MultiSense Sensor
ROS Driver Manual

Contact:

Carnegie Robotics LLC
4501 Hatfield Street
Pittsburgh, PA 15201

http://support.carnegierobotics.com

multisense@carnegierobotics.com

(412) 251-0321

Table of Contents

1. INSTALLATION

1.1. HARDWARE COMPATIBILITY

As shipped from the factory, the MultiSense sensor is configured to communicate on an Ethernet network with a Maximum Transmission Unit (MTU) size of 7200 bytes. On Linux-based systems, the MTU settings of a network interface can be checked by issuing the following command (note that **<*interface_name*>** should be replaced with the name of network interface in question, most likely *eth0*).

```
/sbin/ifconfig <interface_name>
```

The output of this command will include information about the current MTU setting of the network interface. The MTU setting is highlighted in the example output below, which was generated by the command:

```
/sbin/ifconfig eth0
```

```
eth0      Link encap:Ethernet  HWaddr 3c:97:0e:29:85:22
          inet addr:10.66.171.20  Bcast:10.66.171.255  Mask:255.255.255.0
          inet6 addr: fd80::3897:dff:1e29:8522/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:70029406 errors:0 dropped:0 overruns:0 frame:0
          TX packets:30272801 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:69790232908 (64.9 GiB)  TX bytes:35517400326 (33.0 GiB)
          Interrupt:20 Memory:f5200000-f5220000
```

Verify that the network hardware supports the necessary frame size by explicitly setting its MTU to 7200:

```
sudo /sbin/ifconfig <interface_name> mtu 7200
```

where **<*interface_name*>** is again replaced by the name of the network interface in question. An example command would be:

```
sudo /sbin/ifconfig eth0 mtu 7200
```

Check that the change was successful by querying the settings again:

```
/sbin/ifconfig <interface_name>
```

The output should now reflect the updated MTU setting:

```
eth0      Link encap:Ethernet  HWaddr 3c:97:0e:29:85:22
          inet addr:10.66.171.20  Bcast:10.66.171.255  Mask:255.255.255.0
          inet6 addr: fd80::3897:dff:1e29:8522/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:7200  Metric:1
          RX packets:70029406 errors:0 dropped:0 overruns:0 frame:0
          TX packets:30272801 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:69790232908 (64.9 GiB)  TX bytes:35517400326 (33.0 GiB)
          Interrupt:20 Memory:f5200000-f5220000
```

1.2. SOFTWARE COMPATIBILITY

1.2.1. ROS Driver

The *3.1* ROS driver release is designed to be fully backwards compatible with all released firmware versions, back to and including the *2.0* release. If the ROS driver detects an older firmware version, only the appropriate topics and capabilities (including *dynamic_reconfigure* variables) will be presented. In addition the *3.1* ROS driver supports both the rosbuild and catkin build systems.

1.2.2. Firmware

The *3.0_beta* firmware release is designed to be fully backwards compatible with older ROS driver releases, back to and including the *2.0* release. However, older ROS drivers will only be capable of exposing functionality that existed at the time of their release.

1.3. HARDWARE COMPATABILITY

1.3.1. MultiSense Units

The *3.1* ROS driver and documentation support all versions of the Carnegie Robotics MultiSense product line including: MultiSense-SL, MultiSense-S7, MultiSense-S7S, and MultiSense-S21. For more information about the various MultiSense products please visit https://carnegierobotics.com/products/ .

1.4. BUILDING FROM SOURCE

These installation instructions are designed to work with Ubuntu 12.04 (Precise Pangolin) and the ROS Fuerte, Groovy, or Hydro releases. A configuration script (config.sh), located in the root directory of the MultiSense driver installation, is used to switch between the rosbuild and catkin build systems. Running the script with a "–h" option will display the full command line usage.

1.4.1. Fuerte

To get started, add ROS to the machine's apt sources and set up the ROS package-signing keys by following the installation instructions for ROS Fuerte here:

http://www.ros.org/wiki/fuerte/Installation/Ubuntu

Download the full desktop version of ROS Fuerte.

```
sudo apt-get update
sudo apt-get install ros-fuerte-desktop-full
```

Install some tools to work with the source repository:

```
sudo apt-get install mercurial python-rosinstall build-essential cmake
```

Next check out the source code for the MultiSense sensor by executing the following commands:

```
source /opt/ros/fuerte/setup.bash
rosws init <path_to_source> /opt/ros/fuerte/
source <path_to_source>/setup.bash
rosws set multisense --hg https://bitbucket.org/crl/multisense_ros
rosws update multisense
```

where **<*path_to_source*>** should be replaced with the name of the desired local installation directory. If the directory does not exist it will be automatically created.

By default the ROS driver comes configured to build under catkin. To configure the driver to build using rosbuild execute the following commands:

```
<path_to_source>/multisense/config.sh rosbuild
```

Where **<*path_to_source*>** is replaced by the name of the local install directory.

Finally the entire repository should be built by invoking *rosmake*:

```
source <path_to_source>/setup.bash
rosmake multisense
```

where **<*path_to_source*>** is again replaced by the name of the local install directory.

To update the MultiSense ROS driver to the newest Bitbucket release, execute the following commands:

```
source <path_to_source>/setup.bash
rosws update multisense
rosmake multisense --pre-clean
```

1.4.2. Groovy

To get started, add ROS to the machine's apt sources and set up the ROS package-signing keys by following the installation instructions for ROS Groovy here:

http://wiki.ros.org/groovy/Installation/Ubuntu

Install the full desktop version of ROS Groovy

```
sudo apt-get update
sudo apt-get install ros-groovy-desktop-full
```

Install some tools to work with the source repository:

```
sudo apt-get install mercurial python-rosinstall build-essential cmake
```

Next create a catkin workspace and check out the source code for the MultiSense sensor by executing the following commands:

```
mkdir -p <path_to_source>/src
cd <path_to_source>/src
source /opt/ros/groovy/setup.bash
catkin_init_workspace
hg clone https://bitbucket.org/crl/multisense_ros  multisense
```

where <*path_to_source*> should be replaced with the name of the desired local installation directory.

Finally the entire repository should be built by invoking *catkin_make:*

```
cd <path_to_source>
catkin_make
```

where <*path_to_source*> is again replaced by the name of the local install directory.

To update the MultiSense ROS driver to the newest Bitbucket release, execute the following commands:

```
cd <path_to_source>/src/multisense
hg pull
hg up
cd <path_to_source>
rm -r build && rm -r devel
catkin_make
```

1.4.3. Hydro

To get started, add ROS to the machine's apt sources and set up the ROS package-signing keys by following the installation instructions for ROS Groovy here:

http://wiki.ros.org/hydro/Installation/Ubuntu

Install the full desktop version of Hydro

```
sudo apt-get update
sudo apt-get install ros-hydro-desktop-full
```

Install some tools to work with the source repository:

```
sudo apt-get install mercurial python-rosinstall build-essential cmake
```

Next create a catkin workspace and check out the source code for the MultiSense sensor by executing the following commands:

```
mkdir -p <path_to_source>/src
cd <path_to_source>/src
source /opt/ros/hydro/setup.bash
catkin_init_workspace
hg clone https://bitbucket.org/crl/multisense_ros  multisense
```

where <*path_to_source*> should be replaced with the name of the desired local installation directory.

Finally the entire repository should be built by invoking *catkin_make:*

```
cd <path_to_source>
catkin_make
```

where <*path_to_source*> is again replaced by the name of the local install directory.

To update the MultiSense ROS driver to the newest Bitbucket release, execute the following commands:

```
cd <path_to_source>/src/multisense
hg pull
hg up
cd <path_to_source>
rm -r build && rm -r devel
catkin_make
```

1.5. DRIVER LAYOUT

The MultiSense driver consists of five ROS packages:

- **multisense_bringup**

  This is the set of launch files and configuration files used to start the ROS driver, as well as configuration scripts and other configuration files.

- **multisense_cal_check**

  This package provides software for evaluating the quality of the laser calibration stored in the MultiSense-SL non-volatile memory.

  Note this package only supports MultiSense-SL units.

- **multisense_description**

  This package contains the URDF robot description XML file and associated meshes that represent the sensor head, sensor placement and kinematic structure of a MultiSense-SL sensor.

- **multisense_lib**

  This is the library that implements the wire protocol for communication with the MultiSense sensor.

- **multisense_ros**

  This package contains the actual ROS drivers for the MultiSense. Individual drivers are included for the Laser, Camera, and IMU subsystems.

2. RUNNING

When running any of the commands in the remaining sections of this document, make sure the shell environment has first been set by running the following command. As before, replace <*path_to_source*> with the name of the directory in which the ROS driver is installed.

For Fuerte:

```
source <path_to_source>/setup.bash
```

For Groovy and Hydro:

```
Source <path_to_source>/devel/setup.bash
```

2.1. STARTING THE DRIVER

Before running the ROS driver, make sure the MultiSense sensor is powered. For MultiSense-SL units, two LED's on the front of the laser will illuminate on power-up. Next, the RJ45 end of the MultiSense developer's cable should be plugged into the *eth0* network port of the machine on which the ROS driver will be run.

The next step is to configure the network. The *multisense_bringup* package contains an example configuration script called *configureNetwork.sh*. Because the *NetworkManager* daemon may override the settings from *configureNetwork.sh*, it is advisable to stop this daemon before running *configureNetwork.sh*.

