To contact Boston Dynamics:
Boston Dynamics
78 Fourth Avenue
Waltham, MA 02451-7507, USA

Phone: 617-868-5600
Support Email: atlassupport@BostonDynamics.com
Robot Sales Email: atlassupport@BostonDynamics.com

Referenced Documents

Document Number	Title	Revision/Version
ATLAS-01_0018	Atlas Robot Operation and Maintenance Manual	2.0

1 Installation

Presented here are guidelines on how to install the AtlasRobotInterface software onto an Ubuntu 14.04 64-bit system. For example purposes it will be assumed that the software is to be installed into the `/usr/local/share` directory, and that 3.0.0 is the version to be installed. The actual installation directory and version being installed may be different.

In the steps below, the leading '`$`' denotes a Linux command prompt. It should not actually be typed in when entering commands.

1.1 Manual Installation

1.1.1 Unpack the Archive

Go to the installation directory:

```
$ cd /usr/local/share
```

Un-tar and un-gzip the Atlas Robot Interface archive into the installation directory:

```
$ sudo tar xzvf AtlasRobotInterface_3.0.0.tar.gz
```

If the software is being installed into a directory other than a system directory, the `sudo` at the beginning of the command is optional.

1.1.2 Set environment variables

Some Atlas Robot Interface software needs to know where the library is installed. The `ATLAS_ROBOT_INTERFACE` environment variable is used for this.

```
$ export ATLAS_ROBOT_INTERFACE=/usr/local/share/AtlasRobotInterface_3.0.0
```

This environment variable should in general be set by a shell script that sets up a development environment, such as `.bashrc`.

1.1.3 Add Paths to `LD_LIBRARY_PATH`

If the Atlas Robot Interface library directory has not already been added to the `LD_LIBRARY_PATH` environment variable, add it now.

If `LD_LIBRARY_PATH` is not yet set, set it:

```
$ export LD_LIBRARY_PATH=$ATLAS_ROBOT_INTERFACE/lib64
```

If it is set, append to it:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ATLAS_ROBOT_INTERFACE/lib64
```

As with setting `ATLAS_ROBOT_INTERFACE`, setting `LD_LIBRARY_PATH` should in general be set by a shell script that sets up a run-time environment.

1.1.4 Example programs

It is recommended that the `examples` directory be copied to a personal directory. That way you can always revert to or consult the original example code.

1.1.5 Where To Go From Here

The main documentation index can be loaded into a web browser:

```
$ firefox $ATLAS_ROBOT_INTERFACE/doc/html/index.html &
```

1.1.6 Installation Contents

The following directories are in the Atlas Robot Interface installation:

`/doc`	Atlas Robot Interface documentation
`/examples`	Example programs
`/include`	Header files for C++ API
`/lib`	Shared-object library
`/tools`	Tools for downloading information from the robot.

1.2 Updating Software

Boston Dynamics will occasionally provide updates to both the AtlasRobotInterface software and the on-robot software. This section provides information on how to do both.

1.2.1 Updating AtlasRobotInterface

Updated AtlasRobotInterface software will be provided in the same form as previous packages, a compressed tar file. In both cases the Manual Installation instructions should be followed, with the following exceptions.

In general, new distributions should have different version numbers. In this case all instructions should be followed as-is. *Previous versions do not need to be uninstalled*. The `ATLAS_ROBOT_INTERFACE` environment variable just needs to be changed to point to the new version.

If new distributions have the same version number, the existing directory should be renamed so it can be referred to or reverted to if necessary.

