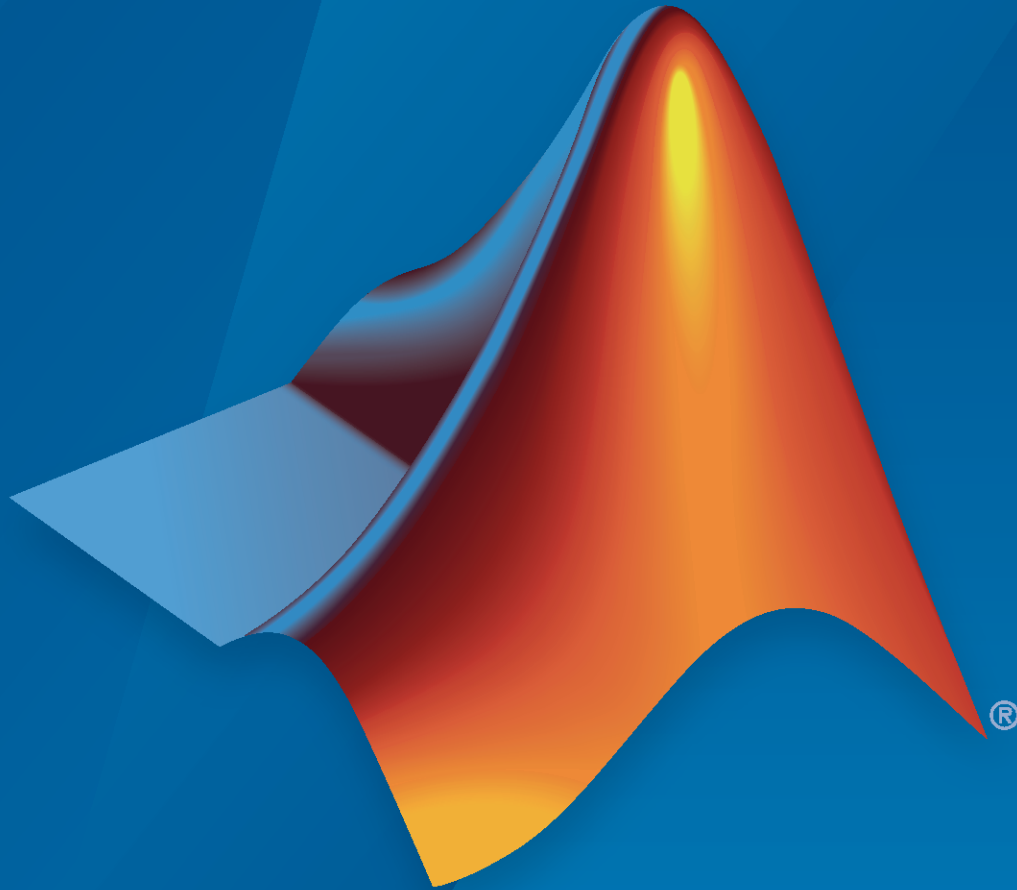


**ROS Toolbox**  
Reference



**MATLAB® & SIMULINK®**

R2020a



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



The MathWorks, Inc.  
1 Apple Hill Drive  
Natick, MA 01760-2098

## *ROS Toolbox Reference*

© COPYRIGHT 2019–2020 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## **Trademarks**

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

## **Patents**

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## **Revision History**

September 2019	Online only	New for Version 1.0 (R2019b)
March 2020	Online only	Rereleased for Version 1.1 (R2020a)

<b>1</b>	<b>Functions</b>
<b>2</b>	<b>Classes</b>
<b>3</b>	<b>Methods</b>
<b>4</b>	<b>Blocks</b>



# Functions

---

## apply

Transform message entities into target frame

### Syntax

```
tfentity = apply(tfmsg,entity)
```

### Description

`tfentity = apply(tfmsg,entity)` applies the transformation represented by the 'TransformStamped' ROS message to the input message object `entity`.

This function determines the message type of `entity` and applies the appropriate transformation method to it. If the object cannot handle a particular message type, then MATLAB® displays an error message.

If you want to use only the most current transformation, call `transform` instead. If you want to store a transformation message for later use, call `getTransform`, and then call `apply`.

### Examples

#### Apply A Transformation To A Point

Connect to a ROS network to get a TransformStamped ROS message. Specify the IP address to connect. Create a transformation tree and get the transformation between desired frames.

```
rosinit('192.168.17.129')
```

```
Initializing global node /matlab_global_node_73610 with NodeURI http://192.168.17.1:55060/
```

```
tftree = rostf;  
pause(1);  
tform = getTransform(tftree,'base_link','camera_link',...  
                    rostime('now'),'Timeout',5);
```

Create a ROS Point message and apply the transformation. You could also get point messages off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');  
pt.Header.FrameId = 'camera_link';  
pt.Point.X = 3;  
pt.Point.Y = 1.5;  
pt.Point.Z = 0.2;
```

```
tfpt = apply(tform,pt);
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_73610 with NodeURI http://192.168.17.1:55060/
```

## Input Arguments

### **tfmsg** – Transformation message

TransformStamped ROS message handle

Transformation message, specified as a TransformStamped ROS message handle. The tfmsg is a ROS message of type: geometry\_msgs/TransformStamped.

### **entity** – ROS message

Message object handle

ROS message, specified as a Message object handle.

Supported messages are:

- geometry\_msgs/PointStamped
- geometry\_msgs/PoseStamped
- geometry\_msgs/PointCloud2Stamped
- geometry\_msgs/QuaternionStamped
- geometry\_msgs/Vector3Stamped

## Output Arguments

### **tfentity** – Transformed ROS message

Message object handle

Transformed ROS message, returned as a Message object handle.

## See Also

getTransform | transform

**Introduced in R2019b**

## call

Call the ROS service server and receive a response

### Syntax

```
response = call(serviceclient)
response = call(serviceclient,requestmsg)
response = call( __ ,Name,Value)
```

### Description

`response = call(serviceclient)` sends a default service request message and waits for a service response. The default service request message is an empty message of type `serviceclient.ServiceType`.

`response = call(serviceclient,requestmsg)` specifies a service request message, `requestmsg`, to be sent to the service.

`response = call( __ ,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments, using any of the arguments from the previous syntaxes.

### Examples

#### Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:50514/.
```

```
Initializing global node /matlab_global_node_67124 with NodeURI http://bat5110win64:50518/
```

Set up a service server and client.

```
server = rossvcserver('/test', 'std_srvs/Empty', @exampleHelperROSEmptyCallback);
client = rossvcclient('/test');
```

Call service server with default message.

```
response = call(client)
```

```
response =
  ROS EmptyResponse message with properties:
```

```
  MessageType: 'std_srvs/EmptyResponse'
```

```
  Use showdetails to show the contents of the message
```

Shut down the ROS network.

```
rosshutdown
```



---

```
Shutting down global node /matlab_global_node_67124 with NodeURI http://bat5110win64:50518/
Shutting down ROS master on http://bat5110win64:50514/.
```

## Call for Response Using Specific Request Message

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:54341/.
Initializing global node /matlab_global_node_28699 with NodeURI http://bat5110win64:54357/
```

Set up a service server and client. This server calculates the sum of two integers and is based on a ROS service tutorial.

```
sumserver = rossvcserver('/sum','roscpp_tutorials/TwoInts',@exampleHelperROSSumCallback);
sumclient = rossvcclient('/sum');
```

Get the request message for the client and modify the parameters.

```
reqMsg = rosmessage(sumclient);
reqMsg.A = 2;
reqMsg.B = 1;
```

Call service and get a response. The response should be the sum of the two integers given in the request message. Wait 5 seconds for the service to time out.

```
response = call(sumclient,reqMsg,'Timeout',5)
```

```
response =
  ROS TwoIntsResponse message with properties:
    MessageType: 'roscpp_tutorials/TwoIntsResponse'
    Sum: 3
```

Use `showdetails` to show the contents of the message

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_28699 with NodeURI http://bat5110win64:54357/
Shutting down ROS master on http://bat5110win64:54341/.
```

## Input Arguments

### **serviceclient** — Service client

`ServiceClient` object handle

Service client, specified as a `ServiceClient` object handle.

### **requestmsg** — Request message

Message object handle

Request message, specified as a `Message` object handle. The default message type is `serviceclient.ServiceType`.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name, Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `"Timeout", 5`

### **Timeout — Timeout for service response in seconds**

`inf` (default) | scalar

Timeout for service response in seconds, specified as a comma-separated pair consisting of `"Timeout"` and a scalar. If the service client does not receive a service response and the timeout period elapses, `call` displays an error message and lets MATLAB continue running the current program. The default value of `inf` prevents MATLAB from running the current program until the service client receives a service response.

### **Output Arguments**

#### **response — Response message**

Message object handle

Response message sent by the service server, returned as a `Message` object handle.

### **See Also**

`rossvcclient`

**Introduced in R2019b**

# cancelAllGoals

Cancel all goals on action server

## Syntax

```
cancelAllGoals(client)
```

## Description

`cancelAllGoals(client)` sends a request from the specified client to the ROS action server to cancel all currently pending or active goals, including goals from other clients.

## Examples

### Send and Cancel ROS Action Goals

This example shows how to send and cancel goals for ROS actions. Action types must be setup beforehand with an action server running.

You must have set up the '/fibonacci' action type. To run this action server, use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

First, set up a ROS action client. Then, send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected to the ROS network using `rosactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.17.129',11311)
```

```
Initializing global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

```
[actClient,goalMsg] = rosactionclient('/fibonacci');
waitForServer(actClient);
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg =
  ROS FibonacciResult message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciResult'
  Sequence: [6x1 int32]
```

Use `showdetails` to show the contents of the message

```
resultState =  
'succeeded'
```

```
showdetails(resultMsg)
```

```
Sequence : [0, 1, 1, 2, 3, 5]
```

Send a new goal message without waiting.

```
goalMsg.Order = 5;  
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

## Input Arguments

### **client** – ROS action client

`SimpleActionClient` object handle

ROS action client, specified as a `SimpleActionClient` object handle. This simple action client enables you to track a single goal at a time.

## See Also

`cancelGoal` | `roaction` | `sendGoal` | `sendGoalAndWait`

### Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

### Introduced in R2019b

# cancelGoal

Cancel last goal sent by client

## Syntax

```
cancelGoal(client)
```

## Description

`cancelGoal(client)` sends a cancel request for the tracked goal, which is the last one sent to the action server. The specified client sends the request.

If the goal is in the 'active' state, the server preempts the execution of the goal. If the goal is 'pending', it is recalled. If this client has not sent a goal, or if the previous goal was achieved, this function returns immediately.

## Examples

### Send and Cancel ROS Action Goals

This example shows how to send and cancel goals for ROS actions. Action types must be setup beforehand with an action server running.

You must have set up the '/fibonacci' action type. To run this action server, use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

First, set up a ROS action client. Then, send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected to the ROS network using `roslaunch`. Specify the action name. Wait for the client to be connected to the server.

```
roslaunch('192.168.17.129', 11311)
```

```
Initializing global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

```
[actClient,goalMsg] = roslaunch('/fibonacci');
waitForServer(actClient);
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg =
  ROS FibonacciResult message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciResult'
```

```
Sequence: [6x1 int32]
```

Use `showdetails` to show the contents of the message

```
resultState =  
'succeeded'
```

```
showdetails(resultMsg)
```

```
Sequence : [0, 1, 1, 2, 3, 5]
```

Send a new goal message without waiting.

```
goalMsg.Order = 5;  
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

## Input Arguments

### **client** – ROS action client

`SimpleActionClient` object handle

ROS action client, specified as a `SimpleActionClient` object handle. This simple action client enables you to track a single goal at a time.

## See Also

`cancelAllGoals` | `roaction` | `sendGoal` | `sendGoalAndWait`

## Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

## Introduced in R2019b

# canTransform

Verify if transformation is available

## Syntax

```
isAvailable = canTransform(tftree, targetframe, sourceframe)
isAvailable = canTransform(tftree, targetframe, sourceframe, sourcetime)

isAvailable = canTransform(bagSel, targetframe, sourceframe)
isAvailable = canTransform(bagSel, targetframe, sourceframe, sourcetime)
```

## Description

`isAvailable = canTransform(tftree, targetframe, sourceframe)` verifies if a transformation between the source frame and target frame is available at the current time in `tftree`. Create the `tftree` object using `rostf`, which requires a connection to a ROS network.

`isAvailable = canTransform(tftree, targetframe, sourceframe, sourcetime)` verifies if a transformation is available for the source time. If `sourcetime` is outside the buffer window, the function returns `false`.

`isAvailable = canTransform(bagSel, targetframe, sourceframe)` verifies if a transformation is available in a rosbag in `bagSel`. To get the `bagSel` input, load a rosbag using `rosbag`.

`isAvailable = canTransform(bagSel, targetframe, sourceframe, sourcetime)` verifies if a transformation is available in a rosbag for the source time. If `sourcetime` is outside the buffer window, the function returns `false`.

## Examples

### Send a Transformation to ROS Network

This example shows how to create a transformation and send it over the ROS network.

Create a ROS transformation tree. Use `rosinit` to connect a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.17.129';
rosinit(ipaddress, 11311)
```

```
Initializing global node /matlab_global_node_33798 with NodeURI http://192.168.17.1:56771/
```

```
tftree = rostf;
pause(2)
```

Verify the transformation you want to send over the network does not already exist. The `canTransform` function returns `false` if the transformation is not immediately available.

```
canTransform(tftree, 'new_frame', 'base_link')
```

```
ans = logical
      0
```

Create a TransformStamped message. Populate the message fields with the transformation information.

```
tform = rosmesssage('geometry_msgs/TransformStamped');
tform.ChildFrameId = 'new_frame';
tform.Header.FrameId = 'base_link';
tform.Transform.Translation.X = 0.5;
tform.Transform.Rotation.Z = 0.75;
```

Send the transformation over the ROS network.

```
sendTransform(tftree,tform)
```

Verify the transformation is now on the ROS network.

```
canTransform(tftree,'new_frame','base_link')
```

```
ans = logical
      1
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_33798 with NodeURI http://192.168.17.1:56771/
```

### **Get ROS Transformations and Apply to ROS Messages**

This example shows how to set up a ROS transformation tree and transform frames based on transformation tree information. It uses time-buffered transformations to access transformations at different times.

Create a ROS transformation tree. Use `rosinit` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.17.129';
rosinit(ipaddress,11311)
```

```
Initializing global node /matlab_global_node_14346 with NodeURI http://192.168.17.1:56312/
```

```
tftree = rostf;
pause(1)
```

Look at the available frames on the transformation tree.

```
tftree.AvailableFrames
```

```
ans = 36x1 cell
      {'base_footprint'      }
      {'base_link'          }
      {'camera_depth_frame' }
```



```

{'camera_depth_optical_frame'}
{'camera_link' }
{'camera_rgb_frame' }
{'camera_rgb_optical_frame' }
{'caster_back_link' }
{'caster_front_link' }
{'cliff_sensor_front_link' }
{'cliff_sensor_left_link' }
{'cliff_sensor_right_link' }
{'gyro_link' }
{'mount_asus_xtion_pro_link' }
{'odom' }
{'plate_bottom_link' }
{'plate_middle_link' }
{'plate_top_link' }
{'pole_bottom_0_link' }
{'pole_bottom_1_link' }
{'pole_bottom_2_link' }
{'pole_bottom_3_link' }
{'pole_bottom_4_link' }
{'pole_bottom_5_link' }
{'pole_kinect_0_link' }
{'pole_kinect_1_link' }
{'pole_middle_0_link' }
{'pole_middle_1_link' }
{'pole_middle_2_link' }
{'pole_middle_3_link' }
:

```

Check if the desired transformation is now available. For this example, check for the transformation from 'camera\_link' to 'base\_link'.

```
canTransform(tftree, 'base_link', 'camera_link')
```

```
ans = logical
      1
```

Get the transformation for 3 seconds from now. The `getTransform` function will wait until the transformation becomes available with the specified timeout.

```
desiredTime = rostime('now') + 3;
tform = getTransform(tftree, 'base_link', 'camera_link', ...
                    desiredTime, 'Timeout', 5);
```

Create a ROS message to transform. Messages can also be retrieved off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_link';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Transform the ROS message to the 'base\_link' frame using the desired time previously saved.

```
tfpt = transform(tftree, 'base_link', pt, desiredTime);
```

*Optional:* You can also use `apply` with the stored `tform` to apply this transformation to the `pt` message.

```
tfpt2 = apply(tform,pt);
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_14346 with NodeURI http://192.168.17.1:56312/
```

## Get Transformations from rosbag File

Get transformations from rosbag (`.bag`) files by loading the rosbag and checking the available frames. From these frames, use `getTransform` to query the transformation between two coordinate frames.

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Get a list of available frames.

```
frames = bag.AvailableFrames;
```

Get the latest transformation between two coordinate frames.

```
tf = getTransform(bag, 'world', frames{1});
```

Check for a transformation available at a specific time and retrieve the transformation. Use `canTransform` to check if the transformation is available. Specify the time using `rostime`.

```
tfTime = rostime(bag.StartTime + 1);  
if (canTransform(bag, 'world', frames{1}, tfTime))  
    tf2 = getTransform(bag, 'world', frames{1}, tfTime);  
end
```

## Input Arguments

### **tfTree** — ROS transformation tree

`TransformationTree` object handle

ROS transformation tree, specified as a `TransformationTree` object handle. Create a transformation tree by calling the `rostopic` function.

### **bagSel** — Selection of rosbag messages

`BagSelection` object handle

Selection of rosbag messages, specified as a `BagSelection` object handle. To create a selection of rosbag messages, use `rosbag`.

### **targetFrame** — Target coordinate frame

string scalar | character vector



## definition

Retrieve definition of ROS message type

### Syntax

```
def = definition(msg)
```

### Description

`def = definition(msg)` returns the ROS definition of the message type associated with the message object, `msg`. The details of the message definition include the structure, property data types, and comments from the authors of that specific message.

### Examples

#### Access ROS Message Definition for Message

Create a Point Message.

```
point = rosmessage('geometry_msgs/Point');
```

Access the definition.

```
def = definition(point)

def =
  '% This contains the position of a point in free space
  double X
  double Y
  double Z
  '
```

### Input Arguments

#### **msg** — ROS message

Message object handle

ROS message, specified as a Message object handle. This message can be created using the `rosmessage` function.

### Output Arguments

#### **def** — Details of message definition

character vector

Details of the information inside the ROS message definition, returned as a character vector.

## **See Also**

rosmesssage | rosmmsg

**Introduced in R2019b**

## del

Delete a ROS parameter

### Syntax

```
del(ptree,paramname)
del(ptree,namespace)
```

### Description

`del(ptree,paramname)` deletes a parameter with name `paramname` from the parameter tree, `ptree`. The parameter is also deleted from the ROS parameter server. If the specified `paramname` does not exist, the function displays an error.

`del(ptree,namespace)` deletes from the parameter tree all parameter values under the specified namespace.

### Examples

#### Delete Parameter on ROS Master

Connect to the ROS network. Create a parameter tree and a 'MyParam' parameter. Check that the parameter exists.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:49221/.
Initializing global node /matlab_global_node_00002 with NodeURI http://bat5110win64:49229/
```

```
ptree = rosparam;
set(ptree,'MyParam','test')
has(ptree,'MyParam')
```

```
ans = logical
      1
```

Delete the parameter. Verify it was deleted. Shut down the ROS network.

```
del(ptree,'MyParam')
has(ptree,'MyParam')
```

```
ans = logical
      0
```

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_00002 with NodeURI http://bat5110win64:49229/
Shutting down ROS master on http://bat5110win64:49221/.
```

---

## Input Arguments

**ptree — Parameter tree**

ParameterTree object handle

Parameter tree, specified as a ParameterTree object handle. Create this object using the `rosparam` function.

**paramname — ROS parameter name**

string scalar | character vector

ROS parameter name, specified as a string scalar or character vector. This string must match the parameter name exactly.

**namespace — ROS parameter namespace**

string scalar | character vector

ROS parameter namespace, specified as a string scalar or character vector. All parameter names starting with this string are listed when calling `rosparam("list", namespace)`.

## See Also

`has` | `rosparam` | `set`

**Introduced in R2019b**

## deleteFile

Delete file from device

### Syntax

```
deleteFile(device, filename)
```

### Description

`deleteFile(device, filename)` deletes the specified file from the ROS device.

### Examples

#### Put, Get, and Delete Files on ROS Device

Put a file from your host computer onto a ROS device, get it back, and then delete it.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.17.128', 'user', 'password');
```

Put a new text file that is in the MATLAB(R) current folder onto the ROS device. The destination folder must exist.

```
putFile(d, 'test_file.txt', '/home/user/test_folder')
```

Get a text file from the ROS device. You can get any file, not just ones added from MATLAB(R). By default, the file is added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/test_file.txt')
```

Delete the text file on the ROS device.

```
deleteFile(d, '/home/user/test_folder/test_file.txt')
```

#### Put, Get, and Delete Files on ROS Device Using Wildcards

Put a file from your host computer onto a ROS device, get it back, and then delete it. Use wildcards to search for all matching files.

**Note:** You must have a valid ROS device to connect to at the IP address specified in the example.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.17.128', 'user', 'password');
```

Put all text files at the specified path onto the ROS device. The destination folder must exist.

```
putFile(d, 'C:/MATLAB/*.txt', '/home/user/test_folder')
```



Get all text files from the ROS device. You can get any files, not just ones added from MATLAB(R). By default, the files are added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/*.txt')
```

Delete all text files on the ROS device at the specified folder.

```
deleteFile(d, '/home/user/test_folder/*.txt')
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **filename** — File to delete

character vector

File to delete, specified as a character vector. When you specify the file name, you can use path information and wildcards.

Example: '/home/user/image.jpg'

Example: '/home/user/\*.jpg'

Data Types: cell

## See Also

dir | getFile | openShell | putFile | rosdevice | system

**Introduced in R2019b**

## dir

List folder contents on device

### Syntax

```
dir(device, folder)
clist = dir(device, folder)
```

### Description

`dir(device, folder)` lists the files in a folder on the ROS device. Wildcards are supported.

`clist = dir(device, folder)` stores the list of files as a structure.

### Examples

#### View Folder Contents on ROS Device

Connect to a ROS device and list the contents of a folder.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.17.129', 'user', 'password');
```

Get the folder list of a Catkin workspace on your ROS device. View the folder as a table.

```
flist = dir(d, '/home/user/Documents/mw_catkin_ws/');
ftable = struct2table(flist)
```

```
ftable=6x4 table
      name                folder                isdir  bytes
-----
{'.'}                    {'/home/user/Documents/mw_catkin_ws'} true    0
{'..'}                   {'/home/user/Documents/mw_catkin_ws'} true    0
{'catkin_workspace'}    {'/home/user/Documents/mw_catkin_ws'} false   98
{'build'}                {'/home/user/Documents/mw_catkin_ws'} true    0
{'devel'}                 {'/home/user/Documents/mw_catkin_ws'} true    0
{'src'}                   {'/home/user/Documents/mw_catkin_ws'} true    0
```

### Input Arguments

#### device — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

#### folder — Folder name

character vector

---

Name of the folder to list the contents of, specified as a character vector.

## Output Arguments

### **cList** — Contents list

structure

Contents list, returned as a structure. The structure contains these fields:

- `name` — File name (char)
- `folder` — Absolute path (char)
- `bytes` — Size of the file in bytes (double)
- `isdir` — Indicator of whether name is a folder (logical)

### See Also

`deleteFile` | `getFile` | `openShell` | `putFile` | `rosdevice` | `system`

**Introduced in R2019b**

## get

Get ROS parameter value

### Syntax

```
pvalue = get(ptree)
pvalue = get(ptree,paramname)
pvalue = get(ptree,namespace)
```

### Description

`pvalue = get(ptree)` returns a dictionary of parameter values under the root namespace: `/`. The dictionary is stored in a structure.

`pvalue = get(ptree,paramname)` gets the value of the parameter with the name `paramname` from the parameter tree object `ptree`.

`pvalue = get(ptree,namespace)` returns a dictionary of parameter values under the specified namespace.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

- 32-bit integer — `int32`
- Boolean — `logical`
- double — `double`
- strings — string scalar, `string`, or character vector, `char`
- list — cell array (`cell`)
- dictionary — structure (`struct`)

### Examples

#### Set and Get Parameter Value

Create the parameter tree. A ROS network must be available using `rosinit`.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:56948/.
```

```
Initializing global node /matlab_global_node_90050 with NodeURI http://bat5110win64:56952/
```

```
ptree = rosparam;
```

Set a parameter value. You can also use the simplified version without a parameter tree:

```
rosparam set 'DoubleParam' 1.0
```

```
set(ptree, 'DoubleParam', 1.0)
```

Get the parameter value.

```
get(ptree, 'DoubleParam')
ans = 1
```

Alternatively, use the simplified versions without using the parameter tree.

```
rosparam set 'DoubleParam' 2.0
rosparam get 'DoubleParam'
2
```

Disconnect from ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_90050 with NodeURI http://bat5110win64:56952/
Shutting down ROS master on http://bat5110win64:56948/.
```

## Input Arguments

### **ptree** — Parameter tree

ParameterTree object handle

Parameter tree, specified as a ParameterTree object handle. Create this object using the `rosparam` function.

### **paramname** — ROS parameter name

string scalar | character vector

ROS parameter name, specified as a character vector. This string must match the parameter name exactly.

### **namespace** — ROS parameter namespace

string scalar | character vector

ROS parameter namespace, specified as a string scalar or character vector. All parameter names starting with this string are listed when calling `rosparam("list", namespace)`.

## Output Arguments

### **pvalue** — ROS parameter value or dictionary of values

int32 | logical | double | character vector | cell array | structure

ROS parameter value, returned as a supported MATLAB data type. When specifying the `namespace` input argument, `pvalue` is returned as a dictionary of parameter values under the specified namespace. The dictionary is represented in MATLAB as a structure.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

- 32-bit integer — `int32`
- Boolean — `logical`

- `double` — `double`
- `string` — character vector (`char`)
- `list` — cell array (`cell`)
- `dictionary` — structure (`struct`)

## Limitations

Base64-encoded binary data and iso 8601 data from ROS are not supported.

## See Also

`rosparam` | `set`

**Introduced in R2019b**

# getFile

Get file from device

## Syntax

```
getFile(device,remoteSource)
getFile(device,remoteSource,localDestination)
```

## Description

`getFile(device,remoteSource)` copies the specified file from the ROS device to the MATLAB current folder. Wildcards are supported.

`getFile(device,remoteSource,localDestination)` copies the remote file to a destination path. Specify a file name at the end of the destination path to copy with a custom file name.

## Examples

### Put, Get, and Delete Files on ROS Device

Put a file from your host computer onto a ROS device, get it back, and then delete it.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.17.128','user','password');
```

Put a new text file that is in the MATLAB(R) current folder onto the ROS device. The destination folder must exist.

```
putFile(d,'test_file.txt','/home/user/test_folder')
```

Get a text file from the ROS device. You can get any file, not just ones added from MATLAB(R). By default, the file is added to the MATLAB current folder.

```
getFile(d,'/home/user/test_folder/test_file.txt')
```

Delete the text file on the ROS device.

```
deleteFile(d,'/home/user/test_folder/test_file.txt')
```

### Put, Get, and Delete Files on ROS Device Using Wildcards

Put a file from your host computer onto a ROS device, get it back, and then delete it. Use wildcards to search for all matching files.

**Note:** You must have a valid ROS device to connect to at the IP address specified in the example.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.17.128', 'user', 'password');
```

Put all text files at the specified path onto the ROS device. The destination folder must exist.

```
putFile(d, 'C:/MATLAB/*.txt', '/home/user/test_folder')
```

Get all text files from the ROS device. You can get any files, not just ones added from MATLAB(R). By default, the files are added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/*.txt')
```

Delete all text files on the ROS device at the specified folder.

```
deleteFile(d, '/home/user/test_folder/*.txt')
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **remoteSource** — Path and name of file on ROS device

source path

Path and name of the file on the ROS device. Specify the path as a character vector. You can use an absolute path or a relative path from the MATLAB current folder. Use the path and file naming conventions of the operating system on your host computer.

Example: '/home/user/test\_folder/test\_file.txt'

Data Types: char

### **localDestination** — Destination folder path and optional file name

character vector

Destination folder path and optional file name, specified as a character vector. Specify a file name at the end of the destination path to copy with a custom file name. Use the host computer path and file naming conventions.

Example: 'C:/User/username/test\_folder'

Data Types: char

## See Also

[deleteFile](#) | [dir](#) | [openShell](#) | [putFile](#) | [rosdevice](#) | [system](#)

**Introduced in R2019b**



# getTransform

Retrieve transformation between two coordinate frames

## Syntax

```
tf = getTransform(tftree, targetframe, sourceframe)
tf = getTransform(tftree, targetframe, sourceframe, sourcetime)
tf = getTransform( ____, "Timeout", timeout)

tf = getTransform(bagSel, targetframe, sourceframe)
tf = getTransform(bagSel, targetframe, sourceframe, sourcetime)
```

## Description

`tf = getTransform(tftree, targetframe, sourceframe)` returns the latest known transformation between two coordinate frames in `tftree`. Create the `tftree` object using `rostopic`, which requires a connection to a ROS network.

Transformations are structured as a 3-D translation (three-element vector) and a 3-D rotation (quaternion).

`tf = getTransform(tftree, targetframe, sourceframe, sourcetime)` returns the transformation from the `tftree` at the given source time. If the transformation is not available at that time, an error is displayed.

`tf = getTransform( ____, "Timeout", timeout)` also specifies a timeout period, in seconds, to wait for the transformation to be available. If the transformation does not become available in the timeout period, the function returns an error. This option can be combined with the previous syntaxes.

`tf = getTransform(bagSel, targetframe, sourceframe)` returns the latest transformation between two frames in the rosbag in `bagSel`. To get the `bagSel` input, load a rosbag using `roslaunch`.

`tf = getTransform(bagSel, targetframe, sourceframe, sourcetime)` returns the transformation at the specified `sourcetime` in the rosbag in `bagSel`.

## Examples

### Get ROS Transformations and Apply to ROS Messages

This example shows how to set up a ROS transformation tree and transform frames based on transformation tree information. It uses time-buffered transformations to access transformations at different times.

Create a ROS transformation tree. Use `roslaunch` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.17.129';
roslaunch(ipaddress, 11311)
```

Initializing global node /matlab\_global\_node\_14346 with NodeURI http://192.168.17.1:56312/

```
tftree = rostf;  
pause(1)
```

Look at the available frames on the transformation tree.

```
tftree.AvailableFrames
```

```
ans = 36x1 cell  
    {'base_footprint'      }  
    {'base_link'          }  
    {'camera_depth_frame' }  
    {'camera_depth_optical_frame'}  
    {'camera_link'        }  
    {'camera_rgb_frame'   }  
    {'camera_rgb_optical_frame' }  
    {'caster_back_link'   }  
    {'caster_front_link'  }  
    {'cliff_sensor_front_link' }  
    {'cliff_sensor_left_link' }  
    {'cliff_sensor_right_link' }  
    {'gyro_link'          }  
    {'mount_asus_xtion_pro_link' }  
    {'odom'               }  
    {'plate_bottom_link'  }  
    {'plate_middle_link'  }  
    {'plate_top_link'     }  
    {'pole_bottom_0_link' }  
    {'pole_bottom_1_link' }  
    {'pole_bottom_2_link' }  
    {'pole_bottom_3_link' }  
    {'pole_bottom_4_link' }  
    {'pole_bottom_5_link' }  
    {'pole_kinect_0_link' }  
    {'pole_kinect_1_link' }  
    {'pole_middle_0_link' }  
    {'pole_middle_1_link' }  
    {'pole_middle_2_link' }  
    {'pole_middle_3_link' }  
    :  
    :
```

Check if the desired transformation is now available. For this example, check for the transformation from 'camera\_link' to 'base\_link'.

```
canTransform(tftree, 'base_link', 'camera_link')
```

```
ans = logical  
     1
```

Get the transformation for 3 seconds from now. The `getTransform` function will wait until the transformation becomes available with the specified timeout.

```
desiredTime = rostime('now') + 3;  
tform = getTransform(tftree, 'base_link', 'camera_link', ...  
                    desiredTime, 'Timeout', 5);
```

Create a ROS message to transform. Messages can also be retrieved off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_link';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Transform the ROS message to the 'base\_link' frame using the desired time previously saved.

```
tfpt = transform(tftree, 'base_link', pt, desiredTime);
```

*Optional:* You can also use `apply` with the stored `tform` to apply this transformation to the `pt` message.

```
tfpt2 = apply(tform, pt);
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_14346 with NodeURI http://192.168.17.1:56312/
```

### Get Buffered Transformations from ROS Network

This example shows how to access time-buffered transformations on the ROS network. Access transformations for specific times and modify the `BufferTime` property based on your desired times.

Create a ROS transformation tree. Use `rosinit` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.17.129';
rosinit(ipaddress, 11311)
```

```
Initializing global node /matlab_global_node_78006 with NodeURI http://192.168.17.1:56344/
```

```
tftree = rostf;
pause(2);
```

Get the transformation from 1 second ago.

```
desiredTime = rostime('now') - 1;
tform = getTransform(tftree, 'base_link', 'camera_link', desiredTime);
```

The transformation buffer time is 10 seconds by default. Modify the `BufferTime` property of the transformation tree to increase the buffer time and wait for that buffer to fill.

```
tftree.BufferTime = 15;
pause(15);
```

Get the transformation from 12 seconds ago.

```
desiredTime = rostime('now') - 12;
tform = getTransform(tftree, 'base_link', 'camera_link', desiredTime);
```

You can also get transformations at a time in the future. The `getTransform` function will wait until the transformation is available. You can also specify a timeout to error if no transformation is found. This example waits 5 seconds for the transformation at 3 seconds from now to be available.

```
desiredTime = rostime('now') + 3;  
tform = getTransform(tftree, 'base_link', 'camera_link', desiredTime, 'Timeout', 5);
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_78006 with NodeURI http://192.168.17.1:56344/
```

### Get Transformations from rosbag File

Get transformations from rosbag (.bag) files by loading the rosbag and checking the available frames. From these frames, use `getTransform` to query the transformation between two coordinate frames.

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Get a list of available frames.

```
frames = bag.AvailableFrames;
```

Get the latest transformation between two coordinate frames.

```
tf = getTransform(bag, 'world', frames{1});
```

Check for a transformation available at a specific time and retrieve the transformation. Use `canTransform` to check if the transformation is available. Specify the time using `rostime`.

```
tftime = rostime(bag.StartTime + 1);  
if (canTransform(bag, 'world', frames{1}, tftime))  
    tf2 = getTransform(bag, 'world', frames{1}, tftime);  
end
```

## Input Arguments

### **tftree** – ROS transformation tree

`TransformationTree` object handle

ROS transformation tree, specified as a `TransformationTree` object handle. You can create a transformation tree by calling the `rostopic` function.

### **bagSel** – Selection of rosbag messages

`BagSelection` object handle

Selection of rosbag messages, specified as a `BagSelection` object handle. To create a selection of rosbag messages, use `rostopic`.

**targetframe — Target coordinate frame**

string scalar | character vector

Target coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation by calling `tftree.AvailableFrames`.

**sourceframe — Initial coordinate frame**

string scalar | character vector

Initial coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation by calling `tftree.AvailableFrames`.

**sourcetime — ROS or system time**

Time object handle

ROS or system time, specified as a Time object handle. By default, `sourcetime` is the ROS simulation time published on the `clock` topic. If the `use_sim_time` ROS parameter is set to `true`, `sourcetime` returns the system time. You can create a Time object using `rostime`.

**timeout — Timeout for receiving transform**

0 (default) | scalar in seconds

Timeout for receiving transform, specified as a scalar in seconds. The function returns an error if the timeout is reached and no transform becomes available.

## Output Arguments

**tf — Transformation between coordinate frames**

TransformStamped object handle

Transformation between coordinate frames, returned as a TransformStamped object handle. Transformations are structured as a 3-D translation (three-element vector) and a 3-D rotation (quaternion).

## Compatibility Considerations

**Empty Transforms***Behavior change in future release*

The behavior of `getTransform` changed in R2018a. When using the `tftree` input argument, the function no longer returns an empty transform when the transform is unavailable and no `sourcetime` is specified. If `getTransform` waits for the specified timeout period and the transform is still not available, the function returns an error. The timeout period is 0 by default.

## See Also

`canTransform` | `roscpp` | `rostopic` | `transform` | `waitForTransform`**Introduced in R2019b**

## has

Check if ROS parameter name exists

### Syntax

```
exists = has(ptree,paramname)
```

### Description

`exists = has(ptree,paramname)` checks if the parameter with name `paramname` exists in the parameter tree, `ptree`.

### Examples

#### Check If ROS Parameter Exists

Connect to a ROS network. Create a parameter tree and check for the 'MyParam' parameter.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:57860/.
```

```
Initializing global node /matlab_global_node_36579 with NodeURI http://bat5110win64:57867/
```

```
ptree = rosparam;  
has(ptree, 'MyParam')
```

```
ans = logical  
     0
```

Set the 'MyParam' parameter and verify it exists. Disconnect from ROS network.

```
set(ptree, 'MyParam', 'test')  
has(ptree, 'MyParam')
```

```
ans = logical  
     1
```

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_36579 with NodeURI http://bat5110win64:57867/  
Shutting down ROS master on http://bat5110win64:57860/.
```

### Input Arguments

#### **ptree** — Parameter tree

ParameterTree object handle

Parameter tree, specified as a ParameterTree object handle. Create this object using the `rosparam` function.

**paramname — ROS parameter name**

string scalar | character vector

ROS parameter name, specified as a string scalar or character vector. This string must match the parameter name exactly.

**Output Arguments****exists — Flag indicating whether the parameter exists**

true | false

Flag indicating whether the parameter exists, returned as true or false.

**See Also**

get | rosparam | search | set

**Introduced in R2019b**

## isCoreRunning

Determine if ROS core is running

### Syntax

```
running = isCoreRunning(device)
```

### Description

`running = isCoreRunning(device)` determines if the ROS core is running on the connected device.

### Examples

#### Run ROS Core on ROS Device

Connect to a remote ROS device and start a ROS core. The ROS core is needed to run ROS nodes to communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.17.128';  
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =  
  rosdevice with properties:  
  
    DeviceAddress: '192.168.17.128'  
    Username: 'user'  
    ROSFolder: '/opt/ros/indigo'  
    CatkinWorkspace: '~/catkin_ws'  
    AvailableNodes: {0x1 cell}
```

Run a ROS core and check if it is running.

```
runCore(d)
```

Another `roscore` / ROS master is already running on the ROS device. Use the `'stopCore'` function to

```
running = isCoreRunning(d)
```

```
running = logical  
         1
```

Stop the ROS core and confirm that it is no longer running.



```
stopCore(d)
running = isCoreRunning(d)

running = logical
    1
```

## Input Arguments

### **device – ROS device**

rosdevice object

ROS device, specified as a rosdevice object.