In the commands below, once again replace <*path_to_source*> with the name of the directory in which the ROS driver is installed:

```
sudo stop network-manager
sudo <path_to_source>/multisense/multisense_bringup/configureNetwork.sh
```

Note that this will change the IP address of the host machine to 10.66.171.20 and configure other network parameters to communicate with the as-shipped MultiSense sensor.

With the network interface configured, the ROS driver can be launched with the command:

```
roslaunch multisense_bringup multisense.launch
```

This will connect to the MultiSense unit using the default IP address and MTU (10.66.171.21 and 7200, respectively).

If the MultiSense has been configured to use a different IP address, or if a different MTU is desired, the change can be specified with a command line argument:

```
roslaunch multisense_bringup multisense.launch ip_address:=”10.66.171.14” mtu:=”9000”
```

2.2. NAMESPACING

The MultiSense ROS driver supports individual namespacing of driver instances. This feature allows for multiple MultiSense units to be run on the same machine without any conflicts. The namespace of a particular MultiSense ROS driver instance can be changed by specifying the namespace parameter when launching the driver:

```
roslaunch multisense_bringup multisense.launch namespace:=”multisense_1”
```

This starts the MultiSense driver with a namespace of */multisense_1.*

Changing the namespace of a MultiSense ROS driver instance appends all the topic and transform names with the new namespace. The default namespace of a MultiSense driver instance is */multisense.* The following documentation assumes the default namespace when describing transforms, topics, and the dynamic reconfigure interface.

2.3. CONFIGURATION

The ROS driver uses dynamic_reconfigure to adjust the LED duty cycle, spindle motor speed, IMU configuration, and camera parameters.

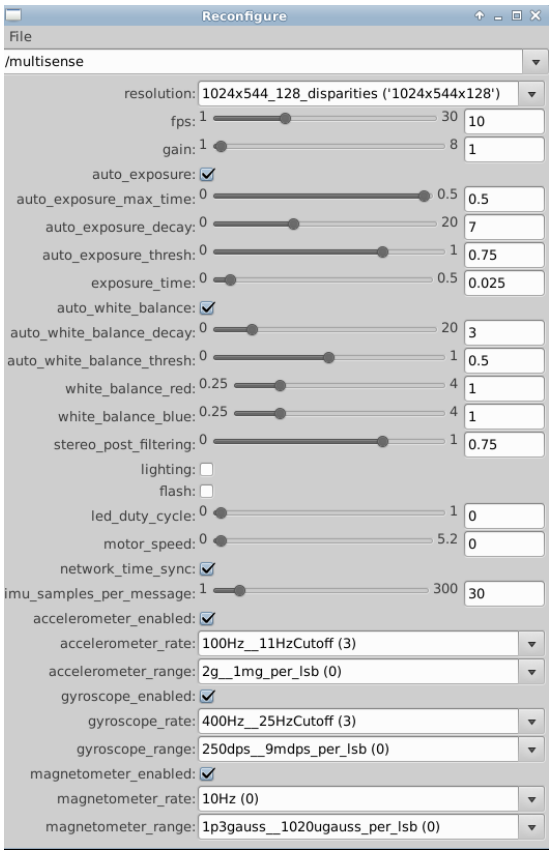
To bring-up the dynamic reconfigure graphical interface for Fuerte installations, execute:

```
rosrun dynamic_reconfigure reconfigure_gui
```

To bring up the dynamic reconfigure graphical interface for Groovy or Hydro installations, execute:

```
rosrun rqt_reconfigure rqt_reconfigure
```

select the "/multisense" namespace from the drop-down list:



Dynamic reconfigure allows the camera resolution to be changed from 0.5 megapixels (1024x544) up to 2 megapixels (2048x1088). *3.0_beta* firmware adds support for non-square vertical resolutions (1024x272 and 2048x544) and varying disparity levels (64, 128, and 256.) Please also note that in the *3.0_beta* and newer firmware release, disparity data is available at all resolutions.

When switching resolutions there is a delay in the image streams while the sensor re-computes internal parameters. This delay can be on the order of 30 seconds.

Note that the spindle motor will not turn unless there is a subscription to the */laser/scan* topic. Similarly the LEDs will not illuminate unless there is a subscription to an image or depth topic. See section 4.5 for more detailed information on these parameters.

2.4. VISUALIZATION

The ROS full install includes a 3D visualization tool called RViz. To start RViz, execute:

```
rosrun rviz rviz
```

Once RViz opens, navigate to "File -> Open Config", and select the file called *rviz_config* in the *multisense_bringup* package, located in the directory:

For Fuerte:

<path_to_source>/multisense/multisense_bringup

For Groovy and Hydro:

<path_to_source>/src/multisense/multisense_bringup

This will load the correct configuration to visualize all sensors on the MultiSense-SL.

Note that in the default Ubuntu 12.04 desktop, the File menu is not displayed on the RViz window, but rather on the panel at the top of the screen, and is only visible when the mouse hovers over the panel. To access the File menu, move the mouse to the top of the screen, and the menu should appear in the upper-left corner.

2.5. CALIBRATION CHECK

The driver comes equipped with a tool to check the laser calibration quality of MultiSense-SL units. The executable requires the MATLAB Compiler Runtime (MCR) to be installed on the computer running the calibration check. To install the MCR navigate to:

http://www.mathworks.com/products/compiler/mcr/index.html

If the ROS driver is being installed on the 64-bit version of Ubuntu 12.04, then download MCR R2013a for 64-bit Linux. If the ROS driver is being installed on the 32 bit version of Ubuntu 12.04, then download MCR R2012a for 32-bit Linux.

Move the downloaded zip into an empty directory, and then unpack it using the "unzip." Navigate into the unpacked directory and find the "install" script. Run the install script as root. On 64-bit Ubuntu 12.04, this will install the MCR here:

/usr/local/MATLAB/MATLAB_Compiler_Runtime/v81/

On 32-bit Ubuntu 12.04, this will install the MCR here:

/usr/local/MATLAB/MATLAB_Compiler_Runtime/v717/

Once the MCR has been installed, and the MultiSense-SL driver has been launched, the calibration utility can be run by executing:

```
rosrun multisense_cal_check cal_check <options>
```

The calibration checking utility connects to the MultiSense-SL captures data, and – by default – saves a ROS bag file (data file) into the directory from which the program is run. After saving the bag file, the calibration checking utility analyzes it and generates a report. For information on how to customize this and other behaviors of the program, run the *cal_check* routine using the –h option:

```
rosrun multisense_cal_check cal_check -h
```

The generated bag file will contain the following:

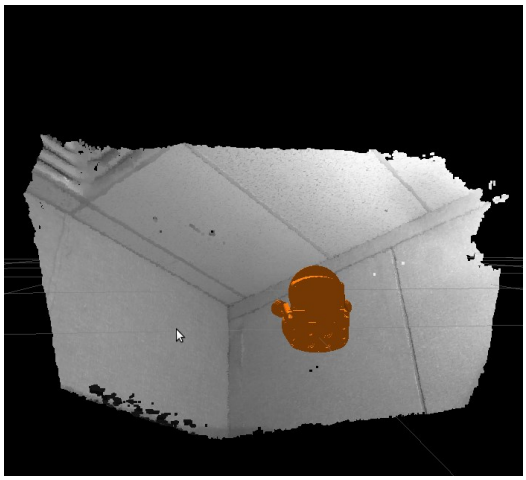
- Information about the MultiSense-SL hardware being checked.
- A synchronized left-rectified and disparity image pair.
- A full revolution of laser scans.

To get the most meaningful calibration results use the following procedure:

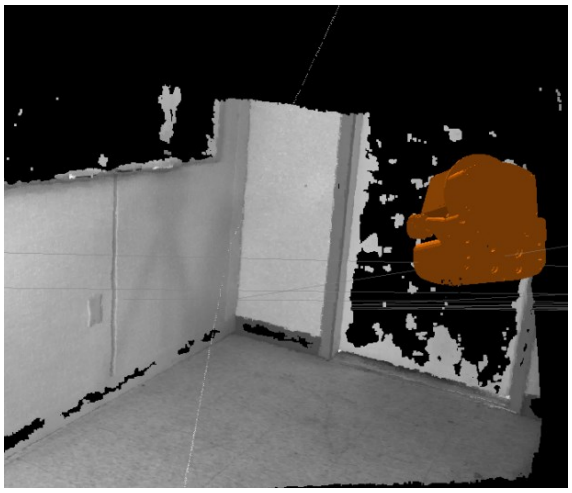
1. Find an interior room corner where two walls meet the floor or ceiling. There must be sufficient visual texture on the walls and ceiling/floor that the stereo algorithm can generate good disparities. It is not important that the surfaces be exactly perpendicular or exactly flat, and it is acceptable to add visual texture by hanging posters, pictures, etc. Industrial carpet or suspended acoustic tile is generally sufficient for floor and ceiling. See below for an example.
2. Place the MultiSense-SL approximately 2.5 meters from the corner, looking directly at the point where the walls and floor/ceiling intersect, and sufficiently far from the walls and floor/ceiling that the camera has a good view of each surface (see below for more detail). Verify that the images from the camera are clear, without large areas of over- or under-exposure. If necessary, the exposure time can be adjusted using the dynamic_reconfigure tool (see Section 2.2 and Section 4.5).
3. Ensure that the MultiSense-SL is securely fastened to a stationary object, and that there is no motion within the field of view of the camera.