```
$ cd ATLAS_ROBOT_INTERFACE=/usr/local/share
$ mv AtlasRobotInterface_3.0.0 AtlasRobotInterface_3.0.0.OLD
```

From here the regular instructions should be followed.

Note that sometimes on-robot software will need to be updated at the same time in order to maintain compatibility between the AtlasRobotInterface and on-robot software.

1.2.2 Updating On-Robot Software

On-robot software will occasionally require an update; for example, the **Walk** behavior has been updated for better stability, or control data structures have changed. Boston Dynamics will provide a software update package in the form of one file that contains everything to be uploaded to the robot. *This file should not be modified in any way.* This file will contain a small amount of human readable text, that identifies the date the package was generated and the package version.

Robot software releases will be labeled with the version, for example: 1.9.0-r3.

- The first number is the major revision.
- The second number is the minor revision. **Important note**: the minor revision refers to the protocol used by the API. The minor revision number for the API and robot software **must match** for the API to communicate correctly with the robot software.
- The third number, 0 in this example, is the point revision. The point revision incremented with significant changes to functionality
- A dash may be added with additional information
    - –b: beta. Reserved for versions not stable enough for full deployment.
    - –rc : release candidate. Software is in testing phase.
    - –r: revision. Revisions represent small changes to fully deployed software, typically minor bug fixes, updated hardware configuration, update calibration data

Robot software is installed using the following procedure:

1. Navigate to 192.168.130.103:3000 in a web browser
2. Click the Browse button to select the desired package (.bde)
3. Click the Upload button
   ***** Do not turn off power to the robot until the installation procedure is complete *****
4. Wait until you see the "Install Succeeded" page (shown in the figure below).
5. Reboot the robot by latching the E-stop, waiting 30 seconds, and unlatching the E-stop

Hardware configuration patches can be used to overwrite hardware configuration files from a robot software release, or another patch. Patches are installed in the same way as robot software releases. Patches are normally created for a specific robot, there is no reason to use a patch created for another robot. In addition, hardware configuration data included in a patch will also be included in future robot software releases.

Installing a robot software release or a hardware configuration patch will overwrite any calibration data generated using the API calibration functionality.

2 General Concepts

This chapter will introduce some of the basic concepts used in the AtlasRobotInterface library. They are:

- **Robot Run States**: The state the robot is in in terms of being controlled by the AtlasRobotInterface.
- **Robot Behaviors**: A mode the robot is in while in the **Control** state that potentially drives the robot with built-in control algorithms.

2.1 Run States

A "run state" (hereafter referred to as just a "state") in the AtlasRobotInterface library describes where the Atlas robot is in preparing to run a control session.

The run state determines what the robot is currently capable of doing:

- Some physical state characteristics the robot has; e.g., hydraulic pump is off, starting, on, or stopping.
- Whether control inputs can be applied.

There are four run states in the AtlasRobotInterface library:

- **Idle**: Robot is physically dormant; there is no hydraulic pressure present, though its electrical systems are in a powered on state and its network connection is active.
- **Start**: Operator has requested that the Robot goes into the **Control** state, for exercising joints and reading sensors. This commonly includes turning on the hydraulic pump.
- **Control**: Robot is actively running. Operator control is enabled. The joints that can be controlled are determined by the robot's Behavior.
- **Stop**: Operator has requested the robot return to the **Idle** state. This will include turning off the hydraulic pump if it is on.

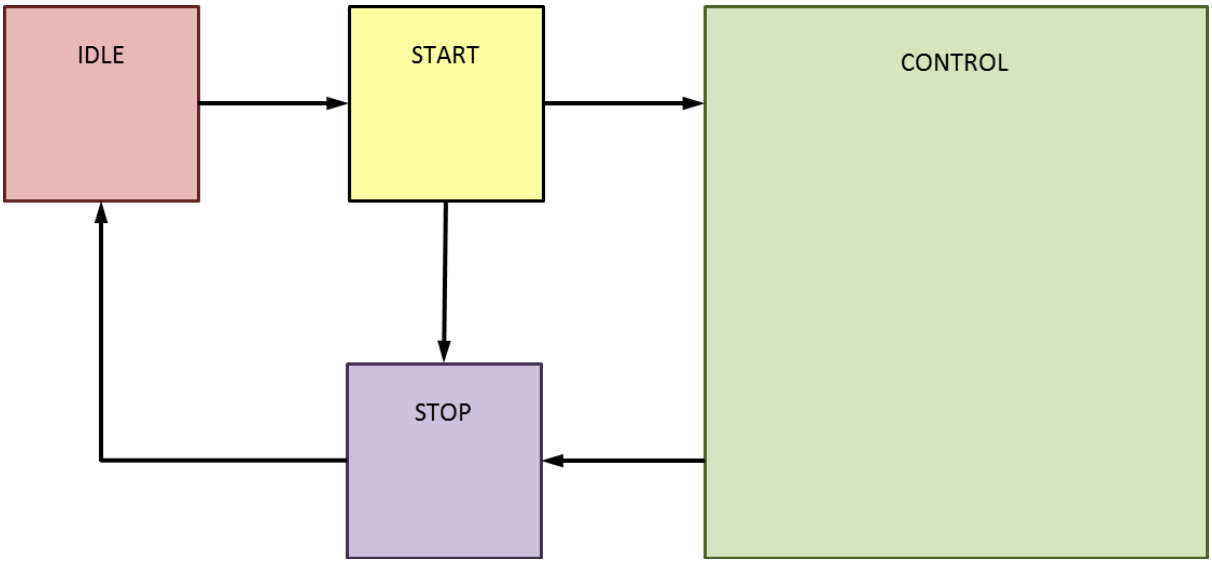


Figure 1: Robot state transition diagram

In the API states are identified by enumerations that are the state name, in all caps, prepended with "RUN_STATE_". e.g., `RUN_STATE_IDLE`.

2.2 States In Detail

Changes to some states will be initiated by API function calls, changes to others may occur automatically.

2.2.1 Idle

The robot always begins in the **Idle** state, and under normal operation should end in the **Idle** state.

In the **Idle** state the robot is physically dormant. There is no hydraulic pressure to drive robot joints. The robot electrical systems should be on and the network connection available, so the Field Computer can connect to the robot. The Field Computer can read sensor values from the robot in the **Idle** state.

The robot will go from **Idle** to **Start** when the API function `start()` is called.

2.2.2 Start

The **Start** state is an intermediate state between **Idle** and **Control**.

Hydraulic pressure may be requested in the API function `start()`, as off or on. If hydraulic pressure has been requested, changing from **Start** to **Control** will happen automatically if the hydraulic pump starts correctly. It may take some time for the hydraulic pump to achieve full pressure.

Start may also change to **Stop** if the pump doesn't start for some reason.

The robot can enter the **Control** state with no hydraulic pressure. If no hydraulic pressure has been requested, the transition from **Idle** to **Control** should happen very quickly.

The robot will return to **Idle** if the API function `stop()` is called.

2.2.3 Control

In the **Control** state the robot is actively running. This is the only state in which Operator joint inputs can be applied.

While in **Control**, the robot will have a number of behaviors available to it. Depending on the robot's current behavior, the Operator may have control of some or all joints. Behaviors are described in the next chapter.

The robot will change from **Control** to **Stop** if the API function `stop()`is called, or if a *critical fault* occurs. Examples of critical faults are if certain components on the robot become too hot, or if communications between the Field Computer and the robot are interrupted for too long.