## Output Arguments

### **running – Status of whether ROS core is running**

true | false

Status of whether ROS core is running, returned as true or false.

## See Also

rosdevice | runCore | stopCore

## Topics

“Generate a Standalone ROS Node from Simulink®”

## Introduced in R2019b

## isNodeRunning

Determine if ROS node is running

### Syntax

```
running = isNodeRunning(device,modelName)
```

### Description

`running = isNodeRunning(device,modelName)` determines if the ROS node associated with the specified Simulink® model is running on the specified `rosdevice`, `device`.

### Examples

#### Run ROS Node on ROS Device

Connect to a remote ROS device and start a ROS node. Run a ROS core so that ROS nodes can communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device already contains the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.203.129';
d = rosdevice(ipaddress,'user','password');
d.ROSFolder = '/opt/ros/indigo';
d.CatkinWorkspace = '~/catkin_ws_test'

d =
  rosdevice with properties:

    DeviceAddress: '192.168.203.129'
    Username: 'user'
    ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws_test'
    AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. Connect MATLAB® to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

```
runCore(d)
rosinit(d.DeviceAddress,11311)
```

```
Initializing global node /matlab_global_node_84497 with NodeURI http://192.168.203.1:56034/
```

Check the available ROS nodes on the connected ROS device. These nodes listed were generated from Simulink® models following the process in the “Get Started with ROS in Simulink®” example.

```
d.AvailableNodes
```

```
ans = 1x2 cell
      {'robotcontroller'}    {'robotcontroller2'}
```

Run a ROS node and specify the node name. Check if the node is running.

```
runNode(d, 'RobotController')
running = isNodeRunning(d, 'RobotController')

running = logical
         1
```

Stop the ROS node. Disconnect from the ROS network. Stop the ROS core.

```
stopNode(d, 'RobotController')
rosshutdown
```

```
Shutting down global node /matlab_global_node_84497 with NodeURI http://192.168.203.1:56034/
stopCore(d)
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **modelName** — Name of the deployed Simulink model

character vector

Name of the deployed Simulink model, specified as a character vector. If the model name is not valid, the function returns false.

## Output Arguments

### **running** — Status of whether ROS node is running

true | false

Status of whether ROS node is running, returned as true or false.

## See Also

rosdevice | runNode | stopNode

## Topics

“Generate a Standalone ROS Node from Simulink®”

## Introduced in R2019b

## openShell

Open interactive command shell to device

### Syntax

```
openShell(device)
```

### Description

`openShell(device)` opens an SSH terminal on your host computer that provides encrypted access to the Linux<sup>®</sup> command shell on the ROS device. When prompted, enter a user name and password.

### Examples

#### Open Command Shell on ROS Device

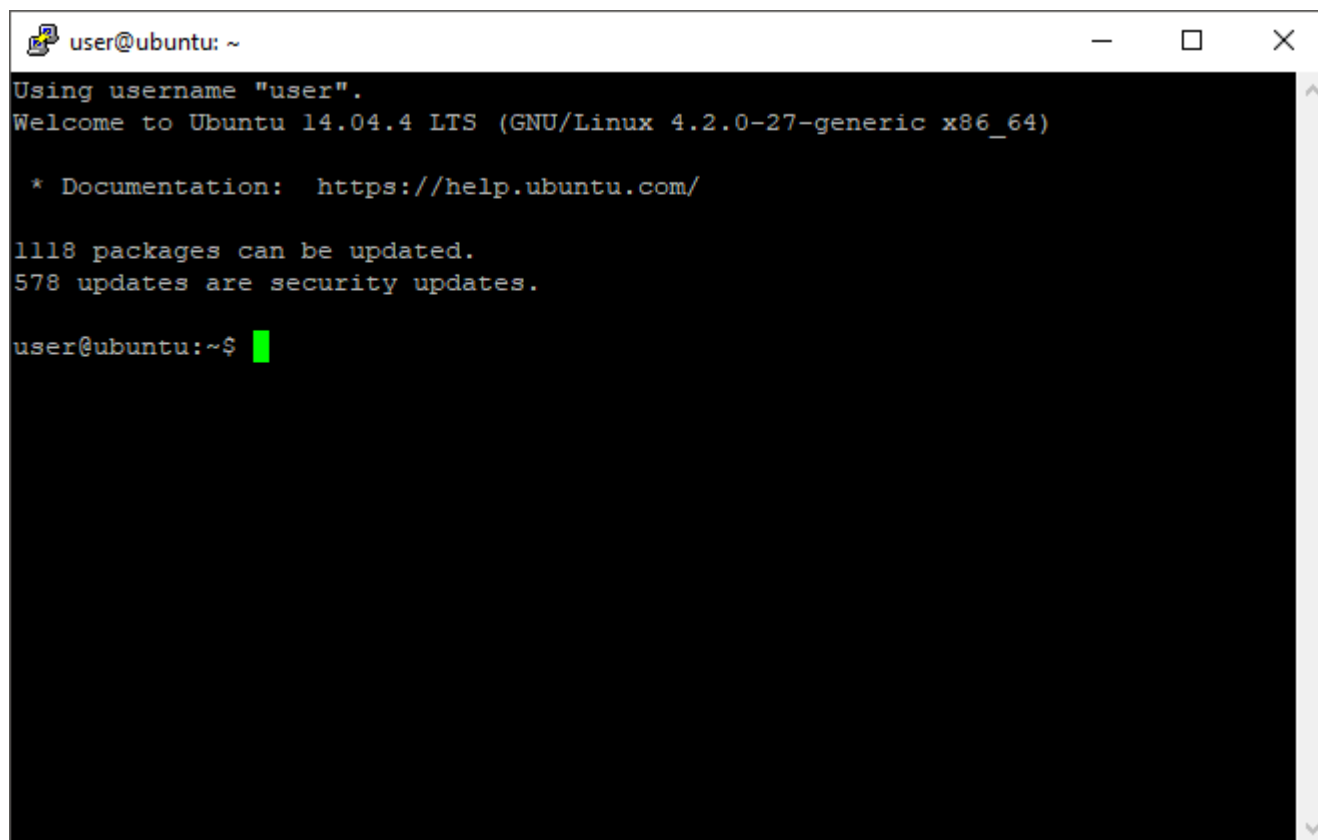
Connect to a ROS device and open the command shell on your host computer.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.17.128', 'user', 'password');
```

Open the command shell.

```
openShell(d);
```

A terminal window titled "user@ubuntu: ~" with standard window controls. The output shows a login message for user "user", the Ubuntu version (14.04.4 LTS), and the Linux kernel version (4.2.0-27-generic x86\_64). It also displays the Ubuntu documentation URL and a notification that 1118 packages can be updated, with 578 of them being security updates. The prompt "user@ubuntu:~\$" is followed by a green cursor.

```
user@ubuntu: ~  
Using username "user".  
Welcome to Ubuntu 14.04.4 LTS (GNU/Linux 4.2.0-27-generic x86_64)  
  
* Documentation:  https://help.ubuntu.com/  
  
1118 packages can be updated.  
578 updates are security updates.  
  
user@ubuntu:~$ █
```

## Input Arguments

### **device** – ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **See Also**

deleteFile | dir | getFile | putFile | rosdevice | system

**Introduced in R2019b**

## plot

Display laser or lidar scan readings

### Syntax

```
plot(scanMsg)
plot(scanObj)
plot(___,Name,Value)
linehandle = plot(___)
```

### Description

`plot(scanMsg)` plots the laser scan readings specified in the input `LaserScan` object message. Axes are automatically scaled to the maximum range that the laser scanner supports.

`plot(scanObj)` plots the lidar scan readings specified in `scanObj`.

`plot(___,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments.

`linehandle = plot(___)` returns a column vector of line series handles, using any of the arguments from previous syntaxes. Use `linehandle` to modify properties of the line series after it is created.

When plotting ROS laser scan messages, MATLAB follows the standard ROS convention for axis orientation. This convention states that **positive x is forward, positive y is left, and positive z is up**. For more information, see [Axis Orientation on the ROS Wiki](#).

### Examples

#### Plot Laser Scan Message

Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

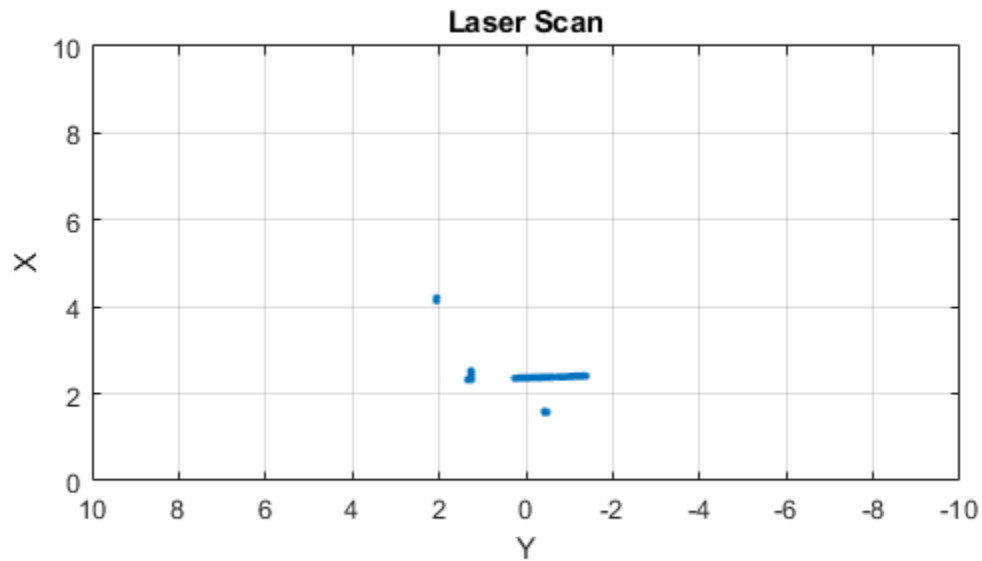
```
rosinit('192.168.17.129')
```

```
Initializing global node /matlab_global_node_90279 with NodeURI http://192.168.17.1:50889/
```

```
sub = rossubscriber('/scan');
scan = receive(sub);
```

Plot the laser scan.

```
plot(scan)
```



Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_90279 with NodeURI http://192.168.17.1:50889/
```

### Plot Laser Scan Message With Maximum Range

Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

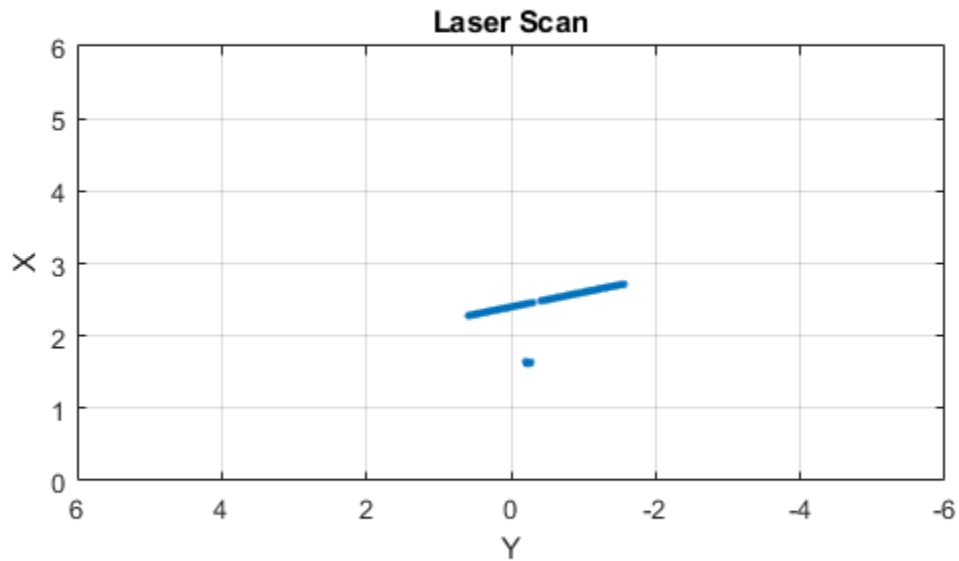
```
rosinit('192.168.17.129')
```

```
Initializing global node /matlab_global_node_31712 with NodeURI http://192.168.17.1:51463/
```

```
sub = rossubscriber('/scan');  
scan = receive(sub);
```

Plot the laser scan specifying the maximum range.

```
plot(scan, 'MaximumRange', 6)
```



Shutdown ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_31712 with NodeURI http://192.168.17.1:51463/
```

### Plot Lidar Scan and Remove Invalid Points

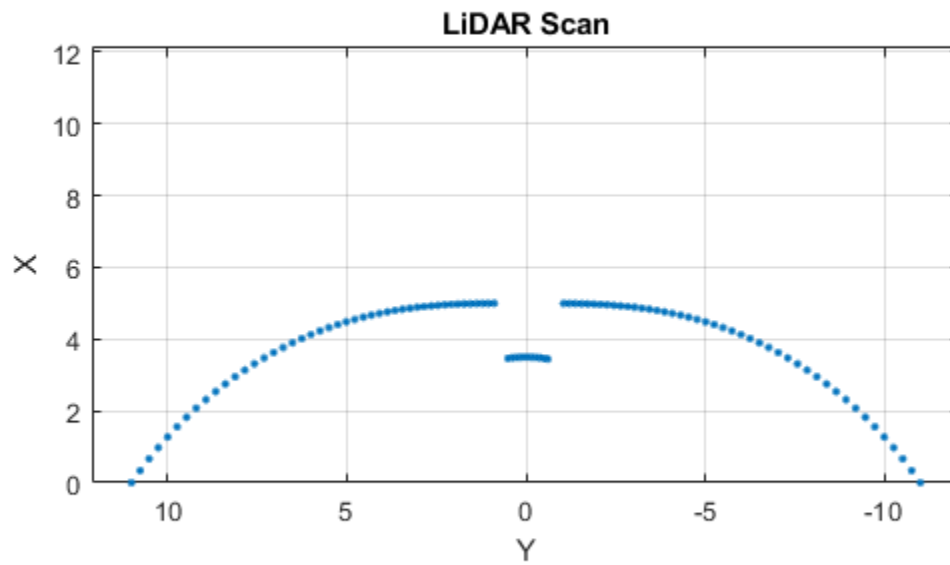
Specify lidar data as vectors of ranges and angles. These values include readings outside of the sensor range.

```
x = linspace(-2,2);  
ranges = abs((1.5).*x.^2 + 5);  
ranges(45:55) = 3.5;  
angles = linspace(-pi/2,pi/2,numel(ranges));
```

Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

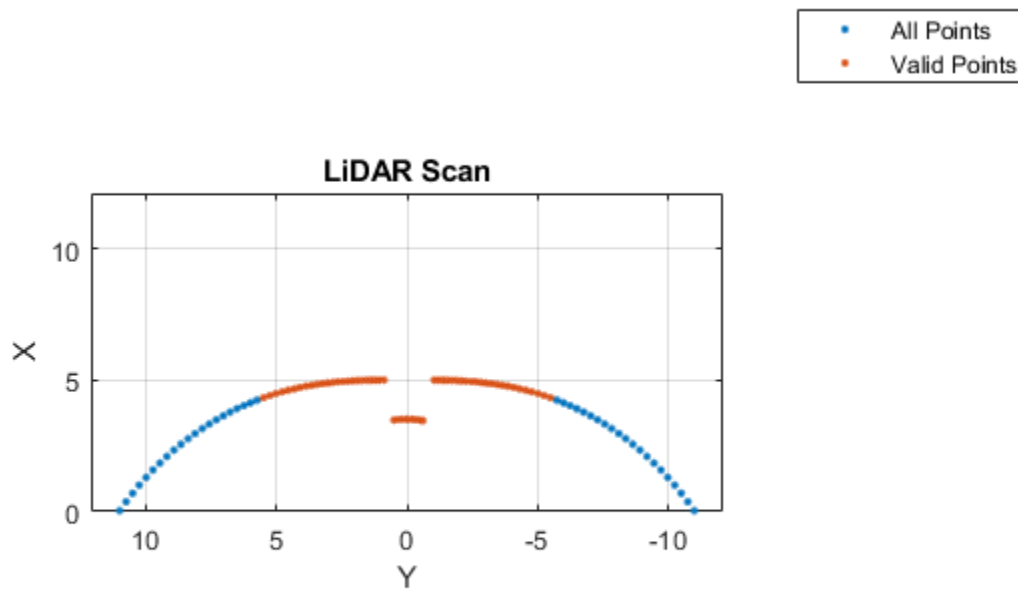
```
scan = lidarScan(ranges,angles);  
plot(scan)
```





Remove invalid points based on a specified minimum and maximum range.

```
minRange = 0.1;  
maxRange = 7;  
scan2 = removeInvalidData(scan, 'RangeLimits', [minRange maxRange]);  
hold on  
plot(scan2)  
legend('All Points', 'Valid Points')
```



## Input Arguments

### **scanMsg** — Laser scan message

LaserScan object handle

sensor\_msgs/LaserScan ROS message, specified as a LaserScan object handle.

### **scanObj** — Lidar scan readings

lidarScan object

Lidar scan readings, specified as a lidarScan object.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: "MaximumRange", 5

### **Parent** — Parent of axes

axes object

Parent of axes, specified as the comma-separated pair consisting of "Parent" and an axes object in which the laser scan is drawn. By default, the laser scan is plotted in the currently active axes.

**MaximumRange — Range of laser scan**`scan.RangeMax (default) | scalar`

Range of laser scan, specified as the comma-separated pair consisting of "MaximumRange" and a scalar. When you specify this name-value pair argument, the minimum and maximum x-axis and the maximum y-axis limits are set based on a specified value. The minimum y-axis limit is automatically determined by the opening angle of the laser scanner.

This name-value pair works only when you input `scanMsg` as the laser scan.

**Outputs****linehandle — One or more chart line objects**`scalar | vector`

One or more chart line objects, returned as a scalar or a vector. These are unique identifiers, which you can use to query and modify properties of a specific chart line.

**See Also**`readCartesian`**Introduced in R2019b**

## putFile

Copy file to device

### Syntax

```
putFile(device,localSource)
putFile(device,localSource,remoteDestination)
```

### Description

`putFile(device,localSource)` copies the specified source file from the MATLAB current folder to the print working directory (pwd) on the ROS device. Wildcards are supported.

`putFile(device,localSource,remoteDestination)` copies the file to a destination path. Specify a file name at the end of the destination path to copy with a custom file name.

### Examples

#### Put, Get, and Delete Files on ROS Device

Put a file from your host computer onto a ROS device, get it back, and then delete it.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.17.128','user','password');
```

Put a new text file that is in the MATLAB(R) current folder onto the ROS device. The destination folder must exist.

```
putFile(d,'test_file.txt','/home/user/test_folder')
```

Get a text file from the ROS device. You can get any file, not just ones added from MATLAB(R). By default, the file is added to the MATLAB current folder.

```
getFile(d,'/home/user/test_folder/test_file.txt')
```

Delete the text file on the ROS device.

```
deleteFile(d,'/home/user/test_folder/test_file.txt')
```

#### Put, Get, and Delete Files on ROS Device Using Wildcards

Put a file from your host computer onto a ROS device, get it back, and then delete it. Use wildcards to search for all matching files.

**Note:** You must have a valid ROS device to connect to at the IP address specified in the example.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.17.128', 'user', 'password');
```

Put all text files at the specified path onto the ROS device. The destination folder must exist.

```
putFile(d, 'C:/MATLAB/*.txt', '/home/user/test_folder')
```

Get all text files from the ROS device. You can get any files, not just ones added from MATLAB(R). By default, the files are added to the MATLAB current folder.

```
getFile(d, '/home/user/test_folder/*.txt')
```

Delete all text files on the ROS device at the specified folder.

```
deleteFile(d, '/home/user/test_folder/*.txt')
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **localSource** — Path and name of file on host computer

character vector

Path and name of the file on the host computer, specified as a character vector. You can use an absolute path or a path relative from the MATLAB current folder. Use the path and file naming conventions of the operating system on your host computer.

Example: 'C:\Work\.profile'

Data Types: char

### **remoteDestination** — Destination folder path and optional file name

character vector

Destination folder path and optional file name, specified as a character vector. Specify a file name at the end of the destination path to copy with a custom file name. Use the Linux path and file naming conventions.

Example: '/home/user/.profile'

Data Types: char

## See Also

[deleteFile](#) | [dir](#) | [getFile](#) | [openShell](#) | [rosdevice](#) | [system](#)

**Introduced in R2019b**

## readAllFieldNames

Get all available field names from ROS point cloud

### Syntax

```
fieldnames = readAllFieldNames(pcloud)
```

### Description

`fieldnames = readAllFieldNames(pcloud)` gets the names of all point fields that are stored in the `PointCloud2` object message, `pcloud`, and returns them in `fieldnames`.

### Examples

#### Read All Fields From Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read all the field names available on the point cloud message.

```
fieldnames = readAllFieldNames(ptcloud)
```

```
fieldnames = 1x4 cell  
    {'x'}    {'y'}    {'z'}    {'rgb'}
```

### Input Arguments

#### **pcloud** — Point cloud

`PointCloud2` object handle

Point cloud, specified as a `PointCloud2` object handle for a 'sensor\_msgs/PointCloud2' ROS message.

### Output Arguments

#### **fieldnames** — List of field names in `PointCloud2` object

cell array of character vectors

List of field names in `PointCloud2` object, returned as a cell array of character vectors. If no fields exist in the object, `fieldname` returns an empty cell array.

### See Also

`PointCloud2` | `readField`

**Introduced in R2019b**

## readCartesian

Read laser scan ranges in Cartesian coordinates

### Syntax

```
cart = readCartesian(scan)
cart = readCartesian( ____,Name,Value)
[cart,angles] = readCartesian( ____,Name,Value)
```

### Description

`cart = readCartesian(scan)` converts the polar measurements of the laser scan object, `scan`, into Cartesian coordinates, `cart`. This function uses the metadata in the message, such as angular resolution and opening angle of the laser scanner, to perform the conversion. Invalid range readings, usually represented as NaN, are ignored in this conversion.

`cart = readCartesian( ____,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes ( `' '` ). You can specify several name-value pair arguments in any order as `Name1,Value1, . . . ,NameN,ValueN`.

`[cart,angles] = readCartesian( ____,Name,Value)` returns the scan angles, `angles`, that are associated with each Cartesian coordinate. Angles are measured counterclockwise around the positive  $z$ -axis, with the zero angle along the  $x$ -axis. The `angles` is returned in radians and wrapped to the  $[-\pi, \pi]$  interval.

### Examples

#### Get Cartesian Coordinates from Laser Scan

Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

```
rosinit('192.168.17.129')
```

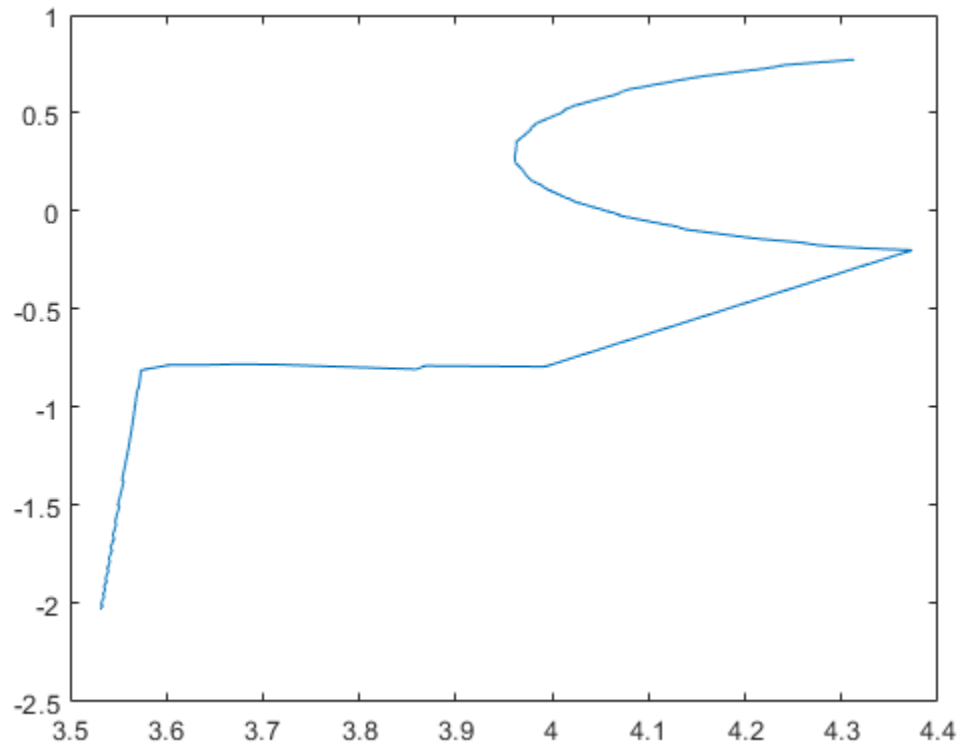
```
Initializing global node /matlab_global_node_40737 with NodeURI http://192.168.17.1:56343/
```

```
sub = rossubscriber('/scan');
scan = receive(sub);
```

Read the Cartesian points from the laser scan. Plot the laser scan.

```
cart = readCartesian(scan);
plot(cart(:,1),cart(:,2))
```





Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_40737 with NodeURI http://192.168.17.1:56343/
```

### Get Cartesian Coordinates from Laser Scan With Scan Range

Connect to ROS network. Subscribe to a laser scan topic, and receive a message.

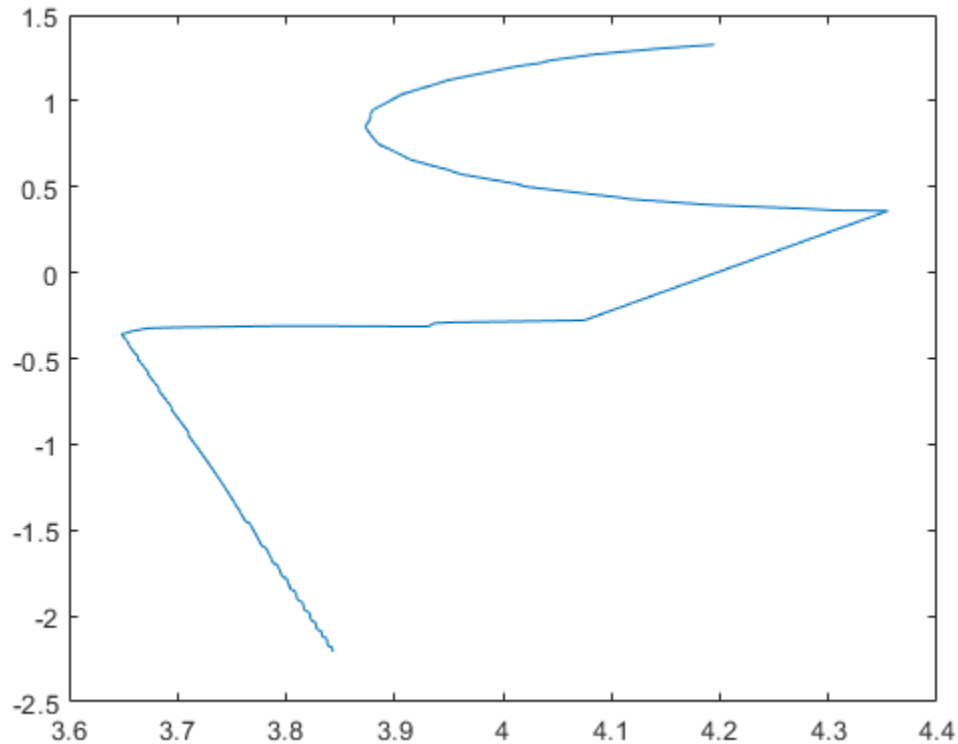
```
rosinit('192.168.17.129')
```

```
Initializing global node /matlab_global_node_12735 with NodeURI http://192.168.17.1:56572/
```

```
sub = rossubscriber('/scan');
scan = receive(sub);
```

Read the Cartesian points from the laser scan with specified range limits. Plot the laser scan.

```
cart = readCartesian(scan, 'RangeLimit', [0.5 6]);
plot(cart(:,1), cart(:,2))
```



Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_12735 with NodeURI http://192.168.17.1:56572/
```

## Input Arguments

### **scan** — Laser scan message

LaserScan object handle

'sensor\_msgs/LaserScan' ROS message, specified as a LaserScan object handle.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of **Name**, **Value** arguments. **Name** is the argument name and **Value** is the corresponding value. **Name** must appear inside quotes. You can specify several name and value pair arguments in any order as **Name1**, **Value1**, ..., **NameN**, **ValueN**.

Example: 'RangeLimits', [-2 2]

### **RangeLimits** — Minimum and maximum range for scan in meters

[scan.RangeMin scan.RangeMax] (default) | 2-element [min max] vector

Minimum and maximum range for a scan in meters, specified as a 2-element [min max] vector. All ranges smaller than min or larger than max are ignored during the conversion to Cartesian coordinates.

## Output Arguments

### **cart** — Cartesian coordinates of laser scan

*n*-by-2 matrix in meters

Cartesian coordinates of laser scan, returned as an *n*-by-2 matrix in meters.

### **angles** — Scan angles for laser scan data

*n*-by-1 matrix in radians

Scan angles for laser scan data, returned as an *n*-by-1 matrix in radians. Angles are measured counterclockwise around the positive *z*-axis, with the zero angle along the *x*-axis. The `angles` is returned in radians and wrapped to the `[-pi, pi]` interval.

## See Also

`plot` | `readScanAngles`

**Introduced in R2019b**

## readField

Read point cloud data based on field name

### Syntax

```
fielddata = readField(pcloud,fieldname)
```

### Description

`fielddata = readField(pcloud,fieldname)` reads the point field from the `PointCloud2` object, `pcloud`, specified by `fieldname` and returns it in `fielddata`. If `fieldname` does not exist, the function displays an error. To preserve the structure of the point cloud data, see “Preserving Point Cloud Structure” on page 1-57.

### Examples

#### Read Specific Field From Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read the 'x' field name available on the point cloud message.

```
x = readField(ptcloud, 'x');
```

### Input Arguments

#### **pcloud** — Point cloud

`PointCloud2` object handle

Point cloud, specified as a `PointCloud2` object handle for a `sensor_msgs/PointCloud2` ROS message.

#### **fieldname** — Field name of point cloud data

string scalar | character vector

Field name of point cloud data, specified as a string scalar or character vector. This string must match the field name exactly. If `fieldname` does not exist, the function displays an error.

### Output Arguments

#### **fielddata** — List of field values from point cloud

matrix

List of field values from point cloud, returned as a matrix. Each row of the matrix is a point cloud reading, where  $n$  is the number of points and  $c$  is the number of values for each point. If the point

cloud object being read has the `PreserveStructureOnRead` property set to `true`, the points are returned as an  $h$ -by- $w$ -by- $c$  matrix. For more information, see “Preserving Point Cloud Structure” on page 1-57.

## Preserving Point Cloud Structure

Point cloud data can be organized in either 1-D lists or in 2-D image styles. 2-D image styles usually come from depth sensors or stereo cameras. The input `PointCloud2` object contains a `PreserveStructureOnRead` property that is either `true` or `false` (default). Suppose you set the property to `true`.

```
pcloud.PreserveStructureOnRead = true;
```

Now calling any read functions (`readXYZ`, `readRGB`, or `readField`) preserves the organizational structure of the point cloud. When you preserve the structure, the output matrices are of size  $m$ -by- $n$ -by- $d$ , where  $m$  is the height,  $n$  is the width, and  $d$  is the number of return values for each point. Otherwise, all points are returned as a  $x$ -by- $d$  list. This structure can be preserved only if the point cloud is organized.

### See Also

`PointCloud2` | `readAllFieldNames`

**Introduced in R2019b**

## readImage

Convert ROS image data into MATLAB image

### Syntax

```
img = readImage(msg)
[img,alpha] = readImage(msg)
```

### Description

`img = readImage(msg)` converts the raw image data in the message object, `msg`, into an image matrix, `img`. You can call `readImage` using either `'sensor_msgs/Image'` or `'sensor_msgs/CompressedImage'` messages.

ROS image message data is stored in a format that is not compatible with further image processing in MATLAB. Based on the specified encoding, this function converts the data into an appropriate MATLAB image and returns it in `img`.

`[img,alpha] = readImage(msg)` returns the alpha channel of the image in `alpha`. If the image does not have an alpha channel, then `alpha` is empty.

### Examples

#### Read ROS Image Data

Load sample ROS messages including a ROS image message, `img`.

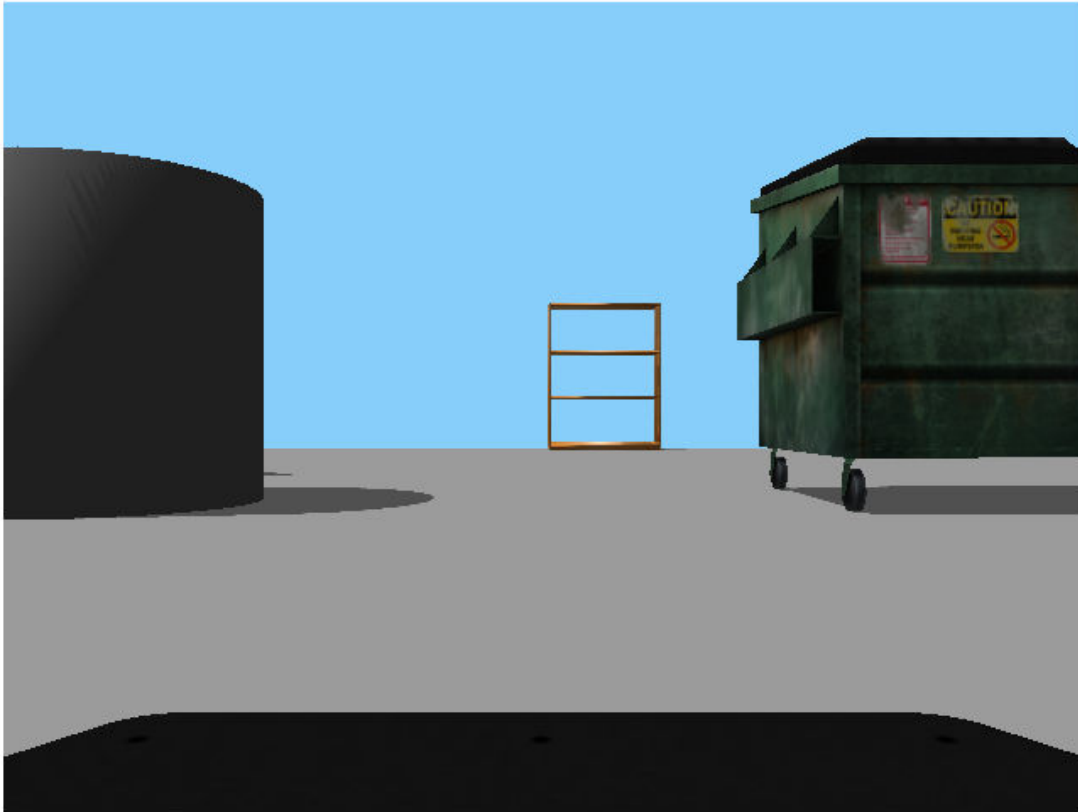
```
exampleHelperROSLoadMessages
```

Read the ROS image message as a MATLAB® image.

```
image = readImage(img);
```

Display the image.

```
imshow(image)
```



## Input Arguments

### **msg** — ROS image message

Image object handle | CompressedImage object handle

'sensor\_msgs/Image' or 'sensor\_msgs/CompressedImage' ROS image message, specified as an Image or Compressed Image object handle.

## Output Arguments

### **img** — Image

grayscale image matrix | RGB image matrix |  $m$ -by- $n$ -by-3 array

Image, returned as a matrix representing a grayscale or RGB image or as an  $m$ -by- $n$ -by-3 array, depending on the sensor image.

### **alpha** — Alpha channel

uint8 grayscale image

Alpha channel, returned as a uint8 grayscale image. If no alpha channel exists, alpha is empty.

---

**Note** For `CompressedImage` messages, you cannot output an Alpha channel.

---

## Supported Image Encodings

ROS image messages can have different encodings. The encodings supported for images are different for `'sensor_msgs/Image'` and `'sensor_msgs/CompressedImage'` message types. Fewer compressed images are supported. The following encodings for raw images of size  $M$ -by- $N$  are supported using the `'sensor_msgs/Image'` message type (**'sensor\_msgs/CompressedImage' support is in bold**):

- **rgb8, rgba8, bgr8, bgra8**: `img` is an `rgb` image of size  $M$ -by- $N$ -by-3. The alpha channel is returned in `alpha`. Each value in the outputs is represented as a `uint8`.
- `rgb16, rgba16, bgr16, and bgra16`: `img` is an `RGB` image of size  $M$ -by- $N$ -by-3. The alpha channel is returned in `alpha`. Each value in the output is represented as a `uint16`.
- **mono8** images are returned as grayscale images of size  $M$ -by- $N$ -by-1. Each pixel value is represented as a `uint8`.
- `mono16` images are returned as grayscale images of size  $M$ -by- $N$ -by-1. Each pixel value is represented as a `uint16`.
- `32fcX` images are returned as floating-point images of size  $M$ -by- $N$ -by- $D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `single`.
- `64fcX` images are returned as floating-point images of size  $M$ -by- $N$ -by- $D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `double`.
- `8ucX` images are returned as matrices of size  $M$ -by- $N$ -by- $D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `uint8`.
- `8scX` images are returned as matrices of size  $M$ -by- $N$ -by- $D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `int8`.
- `16ucX` images are returned as matrices of size  $M$ -by- $N$ -by- $D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `int16`.
- `16scX` images are returned as matrices of size  $M$ -by- $N$ -by- $D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `int16`.
- `32scX` images are returned as matrices of size  $M$ -by- $N$ -by- $D$ , where  $D$  is 1, 2, 3, or 4. Each pixel value is represented as a `int32`.
- `bayer_X` images are returned as either Bayer matrices of size  $M$ -by- $N$ -by-1, or as a converted image of size  $M$ -by- $N$ -by-3 (Image Processing Toolbox™ is required).

The following encoding for raw images of size  $M$ -by- $N$  is supported using the **'sensor\_msgs/CompressedImage'** message type:

- `rgb8, rgba8, bgr8, and bgra8`: `img` is an `rgb` image of size  $M$ -by- $N$ -by-3. The alpha channel is returned in `alpha`. Each output value is represented as a `uint8`.

## See Also

`writeImage`

**Introduced in R2019b**



# readMessages

Read messages from rosbag

## Syntax

```
msgs = readMessages(bag)
msgs = readMessages(bag, rows)
msgs = readMessages( ____, "DataFormat", "struct")
```

## Description

`msgs = readMessages(bag)` returns data from all the messages in the `BagSelection` object, `bag`. The messages are returned in a cell array of messages.

To get a `BagSelection` object, use `rosbag`.

`msgs = readMessages(bag, rows)` returns data from messages in the rows specified by `rows`. The range of the rows is `[1, bag.NumMessages]`.

`msgs = readMessages( ____, "DataFormat", "struct")` returns data as a cell array of structures using either set of the previous input arguments. Using structures can be significantly faster than using message objects, and custom message data can be read directly without loading message definitions using `rosgenmsg`.

## Examples

### Return ROS Messages as a Cell Array

Read rosbag and filter by topic and time.