4. Source the MultiSense-SL ROS *setup.bash* file as described in Section 0, and then run the calibration checking routine. Ensure that nothing moves in the sensor field of view while the routine is running.
5. Inspect the generated file, *YYYY-MM-DD_hh-mm_TZ_SL_SNxxxx_calResults.txt*, and follow any instructions there. If the calibration routine does not pass after implementing any changes suggested in the .txt file, then contact Carnegie Robotics LLC using the information on the front page of this document.

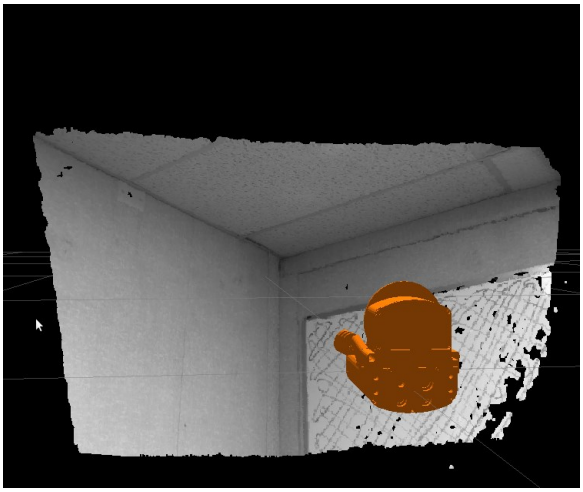
The figures below show an example camera set up and a screenshot of the resulting stereo data. The stereo data visualization was generated using RViz (see Section 2.4). Notice that there is plenty of valid stereo data on each surface. (Note: the walls in these images have a fabric covering that provides excellent texture for disparity generation.)



The next pair of pictures shows a scene in which two surfaces (a whiteboard and a door) do not have sufficient texture to generate a clean stereo image. This scene would not work well with the calibration checker. Subsequent pictures show a similar scene in which the disparities have been improved by adding artificial texture to a whiteboard.



In the following image pair, a marker pattern on the whiteboard provides missing texture and allows the calibration checker to run well.



2.6. COMMAND LINE UTILITIES

The driver contains several command line utilities for querying and setting information stored in non-volatile flash on MultiSense units.

The utilities are packaged as part of the low-level C++ API in the *multisense_lib* package, and can be run using the *rosrun* command. For example, to see the built-in help of the IP address changing utility:

```
rosrun multisense_lib ChangeIpUtility --help
```

2.6.1. Changing the Network Address

The IP address, gateway, and network mask of the MultiSense can be configured by using the *ChangeIpUtility* command:

```
USAGE: ./ChangeIpUtility <options>

Where <options> can be one or more of the possibilities below:
      -a <current_address>    : CURRENT IPV4 address (default=10.66.171.21)
      -A <new_address>        : NEW IPV4 address     (default=10.66.171.21)
      -G <new_gateway>        : NEW IPV4 gateway     (default=10.66.171.1)
      -N <new_netmask>        : NEW IPV4 netmask     (default=255.255.240.0)
      -y                      : disable confirmation prompt
```

For example, assuming the MultiSense's current IP address is 10.66.171.21, the following command:

```
rosrun multisense_lib ChangeIpUtility -A 192.168.0.100 -G 192.168.0.1 -N 255.255.255.0
```

will change the IP address to 192.168.0.100, the gateway to 192.168.0.1, and the network mask to 255.255.255.0.

Note that any changes made will be reflected after the sensor has been power cycled.

2.6.2. Querying and Changing the Stereo Calibration

The stereo calibration can be queried or changed using the *ImageCalUtility* command:

```
USAGE: ImageCalUtility -e <extrinisics_file> -i <intrinsics_file> <options>

Where <options> can be one or more of the possibilities below:
      -a <ip_address>      : ip address (default=10.66.171.21)
      -s                   : set the calibration (default is query)
      -y                   : disable confirmation prompts
```

The default behavior is to query and store the calibration parameters into the filenames specified. For example, assuming the MultiSense's current IP address is 10.66.171.21, the following command:

```
rosrun multisense_lib ImageCalUtility -e saved_extrinsics.yml -I saved_intrinsics.yml
```

will query and store the extrinsics and intrinsics into the "saved_extrinsics.yml" and "saved_intrinsics.yml" files, respectively. If either of the files already exists, a confirmation prompt will be presented.

To program new calibration parameters into the non-volatile flash on the sensor, use the "-s" option:

```
rosrun multisense_lib ImageCalUtility -e new_extrinsics.yml -I new_intrinsics.yml -s
```

Any changes to calibration will be immediately reflected by a query; however, onboard rectification will not see the changes until the sensor has been power cycled.

The utility uses the OpenCV file format for matrix storage, and either YAML or XML formats may be used, determined by the file extension: .yml or .xml, respectively.

Example YAML extrinsics and intrinsics files are provided here:

For Fuerte:

<path_to_source>/multisense/multisense_lib/sensor_api/source/ImageCalUtility

For Groovy and Hydro:

<path_to_source>src/multisense/multisense_lib/sensor_api/source/ImageCalUtility

2.6.3. Querying and Changing the Laser Calibration

The laser calibration on MultiSense-SL units can be queried or changed using the *LidarCalUtility* command:

```
USAGE: LidarCalUtility -f <calibration_file> <options>

Where <options> can be one or more of the possibilities below:
      -a <ip_address>   : ip address (default=10.66.171.21)
      -s                : set the calibration (default is query)
      -y                : disable confirmation prompts
```

The default behavior is to query and store the calibration parameters into the filename specified. For example, assuming the MultiSense-SL's current IP address is 10.66.171.21, the following command:

```
rosrun multisense_lib LidarCalUtility  -f saved_lidar_cal.yml
```

will query and store the laser calibration into the "saved_lidar_cal.yml" file. If the file already exists, a confirmation prompt will be presented.

To program new calibration parameters into the non-volatile flash on the sensor, use the "-s" option:

```
rosrun multisense_lib LidarCalUtility -f new_lidar_cal.yml -s
```

Any changes to calibration will be immediately reflected by a query; however, any ROS nodes communicating with the sensor will require a restart to pick up the change.

The utility uses the OpenCV file format for matrix storage, and either YAML or XML formats may be used, determined by the file extension: .yml or .xml, respectively.

An example YAML laser calibration is provided here:

For Fuerte:

<path_to_source>/multisense_lib/sensor_api/source/LidarCalUtility

For Groovy and Hydro:

<path_to_source>src/multisense_lib/sensor_api/source/LidarCalUtility

2.6.4. Querying Device Information

The factory-set device information can be queried by using the *DeviceInfoUtility* command:

```
USAGE: DeviceInfoUtility <options>

Where <options> can be one or more of the possibilities below:
      -a <ip_address>    : ip address (default=10.66.171.21)
      -k <passphrase>    : passphrase for setting device info
      -s <file_name>     : set device info from file
      -q                 : query device info
      -y                 : disable confirmation prompt
```

For example, assuming the MultiSense's current IP address is 10.66.171.21, the following command:

```
rosrun multisense_lib DeviceInfoUtility -q
```

will query and print the device information to the console. Setting the device information with the '-s' option is not publicly supported.

2.6.5. Querying and Changing the IMU Configuration

The IMU configuration can be queried or changed using the *ImuConfigUtility* command:

```
USAGE: ImuConfigUtility [<options>]
Where <options> are:
        -a <ip_address>     : IPV4 address (default=10.66.171.21)
        -q                  : query and report IMU configuration
        -f                  : store IMU configuration in non-volatile flash
        -s <samples>        : set IMU samples-per-message
        -c "<sensor_config>": IMU sensor configuration string

And "<sensor_config>" is of the following form:
        "<sensor_name> <enabled> <rate_table_index> <range_table_index>"

For example, to enable the accelerometer, and have it use rate index 1 and
range index 2:

        -c "accelerometer true 1 2"

Multiple "-c" options may be specified to configure more than 1 sensor
```

For example, assuming the MultiSense's current IP address is 10.66.17.21, the following command:

```
rosrun multisense_lib ImuConfigUtility -f -s 30 -c "accelerometer true 3 0" -c "gyroscope true 3 0" -c "magnetometer true 0 0"
```

will change the IMU configuration back to the factory defaults, storing the configuration in non-volatile flash on the sensor head.

The "-q" option will print the current configuration along with detailed information about the possible settings for each sensor type.

The "-s" option will tell the MultiSense unit how many IMU sensor samples to queue internally before sending them over the network. This can be used to make tradeoffs between sample rates, processor load, and latency. Please note that low samples-per-message settings combined with high sample rates may interfere with the acquisition and transmission of image and laser data.

See section 4.5 for information on how the IMU configuration is exposed in the ROS dynamic_reconfigure interface.

2.6.6. Upgrading the Onboard Software

The onboard software can be field upgraded by using the *MultiSenseUpdater* command:

```
USAGE: MultiSenseUpdater <ip_address> <update_file>

Where <ip_address> is the sensor's IPV4 address or resolvable hostname, and
      <update_file> is an official ".slf" package file.
```

For example, assuming the MultiSense's IP address is 10.66.171.21, the following command:

```
rosrun multisense_lib MultiSenseUpdater 10.66.171.21 multisense_firmware_v2_3.slf
```

will update the sensor's firmware to version 2.3.

NOTE: This utility will print verbose progress information to the console and may query for user input. The update process may take several minutes to complete.