NOTE: Once the robot has entered the **Control** state, input must be sent to it at a steady, high rate. If input is interrupted for even a short period, the robot will fault and the run will stop automatically. The timeout period is around 0.25 seconds.

2.2.4 Stop

The **Stop** state is an intermediate state between **Control** and **Idle**.

If the hydraulic pump is on, it will be turned off. The robot will remain electrically on, and the network connection to the robot will remain open.

The robot will always leave the **Control** via the **Stop** state, either voluntarily due to an API call, or automatically if a critical fault is detected. Once in **Stop**, no more control settings will be accepted. If the robot leaves **Control** from the **Freeze** behavior, the robot joints will remain in their current frozen positions. From any other behavior the robot joints will be controlled by a "soft freeze" algorithm, which will attempt to position joints to reduce robot damage from a fall.

The robot will automatically transition from **Stop** to **Idle** when appropriate. (e.g., hydraulic pressure has dropped to 0.)

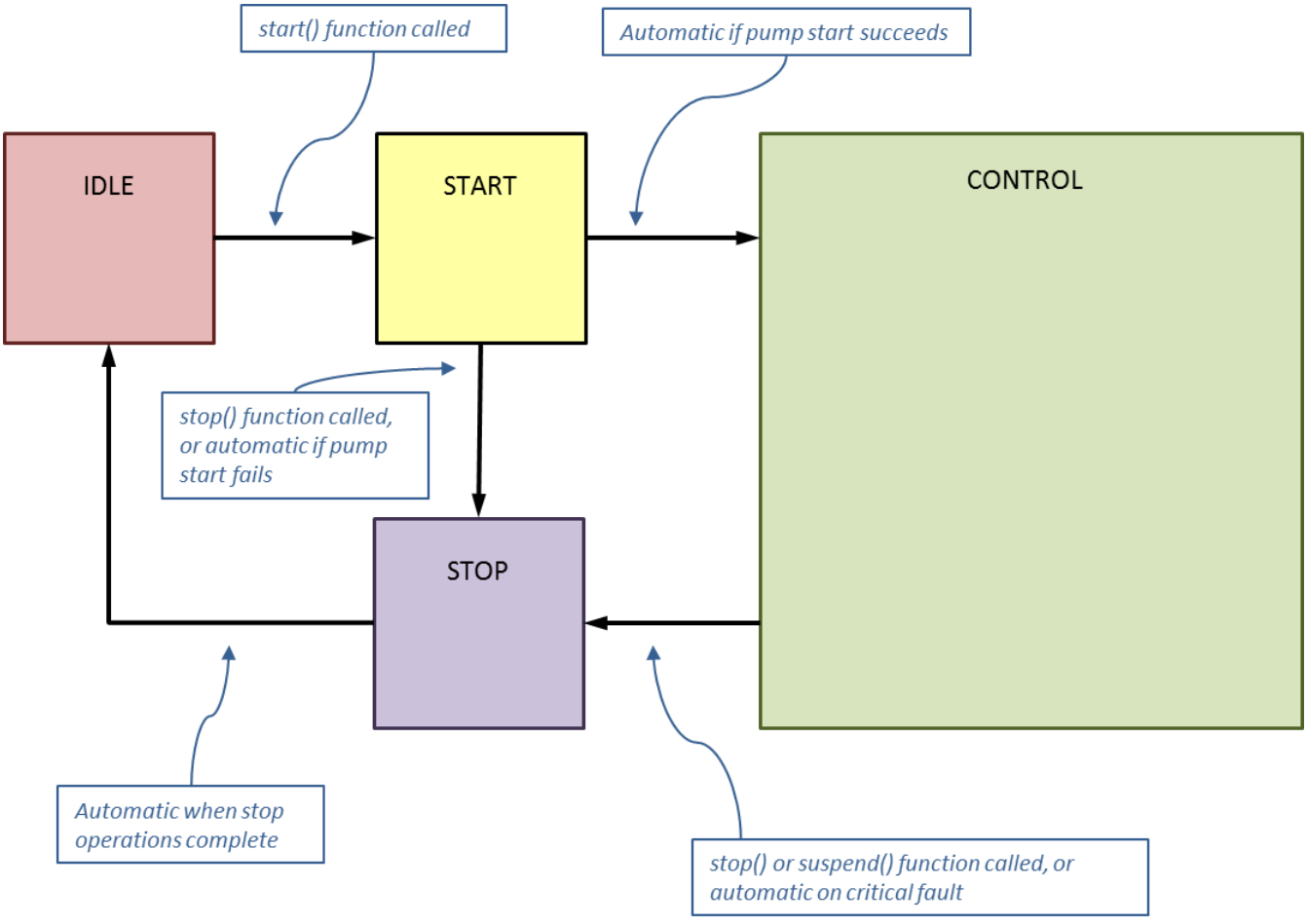


Figure 2: Robot state transition diagram, with transition triggers

3 Robot Behaviors

A "Behavior" in the AtlasRobotInterface library is a mode the Atlas robot can be in while in the **Control** state that controls the setpoints and gains of the robot's joints. The idea behind behaviors is that Operators will be able to enter a behavior to do something, such as walk to a point, and then switch to a different behavior to do something else, such as stand and manipulate an object. The behaviors mentioned here, **Walk** and **Manipulate**, will handle the balance and joint movement to produce the walk and standing motions.

There are nine behaviors (in addition to "none") in the AtlasRobotInterface library:

- **None:** Robot currently not in any behavior; software freeze, controls disabled.
1. **Freeze**: All joints frozen in place.
2. **StandPrep**: Initialization behavior for entering **Stand**.
3. **Stand**: Statically stable stand.
4. **Walk**: Dynamically stable walk using Operator-specified approximate step locations.
5. **Step**: Slow walk with Operator-specified precise step locations, statically stable between steps.
6. **Manipulate**: Statically stable stand, but with upper body joints available for Operator control to enable tool use and environment interaction.
7. **User**: All joints controlled by Operator.
8. **Calibrate**: Robot goes through various motions to update calibration values.
9. **Soft Stop**: Robot tucks its arms and legs while applying soft position gains. Used when robot is falling.

StandPrep, **Stand**, **Walk**, **Step**, and **Manipulate**, are referred to as *auto-control* behaviors (or sometimes as *built-in* behaviors). They are behaviors in which the robot (Boston Dynamics control) takes control over some or all joints.

The **User** behavior provides the most control over the robot; *all* joints are controlled by the Operator. In this behavior maintaining balance is up to the Operator.

Calibrate, like the auto-control behaviors, also takes over control of the entire robot. But calibrate is used only for calibrating various settings on the robot and is not intended to provide users with stable movement.

Soft Stop is also an auto-control behavior. It simply tries to tuck the arms and legs while applying soft position gains to the joints to minimize damage to the robot when falling. Users are encouraged to detect falls and call Soft Stop. However, users are also encouraged to write their own versions of soft stop and apply more advanced damage-avoidance behavior during falls. Note that the built-in behaviors do not select Soft Stop when falling.

In the API behaviors are identified by enumerations that are the behavior name, in all caps, prepended with "BEHAVIOR_". e.g., `BEHAVIOR_FREEZE`.

3.1 Joint Control

The current behavior determines whether the Operator or the robot (auto-control) sets each joint's set-points, gains, etc. In **Freeze** and **Soft Stop**, no joints are under Operator control. In the **User** behavior, all joints are under Operator control. In **StandPrep**, **Stand**, **Walk**, **Step**, and **Manipulate** behaviors,

most upper-body joints are under Operator control, while the legs and overall balance are auto-controlled by the robot.

3.2 Changing Behaviors

Changes between auto-control behaviors must go through **Stand**. The robot can almost always be immediately switched into the **User**, **Freeze,** and **Soft Stop** behaviors. (The **Calibrate** behavior can only switch to **Freeze**.)

Changes in behavior can happen either by request via function call or automatically. An example of a requested change would be calling the API function `set_desired_behavior()` from `BEHAVIOR_STAND` to `BEHAVIOR_WALK`. On the other hand, the **Walk** auto-control behavior will revert automatically to **Stand** if the **Walk** behavior doesn't have its user-supplied steps properly specified.

Requested changes in behavior may not always succeed if constraints needed for the change aren't met. For example, the robot will not change to the **Walk** behavior if its current foot placements have the legs crossed or the robot has fallen.

3.3 Behavior Transitions

It may take some time for a change in behavior to take place, as the robot gets into a posture that makes the change possible. For example, to go from **Stand** to **Walk**, the robot must shift weight to one foot while it prepares to lift the other for the first step.

Transitions into **Freeze, Soft Stop,** and **User** will happen immediately.

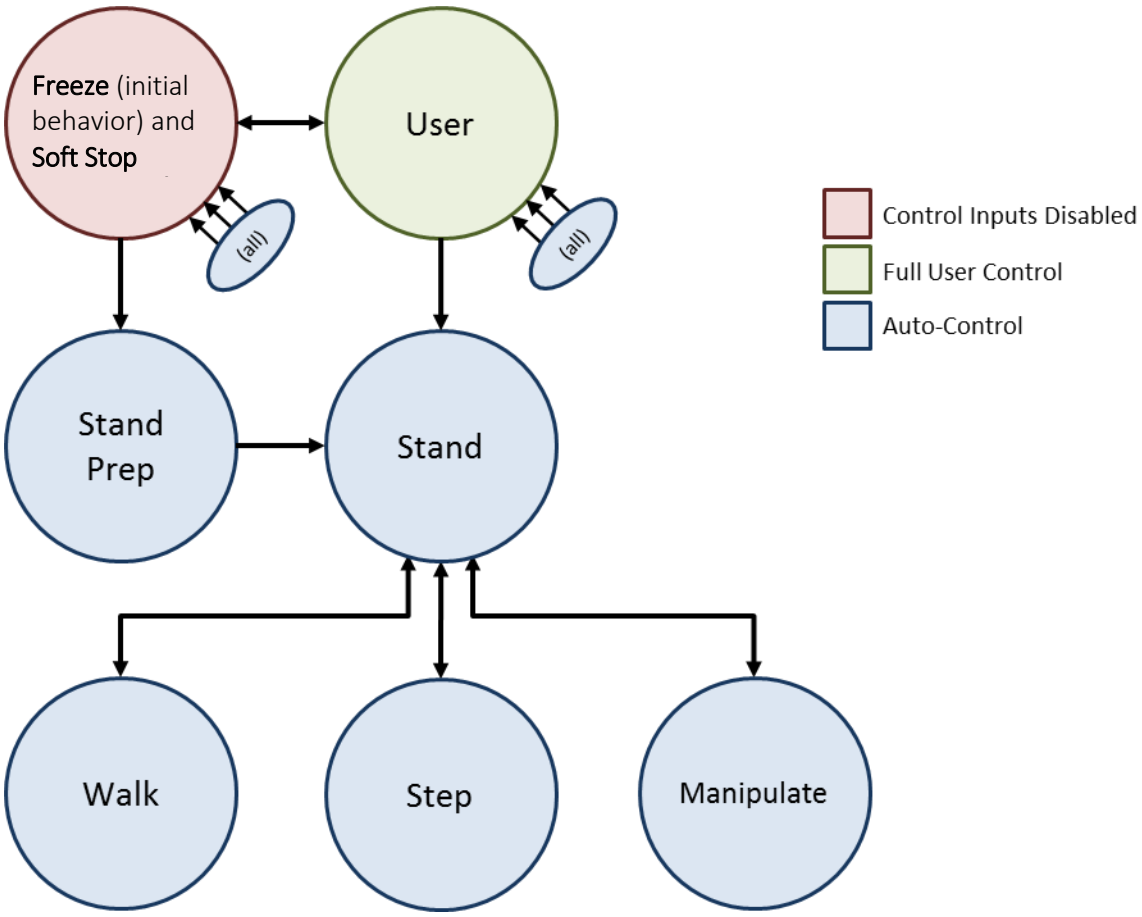


Figure 3: Behavior transition diagram

3.4 Behaviors In Detail

3.4.1 Freeze

Under the **Freeze** behavior, *no* joint setpoints or gains are under Operator control. Joint setpoints and gains will be set to and will retain their current values.

The **Freeze** behavior is the initial behavior the robot will be in when the robot state changes from **Start** to **Control**.

The **Freeze** behavior can be entered immediately from all other behaviors. Note, though, that to get out of **Freeze** behavior and back into **Stand**, the robot will have to go through the **StandPrep** behavior. This usually means the robot is wrangled off the ground until the **StandPrep** has completed.

3.4.2 StandPrep