```
bagselect = rosbag('ex_multiple_topics.bag');
bagselect2 = select(bagselect, 'Time', ...
[bagselect.StartTime bagselect.StartTime + 1], 'Topic', '/odom');
```

Return all messages as a cell array.

```
allMsgs = readMessages(bagselect2);
```

Return the first ten messages as a cell array.

```
firstMsgs = readMessages(bagselect2, 1:10);
```

### Read Messages from a rosbag as a Structure

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Select a specific topic.

```
bSel = select(bag, 'Topic', '/turtle1/pose');
```

Read messages as a structure. Specify the `DataFormat` name-value pair when reading the messages. Inspect the first structure in the returned cell array of structures.

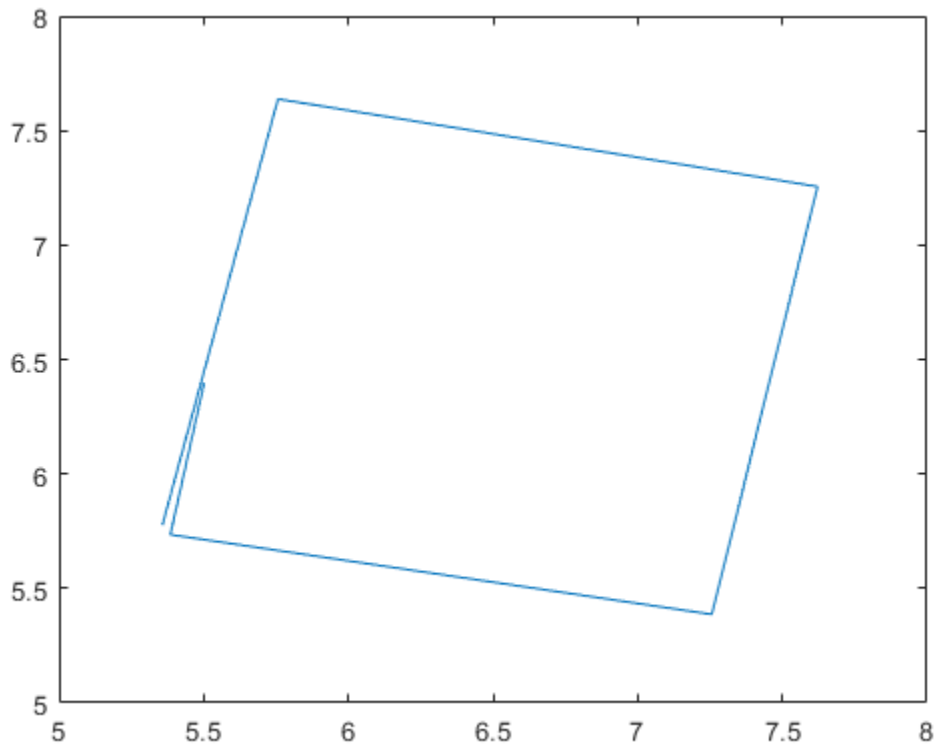
```
msgStructs = readMessages(bSel, 'DataFormat', 'struct');  
msgStructs{1}
```

```
ans = struct with fields:  
    MessageType: 'turtlesim/Pose'  
            X: 5.5016  
            Y: 6.3965  
            Theta: 4.5377  
    LinearVelocity: 1  
    AngularVelocity: 0
```

Extract the xy points from the messages and plot the robot trajectory.

Use `cellfun` to extract all the X and Y fields from the structure. These fields represent the xy positions of the robot during the rosbag recording.

```
xPoints = cellfun(@(m) double(m.X), msgStructs);  
yPoints = cellfun(@(m) double(m.Y), msgStructs);  
plot(xPoints, yPoints)
```



## Input Arguments

### **bag** — Message of rosbag

BagSelection object

All the messages contained within a rosbag, specified as a BagSelection object.

### **rows** — Rows of BagSelection object

*n*-element vector

Rows of BagSelection object, specified as an *n*-element vector, where *n* is the number of rows to retrieve messages from. Each entry in the vector corresponds to a numbered message in the bag. The range of the rows is [1, bag.NumMessage].

## Output Arguments

### **msgs** — ROS message data

object | cell array of message objects | cell array of structures

ROS message data, returned as an object, cell array of message objects, or cell array of structures. Data comes from the BagSelection object created using rosbag. You must specify 'DataFormat', 'struct' in the function to get messages as a cell array of structures. Using structures can be significantly faster than using message objects, and custom message data can be read directly without loading message definitions using rosgenmsg.

## See Also

rosbag | select | timeseries

**Introduced in R2019b**

## readRGB

Extract RGB values from point cloud data

### Syntax

```
rgb = readRGB(pcloud)
```

### Description

`rgb = readRGB(pcloud)` extracts the [r g b] values from all points in the `PointCloud2` object, `pcloud`, and returns them as an  $n$ -by-3 of  $n$  3-D point coordinates. If the point cloud does not contain the RGB field, this function displays an error. To preserve the structure of the point cloud data, see “Preserving Point Cloud Structure” on page 1-64.

### Examples

#### Read RGB Values from ROS Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read the RGB values from the point cloud.

```
rgb = readRGB(ptcloud);
```

### Input Arguments

#### **pcloud** — Point cloud

`PointCloud2` object handle

Point cloud, specified as a `PointCloud2` object handle for a 'sensor\_msgs/PointCloud2' ROS message.

### Output Arguments

#### **rgb** — List of RGB values from point cloud

matrix

List of RGB values from point cloud, returned as a matrix. By default, this is an  $n$ -by-3 matrix. If the point cloud object being read has the `PreserveStructureOnRead` property set to true, the points are returned as an  $h$ -by- $w$ -by-3 matrix. For more information, see “Preserving Point Cloud Structure” on page 1-64.

### Preserving Point Cloud Structure

Point cloud data can be organized in either 1-D lists or in 2-D image styles. 2-D image styles usually come from depth sensors or stereo cameras. The input `PointCloud2` object contains a

`PreserveStructureOnRead` property that is either `true` or `false` (default). Suppose that you set the property to `true`.

```
pcloud.PreserveStructureOnRead = true;
```

Now calling any read functions (`readXYZ`, `readRGB`, or `readField`) preserves the organizational structure of the point cloud. When you preserve the structure, the output matrices are of size  $m$ -by- $n$ -by- $d$ , where  $m$  is the height,  $n$  is the width, and  $d$  is the number of return values for each point. Otherwise, all points are returned as an  $x$ -by- $d$  list. This structure can be preserved only if the point cloud is organized.

## See Also

[PointCloud2](#) | [PointCloud2](#) | [readXYZ](#)

**Introduced in R2019b**

## readScanAngles

Return scan angles for laser scan range readings

### Syntax

```
angles = readScanAngles(scan)
```

### Description

`angles = readScanAngles(scan)` calculates the scan angles, `angles`, corresponding to the range readings in the laser scan message, `scan`. Angles are measured counterclockwise around the positive  $z$ -axis, with the zero angle along the  $x$ -axis. The `angles` is returned in radians and wrapped to the  $[-\pi, \pi]$  interval.

### Examples

#### Read Scan Angles from ROS Laser Scan Message

Load sample ROS messages including a ROS laser scan message, `scan`.

```
exampleHelperROSLoadMessages
```

Read the scan angles from the laser scan.

```
angles = readScanAngles(scan)
```

```
angles = 640x1
```

```
-0.5467  
-0.5450  
-0.5433  
-0.5416  
-0.5399  
-0.5382  
-0.5364  
-0.5347  
-0.5330  
-0.5313  
⋮
```

### Input Arguments

**scan** — Laser scan message

LaserScan object handle

'sensor\_msgs/LaserScan' ROS message, specified as a LaserScan object handle.

## Output Arguments

### **angles** — Scan angles for laser scan data

*n*-by-1 matrix in radians

Scan angles for laser scan data, returned as an *n*-by-1 matrix in radians. Angles are measured counter-clockwise around the positive *z*-axis, with the zero angle along the *x*-axis. The `angles` is returned in radians and wrapped to the `[-pi, pi]` interval.

### See Also

`plot` | `readCartesian`

**Introduced in R2019b**

## readXYZ

Extract XYZ coordinates from point cloud data

### Syntax

```
xyz = readXYZ(pcloud)
```

### Description

`xyz = readXYZ(pcloud)` extracts the  $[x \ y \ z]$  coordinates from all points in the `PointCloud2` object, `pcloud`, and returns them as an  $n$ -by-3 matrix of  $n$  3-D point coordinates. If the point cloud does not contain the  $x$ ,  $y$ , and  $z$  fields, this function returns an error. Points that contain NaN are preserved in the output. To preserve the structure of the point cloud data, see “Preserving Point Cloud Structure” on page 1-69.

### Examples

#### Read XYZ Values from ROS Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read the XYZ values from the point cloud.

```
xyz = readXYZ(ptcloud);
```

### Input Arguments

#### **pcloud** — Point cloud

`PointCloud2` object handle

Point cloud, specified as a `PointCloud2` object handle for a 'sensor\_msgs/PointCloud2' ROS message.

### Output Arguments

#### **xyz** — List of XYZ values from point cloud

matrix

List of XYZ values from point cloud, returned as a matrix. By default, this is a  $n$ -by-3 matrix. If the point cloud object being read has the `PreserveStructureOnRead` property set to true, the points are returned as an  $h$ -by- $w$ -by-3 matrix. For more information, see “Preserving Point Cloud Structure” on page 1-69.



## Preserving Point Cloud Structure

Point cloud data can be organized in either 1-D lists or in 2-D image styles. 2-D image styles usually come from depth sensors or stereo cameras. The input `PointCloud2` object contains a `PreserveStructureOnRead` property that is either `true` or `false` (default). Suppose you set the property to `true`.

```
pcloud.PreserveStructureOnRead = true;
```

Now calling any read functions (`readXYZ`, `readRGB`, or `readField`) preserves the organizational structure of the point cloud. When you preserve the structure, the output matrices are of size  $m$ -by- $n$ -by- $d$ , where  $m$  is the height,  $n$  is the width, and  $d$  is the number of return values for each point. Otherwise, all points are returned as a  $x$ -by- $d$  list. This structure can be preserved only if the point cloud is organized.

### See Also

`PointCloud2` | `readField` | `readRGB`

**Introduced in R2019b**

## receive

Wait for new ROS message

### Syntax

```
msg = receive(sub)
msg = receive(sub,timeout)
```

### Description

`msg = receive(sub)` waits for MATLAB to receive a topic message from the specified subscriber, `sub`, and returns it as `msg`.

`msg = receive(sub,timeout)` specifies in `timeout` the number of seconds to wait for a message. If a message is not received within the timeout limit, the software produces an error.

### Examples

#### Create A Subscriber and Get Data From ROS

Connect to a ROS network. Set up a sample ROS network. The `'/scan'` topic is being published on the network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:55964/.
Initializing global node /matlab_global_node_31941 with NodeURI http://bat5110win64:56008/
```

```
exampleHelperROSCreateSampleNetwork
```

Create a subscriber for the `'/scan'` topic. Wait for the subscriber to register with the master.

```
sub = rossubscriber('/scan');
pause(1);
```

Receive data from the subscriber as a ROS message. Specify a 10-second timeout.

```
msg2 = receive(sub,10)
```

```
msg2 =
  ROS LaserScan message with properties:

    MessageType: 'sensor_msgs/LaserScan'
      Header: [1x1 Header]
      AngleMin: -0.5216
      AngleMax: 0.5243
    AngleIncrement: 0.0016
    TimeIncrement: 0
      ScanTime: 0.0330
      RangeMin: 0.4500
      RangeMax: 10
```

```

    Ranges: [640x1 single]
    Intensities: [0x1 single]

```

Use `showdetails` to show the contents of the message

Shutdown the timers used by sample network.

```
exampleHelperROSShutdownSampleNetwork
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_31941 with NodeURI http://bat5110win64:56008/
Shutting down ROS master on http://bat5110win64:55964/.
```

## Create, Send, and Receive ROS Messages

Set up a publisher and subscriber to send and receive a message on a ROS network.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:55981/.
Initializing global node /matlab_global_node_45365 with NodeURI http://bat5110win64:55985/
```

Create a publisher with a specific topic and message type. You can also return a default message to send using this publisher.

```
[pub,msg] = rospublisher('position','geometry_msgs/Point');
```

Modify the message before sending it over the network.

```
msg.X = 1;
msg.Y = 2;
send(pub,msg);
```

Create a subscriber and wait for the latest message. Verify the message is the one you sent.

```
sub = rossubscriber('position')
```

```
sub =
```

```
Subscriber with properties:
```

```

    TopicName: '/position'
    MessageType: 'geometry_msgs/Point'
    LatestMessage: [0x1 Point]
    BufferSize: 1
    NewMessageFcn: []

```

```
pause(1);
sub.LatestMessage
```

```
ans =
ROS Point message with properties:
```

```
MessageType: 'geometry_msgs/Point'  
  X: 1  
  Y: 2  
  Z: 0
```

Use `showdetails` to show the contents of the message

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_45365 with NodeURI http://bat5110win64:55985/  
Shutting down ROS master on http://bat5110win64:55981/.
```

### Read Specific Field From Point Cloud Message

Load sample ROS messages including a ROS point cloud message, `ptcloud`.

```
exampleHelperROSLoadMessages
```

Read the 'x' field name available on the point cloud message.

```
x = readField(ptcloud, 'x');
```

## Input Arguments

### **sub** — ROS subscriber

Subscriber object handle

ROS subscriber, specified as a `Subscriber` object handle. You can create the subscriber using `rossubscriber`.

### **timeout** — Timeout for receiving a message

scalar in seconds

Timeout for receiving a message, specified as a scalar in seconds.

## Output Arguments

### **msg** — ROS message

Message object handle

ROS message, returned as a `Message` object handle.

## See Also

`rosmessage` | `rospublisher` | `rossubscriber` | `rostopic` | `send`

### Topics

“Exchange Data with ROS Publishers and Subscribers”

**Introduced in R2019b**

## ros2

Retrieve information about ROS 2 network

### Syntax

```
ros2 msg list
ros2 msg show msgType
ros2 node list
ros2 topic list
msgList = ros2("msg", "list")
msgInfo = ros2("msg", "show", msgType)
nodeList = ros2("node", "list")
topicList = ros2("topic", "list")
```

### Description

`ros2 msg list` returns a list of all available ROS 2 message types that can be used in MATLAB.

`ros2 msg show msgType` provides the definition of the ROS 2 message, `msgType`.

`ros2 node list` lists nodes on the ROS 2 network.

`ros2 topic list` lists topic names with registered publishers or subscribers on the ROS 2 network.

`msgList = ros2("msg", "list")` returns a list of all available ROS 2 message types that can be used in MATLAB.

`msgInfo = ros2("msg", "show", msgType)` provides the definition of the ROS 2 message, `msgType`.

`nodeList = ros2("node", "list")` lists nodes on the ROS 2 network.

`topicList = ros2("topic", "list")` lists topic names with registered publishers or subscribers on the ROS 2 network.

### Examples

#### Get Definition of ROS 2 Message

Show the definition of the `geometry_msgs/Accel` message.

```
ros2 msg show geometry_msgs/Accel
```

```
# This expresses acceleration in free space broken into its linear and angular parts.
Vector3 linear
Vector3 angular
```

### Get Definition of ROS 2 Message

Show the definition of the `geometry_msgs/Accel` message.

```
ros2 msg show geometry_msgs/Accel
```

```
# This expresses acceleration in free space broken into its linear and angular parts.
Vector3 linear
Vector3 angular
```

### Get List of ROS 2 Nodes

Create sample node, `myNode`, on the ROS 2 network.

```
node = ros2node("myNode");
```

Lists the nodes on the network.

```
ros2 node list
```

```
/myNode
```

Remove `myNode` from the network.

```
delete(node)
```

### Get List of ROS 2 Topics

List the available ROS 2 topics.

```
ros2 topic list
```

```
/parameter_events
```

## Input Arguments

### `msgType` — Message type

string scalar | character vector

Message type, specified as a string scalar or character vector. The string is case-sensitive and no partial matches are allowed. It must match a message on the list given by calling `ros2("msg", "list")`.

## Output Arguments

### `msgList` — List of all message types available in MATLAB

cell array of character vectors

List of all message types available in MATLAB, returned as a cell array of character vectors.

**msgInfo – Details of message definition**

character vector

Details of the information inside the ROS message definition, returned as a character vector.

**topicList – List of topics from the ROS master**

cell array of character vectors

List of topics from ROS master, returned as a cell array of character vectors.

**nodeList – List of node names available**

cell array of character vectors

List of node names available, returned as a cell array of character vectors.

**See Also**

[ros2message](#) | [ros2node](#) | [ros2publisher](#) | [ros2subscriber](#)

**Introduced in R2019b**



# ros2genmsg

Generate custom messages from ROS 2 definitions

## Syntax

```
ros2genmsg(folderpath)
```

## Description

`ros2genmsg(folderpath)` generates MATLAB interfaces to ROS 2 custom messages. Specify the path to the parent folder that contains the definitions of the custom messages as `.msg` files.

You must have the following installed on your system:

- Python 3.7+
- CMake 3.10+
- C++ compiler to match the build environment
  - Visual Studio 2017
  - Xcode 10.x
  - GCC 6.3.x

## Examples

### ROS 2 Custom Message Support

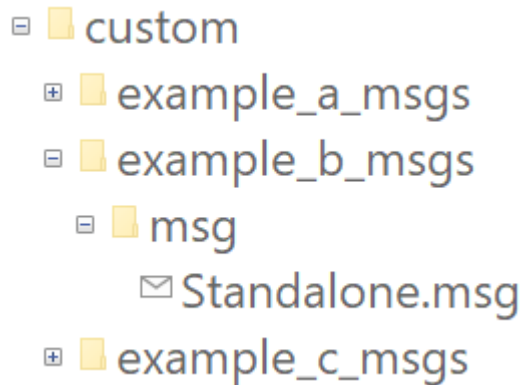
Custom messages are user-defined messages that you can use to extend the set of message types currently supported in ROS 2. If you are sending and receiving supported message types, you do not need to use custom messages. To see a list of supported message types, call `ros2 msg list` in the MATLAB® Command Window. For more information about supported ROS 2 messages, see “Work with Basic ROS 2 Messages”.

To use ROS 2 custom messages, you must have the following installed on your computer:

- Python 3.7+
- CMake 3.10+
- C++ compiler to match the build environment (Visual Studio 2017, Xcode 10.x, or GCC 6.3.x)
- See ROS 2 Model Build Failure in “ROS Simulink Support and Limitations”.

### Custom messages Contents

ROS 2 custom messages are specified in ROS 2 package folders that contain a `msg` directory. The `msg` folder contains all your custom message type definitions. For example, the package `example_b_msgs`, within the `custom` folder, has the below folder and file structure.



The package contains one custom message type, `Standalone.msg`. MATLAB uses these files to generate the necessary files for using the custom messages contained in the package. For more information on message naming conventions, see ROS 2 Interface Definition.

In this example, you go through the procedure for creating ROS 2 custom messages in MATLAB®. You must have a ROS 2 package that contains the required `msg` file.

After ensuring that your custom message package is correct, note the folder path location, and then, call `ros2genmsg` with the specified path. The following example provided three messages `example_package_a`, `example_package_b`, and `example_package_c` that have dependencies. This example also illustrates that you can use a folder containing multiple messages and generate them all at the same time.

To set up custom messages in MATLAB, open MATLAB in a new session. Place your custom message folder in a location and note the folder path. In this example, the custom message interface folder is present in the current directory. If you are creating custom message packages in a separate location, provide the appropriate path to the folder that contains the custom message packages.

```
folderPath = fullfile(pwd, "custom");
copyfile("example_*_msgs", folderPath);
```

Specify the folder path for custom message files and call `ros2genmsg` to create custom messages for MATLAB.

```
ros2genmsg(folderPath)
```

```
Identifying message files in folder 'H:/Documents/MATLAB/Examples/ros-ex44405863/custom'.Done.
Validating message files in folder 'H:/Documents/MATLAB/Examples/ros-ex44405863/custom'.Done.
Generating MATLAB interfaces.Done.
Running colcon build in folder 'H:/Documents/MATLAB/Examples/ros-ex44405863/custom/matlab_msg_gen'
build log
```

Call `ros2 msg list` to verify creation of new custom messages.

You can now use the above created custom message as the standard messages. For more information on sending and receiving messages, see “Exchange Data with ROS 2 Publishers and Subscribers”.

Create a publisher to use `example_package_b/StandAlone` message.

```
node = ros2node("/node_1");
pub = ros2publisher(node, "/example_topic", "example_b_msgs/StandAlone");
```

Create a subscriber on the same topic.

```
sub = ros2subscriber(node, "/example_topic");
```

Create a message and send the message.

```
custom_msg = ros2message("example_b_msgs/Standalone");
custom_msg.int_property = uint32(12);
custom_msg.string_property='This is ROS 2 custom message example';
send(pub, custom_msg);
pause(3)    % Allow a few seconds for the message to arrive
```

Use LatestMessage field to know the recent message received by the subscriber.

```
sub.LatestMessage

ans = struct with fields:
    int_property: 12
    string_property: 'This is ROS 2 custom message example'
```

Remove the created ROS objects.

```
clear node pub sub
```

## Input Arguments

### folderpath — Path to ROS message package

string scalar | character vector

Path to the parent folder of ROS message packages, specified as a string scalar or character vector. The parent folder should contain a `package.xml` file and package folders. These folders contain a `/msg` folder with `.msg` files for message definitions. For more information, see [About ROS 2 Interfaces](#).

Example: `"/opt/ros2/kinetic/share"`

Data Types: `char` | `string`

## Limitations

### Restart Nodes

- After generating custom messages, restart any existing ROS 2 nodes.

### Code Generation with custom messages:

- Custom message and service types can be used with ROS 2 functionality for generating C++ code for a standalone ROS 2 node. The generated code (.tgz archive) will include definitions for the custom messages, but it will not include the ROS 2 custom message packages. When the generated code is built in the destination, it expects the custom message packages to be available in the colcon workspace which should be your current working directory. Please ensure that you either install or copy the custom message package to your system before building the generated code.

### **MATLAB Compiler**

- ROS 2 custom messages and the `ros2genmsg` function are not supported with MATLAB Compiler™.

### **See Also**

`rosgenmsg`

### **Topics**

“ROS Custom Message Support”

“ROS 2 Custom Message Support”

### **External Websites**

About ROS 2 Interfaces

Download Python

Download CMake

### **Introduced in R2019b**

# ros2message

Create ROS 2 message structures

## Syntax

```
msg = ros2message(msgType)
```

## Description

`msg = ros2message(msgType)` creates a structure compatible with ROS 2 messages of type `msgType`.

## Examples

### Create a String Message

Create a ROS 2 string message.

```
strMsg = ros2message('std_msgs/String')
```

```
strMsg = struct with fields:  
    data: ''
```

### Create an empty laser scan message

Create an empty ROS 2 laser scan message.

```
scanMsg = ros2message("sensor_msgs/LaserScan")
```

```
scanMsg = struct with fields:  
    header: [1x1 struct]  
    angle_min: 0  
    angle_max: 0  
    angle_increment: 0  
    time_increment: 0  
    scan_time: 0  
    range_min: 0  
    range_max: 0  
    ranges: 0  
    intensities: 0
```

## Input Arguments

**msgType** — Message type for a ROS 2 topic  
character vector

Message type for a ROS 2 topic, specified as a character vector.

## **Output Arguments**

**msg** — ROS 2 message for a given topic

object handle

ROS 2 message for a given topic, returned as an object handle.

## **See Also**

[ros2](#) | [ros2node](#) | [ros2publisher](#) | [ros2subscriber](#)

**Introduced in R2019b**

# rosaction

Retrieve information about ROS actions

## Syntax

```
rosaction list
rosaction info actionname
rosaction type actionname

actionlist = rosaction("list")
actioninfo = rosaction("info",actionname)
actiontype = rosaction("type",actionname)
```

## Description

`rosaction list` returns a list of available ROS actions from the ROS network.

`rosaction info actionname` returns the action type, message types, action server, and action clients for the specified action name.

`rosaction type actionname` returns the action type for the specified action name.

`actionlist = rosaction("list")` returns a list of available ROS actions from the ROS network.

`actioninfo = rosaction("info",actionname)` returns a structure containing the action type, message types, action server, and action clients for the specified action name.

`actiontype = rosaction("type",actionname)` returns the action type for the specified action name.

## Examples

### Get Information About ROS Actions

Get information about ROS actions that are available from the ROS network. You must be connected to a ROS network using `rosinit`.

Action types must be set up beforehand with a ROS action server running on the network. You must have the set up `'/fibonacci'` action type. To run this action server, use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

Connect to a ROS network. You must be connected to a ROS network to gather information about what actions are available. Replace `ipaddress` with your network address.

```
ipaddress = '192.168.17.129';
rosinit(ipaddress,11311)
```

```
Initializing global node /matlab_global_node_04165 with NodeURI http://192.168.17.1:60617/
```

List the actions available on the network. The only action set up on this network is the '/fibonacci' action.

```
roaction list
```

```
/fibonacci
```

Get information about a specific ROS action type. The action type, message types, action server, and clients are displayed.

```
roaction info /fibonacci
```

```
Action Type: actionlib_tutorials/Fibonacci
```

```
Goal Message Type: actionlib_tutorials/FibonacciGoal
```

```
Feedback Message Type: actionlib_tutorials/FibonacciFeedback
```

```
Result Message Type: actionlib_tutorials/FibonacciResult
```

```
Action Server:
```

```
* /fibonacci (http://192.168.17.129:34793/)
```

```
Action Clients: None
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_04165 with NodeURI http://192.168.17.1:60617/
```

## Input Arguments

**actionname** — ROS action name

string scalar | character vector

ROS action name, specified as a string scalar or character vector. The action name must match one of the topics that `roaction("list")` outputs.

## Output Arguments

**actionlist** — List of actions available

cell array of character vectors

List of actions available on the ROS network, returned as a cell array of character vectors.

**actioninfo** — Information about a ROS action

structure

Information about a ROS action, returned as a structure. `actioninfo`, which contains the following fields:

- ActionType
- GoalMessageType
- FeedbackMessageType



- ResultMessageType
- ActionServer
- ActionClients

For more information about ROS actions, see “ROS Actions Overview”.

**actiontype – Type of ROS action**

character vector

Type of ROS action, returned as a character vector.

**See Also**

cancelGoal | rosmessage | rostopic | sendGoal | waitForServer

**Topics**

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

**Introduced in R2019b**

## rosAddons

Install add-ons for ROS

### Syntax

```
rosAddons
```

### Description

rosAddons allows you to download and install add-ons for ROS Toolbox. Use this function to open the Add-ons Explorer to browse the available add-ons.

### Examples

#### Install Add-ons for ROS Toolbox™

```
rosAddons
```

### See Also

#### Topics

“ROS Custom Message Support”

“Get and Manage Add-Ons” (MATLAB)

“Manage Add-Ons” (MATLAB)

**Introduced in R2019b**

# rosbag

Open and parse rosbag log file

## Syntax

```
bag = rosbag(filename)

bagInfo = rosbag('info',filename)
rosbag info filename
```

## Description

`bag = rosbag(filename)` creates an indexable `BagSelection` object, `bag`, that contains all the message indexes from the rosbag at path `filename`. To get a `BagSelection` object, use `rosbag`. To access the data, call `readMessages` or `timeseries` to extract relevant data.

A rosbag, or bag, is a file format for storing ROS message data. They are used primarily to log messages within the ROS network. You can use these bags for offline analysis, visualization, and storage. See the ROS Wiki page for more information about rosbags.

`bagInfo = rosbag('info',filename)` returns information as a structure, `bagInfo`, which is about the contents of the rosbag at `filename`.

`rosbag info filename` displays information in the MATLAB Command Window about the contents of the rosbag at `filename`. The information includes the number of messages, start and end times, topics, and message types.

## Examples

### Retrieve Information from rosbag

Retrieve information from the rosbag. Specify the full path to the rosbag if it is not already available on the MATLAB® path.

```
bagselect = rosbag('ex_multiple_topics.bag');
```

Select a subset of the messages, filtered by time and topic.

```
bagselect2 = select(bagselect, 'Time', ...
    [bagselect.StartTime bagselect.StartTime + 1], 'Topic', '/odom');
```

### Display rosbag Information from File

To view information about a rosbag log file, use `rosbag info filename`, where `filename` is a rosbag (.bag) file.

```
rosbag info 'ex_multiple_topics.bag'
```

```
Path:      C:\TEMP\Bdoc20a_1326390_8984\ib9D0363\19\tp6ab89118\ros-ex32890909\ex_multiple_topics.
Version:   2.0
Duration:  2:00s (120s)
Start:     Dec 31 1969 19:03:21.34 (201.34)
End:       Dec 31 1969 19:05:21.34 (321.34)
Size:      23.6 MB
Messages:  36963
Types:     gazebo_msgs/LinkStates [48c080191eb15c41858319b4d8a609c2]
           nav_msgs/Odometry      [cd5e73d190d741a2f92e81eda573aca7]
           rosgraph_msgs/Clock    [a9c97c1d230cfc112e270351a944ee47]
           sensor_msgs/LaserScan  [90c7ef2dc6895d81024acba2ac42f369]
Topics:    /clock                  12001 msgs : rosgraph_msgs/Clock
           /gazebo/link_states    11999 msgs : gazebo_msgs/LinkStates
           /odom                  11998 msgs : nav_msgs/Odometry
           /scan                   965 msgs  : sensor_msgs/LaserScan
```

### Get Transformations from rosbag File

Get transformations from rosbag (.bag) files by loading the rosbag and checking the available frames. From these frames, use `getTransform` to query the transformation between two coordinate frames.

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Get a list of available frames.

```
frames = bag.AvailableFrames;
```

Get the latest transformation between two coordinate frames.

```
tf = getTransform(bag, 'world', frames{1});
```

Check for a transformation available at a specific time and retrieve the transformation. Use `canTransform` to check if the transformation is available. Specify the time using `rostime`.

```
tfTime = rostime(bag.StartTime + 1);
if (canTransform(bag, 'world', frames{1}, tfTime))
    tf2 = getTransform(bag, 'world', frames{1}, tfTime);
end
```

### Read Messages from a rosbag as a Structure

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Select a specific topic.

```
bSel = select(bag, 'Topic', '/turtle1/pose');
```

Read messages as a structure. Specify the `DataFormat` name-value pair when reading the messages. Inspect the first structure in the returned cell array of structures.

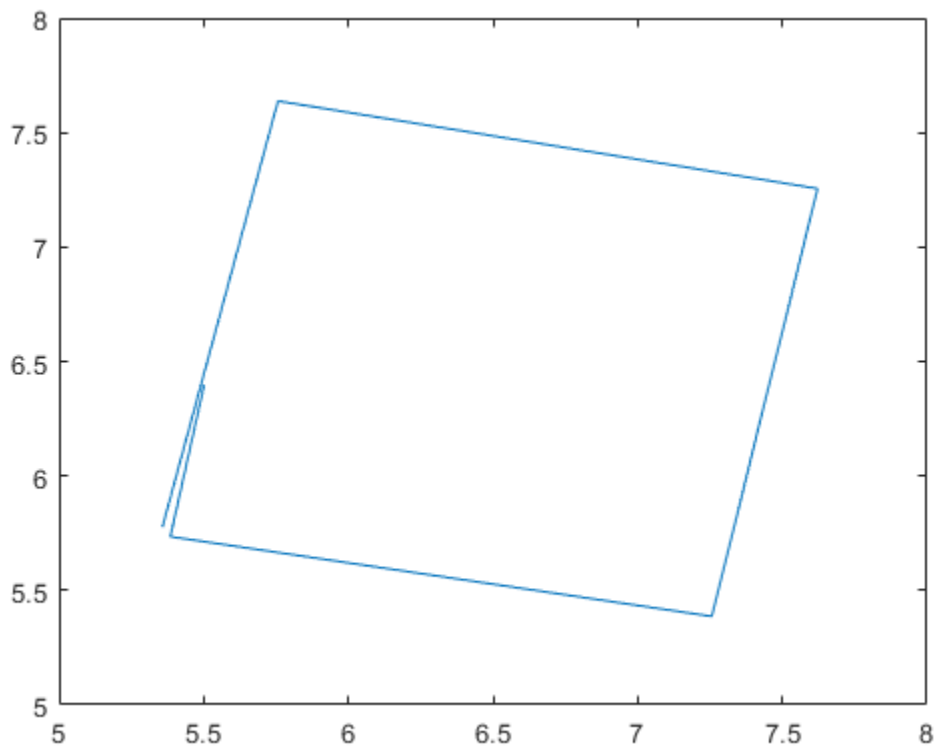
```
msgStructs = readMessages(bSel, 'DataFormat', 'struct');
msgStructs{1}
```

```
ans = struct with fields:
  MessageType: 'turtlesim/Pose'
  X: 5.5016
  Y: 6.3965
  Theta: 4.5377
  LinearVelocity: 1
  AngularVelocity: 0
```

Extract the xy points from the messages and plot the robot trajectory.

Use `cellfun` to extract all the X and Y fields from the structure. These fields represent the xy positions of the robot during the rosbag recording.

```
xPoints = cellfun(@(m) double(m.X),msgStructs);
yPoints = cellfun(@(m) double(m.Y),msgStructs);
plot(xPoints,yPoints)
```



## Input Arguments

**filename** — Name of rosbag file and path

string scalar | character vector

Name of file and path for the rosbag you want to access, specified as a string scalar or character vector. This path can be relative or absolute.

## Output Arguments

### **bag** — Selection of rosbag messages

BagSelection object handle

Selection of rosbag messages, returned as a BagSelection object handle.

### **bagInfo** — Information about contents of rosbag

structure

Information about the contents of the rosbag, returned as a structure. This structure contains fields related to the rosbag file and its contents. A sample output for a rosbag as a structure is:

```
Path:      \ros\data\ex_multiple_topics.bag
Version:   2.0
Duration:  2:00s (120s)
Start:     Dec 31 1969 19:03:21.34 (201.34)
End:       Dec 31 1969 19:05:21.34 (321.34)
Size:      23.6 MB
Messages:  36963
Types:     gazebo_msgs/LinkStates [48c080191eb15c41858319b4d8a609c2]
           nav_msgs/Odometry      [cd5e73d190d741a2f92e81eda573aca7]
           rosgraph_msgs/Clock    [a9c97c1d230cfc112e270351a944ee47]
           sensor_msgs/LaserScan  [90c7ef2dc6895d81024acba2ac42f369]
Topics:    /clock                  12001 msgs  : rosgraph_msgs/Clock
           /gazebo/link_states    11999 msgs  : gazebo_msgs/LinkStates
           /odom                  11998 msgs  : nav_msgs/Odometry
           /scan                  965 msgs   : sensor_msgs/LaserScan
```

## See Also

BagSelection | canTransform | getTransform | readMessages | select | timeseries

**Introduced in R2019b**

# rosduration

Create a ROS duration object

## Syntax

```
dur = rosduration
dur = rosduration(totalSecs)
dur = rosduration(secs, nsecs)
```

## Description

`dur = rosduration` returns a default ROS duration object. The properties for seconds and nanoseconds are set to 0.

`dur = rosduration(totalSecs)` initializes the time values for seconds and nanoseconds based on `totalSecs`, which represents the time in seconds as a floating-point number.

`dur = rosduration(secs, nsecs)` initializes the time values for seconds and nanoseconds individually. Both inputs must be integers. Large values for `nsecs` are wrapped automatically with the remainder added to `secs`.

## Examples

### Work with ROS Duration Objects

Create ROS Duration objects, perform addition and subtraction, and compare duration objects. You can also add duration objects to ROS Time objects to get another Time object.

Create a duration using seconds and nanoseconds.

```
dur1 = rosduration(100, 2000000)

dur1 =
  ROS Duration with properties:
    Sec: 100
    Nsec: 2000000
```

Create a duration using a floating-point value. This sets the seconds using the integer portion and nanoseconds with the remainder.

```
dur2 = rosduration(20.5)

dur2 =
  ROS Duration with properties:
    Sec: 20
    Nsec: 500000000
```

Add the two durations together to get a single duration.

```
dur3 = dur1 + dur2
```

```
dur3 =  
  ROS Duration with properties:  
  
    Sec: 120  
    Nsec: 502000000
```

Subtract durations and get a negative duration. You can initialize durations with negative values as well.

```
dur4 = dur2 - dur1
```

```
dur4 =  
  ROS Duration with properties:  
  
    Sec: -80  
    Nsec: 498000000
```

```
dur5 = rosduration(-1,2000000)
```

```
dur5 =  
  ROS Duration with properties:  
  
    Sec: -1  
    Nsec: 2000000
```

Compare durations.

```
dur1 > dur2
```

```
ans = logical  
    1
```

Add a duration to a ROS Time object.

```
time = rostime('now','system')
```

```
time =  
  ROS Time with properties:  
  
    Sec: 1.5830e+09  
    Nsec: 839000000
```

```
timeFuture = time + dur3
```

```
timeFuture =  
  ROS Time with properties:  
  
    Sec: 1.5830e+09  
    Nsec: 341000000
```



## Input Arguments

### **totalSecs — Total time**

0 (default) | scalar

Total time, specified as a floating-point scalar. The integer portion is set to the Sec property with the remainder applied to the Nsec property of the Duration object.

### **secs — Whole seconds**

0 (default) | integer

Whole seconds, specified as an integer. This value is directly set to the Sec property of the Duration object.

---

**Note** The maximum and minimum values for secs are [-2147483648, 2147483647].

---

### **nsecs — Nanoseconds**

0 (default) | positive integer

Nanoseconds, specified as a positive integer. This value is directly set to the NSec property of the Duration object unless it is greater than or equal to  $10^9$ . The value is then wrapped and the remainders are added to the value of secs.

## Output Arguments

### **dur — Duration**

ROS Duration object

Duration, returned as a ROS Duration object with Sec and Nsec properties.

## See Also

rosmesssage | rostime | seconds

## Introduced in R2019b

## rosgenmsg

Generate custom messages from ROS definitions

### Syntax

```
rosgenmsg(folderpath)
```

### Description

`rosgenmsg(folderpath)` generates ROS custom messages in MATLAB by reading ROS custom message and service definitions in the specified folder path. The function expects ROS package folders inside the folder path. These packages contain the message definitions in `.msg` files and the service definitions in `.srv` files. The packages require a `package.xml` file to define its contents.

After calling this function, you can send and receive your custom messages in MATLAB like all other supported messages. You can create these messages using `rosmmessage` or view the list of messages by calling `rosmmsg list`.

---

**Note** To use the function, you must install the ROS Toolbox Interface for the ROS Custom Messages add-on using `rosAddons`.

---

### Examples

#### Generate MATLAB Code for ROS Custom Messages

After you install the support package and prepare your custom message package folder, specify the path to the parent folder and call `rosgenmsg`.