WARNING: If the firmware update process is not allowed to complete successfully, the MultiSense unit can be left in a non-functional state. To avoid this:

- Before running the update command, make sure to close all other software that communicates with the MultiSense (ROS nodes, utilities, etc.)
- Ensure that the MultiSense unit is on an isolated network and has a stable power source.
- Do not interrupt the update process once underway.

3. TROUBLESHOOTING

Symptom	Recommended Action	Explanation
No communications with the MultiSense	**Check that *NetworkManager* is not running, and stop it if necessary using the command *sudo stop network-manager*. Then reconfigure the network as described in Section 0.**	***NetworkManager* periodically checks network configuration, and may change network settings unpredictably, interrupting communications with the MultiSense.**
No images received	**Check network MTU size on connected computer using the *ifconfig* command. Verify that MTU size is set to 7200.**	**Some network cards do not support setting large MTU sizes.**
No images received	**Try plugging the Ethernet cable from the MultiSense directly into the host computer, then move to a higher quality network switch.**	**The current communications protocol is quite sensitive to dropped packets. Competing network traffic or poor connections can easily result in no images being received.**
No images received or inconsistent image framerate	**Try increasing the size of the kernel's networking RX buffers by adjusting */proc/sys/net/core/rmem_max*. See Section 2.1 and the *configureNetwork.sh* script for more details.**	**The SL can generate large amounts of UDP traffic that may overwhelm typical network settings on some machines.**
Stereo point cloud is not updating in RViz	**Ensure the resolution of the sensor is 0.5 megapixels and not 2 megapixels (see Section 2.2). Or update to a firmware version 3.0 or greater.**	**For firmware versions less than 3.0, disparity images, and stereo point clouds, are not supported at camera resolutions other than 0.5 megapixels. This limitation will change with future firmware revisions.**
Image streams pause during a sensor resolution change.	**Wait approximately 30 seconds.**	**Because of current limitations in the sensor firmware, switching the resolution causes the sensor to recalculate internal parameters.**
Sensor not displayed in RViz	**Navigate to the directory where the MultiSense stack is installed, source the *setup.bash* file, and restart RViz.**	**RViz was run in a terminal in which the ROS environment did not include the MultiSense stack.**
No data is published on the */multisense/pps* topic	**Update the sensor's firmware to version 2.2 or greater.**	**The PPS functionality was added in the 2.2 firmware release.**

4. ROS API DOCUMENTATION

The ROS API for the MultiSense sensor, provided by the *ros_driver* executable, is split into distinct subsystems for each piece of hardware. Each of these subsystems is documented below.

Streams from the sensor are initiated on an "on-demand" basis. That is, on subscription of a given data stream (such as */multisense/left/image_rect*), the driver will initialize the data stream on the sensor. Thus, the driver uses very little bandwidth and CPU when there are no subscriptions to any of the ROS topics.

4.1. CAMERA SUBSYSTEM

Camera topics are supported for all MultiSense products including: MultiSense-SL, MultiSense-S7, MultiSense-S7S, and MultiSense-S21.

4.1.1. Published Topics

4.1.1.1 Depth Camera

/multisense/camera_info (sensor_msgs/CameraInfo)

Camera projection matrix and metadata.

/multisense/depth (sensor_msgs/Image)

Depth image. Uses canonical representation (float32 meters). Depth images replace disparity images as the standard (See REP 118)

Subtopics */compressed* & */theora* are for use with image_transport.

/multisense/image_points2 (sensor_msgs/PointCloud2)

Stereo point cloud. Each point contains 4 fields (x,y,z, lumanince). See the pcl wiki page for more information.

/multisensel/image_points2_color (sensor_msgs/PointCloud2)

Color stereo point cloud. Each point contains 4 fields (x,y,z, rgb). See the pcl wiki page for more information. Color images are rectified on the host machine before combining with range data.

/multisense/left/disparity (sensor_msgs/Image)

Left disparity image in "mono16" format (units are $1/16^{th}$ of a disparity.)

/multisense/right/disparity (sensor_msgs/Image)

3.0+ firmware only. Right disparity image in "mono8" format. In the *3.0_beta* release, these images have not been passed through the left/right-discrepancy or post-stereo filters.

/multisense/left/cost (sensor_msgs/Image)

3.0+ firmware only. Left stereo cost in "mono8" format. Higher values represent less confidence in the disparity value generated for that location. In the *3.0_beta* release, these images have not been passed through the left/right-discrepancy or post-stereo filters.

4.1.1.2 Left Camera

/multisense/left/camera_info (sensor_msgs/CameraInfo)

Camera projection matrix and metadata

/multisense/left/image_rect (sensor_msgs/Image)

Rectified images from left camera in mono8 format

Subtopics */compressed* & */theora* are for use with image_transport.

/multisense/left/image_rect_color (sensor_msgs/Image)

Rectified images from left camera in RGB8 format

Subtopics */compressed* & */theora* are for use with image_transport.

Color images are rectified on the host machine using OpenCV

/multisense/left/image_mono (sensor_msgs/Image)

Unrectified images from left camera in mono8 format

Subtopics */compressed* & */theora* are for use with image_transport.

/multisense/left/image_color (sensor_msgs/Image)

Unrectified images from left camera in RGB8 format

Subtopics */compressed* & */theora* are for use with image_transport.

4.1.1.3 Right Camera

/multisense/right/camera_info (sensor_msgs/CameraInfo)

Camera projection matrix and metadata

/multisense/right/image_rect (sensor_msgs/Image)

Rectified images from right camera in mono8 format

Subtopics */compressed* & */theora* are for use with image_transport.

/multisense/right/image_mono (sensor_msgs/Image)

Unrectified images from right camera in mono8 format

Subtopics */compressed* & */theora* are for use with image_transport.

In the current release, color images are not supported for the right camera.

4.1.1.4 Calibration

/multisense/calibration/device_info (multisense_ros/DeviceInfo)

Hardware and software versioning information.

/multisense/calibration/raw_cam_cal (multisense_ros/RawCamCal)

Rectification and stereo projection matrices: unrectified camera matrix A (left_M, right_M), distortion coefficients (left_D, right_D), rectification transformation matrix (left_R, right_R), new camera matrix A' (left_P, right_P.) These matrices are for the maximum operating resolution of the MultiSense unit.

/multisense/calibration/raw_cam_config (multisense_ros/RawCamCal)

Current operating resolution and camera parameters: fx, fy, cx, cy, tx, ty, tz, roll, pitch, and yaw. These parameters have been scaled to the current operating resolution of the MultiSense unit.

/multisense/calibration/raw_cam_data (multisense_ros/RawCamData)

Synchronized left-rectified and left-disparity images.

4.1.1.5 Histogram

/multisense/histogram (multisense_ros/Histogram)

Image histograms for each of the 4 Bayer channels in the left camera image. Each message contains the image size, fps, exposure time, and number of histogram bins. The data field contains ***channels*bins*** elements, where each histogram is concatenated serially in the order green, red, blue, green. Note that there must be a subscription to a MultiSense image topic for the histogram topic to be published.

4.2. LASER SUBSYSTEM

Laser topics are only supported for MultiSense-SL units.

4.2.1. Published Topics

4.2.1.1 Laser Data

/multisense/lidar_points2 (sensor_msgs/PointCloud2)

Laser point cloud. Each point contains 4 fields: x, y, z, and intensity.

*/multisense/lidar_scan (*sensor_msgs/LaserScan)

Raw laser scan from device.

4.2.1.2 Calibration

/laser/calibration/raw_lidar_cal (multisense_ros/RawLidarCal)

The laser calibration, represented by the homogeneous transform matrices H_M^C (*cameraToSpindleFixed*) and H_L^S (*laserToSpindle*.) See section 5.1 for more information.

/laser/calibration/raw_lidar_data (multisense_ros/RawLidarData)

Contains the raw laser scan data. This includes ranges, intensities, and the timestamps and spindle-angles for the start and end of each scan.

4.3. PPS SUBSYSTEM

PPS topics are supported for all MultiSense products including: MultiSense-SL, MultiSense-S7, MultiSense-S7S, and MultiSense-S21.

4.3.1. Published Topics

This topic requires sensor firmware >= v2.2.

/multisense/pps (std_msgs/Time)

Contains the time of the last PPS in the sensor's clock frame. Published once per second, immediately after the pulse.

Please note that firmware version 2.1 contained a bug that could duplicate PPS events.

4.4. IMU SUBSYSTEM

IMU topics are supported for all MultiSense products including: MultiSense-SL, MultiSense-S7, MultiSense-S7S, and MultiSense-S21.

4.4.1. Published Topics

These topics require sensor firmware >= v2.3.

/multisense/imu/accelerometer (multisense_ros/RawImuData)

Raw accelerometer data. Each message contains 4 fields: time and x,y,z accelerations in g.

/multisense/imu/gyroscope (multisense_ros/RawImuData)

Raw gyroscope data. Each message contains 4 fields: time and x,y,z rates in degrees-per-second.

/multisense/imu/magnetometer (multisense_ros/RawImuData)

Raw magnetometer data. Each message contains 4 fields: time and x,y,z measurements in gauss.

4.5. RECONFIGURABLE PARAMETERS

These parameters employ dynamic_reconfigure, and can be modified at runtime by executing the following commands:

For Fuerte:

```
rosrun dynamic_reconfigure reconfigure_gui
```

For Groovy or Hydro:

```
rosurn rqt_reconfigure rqt_reconfigure
```

Selecting "/multisense" from the drop-down list will bring up the following configurable parameters.