StandPrep is a behavior typically used only as the robot is starting. It assumes that the robot is being held slightly off the ground. **StandPrep** will attempt to move the robot joints into positions that, when the robot is put on the ground, the robot will be able to successfully enter the **Stand** behavior. Once the

StandPrep behavior has been exited, it cannot be reentered except through the **Freeze** behavior, after the robot has been wrangled from the ground.

The **StandPrep** behavior is not needed if the robot is immediately put into the **User** behavior from **Freeze** on control start up.

3.4.3 Stand

In the **Stand** behavior the robot attempts to stand up straight, in a statically stable posture. The **Stand** behavior will attempt to adjust to pushes and other applied forces as necessary to keep the robot balanced.

Stand is the "hub" behavior, in that almost all changes in behaviors must go through **Stand**. For example, the robot can't get to the **Step** behavior from **Walk** until it has successfully transitioned through **Stand**.

The **User** and **Freeze** behaviors can be entered without going through the **Stand** behavior.

To get back into **Stand** from **User**, a number of constraints need to be met: the robot must be mostly upright, with no appreciable linear or angular velocity.

To get into **Stand** from **Freeze**, the robot must go through the **StandPrep** behavior. The same constraints for successful a transition into **Stand** from **User** apply to making a transition from **Freeze** to **StandPrep**; i.e., robot mostly upright, no linear or angular velocity.

3.4.4 Manipulate

The **Manipulate** behavior is a hybrid of the **Stand** and **User** behaviors. The lower body is under robot auto-control, the upper body under Operator control.

The robot will balance with two feet on the ground while the Operator sets reference pelvis orientation, height, and position relative to the center of the feet. Operators will directly control movement of the back and upper body. Behavior will be robust to a known added mass, such as a tool.

Because the **Manipulate** behavior may push the limits of the robot's ability to balance, it is strongly suggested that the placement of the feet be in the most stable configuration possible. If necessary, a step or two in the Step behavior should be used to put the feet into good positions. Care must be taken when driving Operator controlled joints. Extreme poses or quick moves can overwhelm the behavior's ability to keep the robot balanced.

More information on controlling the **Manipulate** behavior is given in the Controlling the Robot chapter.

3.4.5 Walk

Walk is a dynamic walking behavior that requires inputs for at least the next three footsteps. Realized foot positions will be subject to kinematic and balance constraints. The behavior will be robust to some ground height and slope disturbances.

Desired step locations should be set *before* entering the **Walk** behavior. The robot will transition back to **Stand** if at any point not enough steps are provided.

More information on controlling the **Walk** behavior is given in the Controlling the Robot chapter.

There are some constraints on foot placements, as shown in the following figure.

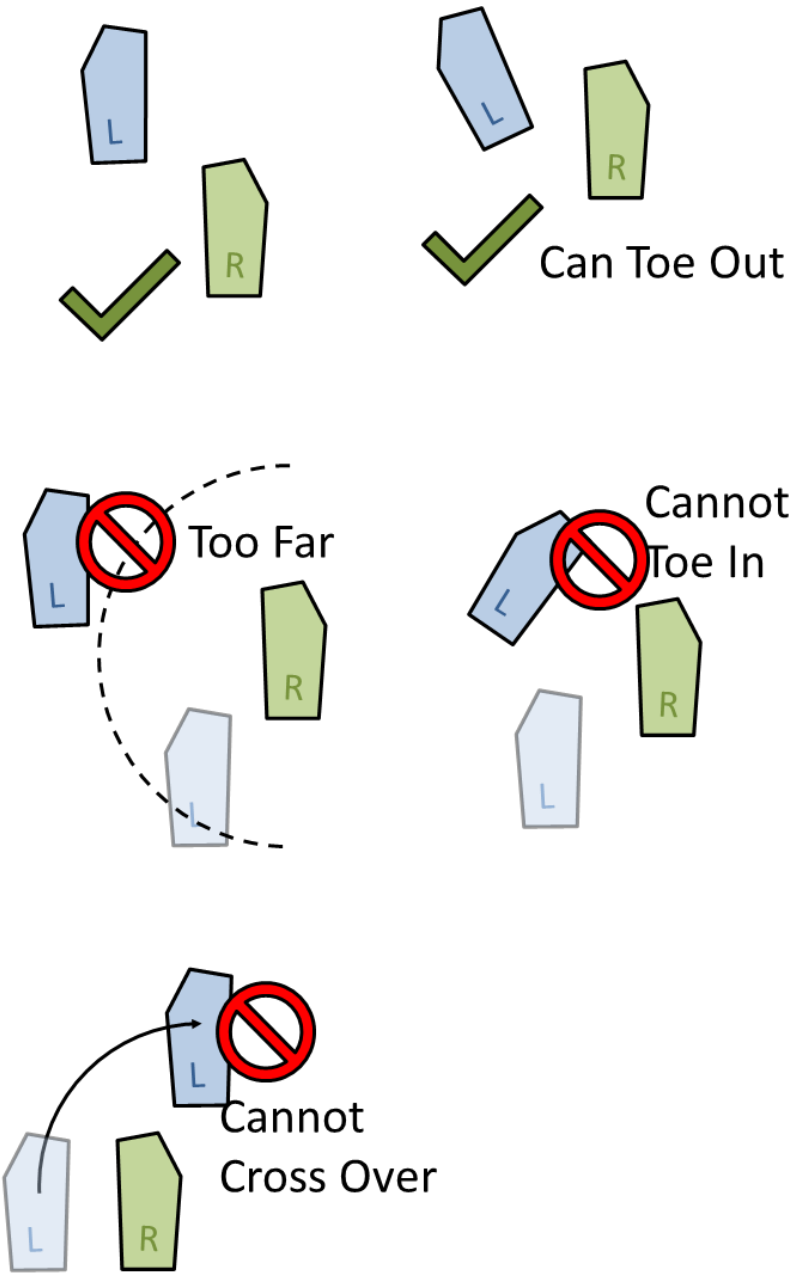


Figure 4: Foot placement constraints

3.4.6 Step

Step is a quasi-static walking behavior that will allow Operators to realize better foot placement accuracy than **Walk**. While foot placement should be more accurate, desired foot placements are still subject to kinematic constraints.

As with **Walk**, the first desired step location should be set *before* entering the **Step** behavior.

If the next desired step is not set before the swing foot comes down, the behavior will make an automatic transition to **Stand**. The desired behavior will have to be re-set to **Step** to re-enter the **Step** behavior.

During the transition from **Step** to **Stand** the **Step** controller will move the robot to have roughly equal support on each foot.

More information on controlling the **Step** behavior is given in the Controlling the Robot chapter.

3.4.7 User

Under the **User** behavior, all joint setpoints and gains are under Operator control. Desired forces computed by the behavior for each joint will be zero.

The **User** behavior can be entered immediately from all other behaviors. Note, though, that to get out of **User** behavior, back into **Stand**, the Operator will have to drive the robot into a posture similar to **Stand**; i.e., the robot must be mostly upright, statically stable, with no appreciable linear or angular velocity.

As with all other behaviors, an immediate transition to **Freeze** can be made from **User**.

3.4.8 Calibrate

Calibrate is a specialized behavior for running a calibration routine on the robot. Currently, it attempts to calibrate the servo-valve null biases for all actuators and the strain-gauge offsets of the feet force sensors. The robot should be wrangled off the ground with nothing touching the robot for this behavior to function properly.

Calibrate can only be entered from the **Freeze** behavior, by calling the `calibrate()` API function. The behavior will automatically transition back to **Freeze** when the calibration is completed. **Freeze** can be entered manually at any time. **Calibrate** cannot go directly to any other behaviors.

3.4.9 Soft Stop

Like **Freeze**, under the **Soft Stop** behavior *no* joint setpoints or gains are under Operator control. This behavior tucks the arms and legs while applying soft position gains to the joints to minimize damage to the robot when falling. The target "tuck" pose is cubically transitioned to over a 1 second period from the initial pose of the robot when the Soft Stop is requested.

The **Soft Stop** behavior can be entered immediately from all other behaviors. Note, though, that to get out of **Soft Stop** behavior and back into **Stand**, the robot will have to go through the **StandPrep** behavior. This usually means the robot is wrangled off the ground until the **StandPrep** has completed.

Users are encouraged to detect falls and call **Soft Stop**. However, users are also encouraged to write their own versions of **Soft Stop** and apply more advanced damage-avoidance behavior during falls. Note that the built-in behaviors do not select **Soft Stop** when falling.

4 C++ Library

This section gives some high-level information about the AtlasBehaviorLibrary C++ API. For the most complete information, refer to the on-line Atlas Robot Interface Reference, here:

```
$ATLAS_ROBOT_INTERFACE/doc/html/index.html
```

If there are any discrepancies between the information here and that in the on-line reference, the on-line reference should be considered correct.

4.1 Library Binary Information

The AtlasRobotInterface C++ API has the following characteristics:

- C++ shared-object (.so) library
- Compiled with gcc 4.6.3 on Ubuntu 14.04
- 64-bit only

4.2 API Structure

Almost all functions in the API are implemented in the C++ class `AtlasInterface`. Operator programs access a static instance of this class by calling the `AtlasInterface::get_instance()` member function.

This section gives only a quick overview of the functions. Refer to the on-line Atlas Robot Interface Reference for details. If there are any differences between this information and the on-line reference, the on-line reference should be considered correct.

The AtlasInterface class has the following categories of functions:

- **Network Functions**: Open and close network connection to robot
- **Run State Functions**: Tell robot to start and stop a control session
- **Behavior Functions**: Set desired behaviors
- **Control Data Functions**: Send and receive control data to and from robot
- **Utility Functions:** Miscellaneous support functions

4.2.1 Network Functions

`open_net_connection_to_robot()`

Opens a network connection to the robot. The connection uses the UDP protocol.

`close_net_connection_to_robot()`

Closes an open network connection to the robot.

`net_connection_open()`

Query whether net connection is open.

`get_robot_ip_address()`

Returns the IP address or hostname in use for the robot. The network connection must be open.

4.2.2 Run State Functions

`start()`

Performs the necessary steps begin a new control session, moving the robot from `RUN_STATE_IDLE`, through `RUN_STATE_START`, to `RUN_STATE_CONTROL`. If necessary this includes attempting to start the hydraulic pump.

`stop()`

Stops the current control session, returning the robot to `RUN_STATE_IDLE`. This includes turning off the hydraulic pump if it was on.

`command_in_progress()`

Query whether a command function is running.

4.2.3 Behavior Functions

`set_desired_behavior()`

Sets the desired behavior of the robot.

`get_desired_behavior()`

Gets the desired behavior of the robot.

4.2.4 Control Data Functions

More specific information on using the `AtlasControlDataToRobot`, `AtlasControlDataFromRobot`, and `AtlasExtendedDataToRobot` data structures for controlling the robot is given in other sections.

`send_control_data_to_robot()`

Sends a frame of control data to the robot, using the `AtlasControlDataToRobot` structure.

`new_control_data_from_robot_available()`

Returns whether new control data from robot is available, and the `get_control_data_from_robot()` function should be called.

`send_ext_data_to_robot()`

Sends a frame of extended data to the robot, using the `AtlasExtendedDataToRobot` structure.

`get_control_data_from_robot()`

Retrieves the most recent control data from robot in an `AtlasControlDataFromRobot` structure.

`clear_faults()`

Clear fault status flags.

`download_robot_log_file()`

Download log file of most recent run from robot. These log files will be essential for Boston Dynamics trouble-shooting of robot problems. The utility script `atlas_log_downloader.py` can also be used to download a log file.

4.2.5 Utility Functions