---

**Note** You must set the `folderpath` based on your ROS package setup.

---

```
folderpath = "C:/Users/user1/Documents/robot_custom_msg/";  
rosgenmsg(folderpath)
```

### Input Arguments

#### **folderpath** — Path to ROS package folders

string scalar | character vector

Path to the parent folder of ROS message packages, specified as a string scalar or character vector. The parent folder should contain a `package.xml` file and package folders. These folders contain message definitions in `.msg` files and the service definitions in `.srv` files. The packages require a `package.xml` file to define its contents.

Example: `"/opt/ros/kinetic/share"`

## Limitations

- You must install the ROS Toolbox Interface for ROS Custom Messages add-on using `rosAddons` to use this function.

## See Also

`rosAddons`

## Topics

“Create Custom Messages from ROS Package”

“ROS Custom Message Support”

## External Websites

ROS Tutorials: Defining Custom Messages

ROS Tutorials: Creating a ROS msg and srv

## Introduced in R2019b

# rosinit

Connect to ROS network

## Syntax

```
rosinit
rosinit(hostname)
rosinit(hostname,port)
rosinit(URI)
rosinit( ____,Name,Value)
```

## Description

`rosinit` starts the global ROS node with a default MATLAB name and tries to connect to a ROS master running on `localhost` and port 11311. If the global ROS node cannot connect to the ROS master, `rosinit` also starts a ROS core in MATLAB, which consists of a ROS master, a ROS parameter server, and a `rosout` logging node.

`rosinit(hostname)` tries to connect to the ROS master at the host name or IP address specified by `hostname`. This syntax uses 11311 as the default port number.

`rosinit(hostname,port)` tries to connect to the host name or IP address specified by `hostname` and the port number specified by `port`.

`rosinit(URI)` tries to connect to the ROS master at the given resource identifier, `URI`, for example, "`http://192.168.1.1:11311`".

`rosinit( ____,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments.

Using `rosinit` is a prerequisite for most ROS-related tasks in MATLAB because:

- Communicating with a ROS network requires a ROS node connected to a ROS master.
- By default, ROS functions in MATLAB operate on the global ROS node, or they operate on objects that depend on the global ROS node.

For example, after creating a global ROS node with `rosinit`, you can subscribe to a topic on the global ROS node. When another node on the ROS network publishes messages on that topic, the global ROS node receives the messages.

If a global ROS node already exists, then `rosinit` restarts the global ROS node based on the new set of arguments.

For more advanced ROS networks, connecting to multiple ROS nodes or masters is possible using the `Node` object.

## Examples

## Start ROS Core and Global Node

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:50465/.  
Initializing global node /matlab_global_node_62964 with NodeURI http://bat5110win64:50469/
```

When you are finished, shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_62964 with NodeURI http://bat5110win64:50469/  
Shutting down ROS master on http://bat5110win64:50465/.
```

## Start Node and Connect to ROS Master at Specified IP Address

```
rosinit('192.168.17.128')
```

```
Initializing global node /matlab_global_node_57409 with NodeURI http://192.168.17.1:57782/
```

Shut down the ROS network when you are finished.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_57409 with NodeURI http://192.168.17.1:57782/
```

## Start Global Node at Given IP and NodeName

```
rosinit('192.168.17.128', 'NodeHost', '192.168.17.1', 'NodeName', '/test_node')
```

```
Initializing global node /test_node with NodeURI http://192.168.17.1:57633/
```

Shut down the ROS network when you are finished.

```
rosshutdown
```

```
Shutting down global node /test_node with NodeURI http://192.168.17.1:57633/
```

## Input Arguments

### hostname — Host name or IP address

string scalar | character vector

Host name or IP address, specified as a string scalar or character vector.

### port — Port number

numeric scalar

Port number used to connect to the ROS master, specified as a numeric scalar.

### URI — URI for ROS master

string scalar | character vector

URI for ROS master, specified as a string scalar or character vector. Standard format for URIs is either `http://ipaddress:port` or `http://hostname:port`

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `"NodeHost", "192.168.1.1"`

### **NodeHost — Host name or IP address**

character vector

Host name or IP address under which the node advertises itself to the ROS network, specified as the comma-separated pair consisting of `"NodeHost"` and a character vector.

Example: `"comp-home"`

### **NodeName — Global node name**

character vector

Global node name, specified as the comma-separated pair consisting of `"NodeName"` and a character vector. The node that is created through `roslaunch` is registered on the ROS network with this name.

Example: `"NodeName", "/test_node"`

## **See Also**

Node | `roslaunch`

## **Topics**

“Connect to a ROS Network”

## **Introduced in R2019b**

# rosmessage

Create ROS messages

## Syntax

```
msg = rosmessage(msgtype)
```

```
msg = rosmessage(pub)
```

```
msg = rosmessage(sub)
```

```
msg = rosmessage(client)
```

```
msg = rosmessage(server)
```

## Description

`msg = rosmessage(msgtype)` creates an empty ROS message object with message type. The `msgtype` string scalar is case-sensitive and no partial matches are allowed. It must match a message on the list given by calling `rosmmsg("list")`.

`msg = rosmessage(pub)` creates an empty message determined by the topic published by `pub`.

`msg = rosmessage(sub)` creates an empty message determined by the subscribed topic of `sub`.

`msg = rosmessage(client)` creates an empty message determined by the service associated with `client`.

`msg = rosmessage(server)` creates an empty message determined by the service type of `server`.

## Examples

### Create Empty String Message

```
strMsg = rosmessage('std_msgs/String')
```

```
strMsg =  
  ROS String message with properties:
```

```
  MessageType: 'std_msgs/String'  
  Data: ''
```

Use `showdetails` to show the contents of the message

### Create ROS Publisher and Send Data

Start ROS master.

```
roscpp
```

```
Initializing ROS master on http://bat5110win64:57316/.
Initializing global node /matlab_global_node_01158 with NodeURI http://bat5110win64:57323/

Create publisher for the '/chatter' topic with the 'std_msgs/String' message type.

chatpub = rospublisher('/chatter', 'std_msgs/String');

Create a message to send. Specify the Data property.

msg = rosmessage(chatpub);
msg.Data = 'test phrase';

Send the message via the publisher.

send(chatpub, msg);

Shut down the ROS network.

rosshutdown

Shutting down global node /matlab_global_node_01158 with NodeURI http://bat5110win64:57323/
Shutting down ROS master on http://bat5110win64:57316/.
```

### Create and Access Array of ROS Messages

You can create an object array to store multiple messages. The array is indexable, similar to any other array. You can modify properties of each object or access specific properties from each element using dot notation.

Create a two - message object array.

```
msgArray = [rosmessage('std_msgs/String') rosmessage('std_msgs/String')]

msgArray =
  1x2 ROS String message array with properties:

  MessageType
  Data
```

Assign data to individual object elements of the array.

```
msgArray(1).Data = 'Some string';
msgArray(2).Data = 'Other string';
```

Read all the Data properties from the message objects into a cell array.

```
allData = {msgArray.Data}

allData = 1x2 cell
    {'Some string'}    {'Other string'}
```

### Preallocate ROS Message Array

To preallocate an array using ROS messages, use the `arrayfun` or `cellfun` functions instead of `repmat`. These functions properly create object or cell arrays for handle classes.



Preallocate an object array of ROS messages.

```
msgArray = arrayfun(@(~) rosmesssage('std_msgs/String'), zeros(1,50));
```

Preallocate a cell array of ROS messages.

```
msgCell = cellfun(@(~) rosmesssage('std_msgs/String'), cell(1,50), 'UniformOutput', false);
```

## Input Arguments

### **messsagetype** — Message type

string scalar | character vector

Message type, specified as a string scalar or character vector. The string is case-sensitive and no partial matches are allowed. It must match a message on the list given by calling `rosmssg("list")`.

### **pub** — ROS publisher

Publisher object handle

ROS publisher, specified as a Publisher object handle. You can create the object using `rospublisher`.

### **sub** — ROS subscriber

Subscriber object handle

ROS subscriber, specified as a Subscriber object handle. You can create the object using `rossubscriber`.

### **client** — ROS service client

ServiceClient object handle

ROS service client, specified as a ServiceClient object handle. You can create the object using `rossvcclient`.

### **server** — ROS service server

ServiceServer object handle

ROS service server, specified as a ServiceServer object handle. You can create the object using `rossvcserver`.

## Output Arguments

### **msg** — ROS message

Message object handle

ROS message, returned as a Message object handle.

## See Also

`rosAddons` | `rosmssg` | `rostopic`

### Topics

“Work with Basic ROS Messages”

“Built-In Message Support”

**Introduced in R2019b**

# rosmg

Retrieve information about ROS messages and message types

## Syntax

```
rosmg show msgtype
rosmg md5 msgtype
rosmg list
```

```
msginfo = rosmg("show", msgtype)
msgmd5 = rosmg("md5", msgtype)
msglist = rosmg("list")
```

## Description

`rosmg show msgtype` returns the definition of the `msgtype` message.

`rosmg md5 msgtype` returns the MD5 checksum of the `msgtype` message.

`rosmg list` returns all available message types that you can use in MATLAB.

`msginfo = rosmg("show", msgtype)` returns the definition of the `msgtype` message as a character vector.

`msgmd5 = rosmg("md5", msgtype)` returns the 'MD5' checksum of the `msgtype` message as a character vector.

`msglist = rosmg("list")` returns a cell array containing all available message types that you can use in MATLAB.

## Examples

### Retrieve Message Type Definition

```
msgInfo = rosmg('show', 'geometry_msgs/Point')
msgInfo =
    '% This contains the position of a point in free space
    double X
    double Y
    double Z
    '
```

### Get the MD5 Checksum of Message Type

```
msgMd5 = rosmg('md5', 'geometry_msgs/Point')
```

```
msgMd5 =  
'4a842b65f413084dc2b10fb484ea7f17'
```

## Input Arguments

### **msgtype** — ROS message type

character vector

ROS message type, specified as a character vector. `msgType` must be a valid ROS message type from ROS that MATLAB supports.

Example: "std\_msgs/Int8"

## Output Arguments

### **msginfo** — Details of message definition

character vector

Details of the information inside the ROS message definition, returned as a character vector.

### **msgmd5** — MD5 checksum hash value

character vector

MD5 checksum hash value, returned as a character vector. The MD5 output is a character vector representation of the 16-byte hash value that follows the MD5 standard.

### **msglist** — List of all message types available in MATLAB

cell array of character vectors

List of all message types available in MATLAB, returned as a cell array of character vectors.

**Introduced in R2019b**

# rosnode

Retrieve information about ROS network nodes

## Syntax

```
rosnode list
rosnode info nodename
rosnode ping nodename

odelist = rosnode("list")
nodeinfo = rosnode("info",nodename)
rosnode("ping",nodename)
```

## Description

`rosnode list` returns a list of all nodes registered on the ROS network. Use these nodes to exchange data between MATLAB and the ROS network.

`rosnode info nodename` returns a structure containing the name, URI, publications, subscriptions, and services of a specific ROS node, `nodename`.

`rosnode ping nodename` pings a specific node, `nodename`, and displays the response time.

`odelist = rosnode("list")` returns a cell array of character vectors containing the nodes registered on the ROS network.

`nodeinfo = rosnode("info",nodename)` returns a structure containing the name, URI, publications, subscriptions, and services of a specific ROS node, `nodename`.

`rosnode("ping",nodename)` pings a specific node, `nodename` and displays the response time.

## Examples

### Retrieve List of ROS Nodes

**Note:** This example requires a valid ROS network to be active with ROS nodes previously set up.

Connect to the ROS network. Specify the IP address for your specific network.

```
rosinit('192.168.17.128')
```

```
Initializing global node /matlab_global_node_99071 with NodeURI http://192.168.17.1:64076/
```

List the nodes available from the ROS master.

```
rosnode list

/gazebo
/laserscan_nodelet_manager
/matlab_global_node_99071
```

```
/mobile_base_nodelet_manager  
/robot_state_publisher  
/rosout
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_99071 with NodeURI http://192.168.17.1:64076/
```

### Retrieve ROS Node Information

Connect to the ROS network. Specify the IP address for your specific network.

```
rosinit('192.168.17.128')
```

```
Initializing global node /matlab_global_node_96994 with NodeURI http://192.168.17.1:64267/
```

Get information on the '/robot\_state\_publisher' node. This node is available on the ROS master.

```
nodeinfo = rosnode('info', '/robot_state_publisher')
```

```
nodeinfo = struct with fields:  
  NodeName: '/robot_state_publisher'  
  URI: 'http://192.168.17.128:43330/'  
  Publications: [3×1 struct]  
  Subscriptions: [2×1 struct]  
  Services: [2×1 struct]
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_96994 with NodeURI http://192.168.17.1:64267/
```

### Ping ROS Node

Connect to the ROS network. Specify the IP address for your specific network.

```
rosinit('192.168.17.128')
```

```
Initializing global node /matlab_global_node_59489 with NodeURI http://192.168.17.1:64471/
```

Ping the '/robot\_state\_publisher' node. This node is available on the ROS master.

```
nodeinfo = rosnode('info', '/robot_state_publisher')
```

```
nodeinfo = struct with fields:  
  NodeName: '/robot_state_publisher'  
  URI: 'http://192.168.17.128:43330/'  
  Publications: [3×1 struct]  
  Subscriptions: [2×1 struct]  
  Services: [2×1 struct]
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_59489 with NodeURI http://192.168.17.1:64471/
```

## Input Arguments

**nodename** — Name of node

string scalar | character vector

Name of node, specified as a string scalar or character vector. The name of the node must match the name given in ROS.

## Output Arguments

**nodeinfo** — Information about ROS node

structure

Information about ROS node, returned as a structure containing these fields: `nodeName`, `URI`, `Publications`, `Subscriptions`, and `Services`. Access these properties using dot syntax, for example, `nodeinfo.NodeName`.

**odelist** — List of node names available

cell array of character vectors

List of node names available, returned as a cell array of character vectors.

## See Also

`rosinit` | `rostopic`

**Introduced in R2019b**

## rosparam

Access ROS parameter server values

### Syntax

```
list = rosparam("list")
list = rosparam("list",namespace)
pvalOut = rosparam("get",pname)
pvalOut = rosparam("get",namespace)
rosparam("set",pname,pval)
rosparam("delete",pname)
rosparam("delete",namespace)

ptree = rosparam
```

### Description

`list = rosparam("list")` returns the list of all ROS parameter names from the ROS master.

**Simplified form:** `rosparam list`

`list = rosparam("list",namespace)` returns the list of all parameter names under the specified ROS namespace.

**Simplified form:** `rosparam list namespace`

`pvalOut = rosparam("get",pname)` retrieves the value of the specified parameter.

**Simplified form:** `rosparam get pname`

`pvalOut = rosparam("get",namespace)` retrieves the values of all parameters under the specified namespace as a structure.

**Simplified form:** `rosparam get namespace`

`rosparam("set",pname,pval)` sets a value for a specified parameter name. If the parameter name does not exist, the function adds a new parameter in the parameter tree.

**Simplified form:** `rosparam set pname pval`

See "Limitations" on page 1-113 for limitations on `pval`.

`rosparam("delete",pname)` deletes a parameter from the parameter tree. If the parameter does not exist, the function displays an error.

**Simplified form:** `rosparam delete pname`

`rosparam("delete",namespace)` deletes all parameters under the given namespace from the parameter tree.

**Simplified form:** `rosparam delete namespace`



`ptree = rosparam` creates a parameter tree object, `ptree`. After `ptree` is created, the connection to the parameter server remains persistent until the object is deleted or the ROS master becomes unavailable.

A ROS parameter tree communicates with the ROS parameter server. The ROS parameter server can store strings, integers, doubles, Booleans and cell arrays. The parameters are accessible by every node in the ROS network. Use the parameters to store static data such as configuration parameters. Use the `get`, `set`, `has`, `search`, and `del` functions to manipulate and view parameter values.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

- 32-bit integer — `int32`
- Boolean — `logical`
- double — `double`
- string — character vector (`char`)
- list — cell array (`cell`)
- dictionary — structure (`struct`)

## Examples

### Get and Set Parameter Values

Connect to a ROS network to set and get ROS parameter values on the ROS parameter tree. You can get lists of parameters in their given namespaces as well. This example uses the simplified form that mimics the ROS command-line interface.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:58928/.
Initializing global node /matlab_global_node_07637 with NodeURI http://bat5110win64:58936/
```

Set parameter values.

```
rosparam set /string_param 'param_value'
rosparam set /double_param 1.2
```

To set a list parameter, use the functional form.

```
rosparam('set', '/list_param', {int32(5), 124.1, -20, 'some_string'});
```

Get the list of parameters using the command-line form.

```
rosparam list
/double_param
/list_param
/string_param
```

List parameters in a specific namespace.

```
rosparam list /double
```

```
/double_param
```

Get the value of a parameter.

```
rosparam get /list_param
```

```
{5, 124.1, -20, some_string}
```

Delete a parameter. List the parameters to verify it was deleted.

```
rosparam delete /double_param  
rosparam list
```

```
/list_param  
/string_param
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_07637 with NodeURI http://bat5110win64:58936/  
Shutting down ROS master on http://bat5110win64:58928/.
```

### Create Parameter Tree Object and View Parameters

Connect to the ROS network. ROS parameters should already be available on the ROS master.

```
rosinit('192.168.17.128')
```

```
Initializing global node /matlab_global_node_91663 with NodeURI http://192.168.17.1:52951/
```

Create a ParameterTree object using rosparam.

```
ptree = rosparam;
```

List the available parameters on the ROS master.

```
ptree.AvailableParameters
```

```
ans = 33x1 cell array  
    {'/bumper2pointcloud/pointcloud_radius' }  
    {'/camera/imager_rate' }  
    {'/camera/rgb/image_raw/compressed/format' }  
    {'/camera/rgb/image_raw/compressed/jpeg_quality'}  
    {'/camera/rgb/image_raw/compressed/png_level' }  
    {'/cmd_vel_mux/yaml_cfg_file' }  
    {'/depthimage_to_laserscan/output_frame_id' }  
    {'/depthimage_to_laserscan/range_max' }  
    {'/depthimage_to_laserscan/range_min' }  
    {'/depthimage_to_laserscan/scan_height' }  
    {'/depthimage_to_laserscan/scan_time' }  
    {'/gazebo/auto_disable_bodies' }  
    {'/gazebo/cfm' }  
    {'/gazebo/contact_max_correcting_vel' }  
    {'/gazebo/contact_surface_layer' }  
    {'/gazebo/erp' }  
    {'/gazebo/gravity_x' }
```

```

{/gazebo/gravity_y'      }
{/gazebo/gravity_z'      }
{/gazebo/max_contacts'   }
{/gazebo/max_update_rate' }
{/gazebo/sor_pgs_iters'  }
{/gazebo/sor_pgs_precon_iters' }
{/gazebo/sor_pgs_rms_error_tol' }
{/gazebo/sor_pgs_w'     }
{/gazebo/time_step'     }
{/robot_description'     }
{/robot_state_publisher/publish_frequency' }
{/roscdistro'           }
{/roslaunch/uris/host_192_168_17_128__34863' }
{/rosversion'           }
{/run_id'               }
{/use_sim_time'         }

```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_91663 with NodeURI http://192.168.17.1:52951/
```

### Set A Dictionary Of Parameter Values

Use structures to specify a dictionary of ROS parameters under a specific namespace.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:56297/.
```

```
Initializing global node /matlab_global_node_59257 with NodeURI http://bat5110win64:56304/
```

Create a dictionary of parameter values. This dictionary contains the information relevant to an image. Display the structure to verify values.

```
image = imread('peppers.png');

pval.ImageWidth = size(image,1);
pval.ImageHeight = size(image,2);
pval.ImageTitle = 'peppers.png';
disp(pval)
```

```

ImageWidth: 384
ImageHeight: 512
ImageTitle: 'peppers.png'

```

Set the dictionary of values using the desired namespace.

```
rosparam('set', 'ImageParam', pval)
```

Get the parameters using the namespace. Verify the values.

```
pval2 = rosparam('get', 'ImageParam')
```

```
pval2 = struct with fields:
  ImageHeight: 512
  ImageTitle: 'peppers.png'
  ImageWidth: 384
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_59257 with NodeURI http://bat5110win64:56304/
Shutting down ROS master on http://bat5110win64:56297/.
```

## Input Arguments

### **namespace** — ROS parameter namespace

string scalar | character vector

ROS parameter namespace, specified as a string scalar or character vector. All parameter names starting with this string are listed when calling `rosparam("list", namespace)`.

### **pname** — ROS parameter name

string scalar | character vector

ROS parameter name, specified as a string scalar or character vector.

### **pval** — ROS parameter value or dictionary of values

int32 | logical | double | string scalar | character vector | cell array | structure

ROS parameter value or dictionary of values, specified as a supported data type.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed:

- 32-bit integers — `int32`
- Booleans — `logical`
- doubles — `double`
- strings — string scalar, `string`, or character vector, `char`
- lists — cell array
- dictionaries — structure

## Output Arguments

### **list** — Parameter list

cell array of character vectors

Parameter list, returned as a cell array of character vectors. This is a list of all parameters available on the ROS master.

### **ptree** — Parameter tree

ParameterTree object handle

Parameter tree, returned as a `ParameterTree` object handle. Use this object to reference parameter information, for example, `ptree.AvailableFrames`.

### **pvalOut — ROS parameter value or dictionary of values**

`int32` | `logical` | `double` | `character vector` | `cell array` | `structure`

ROS parameter value, specified as a supported MATLAB data type. When specifying the namespace input argument, `pvalOut` is returned as a structure of parameter value under the given namespace.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

ROS Data Type	MATLAB Data Type
32-bit integer	<code>int32</code>
Boolean	<code>logical</code>
<code>double</code>	<code>double</code>
<code>string</code>	<code>character vector (char)</code>
<code>list</code>	<code>cell array (cell)</code>
<code>dictionary</code>	<code>structure (struct)</code>

## Limitations

- **Unsupported Data Types:** Base64-encoded binary data and iso8601 data from ROS are not supported.
- **Simplified Commands:** When using the simplified command `rosparam set pname pval`, the parameter value is interpreted as:
  - `logical` — If `pval` is "true" or "false"
  - `int32` — If `pval` is an integer, for example, 5
  - `double` — If `pval` is a fractional number, for example, 1.256
  - `character vector` — If `pval` is any other value

## See Also

### Functions

`del` | `get` | `has` | `search` | `set`

### Objects

`ParameterTree`

### Introduced in R2019b

## rosservice

Retrieve information about services in ROS network

### Syntax

```
rosservice list
rosservice info svcname
rosservice type svcname
rosservice uri svcname
```

```
svclist = rosservice("list")
svcinfo = rosservice("info",svcname)
svctype = rosservice("type",svcname)
svcuri = rosservice("uri",svcname)
```

### Description

`rosservice list` returns a list of service names for all of the active service servers on the ROS network.

`rosservice info svcname` returns information about the specified service, `svcname`.

`rosservice type svcname` returns the service type.

`rosservice uri svcname` returns the URI of the service.

`svclist = rosservice("list")` returns a list of service names for all of the active service servers on the ROS network. `svclist` contains a cell array of service names.

`svcinfo = rosservice("info",svcname)` returns a structure of information, `svcinfo`, about the service, `svcname`.

`svctype = rosservice("type",svcname)` returns the service type of the service as a character vector.

`svcuri = rosservice("uri",svcname)` returns the URI of the service as a character vector.

### Examples

#### View List of ROS Services

Connect to the ROS network. Specify the IP address of your specific network.

```
rosinit('192.168.17.128')
```

```
Initializing global node /matlab_global_node_23375 with NodeURI http://192.168.17.1:64875/
```

List the services available on the ROS master.

```
rosservice list
```

```

/camera/rgb/image_raw/compressed/set_parameters
/camera/set_camera_info
/camera/set_parameters
/depthimage_to_laserscan/set_parameters
/gazebo/apply_body_wrench
/gazebo/apply_joint_effort
/gazebo/clear_body_wrenches
/gazebo/clear_joint_forces
/gazebo/delete_model
/gazebo/get_joint_properties
/gazebo/get_link_properties
/gazebo/get_link_state
/gazebo/get_loggers
/gazebo/get_model_properties
/gazebo/get_model_state
/gazebo/get_physics_properties
/gazebo/get_world_properties
/gazebo/pause_physics
/gazebo/reset_simulation
/gazebo/reset_world
/gazebo/set_joint_properties
/gazebo/set_link_properties
/gazebo/set_link_state
/gazebo/set_logger_level
/gazebo/set_model_configuration
/gazebo/set_model_state
/gazebo/set_parameters
/gazebo/set_physics_properties
/gazebo/spawn_gazebo_model
/gazebo/spawn_sdf_model
/gazebo/spawn_urdf_model
/gazebo/unpause_physics
/laserscan_nodelet_manager/get_loggers
/laserscan_nodelet_manager/list
/laserscan_nodelet_manager/load_nodelet
/laserscan_nodelet_manager/set_logger_level
/laserscan_nodelet_manager/unload_nodelet
/mobile_base_nodelet_manager/get_loggers
/mobile_base_nodelet_manager/list
/mobile_base_nodelet_manager/load_nodelet
/mobile_base_nodelet_manager/set_logger_level
/mobile_base_nodelet_manager/unload_nodelet
/robot_state_publisher/get_loggers
/robot_state_publisher/set_logger_level
/rosout/get_loggers
/rosout/set_logger_level

```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_23375 with NodeURI http://192.168.17.1:64875/
```

### Get Information, Service Type, and URI for ROS Service

Connect to the ROS network. Specify the IP address of your specific network.

```
rosinit('192.168.17.128')
Initializing global node /matlab_global_node_09263 with NodeURI http://192.168.17.1:65083/
Get information on the |gazebo/pause_physics| service.
svcinfo = rosservice('info', 'gazebo/pause_physics')
svcinfo = struct with fields:
    Node: '/gazebo'
    URI: 'rosrpc://192.168.17.128:52059'
    Type: 'std_srvs/Empty'
    Args: {}

Get the service type.
svctype = rosservice('type', 'gazebo/pause_physics')
svctype =
'std_srvs/Empty'

Get the service URI.
svcuri = rosservice('uri', 'gazebo/pause_physics')
svcuri =
'rosrpc://192.168.17.128:52059'

Shut down the ROS network.
rosshutdown
Shutting down global node /matlab_global_node_09263 with NodeURI http://192.168.17.1:65083/
```

## Input Arguments

**svcname** — Name of service  
string scalar | character vector

Name of service, specified as a string scalar or character vector. The service name must match its name in the ROS network.

## Output Arguments

**svcinfo** — Information about a ROS service  
character vector

Information about a ROS service, returned as a character vector.

**svclist** — List of available ROS services  
cell array of character vectors

List of available ROS services, returned as a cell array of character vectors.

**svctype** — Type of ROS service  
character vector



Type of ROS service, returned as a character vector.

**svcuri – URI for accessing service**

character vector

URI for accessing service, returned as a character vector.

**See Also**

rosinit | rosparam

**Introduced in R2019b**

## roshutdown

Shut down ROS system

### Syntax

```
roshutdown
```

### Description

roshutdown shuts down the global node and, if it is running, the ROS master. When you finish working with the ROS network, use roshutdown to shut down the global ROS entities created by rosinit. If the global node and ROS master are not running, this function has no effect.

---

**Note** After calling roshutdown, any ROS entities (objects) that depend on the global node like subscribers created with rossubscriber, are deleted and become unstable.

---

Prior to calling roshutdown, call clear on these objects for a clean removal of ROS entities.

---

### Examples

#### Start ROS Core and Global Node

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:50465/.
```

```
Initializing global node /matlab_global_node_62964 with NodeURI http://bat5110win64:50469/
```

When you are finished, shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_62964 with NodeURI http://bat5110win64:50469/
```

```
Shutting down ROS master on http://bat5110win64:50465/.
```

### See Also

```
rosinit
```

**Introduced in R2019b**

# rostopic

Retrieve information about ROS topics

## Syntax

```
rostopic list
rostopic echo topicname
rostopic info topicname
rostopic type topicname

topiclist = rostopic("list")
msg = rostopic("echo", topicname)
topicinfo = rostopic("info", topicname)
msgtype = rostopic("type", topicname)
```

## Description

`rostopic list` returns a list of ROS topics from the ROS master.

`rostopic echo topicname` returns the messages being sent from the ROS master about a specific topic, `topicname`. To stop returning messages, press **Ctrl+C**.

`rostopic info topicname` returns the message type, publishers, and subscribers for a specific topic, `topicname`.

`rostopic type topicname` returns the message type for a specific topic.

`topiclist = rostopic("list")` returns a cell array containing the ROS topics from the ROS master. If you do not define the output argument, the list is returned in the MATLAB Command Window.

`msg = rostopic("echo", topicname)` returns the messages being sent from the ROS master about a specific topic, `topicname`. To stop returning messages, press **Ctrl+C**. If the output argument is defined, then `rostopic` returns the first message that arrives on that topic.

`topicinfo = rostopic("info", topicname)` returns a structure containing the message type, publishers, and subscribers for a specific topic, `topicname`.

`msgtype = rostopic("type", topicname)` returns a character vector containing the message type for the specified topic, `topicname`.

## Examples

### Get List of ROS Topics

Connect to the ROS network. Specify the IP address of the ROS device.

```
rosinit('192.168.17.129',11311)
```

```
Initializing global node /matlab_global_node_01393 with NodeURI http://192.168.17.1:49865/
```

List the ROS topic available on the ROS master.

```
rostopic list
```

```
/camera/depth/camera_info
/camera/depth/image_raw
/camera/depth/points
/camera/parameter_descriptions
/camera/parameter_updates
/camera/rgb/camera_info
/camera/rgb/image_raw
/camera/rgb/image_raw/compressed
/camera/rgb/image_raw/compressed/parameter_descriptions
/camera/rgb/image_raw/compressed/parameter_updates
/clock
/cmd_vel_mux/active
/cmd_vel_mux/input/navi
/cmd_vel_mux/input/safety_controller
/cmd_vel_mux/input/teleop
/cmd_vel_mux/parameter_descriptions
/cmd_vel_mux/parameter_updates
/depthimage_to_laserscan/parameter_descriptions
/depthimage_to_laserscan/parameter_updates
/fibonacci/cancel
/fibonacci/feedback
/fibonacci/goal
/fibonacci/result
/fibonacci/status
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/joint_states
/laserscan_nodelet_manager/bond
/mobile_base/commands/motor_power
/mobile_base/commands/reset_odometry
/mobile_base/commands/velocity
/mobile_base/events/bumper
/mobile_base/events/cliff
/mobile_base/sensors/bumper_pointcloud
/mobile_base/sensors/core
/mobile_base/sensors/imu_data
/mobile_base_nodelet_manager/bond
/odom
/rosout
/rosout_agg
/scan
/tf
/tf_static
```

### **Get ROS Topic Info**

Connect to ROS network. Specify the IP address of the ROS device.

```
rosinit('192.168.17.129',11311)
```

```
Initializing global node /matlab_global_node_29625 with NodeURI http://192.168.17.1:50079/
```

Show information on a specific ROS topic.

```
rostopic info camera/depth/points
```

```
Type: sensor_msgs/PointCloud2
```

Publishers:

```
* /gazebo (http://192.168.17.129:33044/)
```

Subscribers:

### Get ROS Topic Message Type

Connect to the ROS network. Specify the IP address of the ROS device.

```
rosinit('192.168.17.129',11311)
```

```
Initializing global node /matlab_global_node_19218 with NodeURI http://192.168.17.1:55966/
```

Get the message type for a specific topic. Create a message from the message type to publish to the topic.

```
msgtype = rostopic('type','camera/depth/points');
msg = rosmessag(msgtype);
```

## Input Arguments

### topicname — ROS topic name

string scalar | character vector

ROS topic name, specified as a string scalar or character vector. The topic name must match one of the topics that `rostopic("list")` outputs.

## Output Arguments

### topiclist — List of topics from the ROS master

cell array of character vectors

List of topics from the ROS master, returned as a cell array of character vectors.

### msg — ROS message for a given topic

object handle

ROS message for a given topic, returned as an object handle.

### topicinfo — Information about a given ROS topic

structure

Information about a ROS topic, returned as a structure. The `topicinfo` syntax includes the message type, publishers, and subscribers associated with that topic.

**msgtype — Message type for a ROS topic**

character vector

Message type for a ROS topic, returned as a character vector.

**Introduced in R2019b**

# rostype

Access available ROS message types

## Syntax

```
rostype
```

## Description

`rostype` creates a blank message of a certain type by browsing the list of available message types. You can use tab completion and do not have to rely on typing error-free message type character vectors. By typing `rostype.partialname`, and pressing **Tab**, a list of matching message types appears in a list. By setting the message type equal to a variable, you can create a character vector of that message type. Alternatively, you can create the message by supplying the message type directly into `rosmessage` as an input argument.

## Examples

### Create ROS Message Type and ROS Message

Create Message Type String

```
t = rostype.std_msgs_String
t =
'std_msgs/String'
```

Create ROS Message from ROS Type

```
msg = rosmessage(rostype.std_msgs_String)
msg =
  ROS String message with properties:
    MessageType: 'std_msgs/String'
    Data: ''
```

Use `showdetails` to show the contents of the message

## See Also

`rosmessage` | `rostopic`

## Topics

“Built-In Message Support”

“Work with Basic ROS Messages”

**Introduced in R2019b**

## runCore

Start ROS core

### Syntax

```
runCore(device)
```

### Description

`runCore(device)` starts the ROS core on the connected device. The ROS master uses a default port number of 11311.

### Examples

#### Run ROS Core on ROS Device

Connect to a remote ROS device and start a ROS core. The ROS core is needed to run ROS nodes to communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.17.128';  
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =  
  rosdevice with properties:  
  
    DeviceAddress: '192.168.17.128'  
    Username: 'user'  
    ROSFolder: '/opt/ros/indigo'  
    CatkinWorkspace: '~/catkin_ws'  
    AvailableNodes: {0x1 cell}
```

Run a ROS core and check if it is running.

```
runCore(d)
```

Another `roscore` / ROS master is already running on the ROS device. Use the `'stopCore'` function to

```
running = isCoreRunning(d)
```

```
running = logical  
         1
```

Stop the ROS core and confirm that it is no longer running.



```
stopCore(d)
running = isCoreRunning(d)

running = logical
         1
```

## Input Arguments

### **device – ROS device**

rosdevice object

ROS device, specified as a rosdevice object.

## See Also

[isCoreRunning](#) | [rosdevice](#) | [stopCore](#)

## Topics

“Generate a Standalone ROS Node from Simulink®”

## Introduced in R2019b

## runNode

Start ROS node

### Syntax

```
runNode(device,modelName)
runNode(device,modelName,masterURI)
runNode(device,modelName,masterURI,nodeHost)
```

### Description

`runNode(device,modelName)` starts the ROS node associated with the deployed Simulink model named `modelName`. The ROS node must be deployed in the Catkin workspace specified by the `CatkinWorkspace` property of the input `rosdevice` object, `device`. By default, the node connects to the ROS master that MATLAB is connected to with the `device.DeviceAddress` property.

`runNode(device,modelName,masterURI)` connects to the specified master URI.

`runNode(device,modelName,masterURI,nodeHost)` connects to the specified master URI and node host. The node advertises its address as the host name or IP address given in `nodeHost`.

### Examples

#### Run ROS Node on ROS Device

Connect to a remote ROS device and start a ROS node. Run a ROS core so that ROS nodes can communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device already contains the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.203.129';
d = rosdevice(ipaddress,'user','password');
d.ROSFolder = '/opt/ros/indigo';
d.CatkinWorkspace = '~/catkin_ws_test'

d =
  rosdevice with properties:

    DeviceAddress: '192.168.203.129'
    Username: 'user'
    ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws_test'
    AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. Connect MATLAB® to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

```
runCore(d)
rosinit(d.DeviceAddress,11311)
```

Initializing global node /matlab\_global\_node\_84497 with NodeURI http://192.168.203.1:56034/

Check the available ROS nodes on the connected ROS device. These nodes listed were generated from Simulink® models following the process in the “Get Started with ROS in Simulink®” example.

d.AvailableNodes

```
ans = 1x2 cell
      {'robotcontroller'}   {'robotcontroller2'}
```

Run a ROS node and specify the node name. Check if the node is running.

```
runNode(d, 'RobotController')
running = isNodeRunning(d, 'RobotController')
```

```
running = logical
         1
```

Stop the ROS node. Disconnect from the ROS network. Stop the ROS core.

```
stopNode(d, 'RobotController')
rosshutdown
```

Shutting down global node /matlab\_global\_node\_84497 with NodeURI http://192.168.203.1:56034/

```
stopCore(d)
```

## Run Multiple ROS Nodes

Run multiple ROS nodes on a connected ROS device. ROS nodes can be generated using Simulink® models to perform different tasks on the ROS network. These nodes are then deployed on a ROS device and can be run independently of Simulink®.

This example uses two different Simulink models that have been deployed as ROS nodes. See “Generate a Standalone ROS Node from Simulink®” and follow the instructions to generate and deploy a ROS node. Do this twice and name them 'robotcontroller' and 'robotcontroller2'. The 'robotcontroller' node sends velocity commands to a robot to navigate it to a given point. The 'robotcontroller2' node uses the same model, but doubles the linear velocity to drive the robot faster.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using runNode.

```
ipaddress = '192.168.203.129';
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
rosdevice with properties:
```

```
    DeviceAddress: '192.168.203.129'
```

```
    Username: 'user'  
    ROSFolder: '/opt/ros/indigo'  
    CatkinWorkspace: '~/catkin_ws'  
    AvailableNodes: {0x1 cell}
```

```
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
d =
```

```
  rosdevice with properties:
```

```
    DeviceAddress: '192.168.203.129'  
    Username: 'user'  
    ROSFolder: '/opt/ros/indigo'  
    CatkinWorkspace: '~/catkin_ws_test'  
    AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. The ROS Core is the master enables you to run ROS nodes on your ROS device. Connect MATLAB® to the ROS master using `rosinit`. For this example, the port is set to 11311. `rosinit` can automatically select a port for you without specifying this input.