Note the dynamic reconfigure parameters vary between the different MultiSense sensor types. For instance *motor_speed* is only present when a MultiSense-SL unit is connected to the current instance of the ROS driver.

/multisense

resolution (string)

Resolution of the camera images streamed from the sensor, in the format of "<*width*>x<*height*>x<*numberOfDisparities*>". In the *3.0_beta* and newer releases, each resolution can be configured for 64, 128, or 256 disparities. Since the hardware stereo core has a finite throughput, lower disparities will have the effect of increasing frame-rate at the cost of near-field coverage.

fps (double)

Frames per second.

gain (double)

Camera gain.

auto_exposure (bool)

Enable or disable auto exposure.

auto_exposure_max_time (int)

Maximum time (in seconds) for the auto-exposure algorithm.

auto_exposure_decay (int)

Adjust the auto exposure decay. Increasing this parameter makes the auto exposure algorithm respond more slowly to changes in lighting.

auto_exposure_thresh (double)

Adjust the auto exposure threshold. This parameter changes the overall scene brightness target for the AE algorithm.

exposure_time (double)

Time (in seconds) for camera exposure. Ignored if auto exposure is enabled.

auto_white_balance (bool)

Enable or disable auto white balance.

auto_white_balance_decay (int)

Adjust auto white balance decay. Increasing this parameter makes the auto white balance algorithm respond more slowly to changes.

auto_white_balance_thresh (double)

Adjust the auto white balance threshold.

white_balance_red (double)

Adjust the red white balance. Ignored if auto white balance enabled.

white_balance_blue (double)

Adjust the blue white balance. Ignored if auto white balance enabled.

stereo_post_filtering (double)

Adjust the strength of the hardware stereo post-filter.

lighting (bool)

Enable or disable LEDs.

flash (bool)

Enable or disable flashing of LEDs.

duty_cycle (double)

Change brightness of LEDs (0 = Off, 1 = Full Brightness.)

motor_speed (double)

Change laser spindle speed in radians/second.

network_time_sync (bool)

Enable network-based time sync between the host computer and sensor. If disabled, all datum timestamps will be in the frame of the SL's internal clock, which is free-running from zero on power up.

NOTE:

- The following IMU parameters will only be presented if the ROS driver detects that the MultiSense unit is running firmware v2.3 or greater.
- Any IMU configuration changes will take effect after all IMU topic subscriptions have been closed.
- Refer to section 6 of this document for more information on the IMU sensors.

imu_samples_per_message (int)

Adjust the number of IMU readings (aggregate from accel/gyro/mag sensors) that the MultiSense sensor will accumulate before shipping over the network. This can be used to make tradeoffs between sample rates, processor load, and latency.

accelerometer_enabled (bool)

Enable or disable the accelerometer sensor.

accelerometer_rate (int)

Selects the sample rate index of the accelerometer. Valid values are in [0, 6], corresponding to: 10, 25, 50, 100, 200, 400, and 1344 Hz.

accelerometer_range (int)

Selects the sample range index of the accelerometer. Valid values are in [0, 3], corresponding to: 2, 4, 8, and 16 g.

gyroscope_enabled (bool)

Enable or disable the gyroscope sensor.

gyroscope_rate (int)

Selects the sample rate index of the gyroscope. Valid values are in [0, 3], corresponding to: 100, 200, 400, and 800 Hz.

gyroscope_range (int)

Selects the sample range index of the gyroscope. Valid values are in [0, 2], corresponding to 250, 500, and 2000 degrees-per-second.

magnetometer_enabled (bool)

Enable or disable the magnetometer sensor.

magnetometer_rate (int)

Selects the sample rate index of the magnetometer. Valid values are in [0, 3], corresponding to 10, 25, 50, and 100 Hz.

magnetometer_range (int)

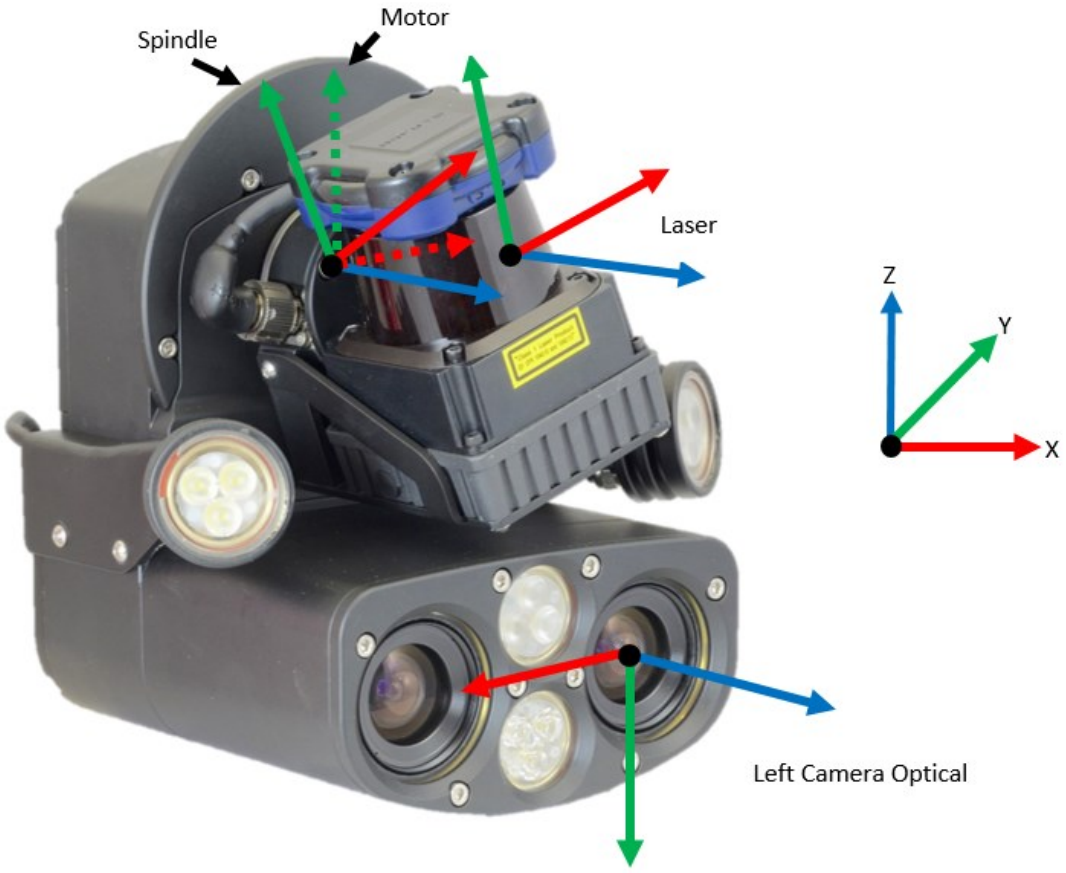
Selects the sample range index of the magnetometer. Valid values are in [0, 6], corresponding to: 1.3, 1.9, 2.5, 4.0, 4.7, 5.6, and 8.1 gauss.

5. MULTISENSE-SL TRANSFORMS

The MultiSense-SL unit stores a unique calibration to transform laser range data into the left camera optical frame. This calibration is comprised of two static transforms; one from the motor frame to the left camera frame, the other from the laser frame to the spindle frame. There is an additional transform between the spindle frame and the motor frame which accounts for the rotation of the laser spindle.

The MultiSense-SL ROS driver automatically generates a transform tree based on these three transforms. Additionally both the ROS driver and the underlying API offer mechanisms to perform custom transformations of laser data. The following sections detail this transform tree and provide a basic example of how to perform this transformation.

5.1. OVERVIEW



The MultiSense-SL has three primary transforms between the four coordinate frames which are used to transform laser range data into the left camera optical frame.

The first transform is a static transform between the fixed motor frame and the left camera frame which is described by the homogeneous transform matrix H_M^C. In the ROS driver this transform is referred to as *cameraToSpindleFixed*.

The second transform is between the spindle frame and the fixed motor frame which is described by the homogeneous transform matrix H_S^M. This transform accounts for the spindle frame's rotation about the z-axis of the fixed motor frame as the laser spins.

The final transform is static between the laser frame and the spindle frame which is described by the homogeneous transform matrix H_L^S. In the ROS driver H_L^S is referred to as *laserToSpindle*. By multiplying these homogeneous transform matrices together a laser range point in Cartesian space can be transformed into the left camera frame.

$$\boldsymbol{d_c} = H_M^C H_S^M H_L^S \boldsymbol{d_L}$$

Where $\boldsymbol{d_L}$ is a 4x1 matrix representing a single laser point in the x-z plane of the laser's local frame and $\boldsymbol{d_c}$ is a 4x1 matrix representing a single laser point in the left camera's optical frame.