```
get_robot_software_version()
get_error_code_text()
get_run_state_as_string()
get_status_flag_as_string()
link_name_from_link_id()
link_id_from_link_name()
joint_name_from_joint_id()
joint_id_from_joint_name()
behavior_name_from_behavior()
behavior_from_behavior_name()
```

The purpose of each of these functions should be relatively clear from the function name.

4.3 Robot Control Data

4.3.1 Communication Structures

Most of the data for controlling the robot are communicated using the `AtlasControlDataToRobot` and `AtlasControlDataFromRobot` structures. Refer to the on-line documentation for in-depth information about the data members of each structure.

More about the meaning of data members in these structures will be given later.

4.3.2 Error Codes

Most API functions return `AtlasErrorCode` values. *Check these values, and take appropriate action*. The `get_error_code_text()` function will return a text description of each error code.

4.3.3 Robot Faults

During robot operation a fault may occur. Critical faults are conditions that could damage the robot hardware if the robot is running. In these cases the robot will automatically shut down, and an `AtlRobotState` fault flag is raised in `AtlasControlDataFromRobot::robot_status_flags`.

5 Controlling the Robot

The top-level data structure for sending data to the robot is `AtlasControlDataToRobot`. In example discussions, it will be referred to as "`data_to_robot`".

The top-level data structure for receiving data from the robot is `AtlasControlDataFromRobot`. In example discussions, it will be referred to as "`data_from_robot`".

5.1 Overview of Control Concepts

5.1.1 Joint Coordinates

The joint coordinate convention used by the Atlas robot is relative to a canonical zero pose (see Figure 5: Canonical zero poses with axis aligned coordinate system). Each joint is signed according to the principle axis to which it is aligned in the zero pose. The principle axes are defined with the Z axis pointing up and the X axis pointing forward. For easy reference, each joint name denotes the axis with which a joint is associated. For example, the joint name for the right knee, `JOINT_R_LEG_KNY`, indicates that, in the zero pose, a counter-clockwise rotation of the knee about the Y axis corresponds to a positive rotation.

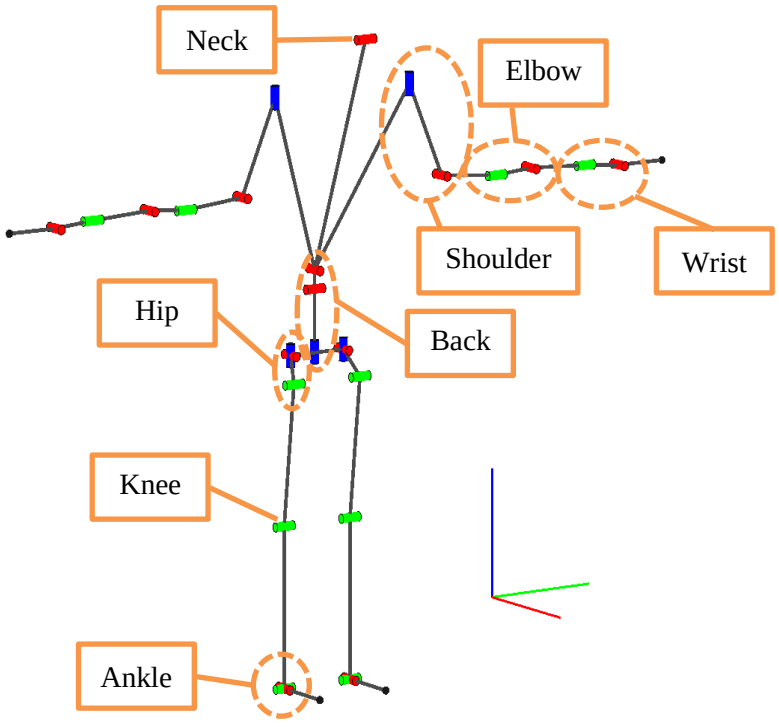


Figure 5: Canonical zero poses with axis aligned coordinate system

5.1.2 Pump Control

Atlas is equipped with an on-board power system that is capable of power-autonomous operation when equipped with a battery. By default the pump controller aims to produce the pressure specified by the `m_desired_pump_psi` value in `Atlas3PacketControlDataToRobot`, rpm set-points are ignored. Alternatively, users can supply pump control parameters via the `pump_params` member of `data_to_robot`. These parameters are active only when the `m_enable_automatic_pump_control` value in `Atlas3PacketControlDataToRobot` is set to 1.

The system is designed to do simple Stand, Step and Walk behaviors with a supply pressure of 1500 psi. Certain maneuvers may require a higher pressure set-point. Most arm motions will require pressure set-points above 2000 psi. Note that the maximum pressure achievable will be flow dependent.

5.1.3 Joint Actuation

All joints (except back Z and neck Y) are actuated with a linear hydraulic actuator controlled by a high performance, electrohydraulic servo valve. The type of the transmission between the actuator and the joint varies. All arm joints as well as the back Z and neck Y joint have a linear transmission. The remaining joints use a variety of linkage types that result in nonlinear transmissions. The mapping between actuators and joints is one-to-one with the exception of the ankle joints, which share two coupled actuators between two joints (see Figure 6). Hydraulic actuators are asymmetric in their force output (i.e., they can push harder than they can pull). Due to this and the nonlinear transmission ratios, the maximum joint torque output will vary with both joint position and direction.

Figure 6: Ankle joint featuring a coupled transmission between two actuators and two intersecting joints

5.1.4 Joint Sensing

The sensing of both positions and forces are done in the actuator coordinate space and then automatically converted to the joint coordinate space. Force sensing is performed by measuring a hydraulic pressure differential in the actuator.

5.1.5 Joint Servo Controllers

The hydraulic servo valves are controlled by on-board servo valve controllers that read sensor data and update the valve commands a rate of 1kHz. In certain behavior modes (e.g. `BEHAVIOR_USER`), users may directly specify set-points and gains, which are used by the servo valve controllers to compute valve

commands, *i*, according to the following equations. Symbols in the above equations are documented in Table 1.

$$i = \frac{i_{max}}{Q_{max}}\left(c_q + c_{qd} + c_f + ff_{const}\right)$$

$$c_q = k_{qp}(q_d - q) + k_{qi}\sum_n (q_d - q)dt$$

$$c_{qd} = k_{qd}(\dot{q}_d - \dot{q}) + ff_{qd}\dot{q} + ff_{qd_d}\dot{q}_d$$

$$c_f = k_{fp}(f_d - f) + ff_{f_d}f_d$$

Q_{max} is the flow through the valve based on a simple orifice model. It is computed based on the valve's specified control flow Q_c, which is achieved by shorting the output ports of the valve with a control supply pressure delta of Δp_c. Values for Q_c and Δp_c are provided in the Table 2, below. p_{supply} and p_{return} are the measured supply and return manifold pressures. These are filtered with a simple two-pole low-pass filter (cutoff frequency = 2Hz). Note that p_{supply} is restricted to be between 800 psi and 3500 psi and p_{return} is restricted to be between 1 psi and 250 psi.

$$Q_{max} = Q_c\sqrt{\frac{p_{supply} - p_{return}}{\Delta p_c}}$$

Table 1: valve command equation symbols and units

Symbol	Description	Units
i	Valve command	A* or V**
i_{max}	Maximum valve command magnitude	A* or V**
Q_{max}	Maximum valve flow for the given valve, p_{supply} and p_{return}	cis***
Q_c	Test flow for a test pressure of Δp_c	cis
Δp_c	Test pressure delta	psi
p_{supply}	Measured supply manifold pressure	psi
p_{return}	Measured return manifold pressure	psi
dt	Time step, nominally 1 ms	s
q_d	Desired joint position (AtlasJointDesired::q_d)	rad
q	Measured joint position (AtlasJointState::q)	rad
$\dot{q}_d$	Desired joint velocity (AtlasJointDesired::qd_d)	rad/s
$\dot{q}$	Measured joint velocity (AtlasJointState::qd)	rad/s
f_d	Desired joint force (AtlasJointDesired::f_d)	Nm
f	Measured joint force (AtlasJointState::f)	Nm
ff_{const}	Feedforward command (AtlasJointControlParams::ff_const)	cis
k_{qp}	Gain on position error (AtlasJointControlParams::k_q_p)	cis / rad
k_{qi}	Gain on integral of position error (AtlasJointControlParams::k_q_i)	cis / (s rad)
k_{qd}	Gain on velocity error (AtlasJointControlParams::k_qd_p)	cis / (rad/s)
ff_{qd}	Gain on measured velocity (AtlasJointControlParams::ff_qd)	cis / (rad/s)
ff_{qd_d}	Gain on desired velocity (AtlasJointControlParams::ff_qd_d)	cis / (rad/s)
k_{fp}	Gain on force error (AtlasJointControlParams::k_f_p)	cis / (Nm)
ff_{f_d}	Gain on desired force (AtlasJointControlParams::ff_f_d)	cis / (Nm)

* Model1 valves, ** Model2 valves, *** cubic inches per second

Table 2: Nominal valve flow

Valve	Joints	Δp_c (psi)	Q_c (cis)
Model1 Valve	bkx, bky, bkz, hpz, aky, akx, shz, shx, ely, elx, wry, wrx	3000	8
Model2 Valve	kny, hpy	3000	23

The current value of integral error servo term (i.e., the summation in the equation above) may be observed in the `data_to_robot.jfeed` structure. The maximum absolute value of this term may be clamped by setting the `AtlasJointControlParams::qerr_int_abs_max` value. The integral error can be zeroed by setting this value to zero for one or more time steps.

5.2 User-specified servo filters

In the servo equation above, an optional set of user-specified difference filters may be used instead of the default, internally filtered values. Parameters related to servo filters are accessible via the `AtlasExtendedDataToRobot` structure.

To enable user-specified servo filters on the joint angles, q, set the `use_q_filt` flag to 1.

To enable user-specified servo filters on the joint velocities, $\dot{q}$, set the `use_qd_filt` flag to 1.

To enable user-specified servo filters on the joint torque, f, and torque error, $f_{err} = (f_d - f)$, set the `use_f_filt` flag to 1.

All user-specified filters are 2nd order "biquad" difference filters, except for the filters on torque error which are 3rd order. Filters are of the form:

$$u[n] = x[n] * b(0) + x[n-1] * b(1) + x[n-2] * b(2) - u[n-1] * a(0) - u[n-2] * a(1),$$

(2nd order) and

$$u[n] = x[n] * b(0) + x[n-1] * b(1) + x[n-2] * b(2) + x[n-3] * b(3) - u[n-1] * a(0) - u[n-2] * a(1) - u[n-3] * a(2),$$

(3rd order), where a and *b* are coefficient vectors and u[i] and x[i] are the filtered output and unfiltered input at time step i. Coefficient vectors are specified via the `AtlasDiffFilt2` and `AtlasDiffFilt3` data structures for each filtered value.

The output of the user-specified filters may be observed in the `data_to_robot.jfeed` data member. Note that the input to these filters are the raw sensor data (or in the case of qd, raw finite differences) and they will have more variance than the internally filtered values reported in the `data_to_robot.j` data member.

5.3 User Behavior and Upper body control