```
runCore(d)  
rosinit(d.DeviceAddress,11311)
```

```
Initializing global node /matlab_global_node_66434 with NodeURI http://192.168.203.1:59395/
```

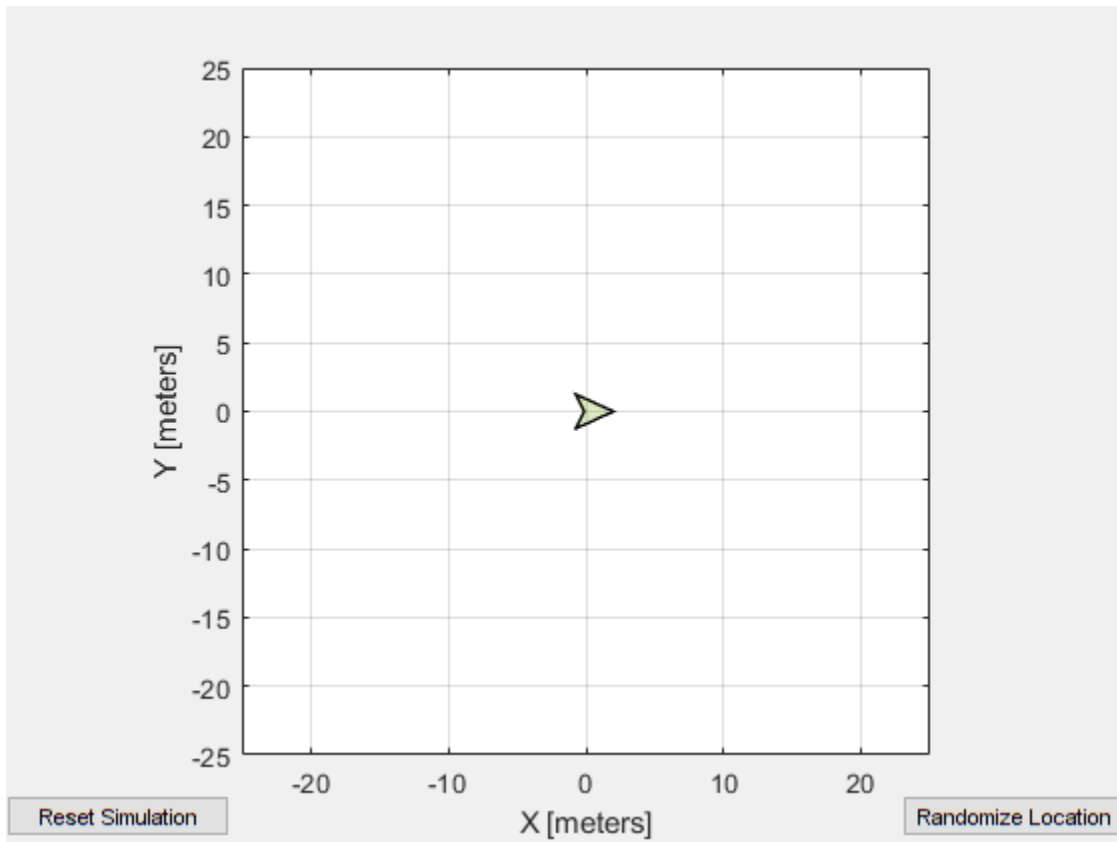
Check the available ROS nodes on the connected ROS device. The nodes listed in this example were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example. Two separate nodes are generated, 'robotcontroller' and 'robotcontroller2', which have the linear velocity set to 1 and 2 in the model respectively.

```
d.AvailableNodes
```

```
ans = 1x2 cell  
    {'robotcontroller'}    {'robotcontroller2'}
```

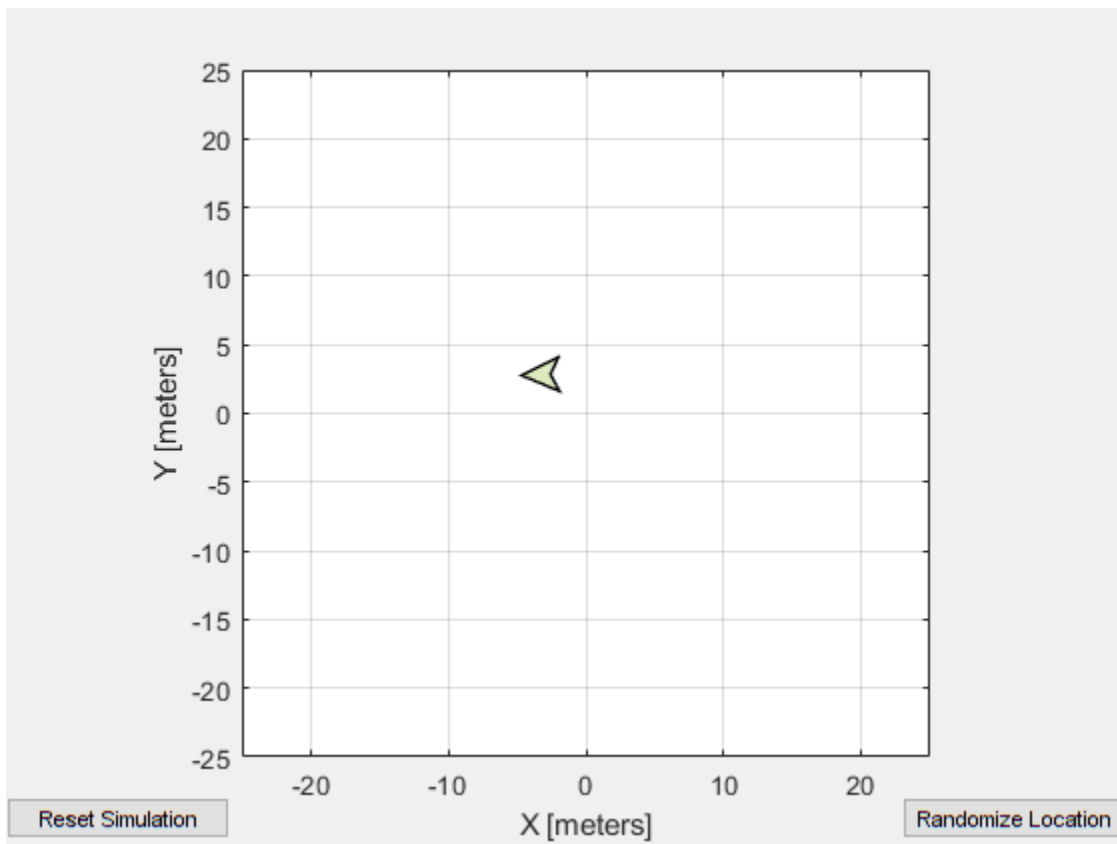
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



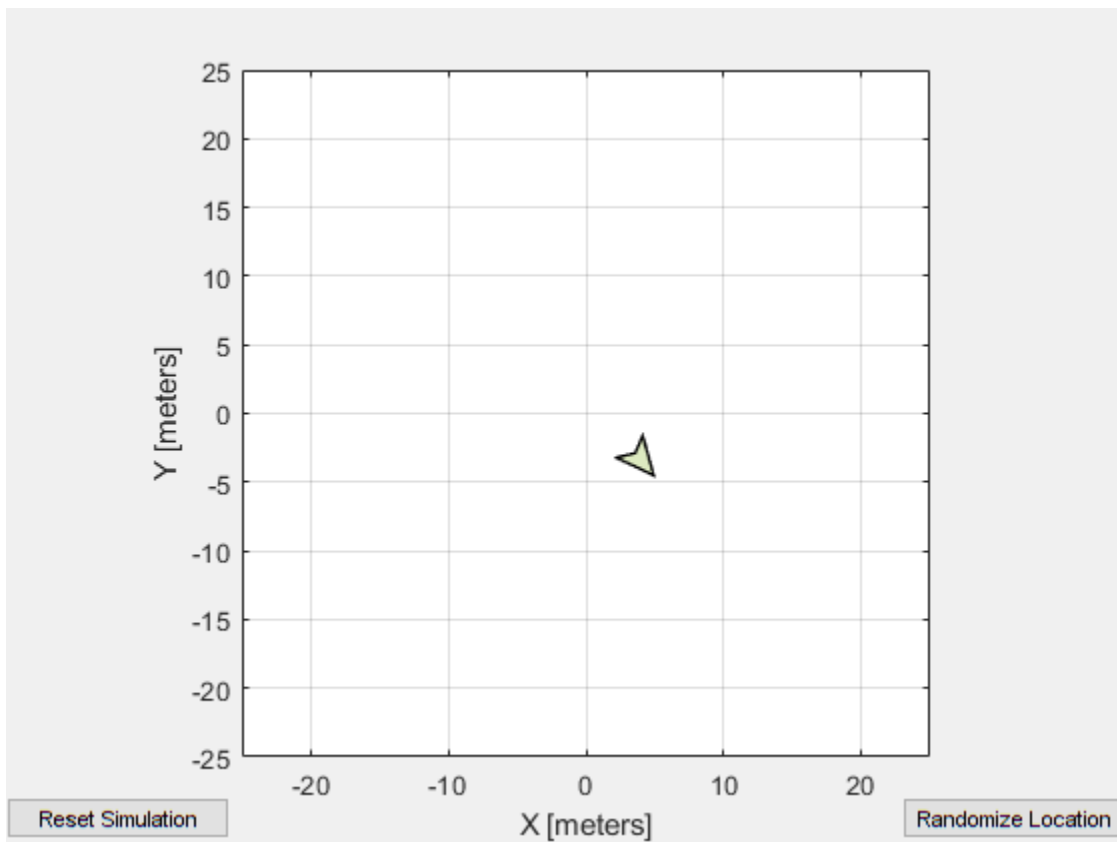
Run a ROS node, specifying the node name. The 'robotcontroller' node commands the robot to a specific location ([ -10 10]). Wait to see the robot drive.

```
runNode(d, 'robotcontroller')  
pause(10)
```



Reset the Robot Simulator to reset the robot position. Alternatively, click **Reset Simulation**. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

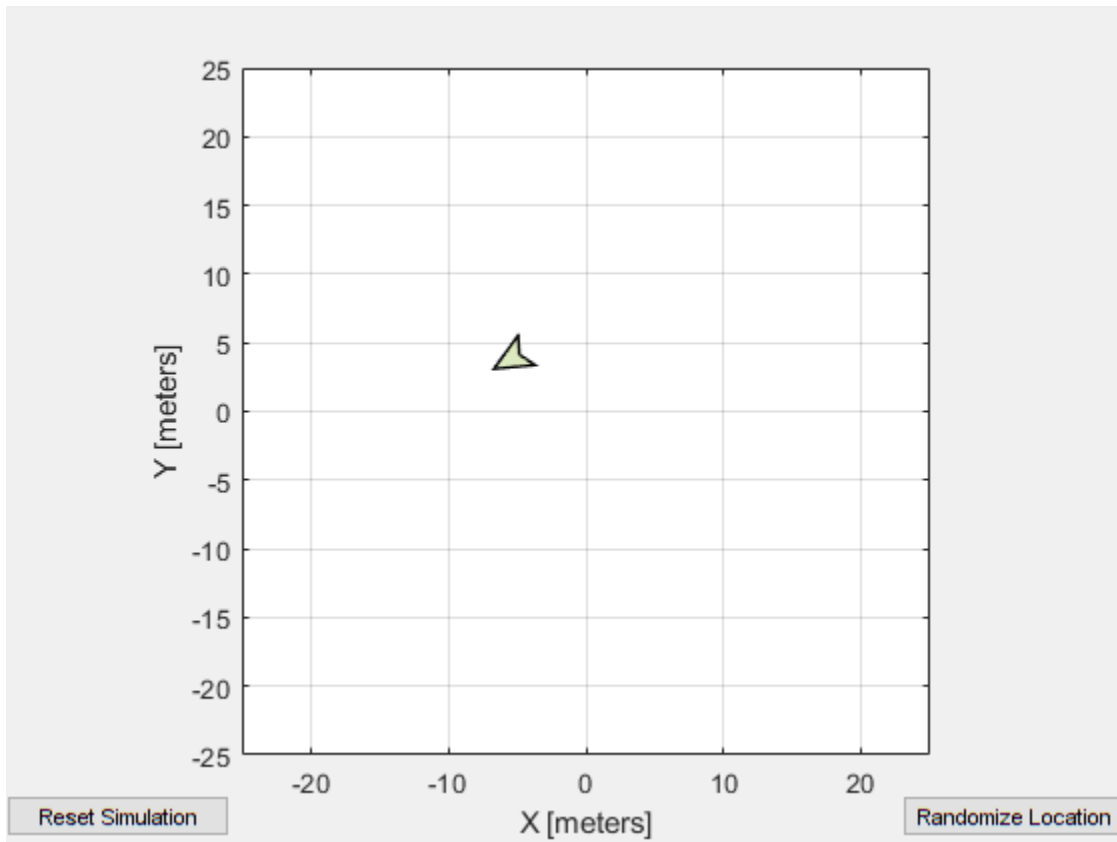
```
resetSimulation(sim.Simulator)  
pause(5)
```



```
stopNode(d, 'robotcontroller')
```

Run the 'robotcontroller2' node. This model drives the robot with twice the linear velocity. Reset the robot position. Wait to see the robot drive. You should see a wider turn due to the increased velocity.

```
runNode(d, 'robotcontroller2')
resetSimulation(sim.Simulator)
pause(10)
```



Close the simulator. Stop the ROS node. Disconnect from the ROS network and stop the ROS core.

```
close
stopNode(d, 'robotcontroller2')
roshutdown
```

```
Shutting down global node /matlab_global_node_66434 with NodeURI http://192.168.203.1:59395/
stopCore(d)
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **modelName** — Name of the deployed Simulink model

character vector

Name of the deployed Simulink model, specified as a character vector. If the model name is not valid, the function returns an error.

### **masterURI** — URI of the ROS master

character vector



URI of the ROS master, specified as a character vector. On startup, the node connects to the ROS master with the given URI.

**nodeHost — Host name for the node**

character vector

Host name for the node, specified as a character vector. The node uses this host name to advertise itself on the ROS network for others to connect to it.

**See Also**

[isNodeRunning](#) | [rosdevice](#) | [stopNode](#)

**Topics**

[“Connect to a ROS Network”](#)

[“Generate a Standalone ROS Node from Simulink®”](#)

**Introduced in R2019b**

## scatter3

Display point cloud in scatter plot

### Syntax

```
scatter3(pcloud)
scatter3(pcloud,Name,Value)
h = scatter3( ___ )
```

### Description

`scatter3(pcloud)` plots the input `pcloud` point cloud as a 3-D scatter plot in the current axes handle. If the data contains RGB information for each point, the scatter plot is colored accordingly.

`scatter3(pcloud,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments. `Name` must appear inside single quotes ( ' ' ). You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`h = scatter3( ___ )` returns the scatter series object, using any of the arguments from previous syntaxes. Use `h` to modify properties of the scatter series after it is created.

When plotting ROS point cloud messages, MATLAB follows the standard ROS convention for axis orientation. This convention states that **positive x is forward, positive y is left, and positive z is up**. If cameras are used, a second frame is defined with an “\_optical” suffix that changes the orientation of the axis. In this case, positive *z* is forward, positive *x* is right, and positive *y* is down. MATLAB looks for the “\_optical” suffix and will adjust the axis orientation of the scatter plot accordingly. For more information, see [Axis Orientation](#) on the ROS Wiki.

### Examples

#### Get and Plot a 3-D Point Cloud

Connect to a ROS network. Subscribe to a point cloud message topic.

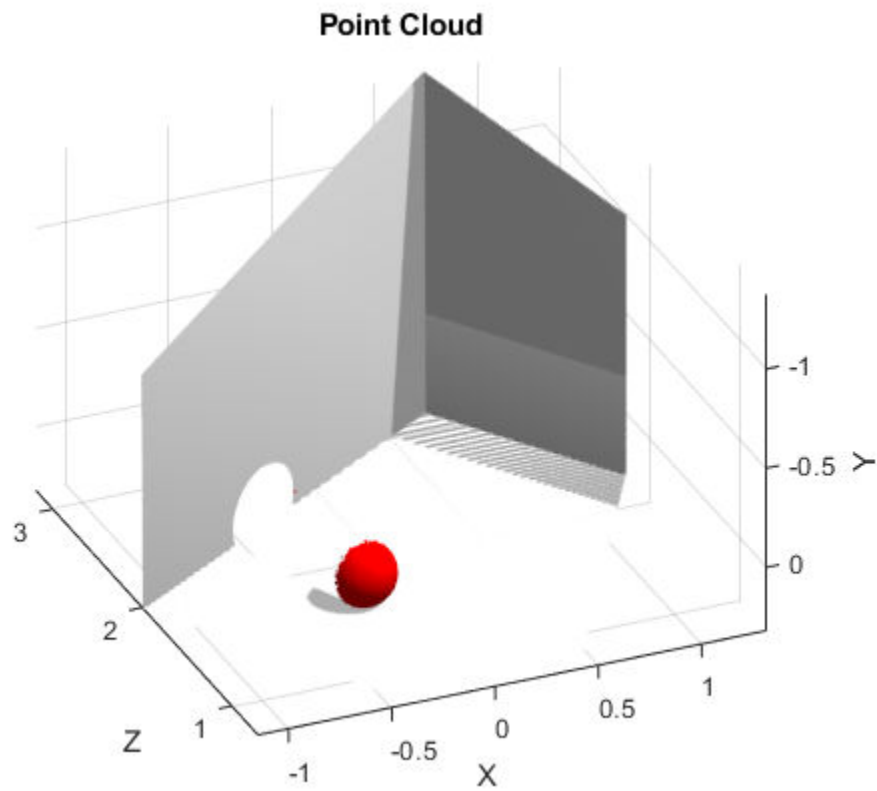
```
rosinit('192.168.17.129')
```

```
Initializing global node /matlab_global_node_65972 with NodeURI http://192.168.17.1:51971/
```

```
sub = rossubscriber('/camera/depth/points');
pause(1)
```

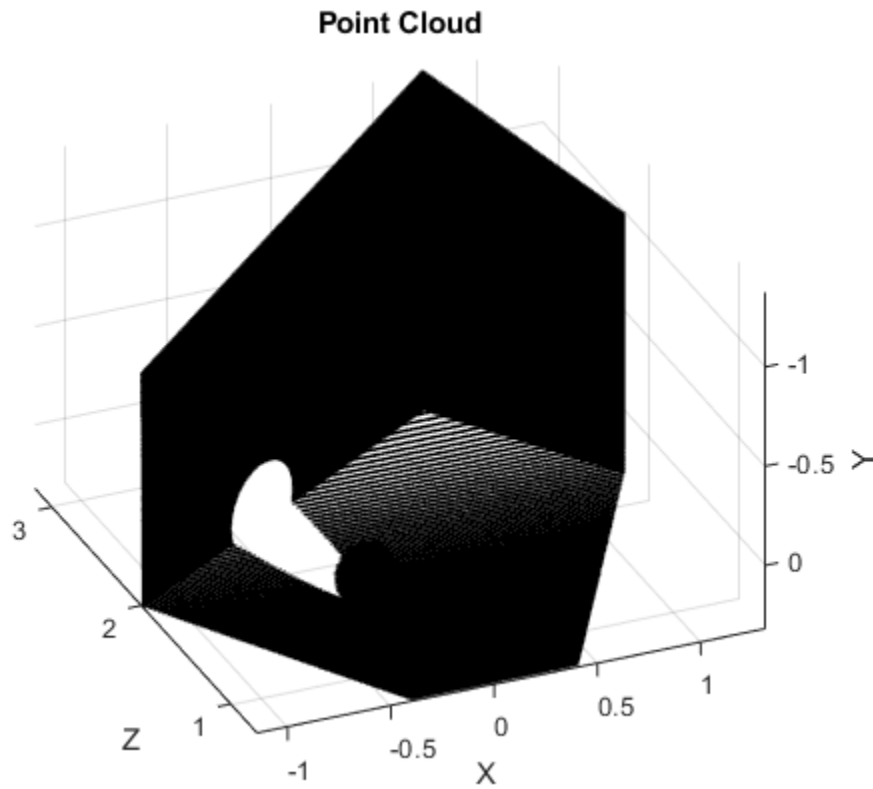
Get the latest point cloud message. Plot the point cloud.

```
pcloud = sub.LatestMessage;
scatter3(pcloud)
```



Plot all points as black dots.

```
scatter3(sub.LatestMessage, 'MarkerEdgeColor', [0 0 0]);
```



## Input Arguments

### **pcloud** — Point cloud

PointCloud2 object handle

Point cloud, specified as a `PointCloud2` object handle for a `'sensor_msgs/PointCloud2'` ROS message.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

Example: `'MarkerEdgeColor', [1 0 0]`

### **MarkerEdgeColor** — Marker outline color

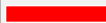







`'flat'` (default) | RGB triplet | hexadecimal color code | `'r'` | `'g'` | `'b'` | ...

Marker outline color, specified `'flat'`, an RGB triplet, a hexadecimal color code, a color name, or a short name. The default value of `'flat'` uses colors from the `CData` property.






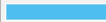

For a custom color, specify an RGB triplet or a hexadecimal color code.

- An RGB triplet is a three-element row vector whose elements specify the intensities of the red, green, and blue components of the color. The intensities must be in the range  $[0, 1]$ ; for example,  $[0.4 \ 0.6 \ 0.7]$ .
- A hexadecimal color code is a character vector or a string scalar that starts with a hash symbol (#) followed by three or six hexadecimal digits, which can range from 0 to F. The values are not case sensitive. Thus, the color codes '#FF8800', '#ff8800', '#F80', and '#f80' are equivalent.

Alternatively, you can specify some common colors by name. This table lists the named color options, the equivalent RGB triplets, and hexadecimal color codes.

Color Name	Short Name	RGB Triplet	Hexadecimal Color Code	Appearance
'red'	'r'	[1 0 0]	'#FF0000'	
'green'	'g'	[0 1 0]	'#00FF00'	
'blue'	'b'	[0 0 1]	'#0000FF'	
'cyan'	'c'	[0 1 1]	'#00FFFF'	
'magenta'	'm'	[1 0 1]	'#FF00FF'	
'yellow'	'y'	[1 1 0]	'#FFFF00'	
'black'	'k'	[0 0 0]	'#000000'	
'white'	'w'	[1 1 1]	'#FFFFFF'	
'none'	Not applicable	Not applicable	Not applicable	No color

Here are the RGB triplets and hexadecimal color codes for the default colors MATLAB uses in many types of plots.

RGB Triplet	Hexadecimal Color Code	Appearance
[0 0.4470 0.7410]	'#0072BD'	
[0.8500 0.3250 0.0980]	'#D95319'	
[0.9290 0.6940 0.1250]	'#EDB120'	
[0.4940 0.1840 0.5560]	'#7E2F8E'	
[0.4660 0.6740 0.1880]	'#77AC30'	
[0.3010 0.7450 0.9330]	'#4DBEEE'	
[0.6350 0.0780 0.1840]	'#A2142F'	

Example:  $[0.5 \ 0.5 \ 0.5]$

Example: 'blue'

Example: '#D2F9A7'

### Parent — Parent of axes

axes object

Parent of axes, specified as the comma-separated pair consisting of 'Parent' and an axes object in which to draw the point cloud. By default, the point cloud is plotted in the active axes.

## Outputs

### **h — Scatter series object**

scalar

Scatter series object, returned as a scalar. This value is a unique identifier, which you can use to query and modify the properties of the scatter object after it is created.

## See Also

[readRGB](#) | [readXYZ](#)

**Introduced in R2019b**

# search

Search ROS network for parameter names

## Syntax

```
pnames = search(ptree,searchstr)
[pnames,pvalues] = search(ptree,searchstr)
```

## Description

`pnames = search(ptree,searchstr)` searches within the parameter tree `ptree` and returns the parameter names that contain the specified search string, `searchstr`.

`[pnames,pvalues] = search(ptree,searchstr)` also returns the parameter values.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed:

- 32-bit integers — `int32`
- Booleans — `logical`
- doubles — `double`
- strings — string scalar, `string`, or character vector, `char`
- lists — cell array
- dictionaries — structure

## Examples

### Search for ROS Parameter Names

Connect to ROS network. Specify the IP address of the ROS master.

```
rosinit('192.168.17.128')
```

```
Initializing global node /matlab_global_node_48144 with NodeURI http://192.168.17.1:54848/
```

Create a parameter tree.

```
ptree = rosparam;
```

Search for parameter names that contain 'gravity'.

```
[pnames,pvalues] = search(ptree,'gravity')
```

```
pnames = 1x3 cell array
    {'/gazebo/gravity_x'}    {'/gazebo/gravity_y'}    {'/gazebo/gravity_z'}
```

```
pvalues = 3x1 cell array
    {[    0]}
```

```
{[ 0]}  
{[-9.8000]}
```

## Input Arguments

### **ptree** — Parameter tree

ParameterTree object handle

Parameter tree, specified as a ParameterTree object handle. Create this object using the `rosparam` function.

### **searchstr** — ROS parameter search string

string scalar | character vector

ROS parameter search string specified as a string scalar or character vector. The `search` function returns all parameters that contain this character vector.

## Output Arguments

### **pnames** — Parameter values

cell array of character vectors

Parameter names, returned as a cell array of character vectors. These character vectors match the parameter names in the ROS master that contain the search character vector.

### **pvalues** — Parameter values

cell array

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed:

- 32-bit integers — `int32`
- Booleans — `logical`
- doubles — `double`
- strings — string scalar, `string`, or character vector, `char`
- lists — cell array
- dictionaries — structure

Base64-encoded binary data and iso 8601 data from ROS are not supported.

## Limitations

Base64-encoded binary data and iso8601 data from ROS are not supported.

## See Also

`get` | `rosparam`

**Introduced in R2019b**



# seconds

Returns seconds of a time or duration

## Syntax

```
secs = seconds(time)
secs = seconds(duration)
```

## Description

`secs = seconds(time)` returns the scalar number, `secs`, in seconds that represents the same value as the time object, `time`.

`secs = seconds(duration)` returns the scalar number, `secs`, in seconds that represents the same value as the duration object, `duration`.

## Examples

### Get Seconds From A Time Object

Use the `seconds` function to get the total seconds of a `Time` object from its `Secs` and `Nsecs` properties.

Create a `Time` object.

```
time = rostime(1,860000000)

time =
  ROS Time with properties:

    Sec: 1
    Nsec: 860000000
```

Get the total seconds from the time object.

```
secs = seconds(time)

secs = 1.8600
```

## Input Arguments

### **time** — Current ROS or system time

Time object handle

ROS or system time, specified as a `Time` object handle. Create a `Time` object using `rostime`.

### **duration** — Duration

ROS Duration object

Duration, specified as a ROS `Duration` object with `Sec` and `Nsec` properties. Create a `Duration` object using `rosduration`.

## Output Arguments

### **secs — Total time**

scalar in seconds

Total time of the `Time` or `Duration` object, returned as a scalar in seconds.

## See Also

`rosduration` | `rostime`

**Introduced in R2019b**

# select

Select subset of messages in rosbag

## Syntax

```
bagsel = select(bag)
bagsel = select(bag,Name,Value)
```

## Description

`bagsel = select(bag)` returns an object, `bagsel`, that contains all of the messages in the `BagSelection` object, `bag`.

This function does not change the contents of the original `BagSelection` object. It returns a new object that contains the specified message selection.

`bagsel = select(bag,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments.

## Examples

### Create Copy of rosbag

Retrieve the rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Use `select` with no selection criteria to create a copy of the rosbag.

```
bagCopy = select(bag);
```

### Select Subset of Messages In rosbag

Retrieve the rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Select all messages within the first second of the rosbag.

```
bag = select(bag, 'Time', [bag.StartTime, bag.StartTime + 1]);
```

## Input Arguments

### bag — Messages of a rosbag

`BagSelection` object

Messages contained within a rosbag, specified as a `BagSelection` object.

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

Example: `"MessageType", "/geometry_msgs/Point"`

### **MessageType — ROS message type**

string scalar | character vector | cell array

ROS message type, specified as a string scalar, character vector, or cell array. Multiple message types can be specified with a cell array.

### **Time — Start and end times**

*n*-by-2 vector

Start and end times of the rosbag selection, specified as an *n*-by-2 vector.

### **Topic — ROS topic name**

string scalar | character vector | cell array

ROS topic name, specified as a string scalar, character vector, or cell array. Multiple topic names can be specified with a cell array.

## **Output Arguments**

### **bagsel — Copy or subset of rosbag messages**

BagSelection object

Copy or subset of rosbag messages, returned as a BagSelection object.

### **See Also**

`readMessages` | `rosbag` | `timeseries`

### **Introduced in R2019b**

# send

Publish ROS message to topic

## Syntax

```
send(pub,msg)
```

## Description

`send(pub,msg)` publishes a message to the topic specified by the publisher, `pub`. This message can be received by all subscribers in the ROS network that are subscribed to the topic specified by `pub`.

## Examples

### Create, Send, and Receive ROS Messages

Set up a publisher and subscriber to send and receive a message on a ROS network.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:55981/.
```

```
Initializing global node /matlab_global_node_45365 with NodeURI http://bat5110win64:55985/
```

Create a publisher with a specific topic and message type. You can also return a default message to send using this publisher.

```
[pub,msg] = rospublisher('position','geometry_msgs/Point');
```

Modify the message before sending it over the network.

```
msg.X = 1;  
msg.Y = 2;  
send(pub,msg);
```

Create a subscriber and wait for the latest message. Verify the message is the one you sent.

```
sub = rossubscriber('position')
```

```
sub =
```

```
Subscriber with properties:
```

```
    TopicName: '/position'  
    MessageType: 'geometry_msgs/Point'  
    LatestMessage: [0x1 Point]  
    BufferSize: 1  
    NewMessageFcn: []
```

```
pause(1);  
sub.LatestMessage
```

```
ans =  
ROS Point message with properties:  
  
  MessageType: 'geometry_msgs/Point'  
      X: 1  
      Y: 2  
      Z: 0
```

Use `showdetails` to show the contents of the message

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_45365 with NodeURI http://bat5110win64:55985/  
Shutting down ROS master on http://bat5110win64:55981/.
```

## Input Arguments

### **pub** — ROS publisher

Publisher object handle

ROS publisher, specified as a `Publisher` object handle. You can create the object using `rospublisher`.

### **msg** — ROS message

Message object handle

ROS message, specified as a `Message` object handle. You can create object using `rosmessage`.

## See Also

`receive` | `rosmessage` | `rospublisher` | `rossubscriber` | `rostopic`

## Topics

“Exchange Data with ROS Publishers and Subscribers”

**Introduced in R2019b**

# sendGoal

Send goal message to action server

## Syntax

```
sendGoal(client,goalMsg)
```

## Description

`sendGoal(client,goalMsg)` sends a goal message to the action server. The specified action client tracks this goal. The function does not wait for the goal to be executed and returns immediately.

If the `ActionFcn`, `FeedbackFcn`, and `ResultFcn` callbacks of the client are defined, they are called when the goal is processing on the action server. All callbacks associated with a previously sent goal are disabled, but the previous goal is not canceled.

## Examples

### Create And Send A ROS Action Goal Message

This example shows how to create goal messages and send to an already active ROS action server on a ROS network. You must create a ROS action client to connect to this server. To run the action server, this command is used on the ROS distribution:

```
roslaunch turtlebot_actions server_turtlebot_move.launch
```

Afterward, connect to the ROS node using `roscpp` with the correct IP address.

```
roscpp('192.168.17.129',11311)
```

```
Initializing global node /matlab_global_node_27318 with NodeURI http://192.168.17.1:60260/
```

Create a ROS action client and get a goal message. The `actClient` object connects to the already running ROS action server. The `goalMsg` is a valid goal message. Update the message parameters with your specific goal.

```
[actClient, goalMsg] = roscppactionclient('/turtlebot_move');
disp(goalMsg)
```

```
ROS TurtlebotMoveGoal message with properties:
```

```
    MessageType: 'turtlebot_actions/TurtlebotMoveGoal'
    TurnDistance: 0
    ForwardDistance: 0
```

```
Use showdetails to show the contents of the message
```

You can also create a message using `roscppmessage` and the action client object. This message sends linear and angular velocity commands to a Turtlebot® robot.

```
goalMsg = rosmessage(actClient);  
disp(goalMsg)
```

ROS TurtlebotMoveGoal message with properties:

```
    MessageType: 'turtlebot_actions/TurtlebotMoveGoal'  
    TurnDistance: 0  
    ForwardDistance: 0
```

Use `showdetails` to show the contents of the message

Modify the goal message parameters and send the goal to the action server.

```
goalMsg.ForwardDistance = 2;  
sendGoal(actClient,goalMsg)
```

### Send and Cancel ROS Action Goals

This example shows how to send and cancel goals for ROS actions. Action types must be setup beforehand with an action server running.

You must have set up the  `'/fibonacci'` action type. To run this action server, use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

First, set up a ROS action client. Then, send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected to the ROS network using `rosactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.17.129',11311)
```

```
Initializing global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

```
[actClient,goalMsg] = rosactionclient('/fibonacci');  
waitForServer(actClient);
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;  
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg =  
    ROS FibonacciResult message with properties:  
  
    MessageType: 'actionlib_tutorials/FibonacciResult'  
    Sequence: [6x1 int32]
```

Use `showdetails` to show the contents of the message

```
resultState =  
'succeeded'
```



```
showdetails(resultMsg)
```

```
Sequence : [0, 1, 1, 2, 3, 5]
```

Send a new goal message without waiting.

```
goalMsg.Order = 5;  
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

## Input Arguments

### **client** — ROS action client

`SimpleActionClient` object handle

ROS action client, specified as a `SimpleActionClient` object handle. This simple action client enables you to track a single goal at a time.

### **goalMsg** — ROS action goal message

`Message` object handle

ROS action goal message, specified as a `Message` object handle. Update this message with your goal details and send it to the ROS action client using `sendGoal` or `sendGoalAndWait`.

## See Also

`cancelGoal` | `roaction` | `roactionclient` | `sendGoalAndWait`

## Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

## Introduced in R2019b

## sendGoalAndWait

Send goal message and wait for result

### Syntax

```
resultMsg = sendGoalAndWait(client,goalMsg)
resultMsg = sendGoalAndWait(client,goalMsg,timeout)
[resultMsg,state,status] = sendGoalAndWait( ___ )
```

### Description

`resultMsg = sendGoalAndWait(client,goalMsg)` sends a goal message using the specified action client to the action server and waits until the action server returns a result message. Press **Ctrl+C** to abort the wait.

`resultMsg = sendGoalAndWait(client,goalMsg,timeout)` specifies a timeout period in seconds. If the server does not return the result in the timeout period, the function displays an error.

`[resultMsg,state,status] = sendGoalAndWait( ___ )` returns the final goal state and associated status text using any of the previous syntaxes. The `state` contains information about whether the goal execution succeeded or not.

### Examples

#### Send and Cancel ROS Action Goals

This example shows how to send and cancel goals for ROS actions. Action types must be setup beforehand with an action server running.

You must have set up the  `'/fibonacci'` action type. To run this action server, use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

First, set up a ROS action client. Then, send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected to the ROS network using `rosactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.17.129',11311)
```

```
Initializing global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

```
[actClient,goalMsg] = rosactionclient('/fibonacci');
waitForServer(actClient);
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg =
  ROS FibonacciResult message with properties:

  MessageType: 'actionlib_tutorials/FibonacciResult'
  Sequence: [6×1 int32]
```

Use `showdetails` to show the contents of the message

```
resultState =
'succeeded'
```

```
showdetails(resultMsg)
```

```
Sequence : [0, 1, 1, 2, 3, 5]
```

Send a new goal message without waiting.

```
goalMsg.Order = 5;
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

## Input Arguments

### **client** — ROS action client

`SimpleActionClient` object handle

ROS action client, specified as a `SimpleActionClient` object handle. This simple action client enables you to track a single goal at a time.

### **goalMsg** — ROS action goal message

`Message` object handle

ROS action goal message, specified as a `Message` object handle. Update this message with your goal details and send it to the ROS action client using `sendGoal` or `sendGoalAndWait`.

### **timeout** — Timeout period

scalar in seconds

Timeout period for receiving a result message, specified as a scalar in seconds. If the client does not receive a new result message in that time period, an error is displayed.

## Output Arguments

### **resultMsg** — Result message

ROS Message object

Result message, returned as a ROS Message object. The result message contains the result data sent by the action server. This data depends on the action type.

### **state** — Final goal state

character vector

Final goal state, returned as one of the following:

- 'pending' — Goal was received, but has not yet been accepted or rejected.
- 'active' — Goal was accepted and is running on the server.
- 'succeeded' — Goal executed successfully.
- 'preempted' — An action client canceled the goal before it finished executing.
- 'aborted' — The goal was aborted before it finished executing. The action server typically aborts a goal.
- 'rejected' — The goal was not accepted after being in the 'pending' state. The action server typically triggers this status.
- 'recalled' — A client canceled the goal while it was in the 'pending' state.
- 'lost' — An internal error occurred in the action client.

### **status** — Status text

character vector

Status text that the server associated with the final goal state, returned as a character vector.

## See Also

[cancelGoal](#) | [rosaction](#) | [rosactionclient](#) | [sendGoal](#)

### Topics

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

### Introduced in R2019b

# sendTransform

Send transformation to ROS network

## Syntax

```
sendTransform(tftree,tf)
```

## Description

`sendTransform(tftree,tf)` broadcasts a transform or array of transforms, `tf`, to the ROS network as a `TransformationStamped` ROS message.

## Examples

### Send a Transformation to ROS Network

This example shows how to create a transformation and send it over the ROS network.

Create a ROS transformation tree. Use `rosinit` to connect a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.17.129';
rosinit(ipaddress,11311)
```

```
Initializing global node /matlab_global_node_33798 with NodeURI http://192.168.17.1:56771/
```

```
tftree = rostf;
pause(2)
```

Verify the transformation you want to send over the network does not already exist. The `canTransform` function returns false if the transformation is not immediately available.

```
canTransform(tftree,'new_frame','base_link')
```

```
ans = logical
      0
```

Create a `TransformStamped` message. Populate the message fields with the transformation information.

```
tform = rosmessage('geometry_msgs/TransformStamped');
tform.ChildFrameId = 'new_frame';
tform.Header.FrameId = 'base_link';
tform.Transform.Translation.X = 0.5;
tform.Transform.Rotation.Z = 0.75;
```

Send the transformation over the ROS network.

```
sendTransform(tftree,tform)
```

Verify the transformation is now on the ROS network.

```
canTransform(tftree, 'new_frame', 'base_link')
```

```
ans = logical  
     1
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_33798 with NodeURI http://192.168.17.1:56771/
```

## Input Arguments

### **tftree** — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a TransformationTree object handle. You can create a transformation tree by calling the `rostopic` function.

### **tf** — Transformations between coordinate frames

TransformStamped object handle | array of object handles

Transformations between coordinate frames, returned as a TransformStamped object handle or as an array of object handles. Transformations are structured as a 3-D translation (3-element vector) and a 3-D rotation (quaternion).

## See Also

`getTransform` | `transform`

**Introduced in R2019b**

## set

Set value of ROS parameter or add new parameter

### Syntax

```
set(ptree,paramname,pvalue)
set(ptree,namespace,pvalue)
```

### Description

`set(ptree,paramname,pvalue)` assigns the value `pvalue` to the parameter with the name `paramname`. This parameter is sent to the parameter tree `ptree`.

`set(ptree,namespace,pvalue)` assigns multiple values as a dictionary in `pvalue` under the specified namespace.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

- 32-bit integer — `int32`
- Boolean — `logical`
- double — `double`
- strings — string scalar, `string`, or character vector, `char`
- list — cell array (`cell`)
- dictionary — structure (`struct`)

### Examples

#### Set and Get Parameter Value

Connect to the ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:59267/.
```

```
Initializing global node /matlab_global_node_21992 with NodeURI http://bat5110win64:59274/
```

Create a ROS parameter tree. Set a double parameter. Get the parameter to verify it was set.

```
ptree = rosparam;
set(ptree,'DoubleParam',1.0)
get(ptree,'DoubleParam')
```

```
ans = 1
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_21992 with NodeURI http://bat5110win64:59274/  
Shutting down ROS master on http://bat5110win64:59267/.
```

## Set A Dictionary Of Parameter Values

Use structures to specify a dictionary of ROS parameters under a specific namespace.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:56297/.  
Initializing global node /matlab_global_node_59257 with NodeURI http://bat5110win64:56304/
```

Create a dictionary of parameter values. This dictionary contains the information relevant to an image. Display the structure to verify values.