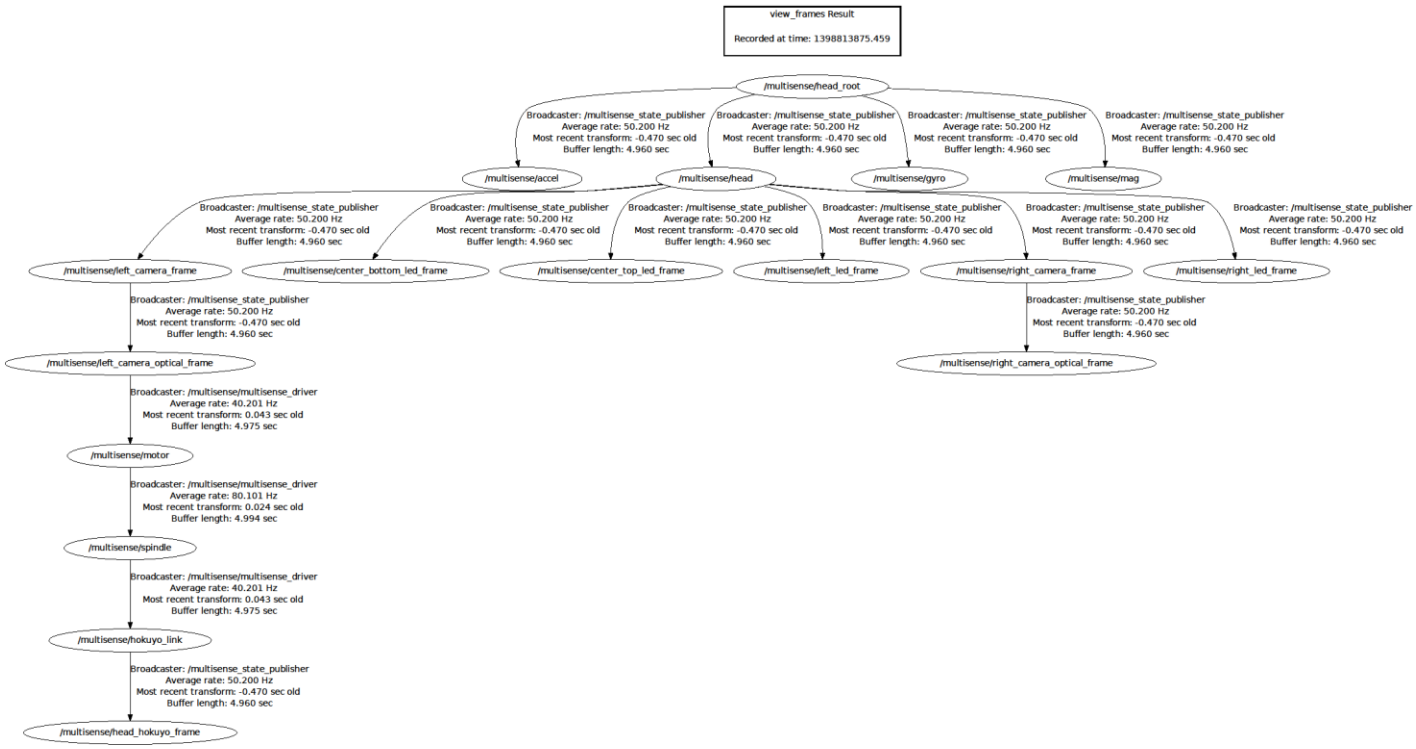
5.2. ROS TRANSFORM TREE

The MultiSense-SL ROS driver uses the robot_state_publisher ROS package to create the initial transform tree using the URDF file in the *multisense_description* package.

The transform H_M^C is accounted for in the transformation between */multisense/motor* and */multisense/left_camera_optical* frames. The H_S^M transform is accounted for between the */multisense/spindle* and */multisense_motor* frames. The transform H_L^S is accounted for between the */multisense/hokuyo_link* and */multisense_spindle* frames.

A final static transformation between */multisense/hokuyo_link* and */multisense/head_hokuyo_frame* is required to properly display the */multisense/lidar_scan* topic in RViz.

The full ROS transform tree is shown below.



5.3. ROS LASER INTERFACE

The MultiSense-SL ROS driver offers three primary mechanisms to process and view laser data: the */multisense/lidar_scan*, */multisense/lidar_points2*, and */multisense/calibration/raw_lidar_data* topics. In addition to laser data, the driver presents the raw calibration matrices on the */multisense/calibration/raw_lidar_cal* topic.

5.3.1. /multisense/lidar_scan Topic

The */multisense/lidar_scan* topic is a sensor_msgs/LaserScan message containing the raw laser range and intensity data in polar coordinates along with information used to interpolate the Cartesian position of each range reading. The transformation of the laser data into the /multisense*/left_camera_optical_frame* is managed by the transform tree constructed by the robot_state_publisher and the per-scan published full H_M^C and H_L^S transforms managed by the ROS driver.

When the laser is spinning the motion between individual range returns is compensated by ROS internally using SLERP. This interpolation can also be used to account for external motions of the head; assuming the MultiSense-SL's transform tree is a child of the frame in which the motion is occurring.

5.3.2. /multisense/lidar_points2 Topic

The */multisense/lidar_points2* topic is a sensor_msgs/PointCloud2 message which contains laser data transformed into the */multisense/left_camera_optical_frame*. Each laser scan is published as an individual point cloud. When transforming the laser data into Cartesian space both the scan angle and the spindle angle are linearly interpolated. This transformation is done without knowledge of external forces that may be acting on the sensor.

5.3.3. /multisense/calibration/raw_lidar_data and /multisense/calibration/raw_lidar_cal Topics

The */multisense/calibration/raw_lidar_data* and */multisense/calibration/raw_lidar_cal* are a custom message types which contain all the information necessary to manually transform the laser data into the */multisense/left_camera_optical_frame*. The */multisense/calibration/raw_lidar_data* topic contains the raw laser data as well as information about the angle of the spindle frame at the start and end of the laser scan.

The */multisense/calibration/raw_lidar_cal* topic contains the two static calibration transforms H_M^C and H_L^S flattened to 16 element arrays in row-major order. Using these two topics the laser data can be manually transformed using a custom interpolation scheme external of the MultiSense-SL ROS driver.

5.4. EXTERNAL TRANSFORMATION EXAMPLE

The following excerpts of code use ROS's TF and Angles packages to transform laser data into the */left_camera_optical_frame*. The example consists of two subscribers; one to the */laser/calibration/raw_lidar_data* topic and the other to the */laser/calibration/raw_lidar_cal* topic.

5.4.1. /multisense/calibration/raw_lidar_cal Subscriber

```
//
// Create Two TF Transforms for the Camera to Spindle Calibration and the Laser to Spindle
Calibration
void rawLaserCalCallback(const multisense_ros::RawLidarCal::ConstPtr& msg)
{
    //
    //Get Transform Matrices from flattened 16 element Homogenous arrays in
    // row-major order
```

```
    _laserToSpindle = makeTransform(msg->laserToSpindle);
    _motorToCamera  = makeTransform(msg->cameraToSpindleFixed);
}

template <typename T>
tf::Transform makeTransform(T data)
{
    //
    // Manually create the rotation matrix
    tf::Matrix3x3 rot = tf::Matrix3x3(data[0],
                                      data[1],
                                      data[2],
                                      data[4],
                                      data[5],
                                      data[6],
                                      data[8],
                                      data[9],
                                      data[10]);

    //
    // Maually create the translation vector
    tf::Vector3 trans = tf::Vector3(data[3], data[7], data[11]);

    return tf::Transform(rot, trans);
}
```

The above code constructs two tf::Transfrom objects, *_laserToSpindle* and *_motorToCamera*, from a single */multisense/calibration/raw_lidar_cal* message. These Frames correspond to H_L^S and H_M^C respectively.