Full control of all joint servos is only available in the **User** behavior mode. During other behaviors, the behavior takes control over the lower body. The neck remains under user control during all behaviors except for **Freeze** and **Stand Prep**. When in the **Manipulate** behavior the user always retains full control of the upper body. In the **Stand**, **Step**, and **Walk** behaviors, setting the `data_to_robot.ful_ub_cntrl` to 1 enables full control of the upper body, otherwise the behavior takes over. To ensure proper operation, it is recommended that this flag get set once before entering the **Stand** state and remain constant during the subsequent behaviors and transitions.

6 Working with Auto-Control Behaviors

Some auto-control behaviors take parameter inputs to modify the control outputs they will produce. The behaviors with adjustable parameters are **Walk**, **Step**, and **Manipulate**.

There are currently no behavior specific parameters for the **Stand** and **StandPrep** behaviors.

Behaviors may be informed of additional mass added to the wrists by specifying the `AtlasHandMass` data structure in `data_to_robot.hand_mass[HAND_LEFT|HAND_RIGHT]`.

6.1 Controlling Manipulate

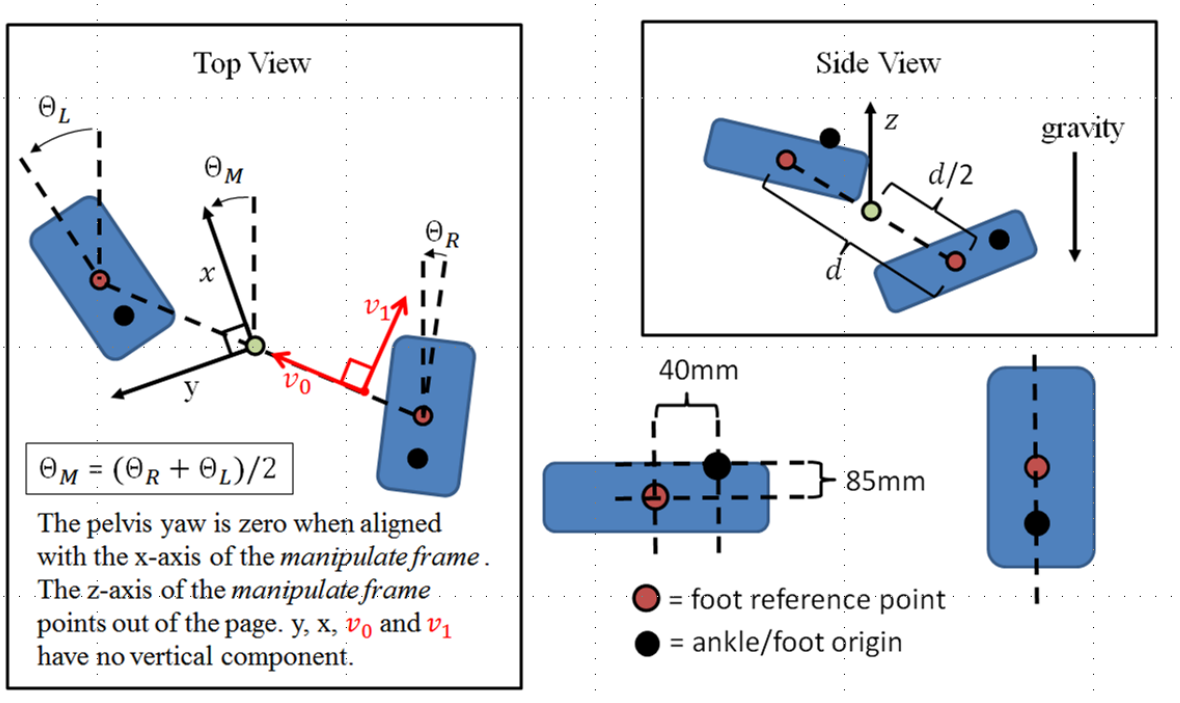
This is preliminary information; final control of this behavior may be slightly different.

Parameters for affecting the **Manipulate** behavior use the `AtlasBehaviorManipulateParams` data structure, accessible as `data_to_robot.manipulate_params`.

`data_to_robot.manipulate_params.use_demo_mode` specifies whether to run **Manipulate** in a demonstration mode that subsumes control of the upper body degrees of freedom.

`data_to_robot.manipulate_params.use_desired` specifies whether the pelvis servo params should be used. A value of 1 means use them, a value of 0 means don't use them.

The `AtlasBehaviorPelvisServoParams` structure, accessible as `data_to_robot.manipulate_params.desired`, is used to specify desired set-points to which the robot attempts to servo the pelvis or center of mass (COM). Servoing is performed relative to the *manipulate frame*, which is defined as follows.



Let the foot reference point be a point located in the frame of the foot with local coordinates (x,y,z) = (0.04, 0.0, -0.085), then the global position of the manipulate frame is a point midway between the global position of the right and left foot reference points. The z-axis of the manipulate frame is aligned with gravity and the yaw of the frame is the mean yaw of the left and right feet.

`data_to_robot.manipulate_params.desired.pelvis_height` is the desired height of the origin of the pelvis in the *manipulate frame*.

`data_to_robot.manipulate_params.desired.pelvis_[yaw|roll|pitch]` are the desired yaw, roll, and pitch angles of the pelvis in the *manipulate frame*. The concatenation order of the Euler angles is *yaw*roll*pitch*.

`data_to_robot.manipulate_params.desired.com_v0` is the desired horizontal offset, in meters, of the COM in the direction of the vector between the two feet. A value of zero centers the COM over the origin of the *manipulate frame*.

`data_to_robot.manipulate_params.desired.com_v1` is the desired horizontal offset, in meters, of the COM in the direction of the vector normal to the vector between the two feet. A value of zero centers the COM over the origin of the *manipulate frame*.

The parameters listed above may be clamped to admissible values, which are reported in `data_from_robot.walk_feedback.clamped`. Even though the values are clamped, the behavior may not succeed in servoing to extreme postures outside the kinematic range of the robot. If such postures are attempted, the robot may fall. It is up to the user to ensure that the robot's posture avoids kinematic singularities and joint limits while in the **Manipulate** behavior.

Additional feedback in `data_from_robot.walk_feedback.internal_desired` and `data_from_robot.walk_feedback.internal_sensed` report the current post-processed desired values and sensed/estimated values.

6.2 Controlling Walk

This is preliminary information; final control of Walk may be slightly different.

Parameters for affecting the **Walk** behavior use the `AtlasBehaviorWalkParams` data structure, accessible as `data_to_robot.walk_params`.

The **Walk** behavior needs to have step data in order to know where to place the robot's feet. The data specification includes:

- `step_index`: which step in the Walk this step data is for
- `foot_index`: which foot (0 for left, 1 for right)
- `foot`: parameters describing the foothold
  - `position`: where to put the foot, in Atlas world frame
  - `yaw`: foot orientation yaw component, in Atlas world frame
  - `normal`: ground normal for the step location
- `action`: parameters describing the step motion
  - `step_duration`: how long the step should take
  - `swing_height`: how high to swing the foot

Step data use the `AtlasBehaviorWalkSpec` data structure, accessible as `data_to_robot.walk_params.walk_spec_queue`. Four step data items can be set at any one time.

Step Indices

The first step should have its `step_index` set to 1, the second to 2, etc. To specify that the data in a step data structure is not valid or should be considered "not set", set the `step_index` to -1.

When to Add Steps

The **Walk** behavior in general needs to know not only the next step, but preferably the following three steps beyond that in order to know how to shift weight and orientation when preparing for the coming steps. Three steps may result in less smooth walking. Two or fewer will likely cause an automatic behavior transition to **Stand**. The foot reference point is located at (x, y, z) = (0.04, 0.0, -0.085) in the foot frame.

If too few steps are specified, the `data_from_robot.walk_feedback.status_flags` `WALK_WARNING_INSUFFICIENT_STEP_DATA` status flag will be raised.

The initial steps of a **Walk** should be specified *before the **Walk** behavior is requested*. So to begin, the first four items of step data should be added, then the desired behavior set to **Walk**.

A **Walk** behavior can use fewer than four steps. In this case all steps would be queued before the walk begins. If there will be only one or two steps, consider using the Step behavior.

All steps can be re-specified at any time, but if existing steps are changed – say, the next step moves right instead of left – the **Walk** behavior will likely not work well, potentially even causing a fall. If steps are to be removed from the queue, the removed steps' `step_index` values should be set to -1.

A **Walk** will go through a series of substates, as specified in the `data_from_robot.walk_feedback.status_flags` value. When preparing to lift the foot on the first step, the `WALK_SUBSTATE_SWAYING` flag will be set. During active walking the `WALK_SUBSTATE_STEPPING` flag will be set. When transitioning back to stand, the `WALK_SUBSTATE_CATCHING` flag will be set.

Once a foot has left the ground, that step (position of foot, yaw, etc.) cannot be changed; the **Walk** behavior is committed to that step. When beginning a **Walk**, the step at index 1 is read immediately. The variable `data_from_robot.walk_feedback.next_step_index_needed` should be watched to know when the next step should be queued.

The variable `data_from_robot.walk_feedback.current_step_index` specifies the most recent step that has been completed. So if the current step is index 1, it will have completed step 1, be committed to stepping to step position 2, and the next step index needed will be 3.

The **Walk** behavior will do its best to reach the specified step data, but may have to modify the steps to fit within constraints. The modified data will be put into `data_from_robot.walk_feedback.walk_spec_queue_saturated`. If the steps are heavily modified, the **Walk** route will likely need to be replanned. The values in `walk_spec_queue_saturated` should be compared to `walk_spec_queue` each time a new step is requested. If there's too much error, the route should be replanned.

Consecutive steps should not use the same foot index. i.e., step *n* can't use the same foot as step *n-1*. If this happens, the robot will begin an automatic transition back to **Stand**. Also, the walk feedback `WALK_ERROR_INCONSISTENT_STEPS` status flag will be raised.

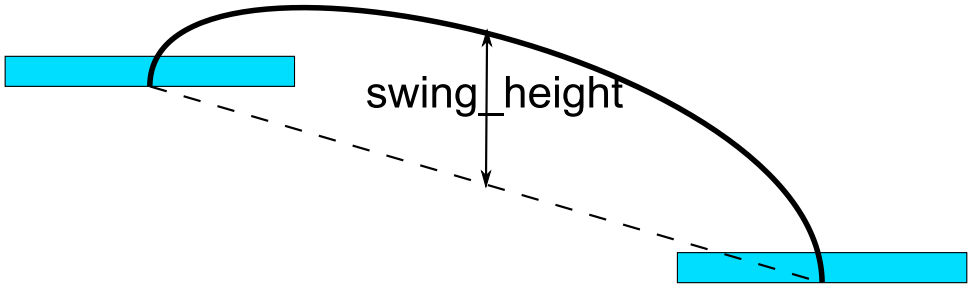


Figure 7: The swing_height adjusts the height of the midpoint of the swing trajectory in Walk

6.3 Controlling Step

As with **Walk**, the first desired step location should be set *before* entering the **Step** behavior.

The **Step** behavior has two phases for each step: a swaying phase to set up the step, and then the actual stepping motion. The behavior will be in either `STEP_SUBSTATE_SWAYING` or `STEP_SUBSTATE_STEPPING` at any given time. The current substate can be checked by reading `control_outputs.step_feedback.m_status` flags.

The **Step** behavior needs to have step data in order to know where to place the robot's feet. The data is specified using the `AtlasBehaviorStepSpec` data structure. The data specification includes:

- `step_index`: which step in the Step this step data is for
- `foot_index`: which foot (0 for left, 1 for right)
- `foot`: parameters describing the foothold
  - `position`: where to put the foot, in Atlas world frame. Note that the foot reference point is located at (x, y, z) = (0.04, 0.0, -0.085) in the foot frame.
  - `yaw`: foot orientation yaw component, in Atlas world frame
  - `normal`: ground normal for the step location
- `action`: parameters describing the step motion
  - `step_duration`: how long the step should take
  - `sway_duration`: how long the motion should spend in double support, setting up the stepping motion