```
image = imread('peppers.png');  
  
pval.ImageWidth = size(image,1);  
pval.ImageHeight = size(image,2);  
pval.ImageTitle = 'peppers.png';  
disp(pval)
```

```
    ImageWidth: 384  
    ImageHeight: 512  
    ImageTitle: 'peppers.png'
```

Set the dictionary of values using the desired namespace.

```
rosparam('set', 'ImageParam', pval)
```

Get the parameters using the namespace. Verify the values.

```
pval2 = rosparam('get', 'ImageParam')
```

```
pval2 = struct with fields:  
    ImageHeight: 512  
    ImageTitle: 'peppers.png'  
    ImageWidth: 384
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_59257 with NodeURI http://bat5110win64:56304/  
Shutting down ROS master on http://bat5110win64:56297/.
```

## Input Arguments

### **ptree** — Parameter tree

ParameterTree object handle



Parameter tree, specified as a `ParameterTree` object handle. Create this object using the `rosparam` function.

**paramname — ROS parameter name**

string scalar | character vector

ROS parameter name, specified as a string scalar or character vector. This string must match the parameter name exactly.

**pvalue — ROS parameter value or dictionary of values**

int32 | logical | double | string scalar | character vector | cell array | structure

ROS parameter value or dictionary of values, specified as a supported MATLAB data type.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

ROS Data Type	MATLAB Data Type
32-bit integer	int32
Boolean	logical
double	double
string	string scalar, string, or character vector, char
list	cell array (cell)
dictionary	structure (struct)

**namespace — ROS parameter namespace**

string scalar | character vector

ROS parameter namespace, specified as a string scalar or character vector. All parameter names starting with this string are listed when calling `rosparam("list", namespace)`.

## Limitations

Base64-encoded binary data and iso 8601 data from ROS are not supported.

## See Also

get | rosparam

## Introduced in R2019b

## showdetails

Display all ROS message contents

### Syntax

```
details = showdetails(msg)
```

### Description

`details = showdetails(msg)` gets all data contents of message object `msg`. The details are stored in `details` or displayed on the command line.

### Examples

#### Create Message and View Details

Create a message. Populate the message with data using the relevant properties.

```
msg = rosmessage('geometry_msgs/Point');  
msg.X = 1;  
msg.Y = 2;  
msg.Z = 3;
```

View the message details.

```
showdetails(msg)
```

```
X : 1  
Y : 2  
Z : 3
```

### Input Arguments

#### **msg** — ROS message

Message object handle

ROS message, specified as a Message object handle.

### Output Arguments

#### **details** — Details of ROS message

character vector

Details of a ROS message, returned as a character vector.

### See Also

`rosmessage`

**Introduced in R2019b**

## stopCore

Stop ROS core

### Syntax

```
stopCore(device)
```

### Description

`stopCore(device)` stops the ROS core on the specified `rosdevice`, `device`. If multiple ROS cores are running on the ROS device, the function stops all of them. If no core is running, the function returns immediately.

### Examples

#### Run ROS Core on ROS Device

Connect to a remote ROS device and start a ROS core. The ROS core is needed to run ROS nodes to communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.17.128';  
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =  
  rosdevice with properties:  
  
    DeviceAddress: '192.168.17.128'  
    Username: 'user'  
    ROSFolder: '/opt/ros/indigo'  
    CatkinWorkspace: '~/catkin_ws'  
    AvailableNodes: {0x1 cell}
```

Run a ROS core and check if it is running.

```
runCore(d)
```

Another `roscore` / ROS master is already running on the ROS device. Use the `'stopCore'` function to

```
running = isCoreRunning(d)
```

```
running = logical  
         1
```

Stop the ROS core and confirm that it is no longer running.

```
stopCore(d)
running = isCoreRunning(d)

running = logical
    1
```

## Input Arguments

### **device – ROS device**

rosdevice object

ROS device, specified as a rosdevice object.

## See Also

isCoreRunning | rosdevice | runCore

## Topics

“Generate a Standalone ROS Node from Simulink®”

## Introduced in R2019b

## stopNode

Stop ROS node

### Syntax

```
stopNode(device,modelName)
```

### Description

`stopNode(device,modelName)` stops a running ROS node that was deployed from a Simulink model named `modelName`. The node is running on the specified `rosdevice` object, `device`. If the node is not running, the function returns immediately.

### Examples

#### Run ROS Node on ROS Device

Connect to a remote ROS device and start a ROS node. Run a ROS core so that ROS nodes can communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device already contains the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.203.129';
d = rosdevice(ipaddress,'user','password');
d.ROSFolder = '/opt/ros/indigo';
d.CatkinWorkspace = '~/catkin_ws_test'

d =
  rosdevice with properties:

    DeviceAddress: '192.168.203.129'
    Username: 'user'
    ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws_test'
    AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. Connect MATLAB® to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

```
runCore(d)
rosinit(d.DeviceAddress,11311)
```

```
Initializing global node /matlab_global_node_84497 with NodeURI http://192.168.203.1:56034/
```

Check the available ROS nodes on the connected ROS device. These nodes listed were generated from Simulink® models following the process in the “Get Started with ROS in Simulink®” example.

```
d.AvailableNodes
```

```
ans = 1x2 cell
      {'robotcontroller'}   {'robotcontroller2'}
```

Run a ROS node and specify the node name. Check if the node is running.

```
runNode(d, 'RobotController')
running = isNodeRunning(d, 'RobotController')

running = logical
         1
```

Stop the ROS node. Disconnect from the ROS network. Stop the ROS core.

```
stopNode(d, 'RobotController')
rosshutdown
```

```
Shutting down global node /matlab_global_node_84497 with NodeURI http://192.168.203.1:56034/
```

```
stopCore(d)
```

## Run Multiple ROS Nodes

Run multiple ROS nodes on a connected ROS device. ROS nodes can be generated using Simulink® models to perform different tasks on the ROS network. These nodes are then deployed on a ROS device and can be run independently of Simulink®.

This example uses two different Simulink models that have been deployed as ROS nodes. See “Generate a Standalone ROS Node from Simulink®” and follow the instructions to generate and deploy a ROS node. Do this twice and name them 'robotcontroller' and 'robotcontroller2'. The 'robotcontroller' node sends velocity commands to a robot to navigate it to a given point. The 'robotcontroller2' node uses the same model, but doubles the linear velocity to drive the robot faster.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using runNode.

```
ipaddress = '192.168.203.129';
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
  rosdevice with properties:

    DeviceAddress: '192.168.203.129'
    Username: 'user'
    ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws'
    AvailableNodes: {0x1 cell}
```

```
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
d =
  rosdevice with properties:
```

```
DeviceAddress: '192.168.203.129'  
Username: 'user'  
ROSFolder: '/opt/ros/indigo'  
CatkinWorkspace: '~/catkin_ws_test'  
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. The ROS Core is the master enables you to run ROS nodes on your ROS device. Connect MATLAB® to the ROS master using `rosinit`. For this example, the port is set to 11311. `rosinit` can automatically select a port for you without specifying this input.

```
runCore(d)  
rosinit(d.DeviceAddress,11311)
```

```
Initializing global node /matlab_global_node_66434 with NodeURI http://192.168.203.1:59395/
```

Check the available ROS nodes on the connected ROS device. The nodes listed in this example were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example. Two separate nodes are generated, 'robotcontroller' and 'robotcontroller2', which have the linear velocity set to 1 and 2 in the model respectively.

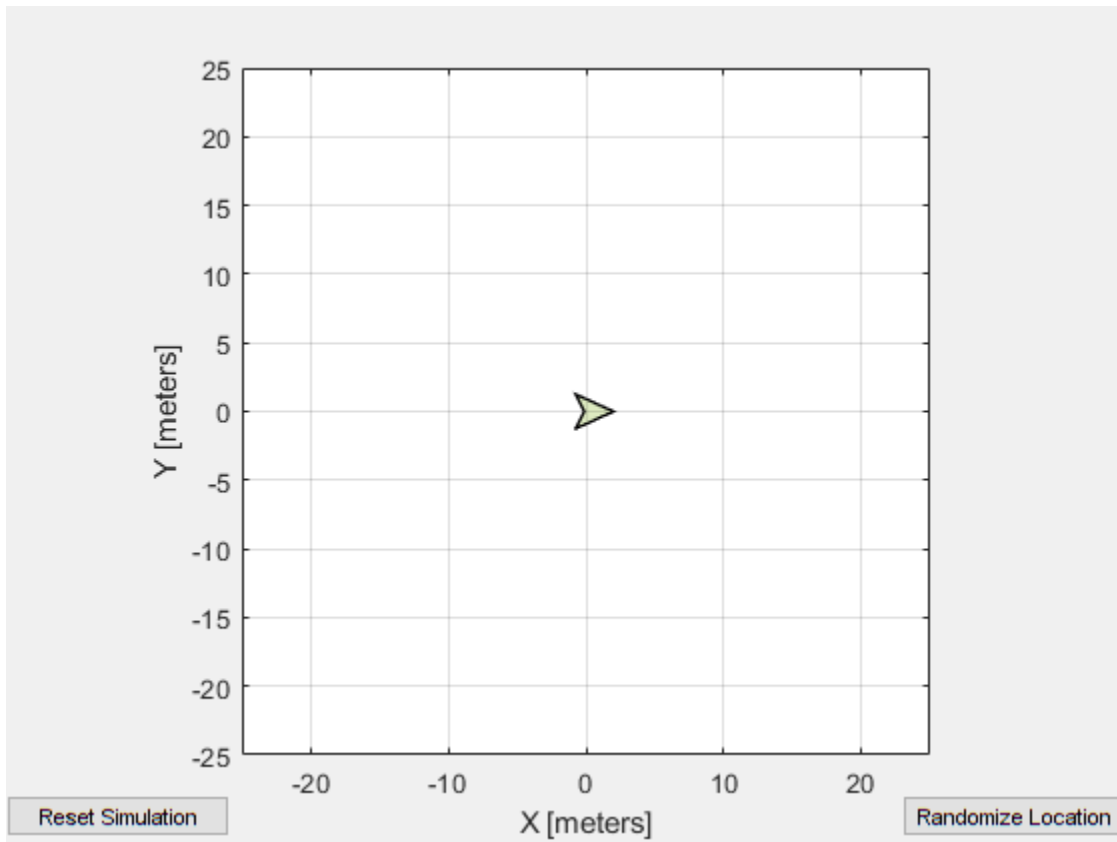
```
d.AvailableNodes
```

```
ans = 1x2 cell  
      {'robotcontroller'}    {'robotcontroller2'}
```

Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

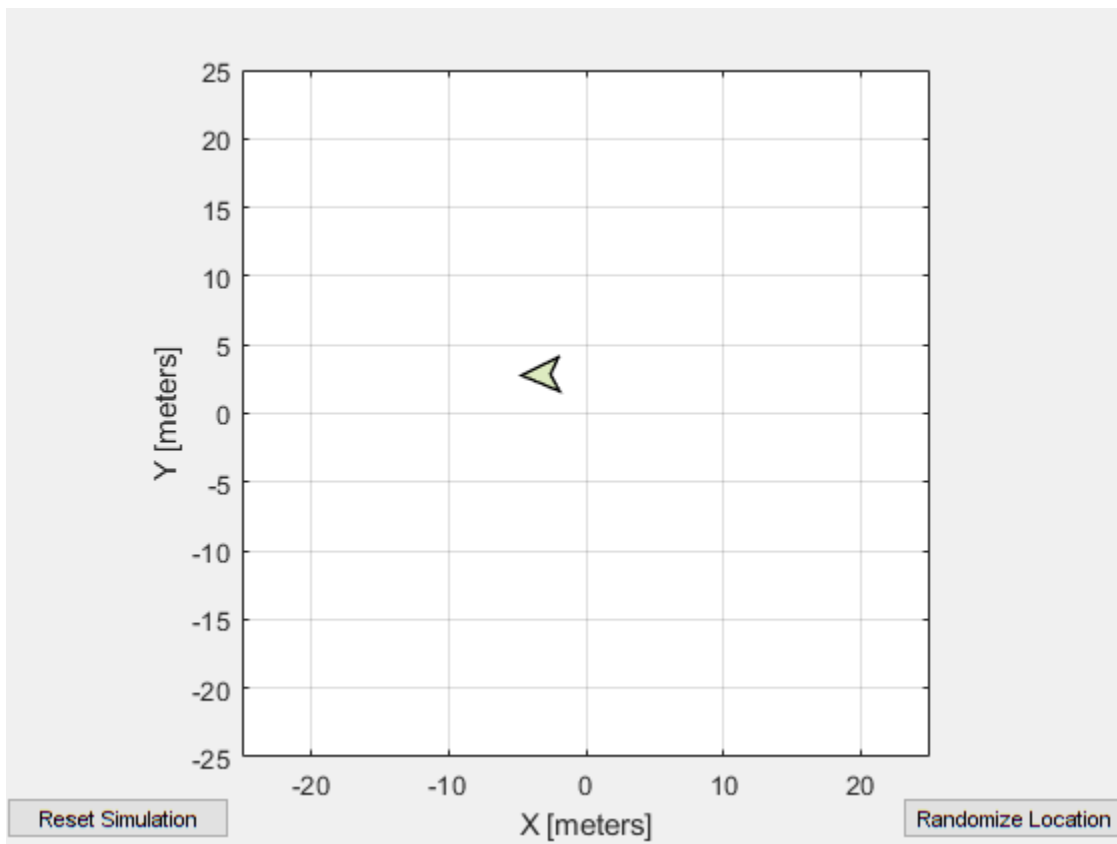
```
sim = ExampleHelperSimulinkRobotROS;
```





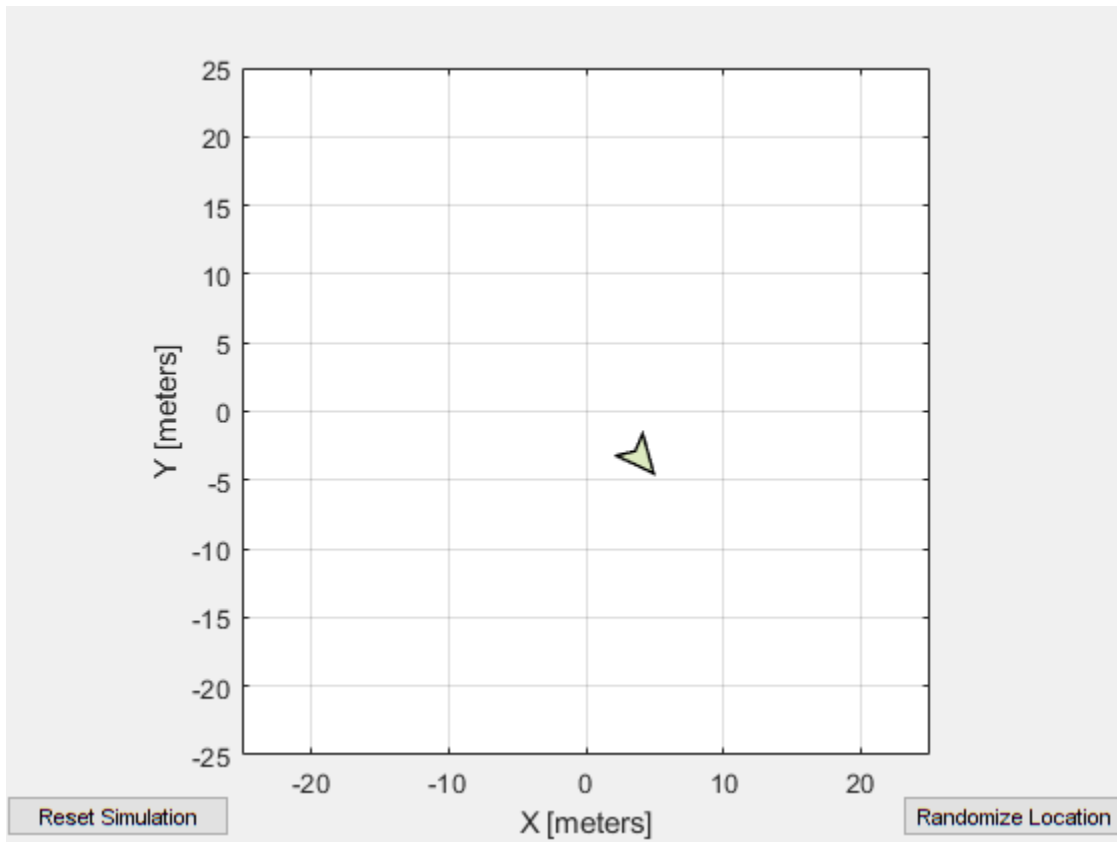
Run a ROS node, specifying the node name. The 'robotcontroller' node commands the robot to a specific location ([ -10 10]). Wait to see the robot drive.

```
runNode(d, 'robotcontroller')  
pause(10)
```



Reset the Robot Simulator to reset the robot position. Alternatively, click **Reset Simulation**. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

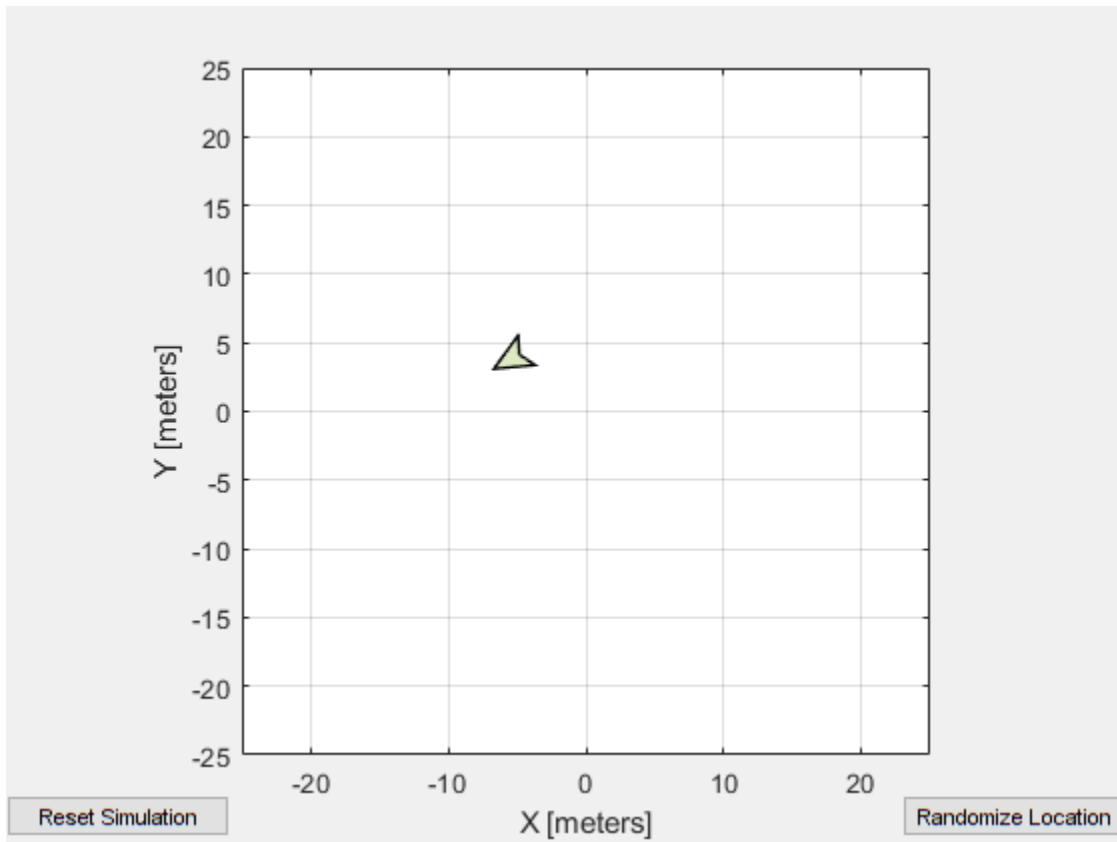
```
resetSimulation(sim.Simulator)  
pause(5)
```



```
stopNode(d, 'robotcontroller')
```

Run the 'robotcontroller2' node. This model drives the robot with twice the linear velocity. Reset the robot position. Wait to see the robot drive. You should see a wider turn due to the increased velocity.

```
runNode(d, 'robotcontroller2')  
resetSimulation(sim.Simulator)  
pause(10)
```



Close the simulator. Stop the ROS node. Disconnect from the ROS network and stop the ROS core.

```
close
stopNode(d, 'robotcontroller2')
roshutdown
```

```
Shutting down global node /matlab_global_node_66434 with NodeURI http://192.168.203.1:59395/
stopCore(d)
```

## Input Arguments

### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

### **modelName** — Name of the deployed Simulink model

character vector

Name of the deployed Simulink model, specified as a character vector. If the model name is not valid, the function returns immediately.

## See Also

isNodeRunning | rosdevice | runNode

**Topics**

“Generate a Standalone ROS Node from Simulink®”

**Introduced in R2019b**

## system

Execute system command on device

### Syntax

```
system(device,command)
system(device,command,'sudo')
response = system( __ )
```

### Description

`system(device,command)` runs a command in the Linux command shell on the ROS device. This function does not allow you to run interactive commands.

`system(device,command,'sudo')` runs a command with superuser privileges.

`response = system( __ )` runs a command using any of the previous syntaxes with the command shell output returned in `response`.

### Examples

#### Run Linux Commands on ROS Device

Connect to a ROS device and run commands on the Linux(R) command shell.

Connect to a ROS device. Specify the device address, user name, and password of your ROS device.

```
d = rosdevice('192.168.17.128','user','password');
```

Run a command that lists the contents of the Catkin workspace folder.

```
system(d,'ls /home/user/catkin_ws_test')
```

```
ans =
    'build
    devel
    src
    '
```

### Input Arguments

#### **device** — ROS device

rosdevice object

ROS device, specified as a rosdevice object.

#### **command** — Linux command

character vector

Linux command, specified as a character vector.

Example: `'ls -al'`

## **Output Arguments**

### **response — Output from Linux shell**

character vector

Output from Linux shell, returned as a character vector.

## **See Also**

`deleteFile` | `dir` | `getFile` | `openShell` | `putFile` | `rosdevice`

**Introduced in R2019b**

## timeseries

Creates a time series object for selected message properties

### Syntax

```
[ts,cols] = timeseries(bag)
[ts,cols] = timeseries(bag,property)
[ts,cols] = timeseries(bag,property,...,propertyN)
```

### Description

`[ts,cols] = timeseries(bag)` creates a time series for all numeric and scalar message properties. The function evaluates each message in the current `BagSelection` object, `bag`, as `ts`. The `cols` output argument stores property names as a cell array of character vectors.

The returned time series object is memory-efficient because it stores only particular message properties instead of whole messages.

`[ts,cols] = timeseries(bag,property)` creates a time series for a specific message property, `property`. Property names can also be nested, for example, `Pose.Pose.Position.X` for the x-axis position of a robot.

`[ts,cols] = timeseries(bag,property,...,propertyN)` creates a time series for a range-specific message properties. Each property is a different column in the time series object.

### Examples

#### Create Time Series from Entire Bag Selection

Load the rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Select a specific topic. Time series supports only single topics.

```
bagSelection = select(bag, 'Topic', '/odom');
```

Create a time series for the '/odom' topic.

```
ts = timeseries(bagSelection);
```

#### Create Time Series from Single Property

Load the rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Select a specific topic. Time series support only single topics.



```
bagSelection = select(bag, 'Topic', '/odom');
```

Create a time series for the 'Pose.Pose.Position.X' property on the '/odom' topic.

```
ts = timeseries(bagSelection, 'Pose.Pose.Position.X');
```

### Create Time Series from Multiple Properties

Load the rosbag. Specify the file path.

```
bag = rosbag('ex_multiple_topics.bag');
```

Select a specific topic. Time series support only single topics.

```
bagSelection = select(bag, 'Topic', '/odom');
```

Create a time series for all the angular 'Twist' properties on the '/odom' topic.

```
ts = timeseries(bagSelection, 'Twist.Twist.Angular.X', ...
               'Twist.Twist.Angular.Y', 'Twist.Twist.Angular.Z');
```

## Input Arguments

### bag — Bag selection

BagSelection object handle

Bag selection, specified as a BagSelection object handle. You can get a bag selection by calling rosbag.

### property — Property names

string scalar | character vector

Property names, specified as a string scalar or character vector. Multiple properties can be specified. Each property name is a separate input and represents a different column in the time series object.

## Output Arguments

### ts — Time series

Time object handle

Time series, returned as a Time object handle.

### cols — List of property names

cell array of character vectors

List of property names, returned as a cell array of character vectors.

## See Also

readMessages | rosbag | select

### Topics

“Time Series” (MATLAB)

**Introduced in R2019b**

# transform

Transform message entities into target coordinate frame

## Syntax

```
tfentity = transform(tftree,targetframe,entity)
tfentity = transform(tftree,targetframe,entity,"msgtime")
tfentity = transform(tftree,targetframe,entity,sourcetime)
```

## Description

`tfentity = transform(tftree,targetframe,entity)` retrieves the latest transformation between `targetframe` and the coordinate frame of `entity` and applies it to `entity`, a ROS message of a specific type. The `tftree` is the full transformation tree containing known transformations between entities. If the transformation from `entity` to `targetframe` does not exist, MATLAB produces an error.

`tfentity = transform(tftree,targetframe,entity,"msgtime")` uses the timestamp in the header of the message, `entity`, as the source time to retrieve and apply the transformation.

`tfentity = transform(tftree,targetframe,entity,sourcetime)` uses the given source time to retrieve and apply the transformation to the message, `entity`.

## Examples

### Get ROS Transformations and Apply to ROS Messages

This example shows how to set up a ROS transformation tree and transform frames based on transformation tree information. It uses time-buffered transformations to access transformations at different times.

Create a ROS transformation tree. Use `rosinit` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.17.129';
rosinit(ipaddress,11311)
```

```
Initializing global node /matlab_global_node_14346 with NodeURI http://192.168.17.1:56312/
```

```
tftree = rostf;
pause(1)
```

Look at the available frames on the transformation tree.

```
tftree.AvailableFrames
```

```
ans = 36x1 cell
    {'base_footprint'      }
    {'base_link'          }
    {'camera_depth_frame' }
```

```

{'camera_depth_optical_frame'}
{'camera_link' }
{'camera_rgb_frame' }
{'camera_rgb_optical_frame' }
{'caster_back_link' }
{'caster_front_link' }
{'cliff_sensor_front_link' }
{'cliff_sensor_left_link' }
{'cliff_sensor_right_link' }
{'gyro_link' }
{'mount_asus_xtion_pro_link' }
{'odom' }
{'plate_bottom_link' }
{'plate_middle_link' }
{'plate_top_link' }
{'pole_bottom_0_link' }
{'pole_bottom_1_link' }
{'pole_bottom_2_link' }
{'pole_bottom_3_link' }
{'pole_bottom_4_link' }
{'pole_bottom_5_link' }
{'pole_kinect_0_link' }
{'pole_kinect_1_link' }
{'pole_middle_0_link' }
{'pole_middle_1_link' }
{'pole_middle_2_link' }
{'pole_middle_3_link' }
:

```

Check if the desired transformation is now available. For this example, check for the transformation from 'camera\_link' to 'base\_link'.

```
canTransform(tftree, 'base_link', 'camera_link')
```

```
ans = logical
      1
```

Get the transformation for 3 seconds from now. The `getTransform` function will wait until the transformation becomes available with the specified timeout.

```
desiredTime = rostime('now') + 3;
tform = getTransform(tftree, 'base_link', 'camera_link', ...
                    desiredTime, 'Timeout', 5);
```

Create a ROS message to transform. Messages can also be retrieved off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_link';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Transform the ROS message to the 'base\_link' frame using the desired time previously saved.

```
tfpt = transform(tftree, 'base_link', pt, desiredTime);
```

*Optional:* You can also use `apply` with the stored `tform` to apply this transformation to the `pt` message.

```
tfpt2 = apply(tform,pt);
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_14346 with NodeURI http://192.168.17.1:56312/
```

### Get Buffered Transformations from ROS Network

This example shows how to access time-buffered transformations on the ROS network. Access transformations for specific times and modify the `BufferTime` property based on your desired times.

Create a ROS transformation tree. Use `rosinit` to connect to a ROS network. Replace `ipaddress` with your ROS network address.

```
ipaddress = '192.168.17.129';
rosinit(ipaddress,11311)
```

```
Initializing global node /matlab_global_node_78006 with NodeURI http://192.168.17.1:56344/
```

```
tftree = rostf;
pause(2);
```

Get the transformation from 1 second ago.

```
desiredTime = rostime('now') - 1;
tform = getTransform(tftree,'base_link','camera_link',desiredTime);
```

The transformation buffer time is 10 seconds by default. Modify the `BufferTime` property of the transformation tree to increase the buffer time and wait for that buffer to fill.

```
tftree.BufferTime = 15;
pause(15);
```

Get the transformation from 12 seconds ago.

```
desiredTime = rostime('now') - 12;
tform = getTransform(tftree,'base_link','camera_link',desiredTime);
```

You can also get transformations at a time in the future. The `getTransform` function will wait until the transformation is available. You can also specify a timeout to error if no transformation is found. This example waits 5 seconds for the transformation at 3 seconds from now to be available.

```
desiredTime = rostime('now') + 3;
tform = getTransform(tftree,'base_link','camera_link',desiredTime,'Timeout',5);
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_78006 with NodeURI http://192.168.17.1:56344/
```

## Input Arguments

### **tftree** — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a TransformationTree object handle. You can create a transformation tree by calling the `rostopic` function.

### **targetframe** — Target coordinate frame

string scalar | character vector

Target coordinate frame that an entity transforms into, specified as a string scalar or character vector. You can view the available frames for transformation calling `tftree.AvailableFrames`.

### **entity** — Initial message entity

Message object handle

Initial message entity, specified as a Message object handle.

Supported messages are:

- `geometry_msgs/PointStamped`
- `geometry_msgs/PoseStamped`
- `geometry_msgs/QuaternionStamped`
- `geometry_msgs/Vector3Stamped`
- `sensor_msgs/PointCloud2`

### **sourcetime** — ROS or system time

scalar | Time object handle

ROS or system time, specified as a scalar or Time object handle. The scalar is converted to a Time object using `rostopic`.

## Output Arguments

### **tfentity** — Transformed entity

Message object handle

Transformed entity, returned as a Message object handle.

## See Also

`canTransform` | `getTransform`

**Introduced in R2019b**

# waitForServer

Wait for action server to start

## Syntax

```
waitForServer(client)
waitForServer(client,timeout)
```

## Description

`waitForServer(client)` waits until the action server is started up and available to send goals. The `IsServerConnected` property of the `SimpleActionClient` shows the status of the server connection. Press **Ctrl+C** to abort the wait.

`waitForServer(client,timeout)` specifies a timeout period in seconds. If the server does not start up in the timeout period, this function displays an error.

## Examples

### Setup a ROS Action Client and Execute an Action

This example shows how to create a ROS action client and execute the action. Action types must be set up beforehand with an action server running.

You must have set up the `'/fibonacci'` action type. To run this action server, use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

Connect to a ROS network. You must be connected to a ROS network to gather information about what actions are available. Replace `ipaddress` with your network address.

```
ipaddress = '192.168.17.129';
rosinit(ipaddress,11311)
```

```
Initializing global node /matlab_global_node_01856 with NodeURI http://192.168.17.1:57693/
```

List actions available on the network. The only action set up on this network is the `'/fibonacci'` action.

```
rosaction list
/fibonacci
```

Create an action client. Specify the action name.

```
[actClient,goalMsg] = rosactionclient('/fibonacci');
```

Wait for the action client to connect to the server.

```
waitForServer(actClient);
```

The fibonacci action will calculate the fibonacci sequence for a given order specified in the goal message. The goal message was returned when creating the action client and can be modified to send goals to the ROS action server.

```
goalMsg.Order = 8
```

```
goalMsg =  
  ROS FibonacciGoal message with properties:  
  
  MessageType: 'actionlib_tutorials/FibonacciGoal'  
  Order: 8
```

Use `showdetails` to show the contents of the message

Send the goal and wait for its completion. Specify a timeout of 10 seconds to complete the action.

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,10)
```

```
resultMsg =  
  ROS FibonacciResult message with properties:  
  
  MessageType: 'actionlib_tutorials/FibonacciResult'  
  Sequence: [10x1 int32]
```

Use `showdetails` to show the contents of the message

```
resultState =  
'succeeded'
```

```
showdetails(resultMsg)
```

```
Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_01856 with NodeURI http://192.168.17.1:57693/
```

### **Send and Cancel ROS Action Goals**

This example shows how to send and cancel goals for ROS actions. Action types must be setup beforehand with an action server running.

You must have set up the  `'/fibonacci'` action type. To run this action server, use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

First, set up a ROS action client. Then, send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.



Connect to a ROS network with a specified IP address. Create a ROS action client connected to the ROS network using `rosactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.17.129',11311)
```

```
Initializing global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

```
[actClient,goalMsg] = rosactionclient('/fibonacci');
waitForServer(actClient);
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg =
```

```
ROS FibonacciResult message with properties:
```

```
  MessageType: 'actionlib_tutorials/FibonacciResult'
  Sequence: [6×1 int32]
```

```
Use showdetails to show the contents of the message
```

```
resultState =
'succeeded'
```

```
showdetails(resultMsg)
```

```
Sequence : [0, 1, 1, 2, 3, 5]
```

Send a new goal message without waiting.

```
goalMsg.Order = 5;
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

## Input Arguments

### **client** – ROS action client

SimpleActionClient object handle

ROS action client, specified as a `SimpleActionClient` object handle. This simple action client enables you to track a single goal at a time.

**timeout — Timeout period**

scalar in seconds

Timeout period for setting up ROS action server, specified as a scalar in seconds. If the client does not connect to the server in the specified time period, an error is displayed.

**See Also**

`cancelGoal` | `roaction` | `roactionclient` | `sendGoalAndWait`

**Topics**

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

**Introduced in R2019b**

# waitForTransform

Wait until a transformation is available

---

**Note** `waitForTransform` will be removed in a future release. Use `getTransform` with a specified timeout instead. Use `inf` to wait indefinitely.

---

## Syntax

```
waitForTransform(tftree, targetframe, sourceframe)
waitForTransform(tftree, targetframe, sourceframe, timeout)
```

## Description

`waitForTransform(tftree, targetframe, sourceframe)` waits until the transformation between `targetframe` and `sourceframe` is available in the transformation tree, `tftree`. This function disables the command prompt until a transformation becomes available on the ROS network.

`waitForTransform(tftree, targetframe, sourceframe, timeout)` specifies a timeout period in seconds. If the transformation does not become available, MATLAB displays an error, but continues running the current program.

## Examples

### Wait for Transformation Between Robot Frames

Connect to the ROS network. Specify the IP address of your network.

```
rosinit('192.168.17.129')
```

```
Initializing global node /matlab_global_node_48383 with NodeURI http://192.168.17.1:54695/
```

Create a ROS transformation tree.

```
tftree = rostf;
```

Wait for the transformation between the target frame, `/camera_depth_frame`, and the source frame, `/base_link`, to be available. Specify a timeout of 5 seconds.

```
waitForTransform(tftree, '/camera_depth_frame', '/base_link', 5);
```

Get the transformation.

```
tform = getTransform(tftree, '/camera_depth_frame', '/base_link');
```

When you are finished, disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_48383 with NodeURI http://192.168.17.1:54695/
```

## Input Arguments

### **tftree** — ROS transformation tree

TransformationTree object handle

ROS transformation tree, specified as a TransformationTree object handle. You can create a transformation tree by calling the `rostopic` function.

### **targetframe** — Target coordinate frame

string scalar | character vector

Target coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation by calling `tftree.AvailableFrames`.

### **sourceframe** — Initial coordinate frame

string scalar | character vector

Initial coordinate frame, specified as a string scalar or character vector. You can view the available frames for transformation using `tftree.AvailableFrames`.

### **timeout** — Timeout period

numeric scalar in seconds

Timeout period, specified as a numeric scalar in seconds. If the transformation does not become available, MATLAB displays an error, but continues running the current program.

## See Also

`getTransform` | `receive` | `transform`

**Introduced in R2019b**

# writeImage

Write MATLAB image to ROS image message

## Syntax

```
writeImage(msg, img)
writeImage(msg, img, alpha)
```

## Description

`writeImage(msg, img)` converts the MATLAB image, `img`, to a message object and stores the ROS compatible image data in the message object, `msg`. The message must be a `'sensor_msgs/Image'` message. `'sensor_msgs/CompressedImage'` messages are not supported. The function does not perform any color space conversion, so the `img` input needs to have the encoding that you specify in the `Encoding` property of the message.

`writeImage(msg, img, alpha)` converts the MATLAB image, `img`, to a message object. If the image encoding supports an alpha channel (`rgba` or `bgra` family), specify this alpha channel in `alpha`. Alternatively, the input image can store the alpha channel as its fourth channel.

## Examples

### Write Image to Message

Read an image.

```
image = imread('imageMap.png');
```

Create a ROS image message. Specify the default encoding for the image. Write the image to the message.