5.4.2. /multisense/calibration/raw_lidar_data Subscriber

```
//
// Transforms Laser Data using onboard calibration
void rawLaserDataCallback(const multisense_ros::RawLidarData::ConstPtr& msg)
{

    //
    // For Visualization
    makePointCloudHeader(msg->distance.size());
    _point_cloud.header.stamp   = msg->time_start;
    uint8_t         *cloudP     = reinterpret_cast<uint8_t*>(&_point_cloud.data[0]);
    const uint32_t  pointSize   = 3 * sizeof(float);


    //
    // Scan angle range of the laser
    const double fullScanAngle = 270 * M_PI / 180.;

    //
    // Laser start at -135 degrees spinning counterclockwise
    const double startScanAngle = -fullScanAngle / 2.;

    //
    // Get the spindle angles for this scan. angle_start and angle_end are in micro-radians
    // Use ros package angles to normalize them to -PI to +PI
    const double spindleAngleStart = angles::normalize_angle(1e-6
                                                * static_cast<double>(msg->angle_start));

    const double spindleAngleEnd   = angles::normalize_angle(1e-6
                                                * static_cast<double>(msg->angle_end));

    const double spindleAngleRange = angles::normalize_angle(spindleAngleEnd
                                                          -  spindleAngleStart);
```

```
    //
    // Loop over all the range data
    for ( unsigned int i = 0; i < msg->distance.size(); i++, cloudP
                                                          += _point_cloud.point_step){

        //
        // Percent through the scan. Used for linear interpolation
        const double percent = static_cast<double>(i)
                             / static_cast<double>(msg->distance.size() -1);

        //
        // Linearly interpolate the laser scan angle from -135 to 135
        const double mirrorAngle = startScanAngle + (percent * fullScanAngle);

        //
        // Linearly interpolate the spindle angle
        const double spindleAngle = spindleAngleStart + (percent * spindleAngleRange);

        //
        // Generate the Homogeneous Transform for the Motor to Spindle Transform
        const double cosSpindle = std::cos(spindleAngle);
        const double sinSpindle = std::sin(spindleAngle);
        _spindleToMotor = tf::Transform(tf::Matrix3x3(cosSpindle, -sinSpindle, 0,
                                                      sinSpindle,  cosSpindle, 0,
                                                      0         ,           0, 1),
                                                      tf::Vector3(0, 0, 0)      );

        //
        // Convert range point to meters and project into the X-Z laser plane
        const double        rangeMeters = static_cast<double>(1e-3 * msg->distance[i]);
        const tf::Vector3   rangePoint  = tf::Vector3(rangeMeters * std::sin(mirrorAngle),
                                                      0,
                                                      rangeMeters * std::cos(mirrorAngle));

        //
        // Transform point into the left camera frame
        const tf::Vector3 pointInCamera = _motorToCamera
                                        * (_spindleToMotor
                                        * (_laserToSpindle * rangePoint));


        //
        // Copy data to point cloud structure
        const float xyz[3] = {static_cast<float>(pointInCamera.getX()),
                              static_cast<float>(pointInCamera.getY()),
                              static_cast<float>(pointInCamera.getZ())};

        memcpy(cloudP, &(xyz[0]), pointSize);
        memcpy((cloudP + pointSize), &(msg->intensity[i]), sizeof(uint32_t));
    }
    _point_pub.publish(_point_cloud);

}

void makePointCloudHeader(const unsigned int pointCount)
{
    const uint32_t cloud_step = 16;

    _point_cloud.data.resize(cloud_step * pointCount);
    _point_cloud.is_bigendian       = (htonl(1) == 1);
    _point_cloud.is_dense           = true;
    _point_cloud.point_step         = cloud_step;
    _point_cloud.height             = 1;
    _point_cloud.header.frame_id    = "/multisense/left_camera_optical_frame";

    _point_cloud.fields.resize(4);
```

```
    _point_cloud.fields[0].name     = "x";
    _point_cloud.fields[0].offset   = 0;
    _point_cloud.fields[0].count    = 1;
    _point_cloud.fields[0].datatype = sensor_msgs::PointField::FLOAT32;
    _point_cloud.fields[1].name     = "y";
    _point_cloud.fields[1].offset   = 4;
    _point_cloud.fields[1].count    = 1;
    _point_cloud.fields[1].datatype = sensor_msgs::PointField::FLOAT32;
    _point_cloud.fields[2].name     = "z";
    _point_cloud.fields[2].offset   = 8;
    _point_cloud.fields[2].count    = 1;
    _point_cloud.fields[2].datatype = sensor_msgs::PointField::FLOAT32;
    _point_cloud.fields[3].name     = "intensity";
    _point_cloud.fields[3].offset   = 12;
    _point_cloud.fields[3].count    = 1;
    _point_cloud.fields[3].datatype = sensor_msgs::PointField::UINT32;

    _point_cloud.row_step           = pointCount;
    _point_cloud.width              = pointCount;
}
```

The above code generates a tf::Vector3 object *pointInCamera* which contains the x, y, and z coordinates of a single laser range point transformed into the */multisense/left_camera_optical_frame*. The code linearly interpolates the laser scan angle as it spins counterclockwise from -135 degrees to 135 degrees. The angle of the spindle, originally published in micro-radians, is also linearly interpolated to compensate for motion due to the rotation of the spindle.

6. ACCELEROMETER, MAGNETOMETER, AND GYROSCOPE

6.1. ACCELEROMETER AND MAGNETOMETER

The accelerometer and magnetometer are combined on a STMicroelectronics LSM303DLHC 3-D linear accelerometer and magnetometer microchip. For more information on the chip's operation please see the full datasheet at:

http://www.st.com/web/catalog/sense_power/FM89/SC1449/PF251940.

The measured outputs for the accelerometer and magnetometer are in units of g-force and gauss respectively.

6.2. GYROSCOPE

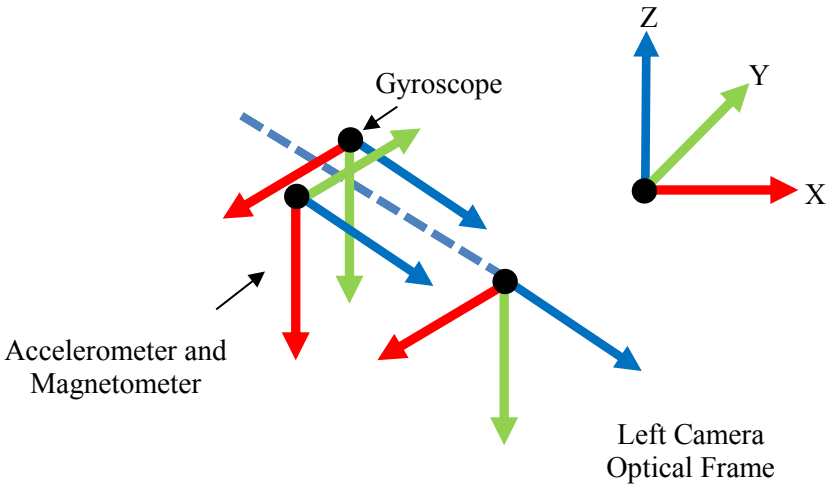
The gyroscope is a STMicroelectronics L3G4200D three-axis gyroscope. For information on the chip's operation please see the full datasheet at:

http://www.st.com/web/catalog/sense_power/FM89/SC1288/PF250373.

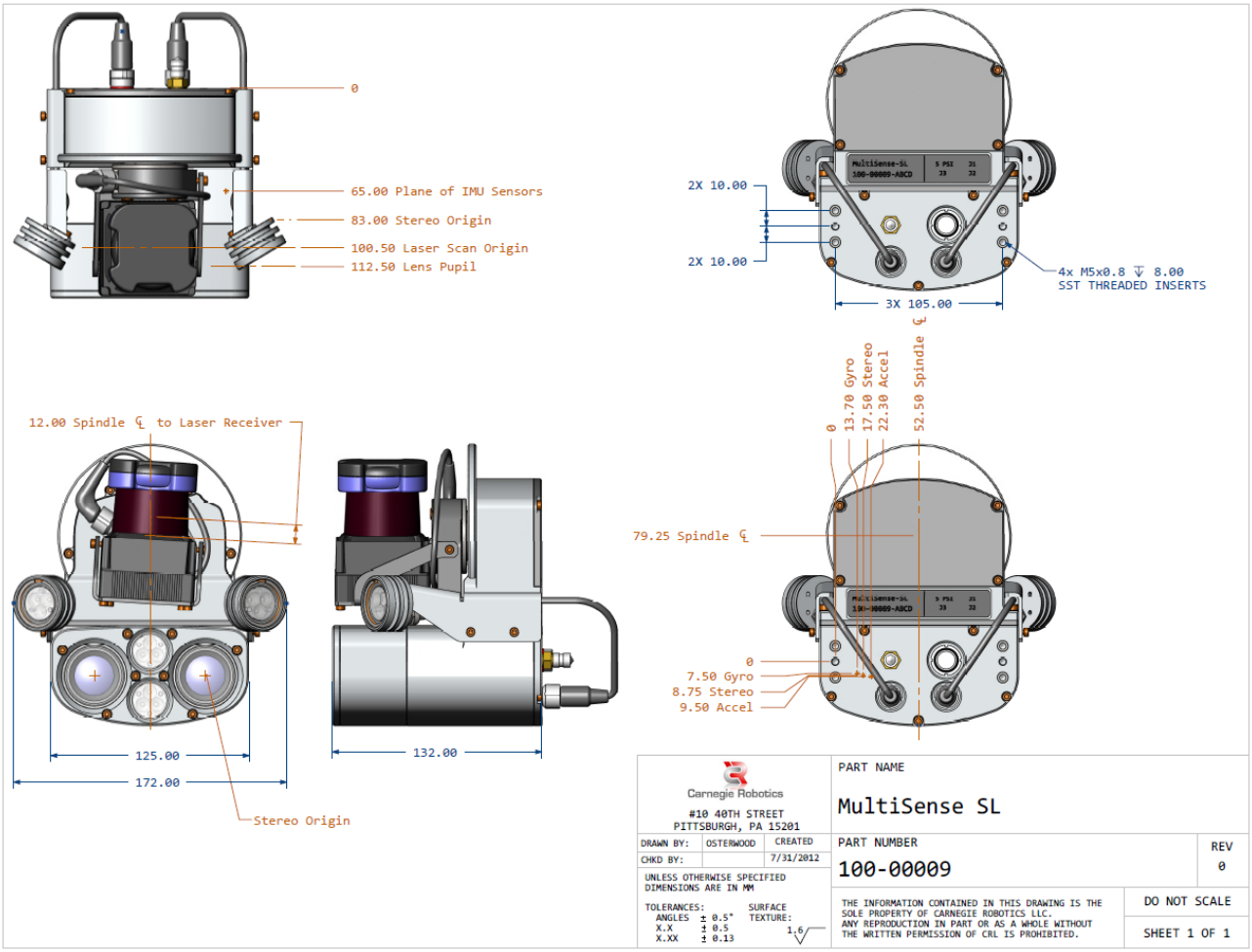
The measured output is in degrees per second.