  - `swing_height`: how high to swing the foot, after it has been lifted vertically
  - `lift_height`: how high to lift the foot vertically above the maximum of the start and end positions of the foot. See Figure 8 for how this affects the swing trajectory.
  - `toe_off`: control whether or not the robot will use a toe-off during the sway.
  - `knee_nominal`: set the desired knee angle during the motion, which will affect how crouched the robot walks
  - `max_body_accel`: maximum body acceleration during the sway
  - `max_foot_vel`: maximum foot velocity during the step
  - `sway_end_dist`: Tuning – distance to hold back from the foot center at the end.
  - `step_end_dist`: Tuning – distance to lean into the next step location.

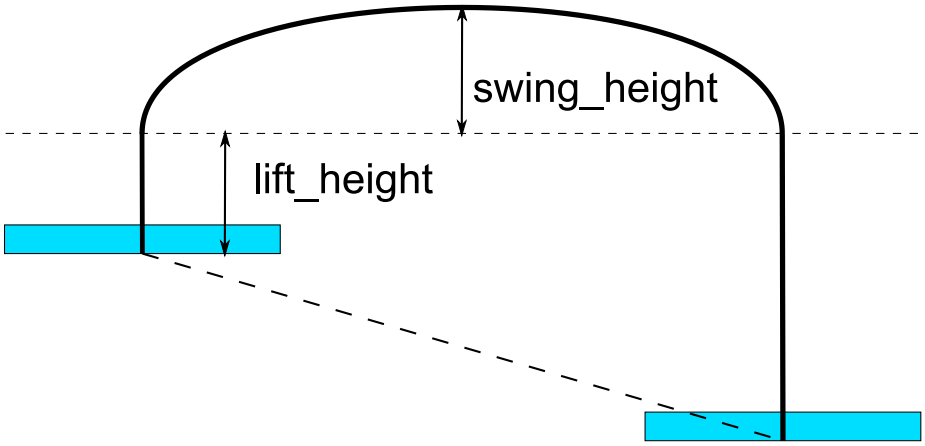


Figure 8: Swing trajectory parameters for the Step behavior

The next desired step should be set while the behavior is in `STEP_SUBSTATE_SWAYING`, or during the previous `STEP_SUBSTATE_STEPPING` phase. Unlike with **Walk**, the same foot can be moved in consecutive steps.

The orientation of the pelvis will be determined automatically based on the step location orientations. The quaternion `data_to_robot.step_params.pelvis_orientation_offset` is used to specify an offset to that orientation applied throughout the walk. It is read at the beginning of each stepping phase and will be fully applied by the end of that phase.

The **Step** behavior will do its best to reach the specified step data, but may have to modify the step to fit within constraints. The modified data will be put into `data_from_robot.step_feedback.desired_step_spec_saturated`.

The `toe_off` value can be used to control the toe-off behavior for each step. When set to `TOE_OFF_ENABLE`, the **Step** behavior will decide whether a toe-off is necessary to aid reachability during the sway. This mode can be changed to `TOE_OFF_DISABLE`, which will keep the feet flat, or `TOE_OFF_FORCE_ENABLE`, which will instruct the **Step** behavior to use a toe-off even if its internal check does not indicate a toe-off is needed. The resulting choice of whether to use a toe-off or not will be indicated in the `data_from_robot.step_feedback.desired_step_spec_saturated` structure with `TOE_OFF_DISABLE` or `TOE_OFF_ENABLE.`

The step indexing works similarly to **Walk**, i.e. the variable `data_from_robot.step_feedback.next_step_index_needed` is the index of the next step data that needs to be specified, and `data_from_robot.step_feedback.current_step_index` is the index of the currently completed step. See Figure 9 for an illustration of step indexing during stepping. The next_step_index_needed variable increments when the robot takes a step and commits to its stepping location. When the robot finishes its step, it sets the current_step_index to that step's index.

If the next desired step is not set before the swing foot comes down, the behavior will make an automatic transition to **Stand**. The desired behavior will have to be re-set to **Step** to re-enter the **Step** behavior.

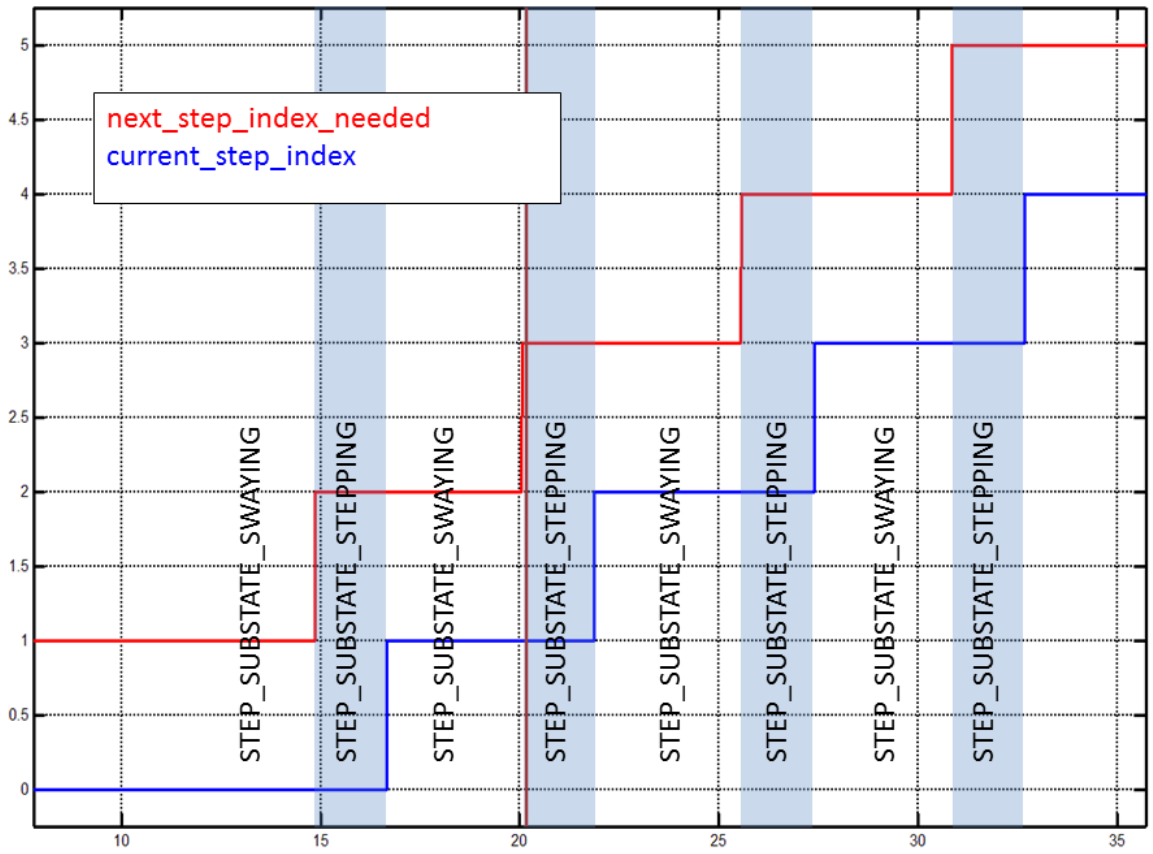


Figure 9: Step indexing

During the transition from **Step** to **Stand**, the **Step** controller will move the robot to have roughly equal support on each foot. It is important that when transitioning out of **Step**, the robot have both feet flat. This can fail if during a sway phase that includes a toe-off, the step is cancelled by either **a)** a transition out of **Step** is requested or **b)** `data_to_robot.step_params.desired_step_spec.step_index` is set to less than `data_from_robot.step_feedback.next_step_index_needed`.

6.3.1 Step Timing

The `sway_duration` value in `AtlasBehaviorStepData` will determine how long `STEP_SUBSTATE_SWAYING` will last. Likewise, `step_duration` specifies how long `STEP_SUBSTATE_STEPPING` will last. Both of these durations can be arbitrarily long, as the robot should be stable at any given time, but there are constraints on how short these durations can be. The minimum duration of the sway will be based on `max_body_accel`, and the minimum step duration will be based on `max_foot_vel`. Increasing `max_body_accel` and/or `max_foot_vel` will allow the robot to move faster, but can adversely affect its stability. The modified durations can be read from `data_from_robot.step_feedback.desired_step_spec_saturated`. The `sway_duration` with be limited at the start of `STEP_SUBSTATE_SWAYING`, and `step_duration` will be limited at the start of `STEP_SUBSTATE_STEPPING`.

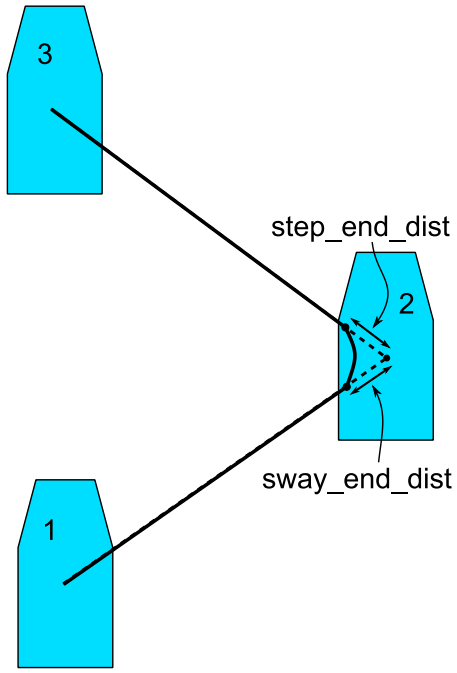
6.3.2 Step Tuning

In practice, certain step motions may be unstable, and always fall in the same way. The following tuning parameters are provided to allow tweaking how the gait performs for a step:

The position at the end of the sway is modified by `sway_end_dist`. This distance pulls the robot back so that it does not sway over as far. Increase this distance if the robot consistently falls to the outside on a particular step.

The position at the end of the step is modified by `step_end_dist`. This is a distance towards the next step location. If the foot touchdown pushes the robot over, increase this parameter.

The default values of these parameters can be read from `data_from_robot.step_feedback`. Ideally these parameters should not be changed, and should only be set for steps which are problematic.



In some situations, required knee strength may exceed the actuator's limit, causing the knee to buckle. There are several tweaks that can lessen the required knee torque: Reducing step size or decreasing `knee_nominal` in order to keep the knee straigher, or moving the center of mass forward by adjusting the upper body pose or leaning forward via the `pelvis_orientation_offset`.

7 Tools

The `tools` subdirectory of the AtlasRobotInterface installation contains some scripts and programs necessary for various operations.

7.1 atlas_log_downloader.py

The `atlas_log_downloader.py` Python script provides a simple user interface for downloading log files from the Atlas robot. It needs three pieces of information:

- `Robot IP Addr`: IP address of the robot; defaults to 192.168.130.103.
- `Directory`: directory to download to; defaults to `$ATLAS_ROBOT_INTERFACE/logs`
- `Duration`: how much time to download, in seconds; defaults to 30

The filename of the log file will be generated automatically, based on date and time.

The script calls a program in the tools directory called `jef-client`, which handles downloading of the files.

The API function `download_robot_log_file()` can also be used to download a log file from within a program.

7.2 jef-client

This program is used by `atlas_log_downloader.py` Python script for downloading log files. Boston Dynamics may request a log file with specific arguments to the `jef-client` program if additional or different data is needed.

8 IP Addresses

This table summarizes the IP addresses assigned to all robot hardware:

Atlas robots	192.168.130.103
MultiSense-SL	192.168.100.101
Left Hand	192.168.120.101
Right Hand	192.168.121.101
Main SA Camera Left	192.168.110.101
Main SA Camera Right	192.168.111.101
Spare to switch	192.168.130.104
Radio	192.168.130.3
Rear SA Camera (mic) / Spare (CPU0)	192.168.101.101
On-robot switch	192.168.130.2

9 ROS

The AtlasRobotInterface library does not implement a ROS node. It is up to the application that uses the library to publish and subscribe data to ROS topics if so desired.

We recommend that the code that controls the Atlas robot be in the same program that reads and writes the data to the robot. In other words, control inputs should not be published onto ROS topics, but made directly into the API. This is to avoid network overhead that could be potentially detrimental to control.

10 Other Vendor Components

The Atlas robot has several components provided by vendors other than Boston Dynamics. Some of these provide a ROS interface, some other software interfaces. This is a quick overview of these components.

10.1 Carnegie Robotics MultiSense-SL

The MultiSense-SL sensor head provides a ROS interface to read sensor data and write commands. Use of its ROS interface is detailed in the MultiSense-SL documentation.

10.2 Sandia Robotic Hands

The Sandia robotic hands provide a ROS interface to read sensor data and write hand commands. Use of its ROS interface is detailed in the Sandia hand documentation.

10.3 iRobot Hands

The iRobot robotic hands provide a ROS interface to read sensor data and write hand commands. Use of its ROS interface is detailed in the iRobot hand documentation.

10.4 Main Situational Awareness Cameras

Data from the Main Situational Awareness Cameras will be available on the two IP addresses listed in the section IP Addresses.

The model numbers of the cameras are in the Atlas Robot Operation and Maintenance Manual, Third Party Hardware chapter. Documentation for the cameras is available on-line.

10.5 Rear Situational Awareness Camera

Data from the read situational camera will be available on the IP address listed in the section IP Addresses.

The model number of the camera is in the Atlas Robot Operation and Maintenance Manual, Third Party Hardware chapter. Documentation for the cameras is available on-line.

11 3rdparty Libraries

openssl	© Eric Young	openssl.org
zlib	© Jean-loup Gailly and Mark Adler	zlib.net
qpOASES	© 2007-2009 Luca Di Gaspero, © 2009 Eric Moyer	www.kuleuven.be/optec/software/qpOASES
ROS		www.ros.org
eigen		eigen.tuxfamily.org/index.php?title=Main_Page
blas		www.netlib.org/blas/
lapack		www.netlib.org/lapack/