```
msg = rosmessage('sensor_msgs/Image');
msg.Encoding = 'rgb8';
writeImage(msg, image);
```

## Input Arguments

### **msg** — ROS image message

Image object handle

`'sensor_msgs/Image'` ROS image message, specified as an Image object handle. `'sensor_msgs/Image'` image messages are not supported.

### **img** — Image

grayscale image matrix | RGB image matrix | *m*-by-*n*-by-3 array

Image, specified as a matrix representing a grayscale or RGB image or as an *m*-by-*n*-by-3 array, depending on the sensor image.

**alpha — Alpha channel**`uint8` grayscale image

Alpha channel, specified as a `uint8` grayscale image. Alpha must be the same size and data type as `img`.

**ROS Image Encoding**

You must specify the correct encoding of the input image in the `Encoding` property of the image message. If you do not specify the image encoding before calling the function, the default encoding, `rgb8`, is used (3-channel RGB image with `uint8` values). The function does not perform any color space conversion, so the `img` input needs to have the encoding that you specify in the `Encoding` property of the message.

All encoding types supported for the `readImage` are also supported in this function. For more information on supported encoding types and their representations in MATLAB, see `readImage`.

Bayer-encoded images (`bayer_rggb8`, `bayer_bggr8`, `bayer_gbrg8`, `bayer_grbg8`, and their 16-bit equivalents) must be given as 8-bit or 16-bit single-channel images or they do not encode.

**See Also**`readImage`**Introduced in R2019b**

# Classes

---

## BagSelection

Object for storing rosbag selection

### Description

The `BagSelection` object is an index of the messages within a rosbag. You can use it to extract message data from a rosbag, select messages based on specific criteria, or create a `timeseries` of the message properties.

Use `rosbag` to load a rosbag and create the `BagSelection` object.

Use `select` to filter the rosbag by criteria such as time and topic.

### Creation

#### Syntax

```
bag = rosbag(filename)
```

```
bagsel = select(bag)
```

#### Description

`bag = rosbag(filename)` creates an indexable `BagSelection` object, `bag`, that contains all the message indexes from the rosbag at the input path, `filename`. To access the data, you can call `readMessages` or `timeseries` to extract relevant data.

See `rosbag` for other syntaxes.

`bagsel = select(bag)` returns an object, `bagsel`, that contains all the messages in the `BagSelection` object, `bag`.

This function does not change the contents of the original `BagSelection` object. The return object, `bagsel`, is a new object that contains the specified message selection.

See `select` for other syntaxes and to filter by criteria such as time and topic.

### Properties

#### **FilePath** — Absolute path to rosbag file

character vector

This property is read-only.

Absolute path to the rosbag file, specified as a character vector.

Data Types: `char`



**StartTime — Timestamp of first message in selection**

scalar

This property is read-only.

Timestamp of the first message in the selection, specified as a scalar in seconds.

Data Types: double

**EndTime — Timestamp of last message in selection**

scalar

This property is read-only.

Timestamp of the last message in the selection, specified as a scalar in seconds.

Data Types: double

**NumMessages — Number of messages in selection**

scalar

This property is read-only.

Number of messages in the selection, specified as a scalar. When you first load a rosbag, this property contains the number of messages in the rosbag. Once you select a subset of messages with `select`, the property shows the number of messages in this subset.

Data Types: double

**AvailableTopics — Table of topics in selection**

table

This property is read-only.

Table of topics in the selection, specified as a table. Each row in the table lists one topic, the number of messages for this topic, the message type, and the definition of the type. For example:

	NumMessages	MessageType	MessageDefinition
/odom	99	nav_msgs/Odometry	'# This represents an estimate of a position and

Data Types: table

**AvailableFrames — List of available coordinate frames**

cell array of character vectors

This property is read-only.

List of available coordinate frames, returned as a cell array of character vectors. Use `canTransform` to check whether specific transformations between frames are available, or `getTransform` to query a transformation.

Data Types: cell array

**MessageList — List of messages in selection**

table

This property is read-only.

List of messages in the selection, specified as a table. Each row in the table lists one message.

Data Types: `table`

## Object Functions

<code>canTransform</code>	Verify if transformation is available
<code>getTransform</code>	Retrieve transformation between two coordinate frames
<code>readMessages</code>	Read messages from rosbag
<code>select</code>	Select subset of messages in rosbag
<code>timeseries</code>	Creates a time series object for selected message properties

## Examples

### Create rosbag Selection Using BagSelection Object

Load a rosbag log file and parse out specific messages based on the selected criteria.

Create a `BagSelection` object of all the messages in the rosbag log file.

```
bagMsgs = rosbag('ex_multiple_topics.bag');
```

Select a subset of the messages based on their timestamp and topic.

```
bagMsgs2 = select(bagMsgs, 'Time', ...  
                [bagMsgs.StartTime bagMsgs.StartTime + 1], 'Topic', '/odom');
```

Retrieve the messages in the selection as a cell array.

```
msgs = readMessages(bagMsgs2);
```

Return certain message properties as a time series.

```
ts = timeseries(bagMsgs2, 'Pose.Pose.Position.X', ...  
               'Twist.Twist.Angular.Y');
```

### Retrieve Information from rosbag

Retrieve information from the rosbag. Specify the full path to the rosbag if it is not already available on the MATLAB® path.

```
bagselect = rosbag('ex_multiple_topics.bag');
```

Select a subset of the messages, filtered by time and topic.

```
bagselect2 = select(bagselect, 'Time', ...  
                  [bagselect.StartTime bagselect.StartTime + 1], 'Topic', '/odom');
```

### Display rosbag Information from File

To view information about a rosbag log file, use `rosbag info filename`, where *filename* is a rosbag (.bag) file.

```
rosbag info 'ex_multiple_topics.bag'
```

```
Path:      C:\TEMP\Bdoc20a_1326390_8984\ib9D0363\19\tp6ab89118\ros-ex32890909\ex_multiple_topics.l
Version:   2.0
Duration:  2:00s (120s)
Start:     Dec 31 1969 19:03:21.34 (201.34)
End:       Dec 31 1969 19:05:21.34 (321.34)
Size:      23.6 MB
Messages:  36963
Types:     gazebo_msgs/LinkStates [48c080191eb15c41858319b4d8a609c2]
           nav_msgs/Odometry     [cd5e73d190d741a2f92e81eda573aca7]
           rosgraph_msgs/Clock   [a9c97c1d230cfc112e270351a944ee47]
           sensor_msgs/LaserScan [90c7ef2dc6895d81024acba2ac42f369]
Topics:    /clock                12001 msgs : rosgraph_msgs/Clock
           /gazebo/link_states  11999 msgs : gazebo_msgs/LinkStates
           /odom                11998 msgs : nav_msgs/Odometry
           /scan                965 msgs  : sensor_msgs/LaserScan
```

### Get Transformations from rosbag File

Get transformations from rosbag (.bag) files by loading the rosbag and checking the available frames. From these frames, use `getTransform` to query the transformation between two coordinate frames.

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Get a list of available frames.

```
frames = bag.AvailableFrames;
```

Get the latest transformation between two coordinate frames.

```
tf = getTransform(bag, 'world', frames{1});
```

Check for a transformation available at a specific time and retrieve the transformation. Use `canTransform` to check if the transformation is available. Specify the time using `rostime`.

```
tfTime = rostime(bag.StartTime + 1);
if (canTransform(bag, 'world', frames{1}, tfTime))
    tf2 = getTransform(bag, 'world', frames{1}, tfTime);
end
```

### Read Messages from a rosbag as a Structure

Load the rosbag.

```
bag = rosbag('ros_turtlesim.bag');
```

Select a specific topic.

```
bSel = select(bag, 'Topic', '/turtle1/pose');
```

Read messages as a structure. Specify the `DataFormat` name-value pair when reading the messages. Inspect the first structure in the returned cell array of structures.

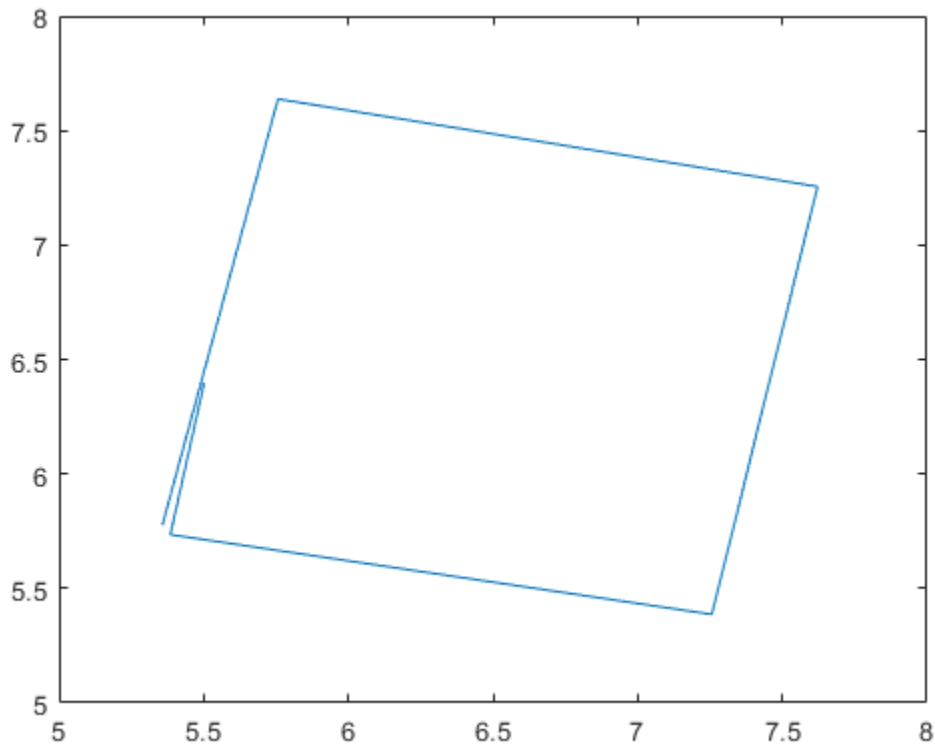
```
msgStructs = readMessages(bSel, 'DataFormat', 'struct');  
msgStructs{1}
```

```
ans = struct with fields:  
    MessageType: 'turtlesim/Pose'  
        X: 5.5016  
        Y: 6.3965  
        Theta: 4.5377  
    LinearVelocity: 1  
    AngularVelocity: 0
```

Extract the xy points from the messages and plot the robot trajectory.

Use `cellfun` to extract all the X and Y fields from the structure. These fields represent the xy positions of the robot during the rosbag recording.

```
xPoints = cellfun(@(m) double(m.X),msgStructs);  
yPoints = cellfun(@(m) double(m.Y),msgStructs);  
plot(xPoints,yPoints)
```



## See Also

[canTransform](#) | [getTransform](#) | [readMessages](#) | [rosbag](#) | [select](#) | [timeseries](#)

## Topics

“Work with rosbag Logfiles”

“ROS Log Files (rosbags)”

**Introduced in R2019b**

## Core

Create ROS Core

## Description

The ROS Core encompasses many key components and nodes that are essential for the ROS network. You must have exactly one ROS core running in the ROS network for nodes to communicate. Using this class allows the creation of a ROS core in MATLAB. Once the core is created, you can connect to it by calling `rosinit` or `ros.Node`.

## Creation

### Syntax

```
core = ros.Core  
core = ros.Core(port)
```

### Description

`core = ros.Core` returns a `Core` object and starts a ROS core in MATLAB. This ROS core has a default port of 11311. MATLAB allows the creation of only one core on any given port and displays an error if another core is detected on the same port.

`core = ros.Core(port)` starts a ROS core at the specified port, `port`.

## Properties

### Port — Network port at which the ROS master is listening

11311 (default) | scalar

This property is read-only.

Network port at which the ROS master is listening, returned as a scalar.

### MasterURI — The URI on which the ROS master can be reached

'http://<HOSTNAME>:11311' (default) | character vector

This property is read-only.

The URI on which the ROS master can be reached, returned as a character vector. The `MasterURI` is constructed based on the host name of your computer. If your host name is not valid, the IP address of your first network interface is used.

## Examples

## Create ROS Core

Create a ROS core on localhost and default port 11311.

```
core = ros.Core
core =
  Core with properties:
    Port: 57113
    MasterURI: 'http://bat5110win64:57113/'
```

Clear the ROS core to shut down the ROS network.

```
clear('core')
```

## Create ROS Core On Specific Port

Create a ROS core on localhost and port 12000.

```
core = ros.Core(12000)
core =
  Core with properties:
    Port: 12000
    MasterURI: 'http://bat5110win64:12000/'
```

Clear the ROS core to shut down the ROS network.

```
clear('core')
```

## See Also

Node | `rosinit`

## Topics

“Connect to a ROS Network”

“ROS Network Setup”

## External Websites

ROS Core

## Introduced in R2019b

# CompressedImage

Create compressed image message

## Description

The `CompressedImage` object is an implementation of the `sensor_msgs/CompressedImage` message type in ROS. The object contains the compressed image and meta-information about the message. You can create blank `CompressedImage` messages and populate them with data, or subscribe to image messages over the ROS network. To convert the image to a MATLAB image, use the `readImage` function.

Only images that are sent through the ROS Image Transport package are supported for conversion to MATLAB images.

## Creation

### Syntax

```
msg = rosmesssage('sensor_msgs/CompressedImage')
```

### Description

`msg = rosmesssage('sensor_msgs/CompressedImage')` creates an empty `CompressedImage` object. To specify image data, use the `msg.Data` property. You can also get these image messages off the ROS network using `rossubscriber`.

## Properties

### MessageType — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: `char`

### Header — ROS Header message

Header object

This property is read-only.

ROS Header message, returned as a `Header` object. This header message contains the `MessageType`, sequence (`Seq`), timestamp (`Stamp`), and `FrameId`.

### Format — Image format

character vector

Image format, specified as a character vector.



Example: 'bgr8; jpeg compressed bgr8'

### Data — Image data

uint8 array

Image data, specified as a uint8 array.

## Object Functions

`readImage` Convert ROS image data into MATLAB image

## Examples

### Read and Write CompressedImage Messages

Read and write a sample ROS `CompressedImage` message by converting it.

Load sample ROS messages and inspect the image message. The `imgcomp` object is a sample ROS `CompressedImage` message object.

```
exampleHelperROSLoadMessages
```

```
imgcomp
```

```
imgcomp =
```

```
ROS CompressedImage message with properties:
```

```
    MessageType: 'sensor_msgs/CompressedImage'  
        Header: [1x1 Header]  
        Format: 'bgr8; jpeg compressed bgr8'  
        Data: [30376x1 uint8]
```

Use `showdetails` to show the contents of the message

Create a MATLAB image from the `CompressedImage` message using `readImage` and display it.

```
I = readImage(imgcomp);  
imshow(I)
```



### Create Blank Compressed Image Message

```
compImg = rosmesssage('sensor_msgs/CompressedImage')
```

```
compImg =  
  ROS CompressedImage message with properties:  
  
    MessageType: 'sensor_msgs/CompressedImage'  
    Header: [1x1 Header]  
    Format: ''  
    Data: [0x1 uint8]
```

Use `showdetails` to show the contents of the message

### See Also

`readImage` | `rosmesssage` | `rossubscriber`

**Topics**

“Work with Specialized ROS Messages”

**Introduced in R2019b**

# Image

Create image message

## Description

The `Image` object is an implementation of the `sensor_msgs/Image` message type in ROS. The object contains the image and meta-information about the message. You can create blank `Image` messages and populate them with data, or subscribe to image messages over the ROS network. To convert the image to a MATLAB image, use the `readImage` function.

## Creation

### Syntax

```
msg = rosmesssage('sensor_msgs/Image')
```

### Description

`msg = rosmesssage('sensor_msgs/Image')` creates an empty `Image` object. To specify image data, use the `msg.Data` property. You can also get these image messages off the ROS network using `rossubscriber`.

## Properties

### MessageType — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: `char`

### Header — ROS Header message

Header object

This property is read-only.

ROS Header message, returned as a `Header` object. This header message contains the `MessageType`, sequence (`Seq`), timestamp (`Stamp`), and `FrameId`.

### Height — Image height in pixels

scalar

Image height in pixels, specified as a scalar.

### Width — Image width in pixels

scalar

Image width in pixels, specified as a scalar.

### Encoding — Image encoding

character vector

Image encoding, specified as a character vector.

Example: 'rgb8'

### IsBigendian — Image byte sequence

true | false

Image byte sequence, specified as true or false.

- `true` —Big endian sequence. Stores the most significant byte in the smallest address.
- `false` —Little endian sequence. Stores the least significant byte in the smallest address.

### Step — Full row length in bytes

integer

Full row length in bytes, specified as an integer. This length depends on the color depth and the pixel width of the image. For example, an RGB image has 3 bytes per pixel, so an image with width 640 has a step of 1920.

### Data — Image data

uint8 array

Image data, specified as a uint8 array.

## Object Functions

`readImage` Convert ROS image data into MATLAB image

`writeImage` Write MATLAB image to ROS image message

## Examples

### Read and Write Image Messages

Read and write a sample ROS Image message by converting it to a MATLAB image. Then, convert a MATLAB® image to a ROS message.

Load sample ROS messages and inspect the image message data. The `img` object is a sample ROS Image message object.

```
exampleHelperROSLoadMessages
```

```
img
```

```
img =
```

```
ROS Image message with properties:
```

```
    MessageType: 'sensor_msgs/Image'
```

```
        Header: [1x1 Header]
```

```
        Height: 480
```

```
        Width: 640
```

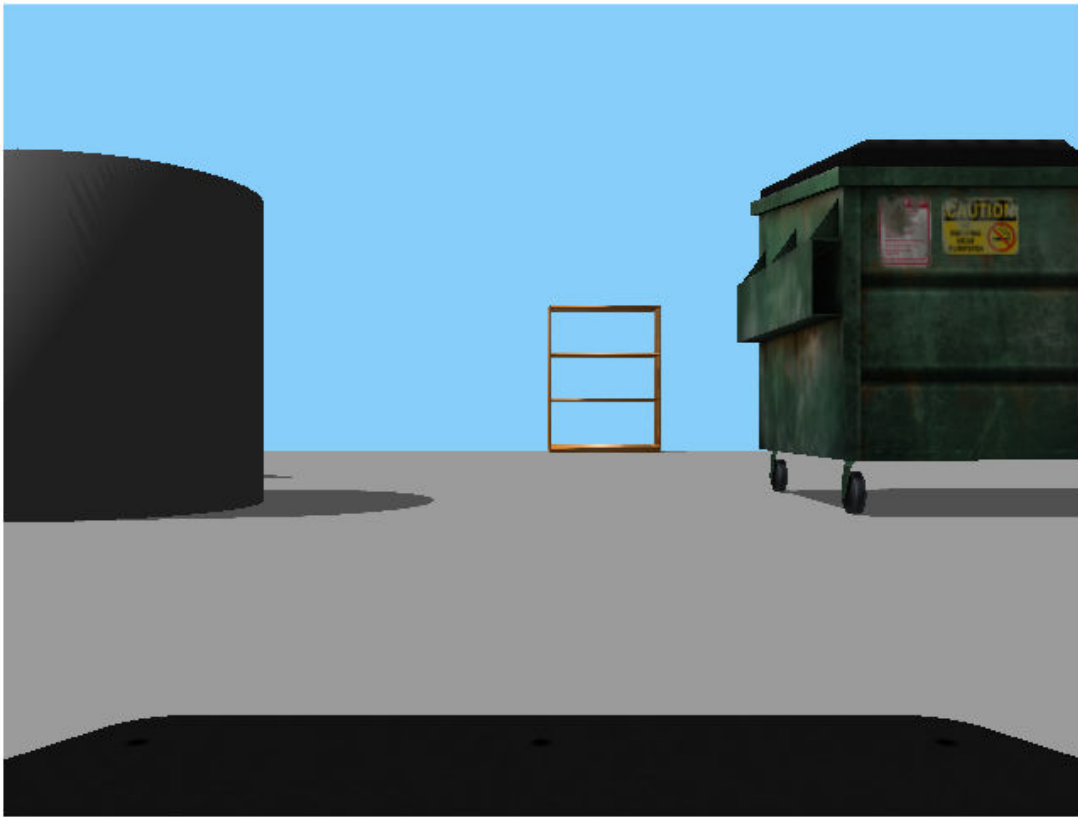
```
        Encoding: 'rgb8'
```

```
IsBigendian: 0
Step: 1920
Data: [921600x1 uint8]
```

Use `showdetails` to show the contents of the message

Create a MATLAB image from the `Image` message using `readImage` and display it.

```
I = readImage(img);
imshow(I)
```



Create a ROS `Image` message from a MATLAB image.

```
imgMsg = rosmesssage('sensor_msgs/Image');
imgMsg.Encoding = 'rgb8'; % Specifies Image Encoding Type
writeImage(imgMsg,I)
imgMsg
```

```
imgMsg =
  ROS Image message with properties:
```

```
  MessageType: 'sensor_msgs/Image'
  Header: [1x1 Header]
```

```
    Height: 480
    Width: 640
    Encoding: 'rgb8'
    IsBigendian: 0
    Step: 1920
    Data: [921600x1 uint8]
```

Use `showdetails` to show the contents of the message

### Create Blank Image Message

```
msg = rosmessage('sensor_msgs/Image')
```

```
msg =
  ROS Image message with properties:
```

```
    MessageType: 'sensor_msgs/Image'
    Header: [1x1 Header]
    Height: 0
    Width: 0
    Encoding: ''
    IsBigendian: 0
    Step: 0
    Data: [0x1 uint8]
```

Use `showdetails` to show the contents of the message

### See Also

[readImage](#) | [rosmessage](#) | [rossubscriber](#) | [writeImage](#)

### Topics

“Work with Specialized ROS Messages”

### Introduced in R2019b

# LaserScan

Create laser scan message

## Description

The LaserScan object is an implementation of the `sensor_msgs/LaserScan` message type in ROS. The object contains meta-information about the message and the laser scan data. You can extract the ranges and angles using the `Ranges` property and the `readScanAngles` function. To access points in Cartesian coordinates, use `readCartesian`.

You can also convert this object to a `lidarScan` object to use with other robotics algorithms such as `matchScans`, `controllerVFH`, or `monteCarloLocalization`.

## Creation

### Syntax

```
scan = rosmessage('sensor_msgs/LaserScan')
```

### Description

`scan = rosmessage('sensor_msgs/LaserScan')` creates an empty LaserScan object. You can specify scan info and data using the properties, or you can get these messages off a ROS network using `rossubscriber`.

## Properties

### MessageType — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: `char`

### Header — ROS Header message

Header object

This property is read-only.

ROS Header message, returned as a Header object. This header message contains the `MessageType`, sequence (`Seq`), timestamp (`Stamp`), and `FrameId`. Timestamp relates to the acquisition time of the first ray in the scan.

### AngleMin — Minimum angle of range data

scalar



Minimum angle of range data, specified as a scalar in radians. Positive angles are measured from the forward direction of the robot.

#### **AngleMax — Maximum angle of range data**

scalar

Maximum angle of range data, specified as a scalar in radians. Positive angles are measured from the forward direction of the robot.

#### **AngleIncrement — Angle increment of range data**

scalar

Angle increment of range data, specified as a scalar in radians.

#### **TimeIncrement — Time between individual range data points in seconds**

scalar

Time between individual range data points in seconds, specified as a scalar.

#### **ScanTime — Time to complete a full scan in seconds**

scalar

Time to complete a full scan in seconds, specified as a scalar.

#### **RangeMin — Minimum valid range value**

scalar

Minimum valid range value, specified as a scalar.

#### **RangeMax — Maximum valid range value**

scalar

Maximum valid range value, specified as a scalar.

#### **Ranges — Range readings from laser scan**

vector

Range readings from laser scan, specified as a vector. To get the corresponding angles, use `readScanAngles`.

#### **Intensities — Intensity values from range readings**

vector

Intensity values from range readings, specified as a vector. If no valid intensity readings are found, this property is empty.

### **Object Functions**

<code>lidarScan</code>	Create object for storing 2-D lidar scan
<code>plot</code>	Display laser or lidar scan readings
<code>readCartesian</code>	Read laser scan ranges in Cartesian coordinates
<code>readScanAngles</code>	Return scan angles for laser scan range readings

### **Examples**

### Inspect Sample Laser Scan Message

Load, inspect, and display a sample laser scan message.

Create sample messages and inspect the laser scan message data. The scan object is a sample ROS LaserScan message object.

```
exampleHelperROSLoadMessages
scan
```

```
scan =
  ROS LaserScan message with properties:

  MessageType: 'sensor_msgs/LaserScan'
  Header: [1x1 Header]
  AngleMin: -0.5467
  AngleMax: 0.5467
  AngleIncrement: 0.0017
  TimeIncrement: 0
  ScanTime: 0.0330
  RangeMin: 0.4500
  RangeMax: 10
  Ranges: [640x1 single]
  Intensities: [0x1 single]
```

Use showdetails to show the contents of the message

Get ranges and angles from the object properties. Check that the ranges and angles are the same size.

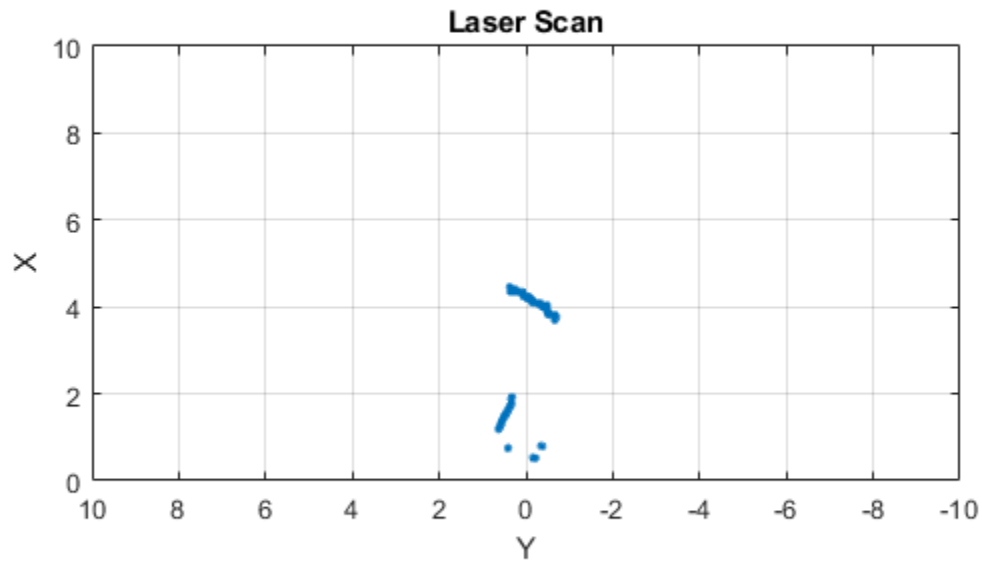
```
ranges = scan.Ranges;
angles = scan.readScanAngles;
size(ranges)
```

```
ans = 1x2
    640     1
```

```
size(angles)
```

```
ans = 1x2
    640     1
```

```
plot(scan)
```



### Create Empty LaserScan Message

```
scan = rosmesssage('sensor_msgs/LaserScan')
```

```
scan =
```

```
ROS LaserScan message with properties:
```

```
  MessageType: 'sensor_msgs/LaserScan'  
  Header: [1x1 Header]  
  AngleMin: 0  
  AngleMax: 0  
AngleIncrement: 0  
TimeIncrement: 0  
  ScanTime: 0  
  RangeMin: 0  
  RangeMax: 0  
  Ranges: [0x1 single]  
  Intensities: [0x1 single]
```

Use `showdetails` to show the contents of the message

**See Also**

lidarScan | plot | readCartesian | readScanAngles | rosmessage | rossubscriber | showdetails

**Topics**

“Work with Specialized ROS Messages”

**Introduced in R2019b**

# Node

Start ROS node and connect to ROS master

## Description

The `ros.Node` object represents a ROS node in the ROS network. The object enables you to communicate with the rest of the ROS network. You must create a node before you can use other ROS functionality, such as publishers, subscribers, and services.

You can create a ROS node using the `rosinit` function, or by calling `ros.Node`:

- `rosinit` — Creates a single ROS node in MATLAB. You can specify an existing ROS master, or the function creates one for you. The `Node` object is not visible.
- `ros.Node`— Creates multiple ROS nodes for use on the same ROS network in MATLAB.

## Creation

### Syntax

```
N = ros.Node(Name)
N = ros.Node(Name,Host)
N = ros.Node(Name,Host,Port)
N = ros.Node(Name,MasterURI,Port)
N = ros.Node( ___, 'NodeHost',HostName)
```

### Description

`N = ros.Node(Name)` initializes the ROS node with `Name` and tries to connect to the ROS master at default URI, `http://localhost:11311`.

`N = ros.Node(Name,Host)` tries to connect to the ROS master at the specified IP address or host name, `Host` using the default port number, `11311`.

`N = ros.Node(Name,Host,Port)` tries to connect to the ROS master with port number, `Port`.

`N = ros.Node(Name,MasterURI,Port)` tries to connect to the ROS master at the specified IP address, `MasterURI`.

`N = ros.Node( ___, 'NodeHost',HostName)` specifies the IP address or host name that the node uses to advertise itself to the ROS network. Examples include `"192.168.1.1"` or `"comp-home"`. You can use any of the arguments from the previous syntaxes.

## Properties

### Name — Name of the node

string scalar | character vector

Name of the node, specified as a string scalar or character vector. The node name must be a valid ROS graph name. See ROS Names.

**MasterURI — URI of the ROS master**

string scalar | character vector

URI of the ROS master, specified as a string scalar or character vector. The node is connected to the ROS master with the given URI.

**NodeURI — URI for the node**

string scalar | character vector

URI for the node, specified as a string scalar or character vector. The node uses this URI to advertise itself on the ROS network for others to connect to it.

**CurrentTime — Current ROS network time**

Time object

Current ROS network time, specified as a Time object. For more information, see `rostime`.

## Examples

**Create Multiple ROS Nodes**

Create multiple ROS nodes. Use the `Node` object with publishers, subscribers, and other ROS functionality to specify the node the you are connecting to.

Create a ROS master.

```
master = ros.Core;
```

Initialize multiple nodes.

```
node1 = ros.Node('/test_node_1');  
node2 = ros.Node('/test_node_2');
```

Use these nodes to perform separate operations and send separate messages. A message published by `node1` can be accessed by a subscriber running in `node2`.

```
pub = ros.Publisher(node1, '/chatter', 'std_msgs/String');  
sub = ros.Subscriber(node2, '/chatter', 'std_msgs/String');
```

```
msg = rosmessage('std_msgs/String');  
msg.Data = 'Message from Node 1';
```

Send a message from `node1`. The subscriber attached to `node2` will receive the message.

```
send(pub,msg) % Sent from node 1  
pause(1) % Wait for message to update  
sub.LatestMessage
```

```
ans =  
ROS String message with properties:
```

```
    MessageType: 'std_msgs/String'
```

```
Data: 'Message from Node 1'
```

```
Use showdetails to show the contents of the message
```

Clear the ROS network of publisher, subscriber, and nodes. Delete the Core object to shut down the ROS master.

```
clear('pub','sub','node1','node2')
clear('master')
```

### Connect to Multiple ROS Masters

Connecting to multiple ROS masters is possible using MATLAB®. These separate ROS masters do not share information and must have different port numbers. Connect ROS nodes to each master based on how you want to distribute information across the network.

Create two ROS masters on different ports.

```
m1 = ros.Core; % Default port of 11311
m2 = ros.Core(12000);
```

Connect separate ROS nodes to each ROS master.

```
node1 = ros.Node('/test_node_1','localhost');
node2 = ros.Node('/test_node_2','localhost',12000);
```

Clear the ROS nodes. Shut down the ROS masters.

```
clear('node1','node2')
clear('m1','m2')
```

### See Also

[rosinit](#) | [rosshutdown](#)

### Topics

“ROS Network Setup”

### External Websites

ROS Nodes

### Introduced in R2019b

## ros2node

Create a ROS 2 node on the specified network

### Description

This class represents a ROS 2 node, and allows you to communicate with the rest of the ROS 2 network. You have to create a node before you can create publishers and subscribers.

### Creation

#### Syntax

```
node = ros2node(Name)
node = ros2node(Name, ID)
```

#### Description

`node = ros2node(Name)` initializes a ROS 2 node with the given `Name`. The node will be on the network specified by the domain identification `0`, unless otherwise specified by the `ROS_DOMAIN_ID` environment variable.

`node = ros2node(Name, ID)` will initialize the ROS 2 node with `Name` and connect to the network using domain `ID`.

#### Input Arguments

##### **Name — Name of the node**

string | char array

The name of the node on the ROS 2 network.

---

**Note** In ROS 1, node names are unique and this is being enforced by shutting down existing nodes when a new node with the same name is started. In ROS 2, the uniqueness of node names is not yet enforced. When creating a new node, use `ros2` function to list existing nodes.

---

##### **ID — Domain identification of the network**

non-negative scalar integer

The domain identification of the ROS 2 network.

Data Types: double

### Properties

##### **Name — Name of the node**

char array



This property is read-only.

The name of the node on the ROS 2 network.

Example: `"/node_1"`

Data Types: `char`

### **ID – Domain identification of the network**

non-negative scalar integer between 0 and 232

This property is read-only.

The domain identification of the ROS 2 network, specified as a non-negative scalar integer between 0 and 232.

Example: `2`

Data Types: `double`

## **Object Functions**

`delete` Remove reference to ROS 2 node

## **Examples**

### **Initialize a Node on Default ROS 2 Network**

Initialize the node, `"/node_1"`, on the default ROS 2 network.

```
node1 = ros2node('/node_1')
```

```
node1 =  
  ros2node with properties:
```

```
  Name: '/node_1'  
  ID: 0
```

### **Initialize a Node on Specified ROS 2 Network**

Initialize the node, `"/node_2"`, on the ROS 2 network identified with domain 2.

```
node2 = ros2node("/node_2", 2)
```

```
node2 =  
  ros2node with properties:
```

```
  Name: '/node_2'  
  ID: 2
```

**See Also**

ros2publisher | ros2subscriber

**Introduced in R2019b**

# OccupancyGrid

Create occupancy grid message

## Description

The `OccupancyGrid` object is an implementation of the `nav_msgs/OccupancyGrid` message type in ROS. The object contains meta-information about the message and the occupancy grid data. To create a `binaryOccupancyMap` object from a ROS message, use `readBinaryOccupancyGrid`.

## Creation

### Syntax

```
msg = rosmesssage('nav_msgs/OccupancyGrid');
```

### Description

`msg = rosmesssage('nav_msgs/OccupancyGrid');` creates an empty `OccupancyGrid` object. To specify map information and data, use the `map.Info` and `msg.Data` properties. You can also get the occupancy grid messages off the ROS network using `rossubscriber`.

## Properties

### MessageType — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: `char`

### Header — ROS Header message

Header object

This property is read-only.

ROS Header message, returned as a `Header` object. This header message contains the `MessageType`, sequence (`Seq`), timestamp (`Stamp`), and `FrameId`.

### Info — Information about the map

`MapMetaData` object

Information about the map, specified as a `MapMetaData` object. It contains the width, height, resolution, and origin of the map.

### Data — Map data

vector

Map data, specified as a vector. The vector is all the occupancy data from each grid location in a single 1-D array.

## Object Functions

`readBinaryOccupancyGrid` Read binary occupancy grid  
`writeBinaryOccupancyGrid` Write values from grid to ROS message

## Examples

### Create Occupancy Grid from 2-D Map

Load two maps, `simpleMap` and `complexMap`, as logical matrices. Use `whos` to display the map.

```
load exampleMaps.mat
whos *Map*
```

Name	Size	Bytes	Class	Attributes
<code>complexMap</code>	41x52	2132	logical	
<code>emptyMap</code>	26x27	702	logical	
<code>simpleMap</code>	26x27	702	logical	
<code>ternaryMap</code>	501x501	2008008	double	

Create a ROS message from `simpleMap` using a `binaryOccupancyMap` object. Write the `OccupancyGrid` message using `writeBinaryOccupancyGrid`.

```
bogMap = binaryOccupancyMap(double(simpleMap));
mapMsg = rosmesssage('nav_msgs/OccupancyGrid');
writeBinaryOccupancyGrid(mapMsg,bogMap)
mapMsg
```

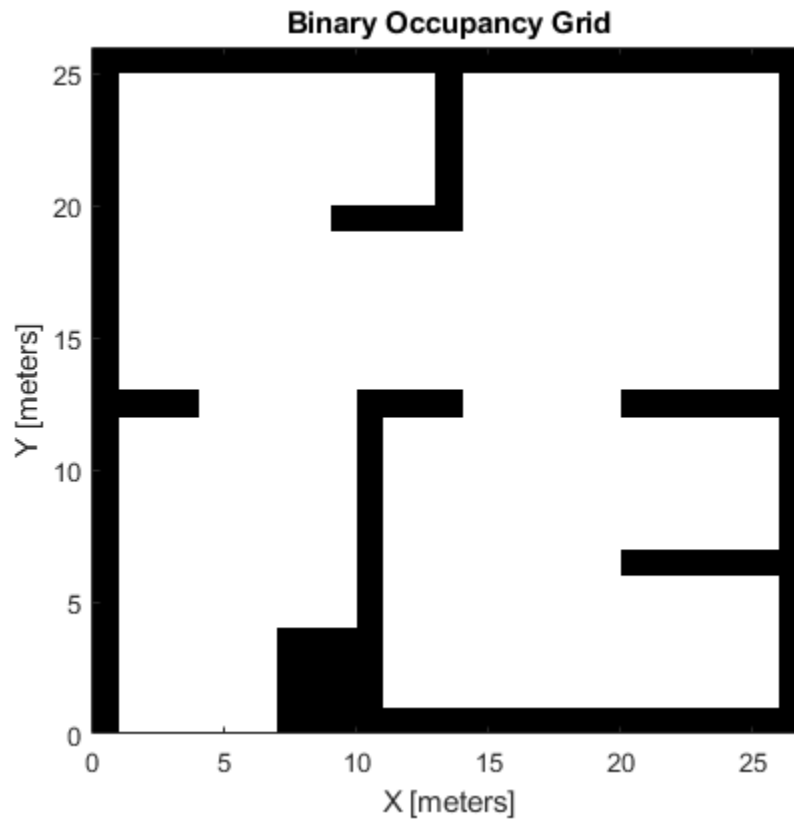
```
mapMsg =
  ROS OccupancyGrid message with properties:
```

```
  MessageType: 'nav_msgs/OccupancyGrid'
  Header: [1x1 Header]
  Info: [1x1 MapMetaData]
  Data: [702x1 int8]
```

Use `showdetails` to show the contents of the message

Use `readBinaryOccupancyGrid` to convert the ROS message to a `binaryOccupancyMap` object. Use the object function `show` to display the map.

```
bogMap2 = readBinaryOccupancyGrid(mapMsg);
show(bogMap2);
```



### See Also

[binaryOccupancyMap](#) | [readBinaryOccupancyGrid](#) | [rosmesssage](#) | [rosubscrber](#) | [writeBinaryOccupancyGrid](#)

### Topics

“Occupancy Grids” (Navigation Toolbox)

**Introduced in R2019b**

# PointCloud2

Access point cloud messages

## Description

The `PointCloud2` object is an implementation of the `sensor_msgs/PointCloud2` message type in ROS. The object contains meta-information about the message and the point cloud data. To access the actual data, use `readXYZ` to get the point coordinates and `readRGB` to get the color information, if available.

## Creation

### Syntax

```
ptcloud = rosmesssage('sensor_msgs/PointCloud2')
```

### Description

`ptcloud = rosmesssage('sensor_msgs/PointCloud2')` creates an empty `PointCloud2` object. To specify point cloud data, use the `ptcloud.Data` property. You can also get point cloud data messages off the ROS network using `rossubscriber`.

## Properties

### **PreserveStructureOnRead — Preserve the shape of point cloud matrix**

`false` (default) | `true`

This property is read-only.

Preserve the shape of point cloud matrix, specified as `false` or `true`. When the property is `true`, the output data from `readXYZ` and `readRGB` are returned as matrices instead of vectors.

### **MessageType — Message type of ROS message**

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: `char`

### **Header — ROS Header message**

Header object

This property is read-only.

ROS Header message, returned as a `Header` object. This header message contains the `MessageType`, sequence (`Seq`), timestamp (`Stamp`), and `FrameId`.

**Height — Point cloud height in pixels**

integer

Point cloud height in pixels, specified as an integer.

**Width — Point cloud width in pixels**

integer

Point cloud width in pixels, specified as an integer.

**IsBigendian — Image byte sequence**

true | false

Image byte sequence, specified as `true` or `false`.

- `true` —Big endian sequence. Stores the most significant byte in the smallest address.
- `false` —Little endian sequence. Stores the least significant byte in the smallest address.

**PointStep — Length of a point in bytes**

integer

Length of a point in bytes, specified as an integer.

**RowStep — Full row length in bytes**

integer

Full row length in bytes, specified as an integer. The row length equals the `PointStep` property multiplied by the `Width` property.

**Data — Point cloud data**

uint8 array

Point cloud data, specified as a `uint8` array. To access the data, use the “Object Functions” on page 2-33.

**Object Functions**

<code>readAllFieldNames</code>	Get all available field names from ROS point cloud
<code>readField</code>	Read point cloud data based on field name
<code>readRGB</code>	Extract RGB values from point cloud data
<code>readXYZ</code>	Extract XYZ coordinates from point cloud data
<code>scatter3</code>	Display point cloud in scatter plot
<code>showdetails</code>	Display all ROS message contents

**Examples****Inspect Point Cloud Image**

Access and visualize the data inside a point cloud message.

Create sample ROS messages and inspect a point cloud image. `ptcloud` is a sample ROS `PointCloud2` message object.