6.3. MOUNTING LOCATION

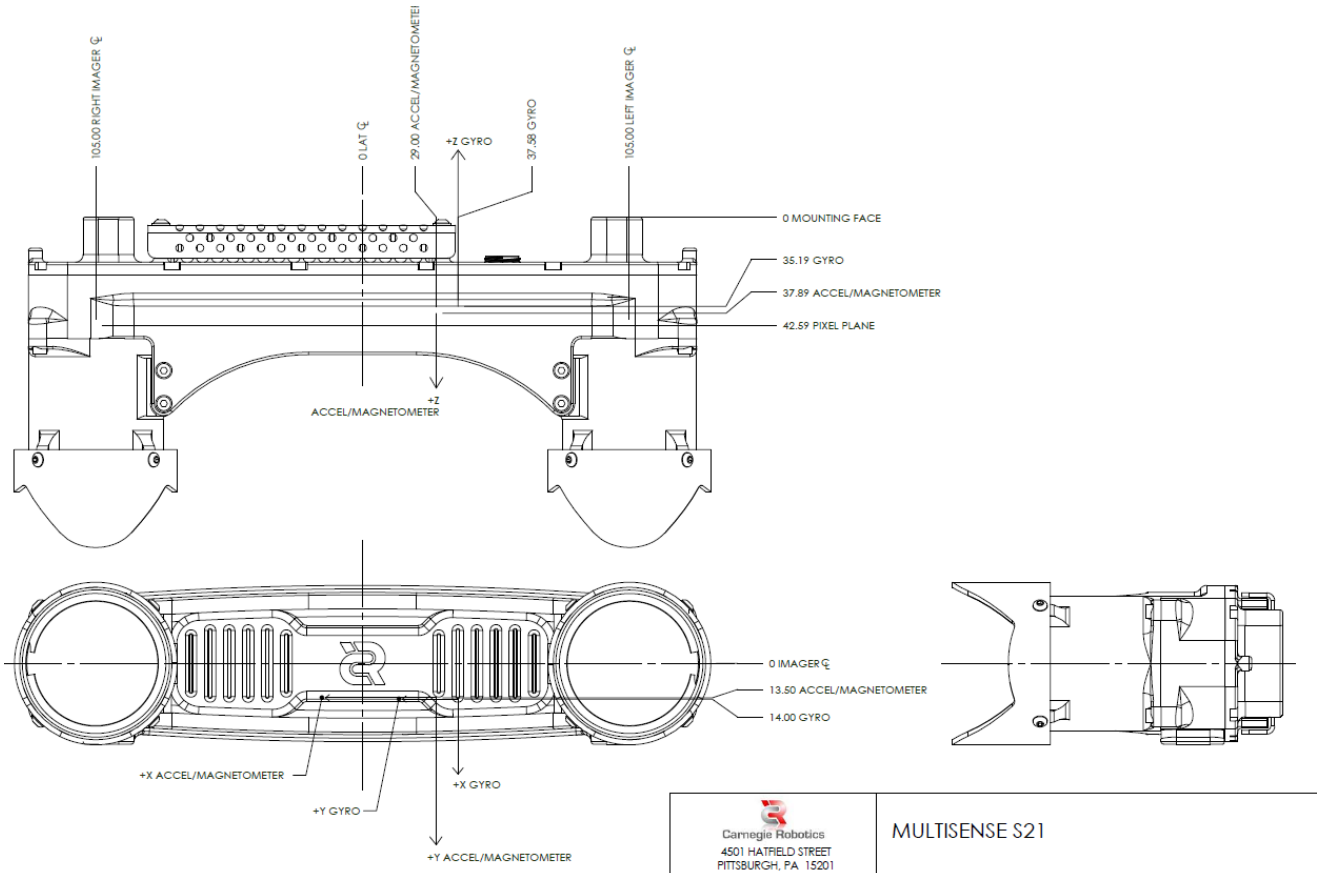
The accelerometer, magnetometer, and gyroscope are all located under the left imager. Their exact locations correspond with the */multisense/accel*, */multisense/mag*, and */multisense/gyro* frames in the /tf tree. The accelerometer and magnetometer are mounted with their x-axes pointed downward while the gyroscope is mounted with its y-axis pointed downward.



The exact locations in MultiSense-SL and MultiSense-S7 units are described in the CAD drawing below:



Note: “Lens Pupil” corresponds to */multisense/left_camera_optical_frame* in the ROS /tf.



GYRO AND ACCEL/MAGNETOMETER DEFAULT AXES ARE SHOWN
AXES FOLLOW RIGHT HAND RULE CONVENTION

Carnegie Robotics
4501 HATFIELD STREET
PITTSBURGH, PA 15201

MULTISENSE S21

DRAWN BY:	R. DOLL	CREATED
CHKD BY:		3/25/2014
UNLESS OTHERWISE SPECIFIED		

SIZE A | PART NUMBER: 100-00033 | REV 0

THE INFORMATION CONTAINED IN THIS DRAWING IS THE SOLE PROPERTY OF CARNEGIE ROBOTICS LLC. ANY REPRODUCTION IN PART OR AS A WHOLE WITHOUT THE WRITTEN PERMISSION OF CRL IS PROHIBITED.

DIMENSIONS ARE IN MM
TOLERANCES:
ANGULAR: MACH± .5 DEG
NO DECIMAL ± 1.3
ONE PLACE DECIMAL ± .3
TWO PLACE DECIMAL ± .13

125

MATERIAL: SEE NOTES | FINISH: SEE NOTES | DO NOT SCALE | SHEET 2 OF 3

7. DOCUMENT RELEASE NOTES

Release 3.2 – August 4th, 2014

- Updated the Release version to 3.2
- Removed reference to joint_states publishing
- Added documentation on Histogram topic

Release 3.1 – May 10th, 2014

- Added catkin instructions for Groovy and Hydro.
- Updated MultiSense topic names to reflect the new namespaceing scheme.
- Updated transform tree overview to reflect removal of KDL.
- Updated the example transformation code to use TF.
- Updated documentation to support different sensor types (SL, S7, S7S, S21)

Release 3.0_beta – February 14th, 2014

- Added notes and topics for the *3.0_beta* firmware release.
- Added software compatibility section.

Release 2.3 – December 4th, 2013

- Removed dashboard and diagnostic sections.
- Added "MultiSenseUpdater" documentation in place of "FlashUtility."
- Moved "Reconfigurable Parameters" to its own section.
- Added IMU topic, command-line utility, and dynamic_reconfigure sections.
- Added TF Tree overview
- Added IMU section

Release 2.2 – October 17th, 2013

- Added note about PPS bug in version 2.1 firmware.
- Removed deprecation warning about /laser/points2 topic.

Release 2.1 – September 23, 2013

- Added topic description and troubleshooting section for PPS.
- Added section 2.6.5 for the FlashUtility command line tool

Release 2.0 – July 9, 2013

- Added description of command-line utilities to configure sensor heads.
- Updated ROS API documentation.
- Updated low-level communications and wire protocol.
- Added documentation for calibration check program.
- Removed documentation for calibration check service.
- General formatting and cleanup.

Release 1.3 – May 23, 2013

- Added documentation of raw_snapshot tool to multisense_ros.
- Added documentation for temporary calibration check service.

Release 1.1 – April 30, 2013

- Initial public release.

8. ROS DRIVER RELEASE NOTES

Release 3.2 – August 4th, 2014

- Added a histogram topic to publish image histograms for each image channel
- Fixed building with rosbuild under Groovy, Hydro, and Indigo
- Small bug fixes
- Update license from LGPL to BSD

Release 3.1 – May 9th, 2014

- Add support for *3.1* firmware, including S21 product line.
- Add support for catkin build system (Groovy and Hydro.)
- Simplify laser /tf publishing.

Release 3.0_beta – February 14th, 2014

- Add support for *3.0_beta* firmware
    - SGM (semi-global matching) hardware stereo core.
    - Hardware stereo post-filter support and tuning.
    - Support for non-square (2x binned) vertical resolutions.
    - Support for variable disparity levels (64, 128, and 256.)
    - Support for right-disparity (8-bit) and left stereo cost-map (8-bit) images. Please note that in the *3.0_beta* release, these images have not been passed through the left-right-discrepancy and post-stereo filters.
- Add colorized (rgb) point cloud topic (please note that the color image rectification is done on the host CPU, which may reduce the overall throughput of this topic compared to the monochrome version.)
- Add configurable range and edge filtering to point cloud generation.
- Add raw left-disparity image topic (16 bit.) Each LSB is $1/16^{th}$ of a disparity.

Release 2.3 – December 4th, 2013

- Added support for firmware v2.3 (IMU and CMV4000 support.)
- Added smart presentation of dynamic_reconfigure interfaces based off of hardware configuration and sensor firmware version.
- Removed multisense_dashboard and multisense_diagnostics
- Many small bugfixes and feature enhancements.

Release 2.2 – October 17th, 2013

- Add check for firmware version 2.2 before enabling PPS output

Release 2.1 – September 23, 2013

- Added topic for publishing the sensor's clock at the last PPS.
- Added "network-time-sync" option to dynamic-reconfigure.
- Corrected step size for color images (now display correctly with image_view.)

- Corrected projection center in cached camera intrinsics.
- Updated calibration check utility.

Release 2.0 – July 9, 2013

- Added command-line utilities to configure sensor heads.
- Updated ROS API to more closely match DRC Gazebo simulator.
- Updated low-level communications and wire protocol to match MultiSense-SL firmware Release 2.0.
- Added raw_snapshot tool to multisense_ros.
- Added calibration check program.

Release 1.1 – May 1, 2013

- Initial public release.