```
exampleHelperROSLoadMessages  
ptcloud
```

```
ptcloud =  
  ROS PointCloud2 message with properties:  
  
  PreserveStructureOnRead: 0  
    MessageType: 'sensor_msgs/PointCloud2'  
    Header: [1x1 Header]  
    Height: 480  
    Width: 640  
  IsBigendian: 0  
  PointStep: 32  
  RowStep: 20480  
  IsDense: 0  
  Fields: [4x1 PointField]  
  Data: [9830400x1 uint8]
```

Use `showdetails` to show the contents of the message

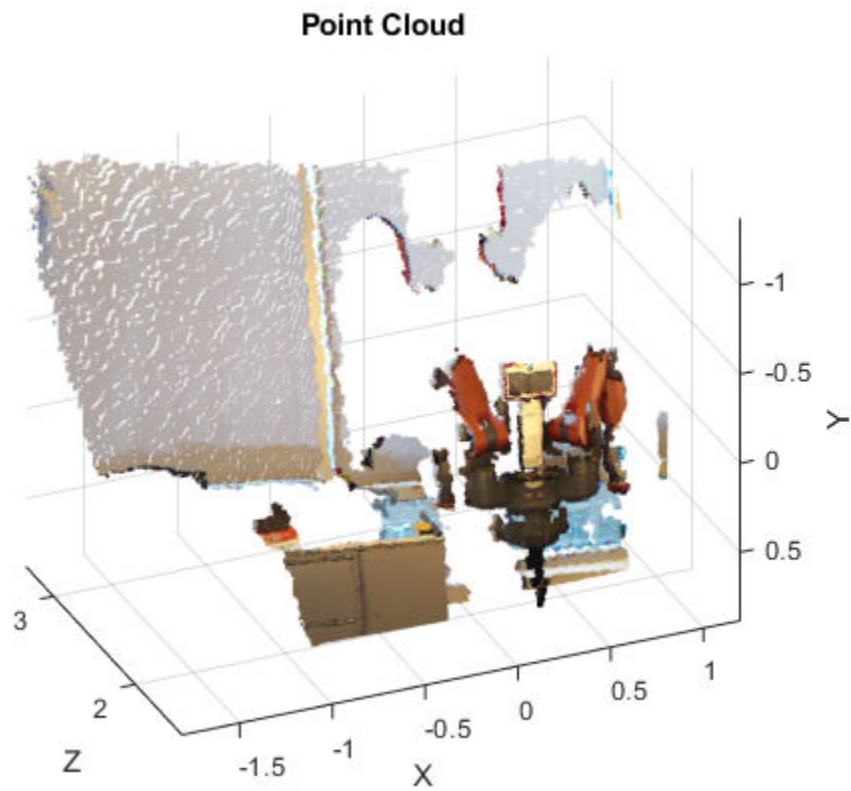
Get RGB info and xyz-coordinates from the point cloud using `readXYZ` and `readRGB`.

```
xyz = readXYZ(ptcloud);  
rgb = readRGB(ptcloud);
```

Display the point cloud in a figure using `scatter3`.

```
scatter3(ptcloud)
```





### Create pointCloud Object Using Point Cloud Message

Convert a ROS Toolbox™ point cloud message into a Computer Vision System Toolbox™ pointCloud object.

Load sample messages.

```
exampleHelperROSLoadMessages
```

Convert a ptcloud message to the pointCloud object.

```
pcobj = pointCloud(readXYZ(ptcloud), 'Color', uint8(255*readRGB(ptcloud)))
```

```
pcobj =
  pointCloud with properties:
    Location: [307200x3 single]
    Color: [307200x3 uint8]
    Normal: []
    Intensity: []
    Count: 307200
    XLimits: [-1.8147 1.1945]
    YLimits: [-1.3714 0.8812]
    ZLimits: [1.4190 3.3410]
```

**See Also**

readAllFieldNames | readField | readRGB | readXYZ | rosmessage | rossubscriber | scatter3 | showdetails

**Topics**

“Work with Specialized ROS Messages”

**Introduced in R2019b**

# rosdevice

Connect to remote ROS device

## Description

The `rosdevice` object is used to create a connection with a ROS device. The object contains the necessary login information and other parameters of the ROS distribution. Once a connection is made using `rosdevice`, you can run and stop a ROS core or ROS nodes and check the status of the ROS network. Before running ROS nodes, you must connect MATLAB to the ROS network using the `rosinit` function.

You can deploy ROS nodes to a ROS device using Simulink models. For an example, see “Generate a Standalone ROS Node from Simulink®”.

---

**Note** To connect to a ROS device, an SSH server must be installed on the device.

---

## Creation

### Syntax

```
device = rosdevice(deviceAddress,username,password)
device = rosdevice
```

### Description

`device = rosdevice(deviceAddress,username,password)` creates a `rosdevice` object connected to the ROS device at the specified address and with the specified user name and password.

`device = rosdevice` creates a `rosdevice` object connected to a ROS device using the saved values for `deviceAddress`, `username`, and `password`.

## Properties

### DeviceAddress — Host name or IP address of the ROS device

character vector

This property is read-only.

Host name or IP address of the ROS device, specified as a character vector.

Example: '192.168.1.10'

Example: 'samplehost.foo.com'

### UserName — User name used to connect to the ROS device

character vector

This property is read-only.

User name used to connect to the ROS device, specified as a character vector.

Example: 'user'

### **ROSFolder — Location of ROS installation**

character vector

Location of ROS installation, specified as a character vector. If a folder is not specified, MATLAB tries to determine the correct folder for you. When you deploy a ROS node, set this value from Simulink in the Configuration Parameters dialog box, under **Hardware Implementation**.

Example: '/opt/ros/hydro'

### **CatkinWorkspace — Catkin folder where models are deployed on device**

character vector

Catkin folder where models are deployed on device, specified as a character vector. When you deploy a ROS node, set this value from Simulink in the Configuration Parameters dialog box, under **Hardware Implementation**.

Example: '~/catkin\_ws\_test'

### **AvailableNodes — Nodes available to run on ROS device**

cell array of character vectors

This property is read-only.

Nodes available to run on a ROS device, returned as a cell array of character vectors. Nodes are only listed if they are part of the `CatkinWorkspace` and have been deployed to the device using Simulink.

Example: {'robotcontroller', 'publishernode'}

## **Object Functions**

<code>runNode</code>	Start ROS node
<code>stopNode</code>	Stop ROS node
<code>isNodeRunning</code>	Determine if ROS node is running
<code>runCore</code>	Start ROS core
<code>stopCore</code>	Stop ROS core
<code>isCoreRunning</code>	Determine if ROS core is running
<code>system</code>	Execute system command on device
<code>putFile</code>	Copy file to device
<code>getFile</code>	Get file from device
<code>deleteFile</code>	Delete file from device
<code>dir</code>	List folder contents on device
<code>openShell</code>	Open interactive command shell to device

## **Examples**

### **Run ROS Core on ROS Device**

Connect to a remote ROS device and start a ROS core. The ROS core is needed to run ROS nodes to communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.17.128';
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
  rosdevice with properties:

    DeviceAddress: '192.168.17.128'
    Username: 'user'
    ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws'
    AvailableNodes: {0x1 cell}
```

Run a ROS core and check if it is running.

```
runCore(d)
```

Another roscore / ROS master is already running on the ROS device. Use the 'stopCore' function to

```
running = isCoreRunning(d)
```

```
running = logical
         1
```

Stop the ROS core and confirm that it is no longer running.

```
stopCore(d)
running = isCoreRunning(d)
```

```
running = logical
         1
```

## Run ROS Node on ROS Device

Connect to a remote ROS device and start a ROS node. Run a ROS core so that ROS nodes can communicate via a ROS network. You can run and stop a ROS core or node and check their status using a `rosdevice` object.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device already contains the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.203.129';
d = rosdevice(ipaddress, 'user', 'password');
d.ROSFolder = '/opt/ros/indigo';
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
d =
  rosdevice with properties:

    DeviceAddress: '192.168.203.129'
```

```
Username: 'user'  
ROSFolder: '/opt/ros/indigo'  
CatkinWorkspace: '~/catkin_ws_test'  
AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. Connect MATLAB® to the ROS master using `rosinit`. This core enables you to run ROS nodes on your ROS device.

```
runCore(d)  
rosinit(d.DeviceAddress,11311)
```

```
Initializing global node /matlab_global_node_84497 with NodeURI http://192.168.203.1:56034/
```

Check the available ROS nodes on the connected ROS device. These nodes listed were generated from Simulink® models following the process in the “Get Started with ROS in Simulink®” example.

```
d.AvailableNodes
```

```
ans = 1x2 cell  
      {'robotcontroller'}      {'robotcontroller2'}
```

Run a ROS node and specify the node name. Check if the node is running.

```
runNode(d, 'RobotController')  
running = isNodeRunning(d, 'RobotController')
```

```
running = logical  
         1
```

Stop the ROS node. Disconnect from the ROS network. Stop the ROS core.

```
stopNode(d, 'RobotController')  
rosshutdown
```

```
Shutting down global node /matlab_global_node_84497 with NodeURI http://192.168.203.1:56034/
```

```
stopCore(d)
```

## Run Multiple ROS Nodes

Run multiple ROS nodes on a connected ROS device. ROS nodes can be generated using Simulink® models to perform different tasks on the ROS network. These nodes are then deployed on a ROS device and can be run independently of Simulink®.

This example uses two different Simulink models that have been deployed as ROS nodes. See “Generate a Standalone ROS Node from Simulink®” and follow the instructions to generate and deploy a ROS node. Do this twice and name them 'robotcontroller' and 'robotcontroller2'. The 'robotcontroller' node sends velocity commands to a robot to navigate it to a given point. The 'robotcontroller2' node uses the same model, but doubles the linear velocity to drive the robot faster.

Create a connection to a ROS device. Specify the address, user name, and password of your specific ROS device. The device contains information about the ROS device, including the available ROS nodes that can be run using `runNode`.

```
ipaddress = '192.168.203.129';
d = rosdevice(ipaddress, 'user', 'password')
```

```
d =
  rosdevice with properties:

    DeviceAddress: '192.168.203.129'
    Username: 'user'
    ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws'
    AvailableNodes: {0x1 cell}
```

```
d.CatkinWorkspace = '~/catkin_ws_test'
```

```
d =
  rosdevice with properties:

    DeviceAddress: '192.168.203.129'
    Username: 'user'
    ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws_test'
    AvailableNodes: {'robotcontroller' 'robotcontroller2'}
```

Run a ROS core. The ROS Core is the master enables you to run ROS nodes on your ROS device. Connect MATLAB® to the ROS master using `rosinit`. For this example, the port is set to 11311. `rosinit` can automatically select a port for you without specifying this input.

```
runCore(d)
rosinit(d.DeviceAddress, 11311)
```

```
Initializing global node /matlab_global_node_66434 with NodeURI http://192.168.203.1:59395/
```

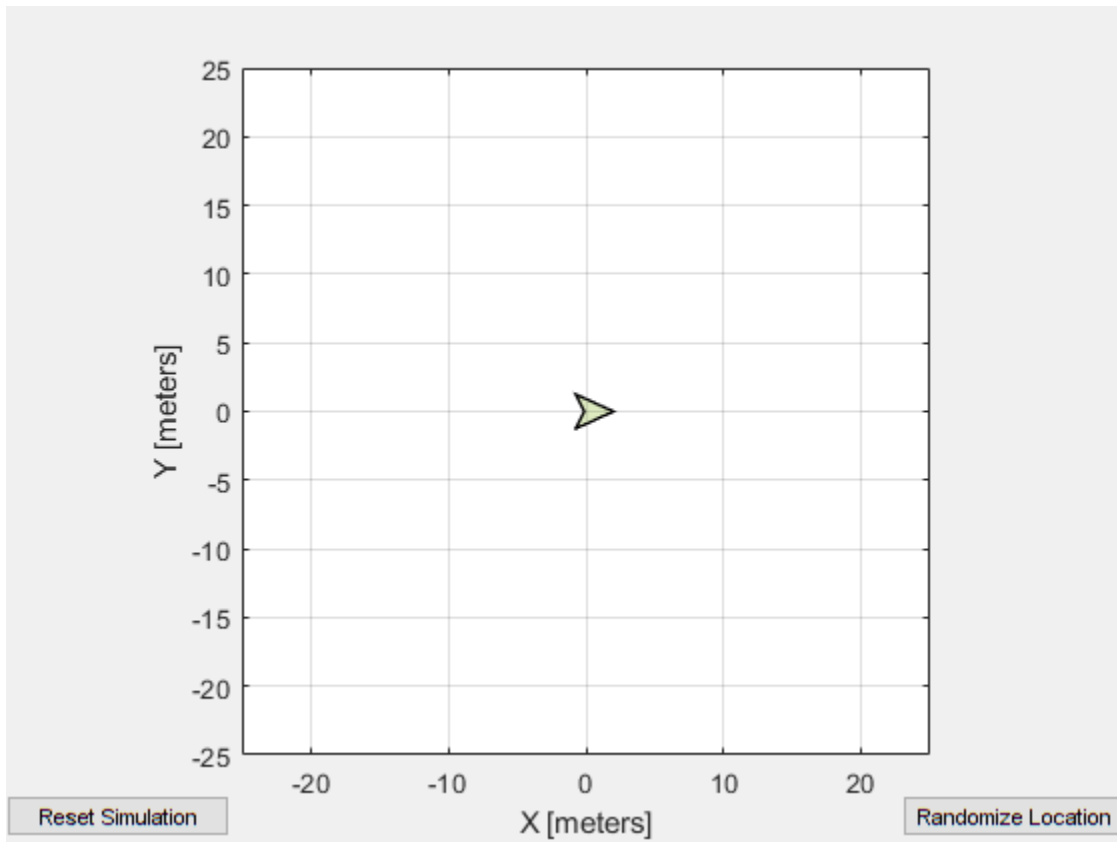
Check the available ROS nodes on the connected ROS device. The nodes listed in this example were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink®” example. Two separate nodes are generated, 'robotcontroller' and 'robotcontroller2', which have the linear velocity set to 1 and 2 in the model respectively.

```
d.AvailableNodes
```

```
ans = 1x2 cell
    {'robotcontroller'}    {'robotcontroller2'}
```

Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

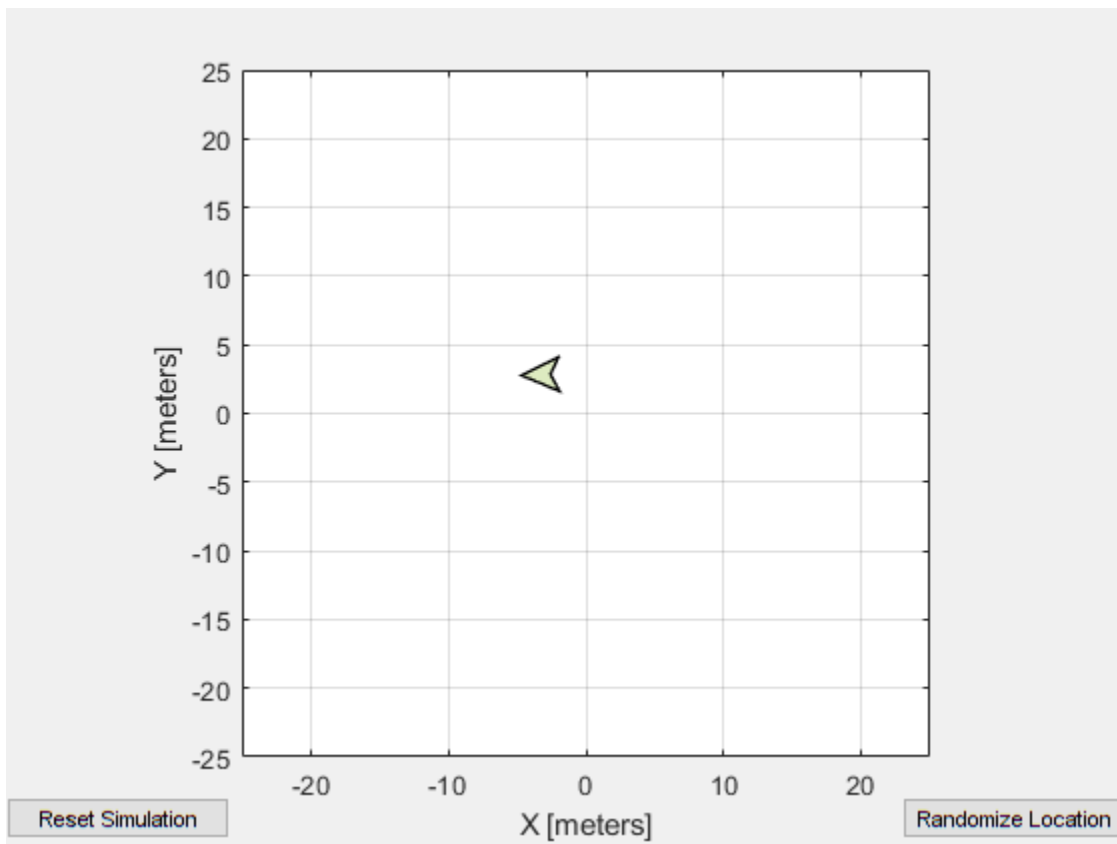
```
sim = ExampleHelperSimulinkRobotROS;
```



Run a ROS node, specifying the node name. The 'robotcontroller' node commands the robot to a specific location ([ -10 10]). Wait to see the robot drive.

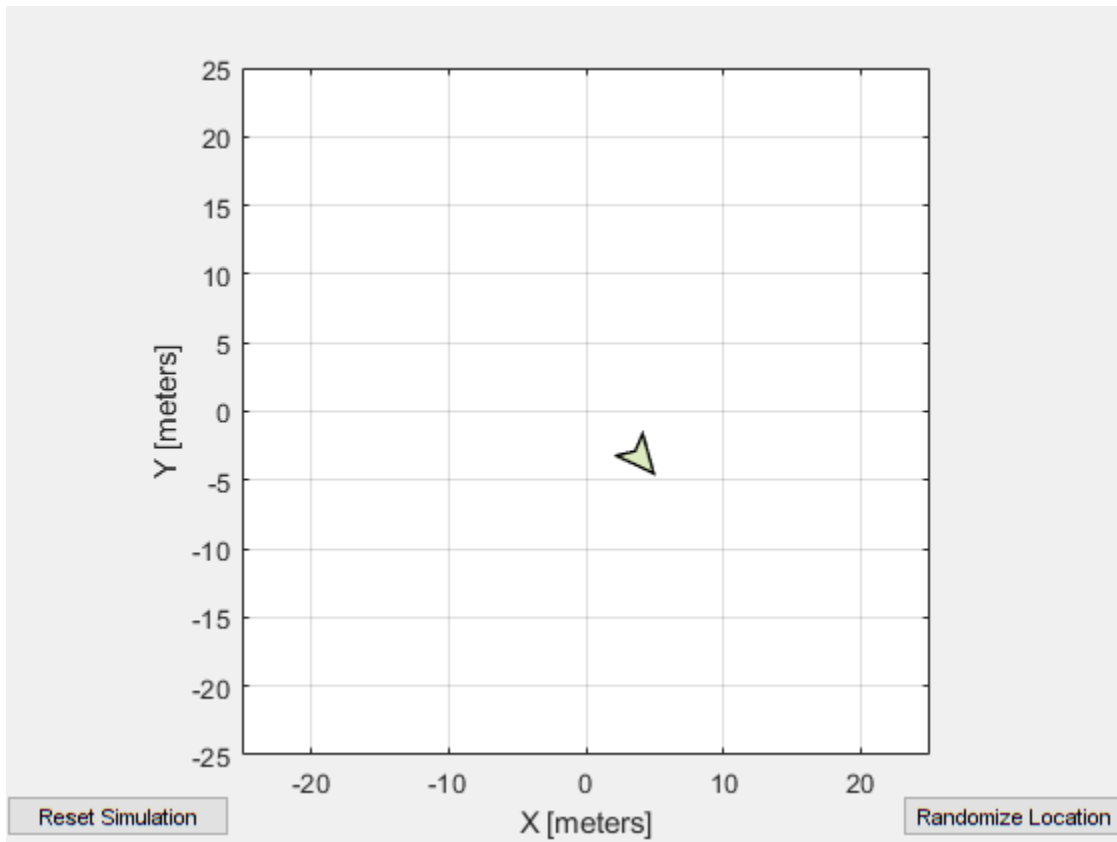
```
runNode(d, 'robotcontroller')  
pause(10)
```





Reset the Robot Simulator to reset the robot position. Alternatively, click **Reset Simulation**. Because the node is still running, the robot continues back to the specific location. To stop sending commands, stop the node.

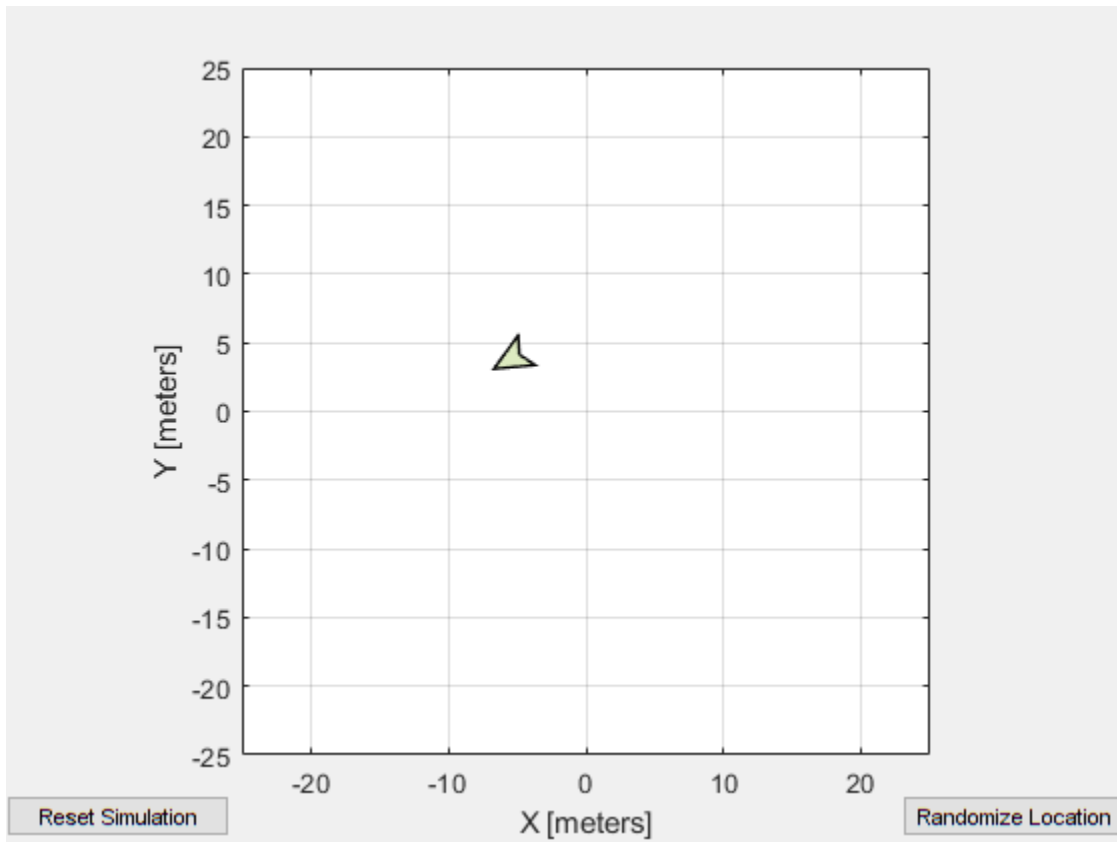
```
resetSimulation(sim.Simulator)
pause(5)
```



```
stopNode(d, 'robotcontroller')
```

Run the 'robotcontroller2' node. This model drives the robot with twice the linear velocity. Reset the robot position. Wait to see the robot drive. You should see a wider turn due to the increased velocity.

```
runNode(d, 'robotcontroller2')
resetSimulation(sim.Simulator)
pause(10)
```



Close the simulator. Stop the ROS node. Disconnect from the ROS network and stop the ROS core.

```
close
stopNode(d, 'robotcontroller2')
roshutdown
```

```
Shutting down global node /matlab_global_node_66434 with NodeURI http://192.168.203.1:59395/
stopCore(d)
```

## See Also

[isNodeRunning](#) | [runCore](#) | [runNode](#) | [stopNode](#)

## Topics

“Generate a Standalone ROS Node from Simulink®”

**Introduced in R2019b**

## roactionclient

Create ROS action client

### Description

Use the `roactionclient` to connect to an action server using a `SimpleActionClient` object and request the execution of action goals. You can get feedback on the execution process and cancel the goal at any time. The `SimpleActionClient` object encapsulates a simple action client and enables you to track a single goal at a time.

### Creation

#### Syntax

```
client = roactionclient(actionname)
client = roactionclient(actionname,actiontype)
[client,goalMsg] = roactionclient( ___ )

client = ros.SimpleActionClient(node,actionname)
client = ros.SimpleActionClient(node,actionname,actiontype)
```

#### Description

`client = roactionclient(actionname)` creates a client for the specified ROS `ActionName`. The client determines the action type automatically. If the action is not available, this function displays an error.

Use `roactionclient` to connect to an action server and request the execution of action goals. You can get feedback on the execution progress and cancel the goal at any time.

`client = roactionclient(actionname,actiontype)` creates an action client with the specified name and type (`ActionType`). If the action is not available, or the name and type do not match, the function displays an error.

`[client,goalMsg] = roactionclient( ___ )` returns a goal message to send the action client created using any of the arguments from the previous syntaxes. The `Goal` message is initialized with default values for that message.

If the `ActionFcn`, `FeedbackFcn`, and `ResultFcn` callbacks are defined, they are called when the goal is processing on the action server. All callbacks associated with a previously sent goal are disabled, but the previous goal is not canceled.

`client = ros.SimpleActionClient(node,actionname)` creates a client for the specified ROS action name. The `node` is the `Node` object that is connected to the ROS network. The client determines the action type automatically. If the action is not available, the function displays an error.

`client = ros.SimpleActionClient(node, actionname, actiontype)` creates an action client with the specified name and type. You can get the type of an action using `rosaction type actionname`.

## Properties

### ActionName — ROS action name

character vector

ROS action name, returned as a character vector. The action name must match one of the topics that `rosaction("list")` outputs.

### ActionType — Action type for a ROS action

string scalar | character vector

Action type for a ROS action, returned as a string scalar or character vector. You can get the action type of an action using `rosaction type <action_name>`. For more details, see `rosaction`.

### IsServerConnected — Indicates if client is connected to ROS action server

false (default) | true

Indicator of whether the client is connected to a ROS action server, returned as `false` or `true`. Use `waitForServer` to wait until the server is connected when setting up an action client.

### Goal — Tracked goal

ROS message

Tracked goal, returned as a ROS message. This message is the last goal message this client sent. The goal message depends on the action type.

### GoalState — Goal state

character vector

Goal state, returned as one of the following:

- 'pending' — Goal was received, but has not yet been accepted or rejected.
- 'active' — Goal was accepted and is running on the server.
- 'succeeded' — Goal executed successfully.
- 'preempted' — An action client canceled the goal before it finished executing.
- 'aborted' — The goal was aborted before it finished executing. The action server typically aborts a goal.
- 'rejected' — The goal was not accepted after being in the 'pending' state. The action server typically triggers this status.
- 'recalled' — A client canceled the goal while it was in the 'pending' state.
- 'lost' — An internal error occurred in the action client.

### ActivationFcn — Activation function

@(~) disp('Goal is active.')(default) | function handle

Activation function, returned as a function handle. This function executes when `GoalState` is set to 'active'. By default, the function displays 'Goal is active.'. You can set the function to `[]` to have the action client do nothing upon activation.

**FeedbackFcn — Feedback function**

```
@(~,msg) disp(['Feedback: ', showdetails(msg)]) (default) | function handle
```

Feedback function, returned as a function handle. This function executes when a new feedback message is received from the action server. By default, the function displays the details of the message. You can set the function to [] to have the action client not give any feedback.

**ResultFcn — Result function**

```
@(~,msg,s,~) disp(['Result with state ' s ': ', showdetails(msg)]) (default) | function handle
```

Result function, returned as a function handle. This function executes when the server finishes executing the goal and returns a result state and message. By default, the function displays the state and details of the message. You can set the function to [] to have the action client do nothing once the goal is completed.

**Object Functions**

cancelGoal	Cancel last goal sent by client
cancelAllGoals	Cancel all goals on action server
rosmessage	Create ROS messages
sendGoal	Send goal message to action server
sendGoalAndWait	Send goal message and wait for result
waitForServer	Wait for action server to start

**Examples****Setup a ROS Action Client and Execute an Action**

This example shows how to create a ROS action client and execute the action. Action types must be set up beforehand with an action server running.

You must have set up the '/fibonacci' action type. To run this action server, use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

Connect to a ROS network. You must be connected to a ROS network to gather information about what actions are available. Replace `ipaddress` with your network address.

```
ipaddress = '192.168.17.129';
rosinit(ipaddress,11311)
```

```
Initializing global node /matlab_global_node_01856 with NodeURI http://192.168.17.1:57693/
```

List actions available on the network. The only action set up on this network is the '/fibonacci' action.

```
rosaction list
/fibonacci
```

Create an action client. Specify the action name.

```
[actClient,goalMsg] = rosactionclient('/fibonacci');
```

Wait for the action client to connect to the server.

```
waitForServer(actClient);
```

The fibonacci action will calculate the fibonacci sequence for a given order specified in the goal message. The goal message was returned when creating the action client and can be modified to send goals to the ROS action server.

```
goalMsg.Order = 8
```

```
goalMsg =
  ROS FibonacciGoal message with properties:
    MessageType: 'actionlib_tutorials/FibonacciGoal'
    Order: 8
```

Use `showdetails` to show the contents of the message

Send the goal and wait for its completion. Specify a timeout of 10 seconds to complete the action.

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,10)
```

```
resultMsg =
  ROS FibonacciResult message with properties:
    MessageType: 'actionlib_tutorials/FibonacciResult'
    Sequence: [10x1 int32]
```

Use `showdetails` to show the contents of the message

```
resultState =
'succeeded'
```

```
showdetails(resultMsg)
  Sequence : [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Disconnect from the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_01856 with NodeURI http://192.168.17.1:57693/
```

### Send and Cancel ROS Action Goals

This example shows how to send and cancel goals for ROS actions. Action types must be setup beforehand with an action server running.

You must have set up the  `'/fibonacci'` action type. To run this action server, use the following command on the ROS system:

```
roslaunch actionlib_tutorials fibonacci_server
```

First, set up a ROS action client. Then, send a goal message with modified parameters. Finally, cancel your goal and all goals on the action server.

Connect to a ROS network with a specified IP address. Create a ROS action client connected to the ROS network using `roactionclient`. Specify the action name. Wait for the client to be connected to the server.

```
rosinit('192.168.17.129',11311)
```

```
Initializing global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

```
[actClient,goalMsg] = roactionclient('/fibonacci');  
waitForServer(actClient);
```

Send a goal message with modified parameters. Wait for the goal to finish executing.

```
goalMsg.Order = 4;  
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg =  
  ROS FibonacciResult message with properties:  
  
  MessageType: 'actionlib_tutorials/FibonacciResult'  
  Sequence: [6×1 int32]
```

Use `showdetails` to show the contents of the message

```
resultState =  
'succeeded'
```

```
showdetails(resultMsg)  
  
  Sequence : [0, 1, 1, 2, 3, 5]
```

Send a new goal message without waiting.

```
goalMsg.Order = 5;  
sendGoal(actClient,goalMsg)
```

Cancel the goal on the ROS action client, `actClient`.

```
cancelGoal(actClient)
```

Cancel all the goals on the action server that `actClient` is connected to.

```
cancelAllGoals(actClient)
```

Delete the action client.

```
delete(actClient)
```

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_59254 with NodeURI http://192.168.17.1:59729/
```

## See Also

`cancelGoal` | `roaction` | `rosmesssage` | `sendGoal` | `waitForServer`



**Topics**

“ROS Actions Overview”

“Move a Turtlebot Robot Using ROS Actions”

**External Websites**

ROS Actions

**Introduced in R2019b**

# ParameterTree

Access ROS parameter server

## Description

A `ParameterTree` object communicates with the ROS parameter server. The ROS parameter server can store strings, integers, doubles, Booleans, and cell arrays. The parameters are accessible globally over the ROS network. You can use these parameters to store static data such as configuration parameters.

To directly set, get, or access ROS parameters without creating a `ParameterTree` object, see `rosparam`.

The following ROS data types are supported as values of parameters. For each ROS data type, the corresponding MATLAB data type is also listed.

ROS Data Type	MATLAB Data Type
32-bit integer	<code>int32</code>
boolean	<code>logical</code>
double	<code>double</code>
string	character vector ( <code>char</code> )
list	cell array ( <code>cell</code> )
dictionary	structure ( <code>struct</code> )

## Creation

### Syntax

```
ptree = rosparam
```

```
ptree = ros.ParameterTree(node)
```

### Description

`ptree = rosparam` creates a parameter tree object, `ptree`. After `ptree` is created, the connection to the parameter server remains persistent until the object is deleted or the ROS master becomes unavailable.

`ptree = ros.ParameterTree(node)` returns a `ParameterTree` object to communicate with the ROS parameter server. The parameter tree attaches to the ROS node, `node`. To connect to the global node, specify `node` as `[]`.

## Properties

### AvailableParameters — List of parameter names on the server

cell array

This property is read-only.

List of parameter names on the server, specified as a cell array.

Example: {'/myParam'; '/robotSize'; '/hostname'}

Data Types: cell

## Object Functions

get	Get ROS parameter value
has	Check if ROS parameter name exists
search	Search ROS network for parameter names
set	Set value of ROS parameter or add new parameter
del	Delete a ROS parameter

## Examples

### Create ROS ParameterTree Object and Modify Parameters

Start the ROS master and create a ROS node.

```
master = ros.Core;
node = ros.Node('/test1');
```

Create the parameter tree object.

```
ptree = ros.ParameterTree(node);
```

Set multiple parameters.

```
set(ptree, 'DoubleParam', 1.0)
set(ptree, 'CharParam', 'test')
set(ptree, 'CellParam', {'test'}, {1,2});
```

View the available parameters.

```
parameters = ptree.AvailableParameters
```

```
parameters = 3x1 cell
    {'/CellParam' }
    {'/CharParam' }
    {'/DoubleParam'}
```

Get a parameter value.

```
data = get(ptree, 'CellParam')
```

```
data=1x2 cell array
    {1x1 cell}    {1x2 cell}
```

Search for a parameter name.

```
search(ptree, 'char')  
  
ans = 1x1 cell array  
    {'/CharParam'}
```

Delete the parameter tree and ROS node. Shut down the ROS master.

```
clear('ptree','node')  
clear('master')
```

### **Set A Dictionary Of Parameter Values**

Use structures to specify a dictionary of ROS parameters under a specific namespace.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:56297/.  
Initializing global node /matlab_global_node_59257 with NodeURI http://bat5110win64:56304/
```

Create a dictionary of parameter values. This dictionary contains the information relevant to an image. Display the structure to verify values.

```
image = imread('peppers.png');  
  
pval.ImageWidth = size(image,1);  
pval.ImageHeight = size(image,2);  
pval.ImageTitle = 'peppers.png';  
disp(pval)  
  
    ImageWidth: 384  
    ImageHeight: 512  
    ImageTitle: 'peppers.png'
```

Set the dictionary of values using the desired namespace.

```
rosparam('set', 'ImageParam', pval)
```

Get the parameters using the namespace. Verify the values.

```
pval2 = rosparam('get', 'ImageParam')  
  
pval2 = struct with fields:  
    ImageHeight: 512  
    ImageTitle: 'peppers.png'  
    ImageWidth: 384
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_59257 with NodeURI http://bat5110win64:56304/  
Shutting down ROS master on http://bat5110win64:56297/.
```

## **See Also**

del | get | has | rosparam | search | set

## **Topics**

“Access the ROS Parameter Server”

**Introduced in R2019b**

## rospublisher

Publish message on a topic

### Description

Use `rospublisher` to create a ROS publisher for sending messages via a ROS network. To create ROS messages, use `rosmessage`. Send these messages via the ROS publisher with the `send` function.

The `Publisher` object created by the function represents a publisher on the ROS network. The object publishes a specific message type on a given topic. When the `Publisher` object publishes a message to the topic, all subscribers to the topic receive this message. The same topic can have multiple publishers and subscribers.

The publisher gets the topic message type from the topic list on the ROS master. When the MATLAB global node publishes messages on that topic, ROS nodes that subscribe to that topic receive those messages. If the topic is not on the ROS master topic list, this function displays an error message. If the ROS master topic list already contains a matching topic, the ROS master adds the MATLAB global node to the list of publishers for that topic. To see a list of available topic names, at the MATLAB command prompt, type `rostopic list`.

You can create a `Publisher` object using the `rospublisher` function, or by calling `ros.Publisher`:

- `rospublisher` only works with the global node using `rosinit`. It does not require a node object handle as an argument.
- `ros.Publisher` works with additional nodes that are created using `ros.Node`. It requires a node object handle as the first argument.

### Creation

#### Syntax

```
pub = rospublisher(topicname)
pub = rospublisher(topicname,msgtype)
pub = rospublisher( ___,Name,Value)
[pub,msg] = rospublisher( ___ )

pub = ros.Publisher(node,topicname)
pub = ros.Publisher(node,topicname,type)
pub = ros.Publisher( ___, "IsLatching", value)
```

#### Description

`pub = rospublisher(topicname)` creates a publisher for a specific topic name and sets the `TopicName` property. The topic must already exist on the ROS master topic list with an established `MessageType`.

`pub = rospublisher(topicname,msgtype)` creates a publisher for a topic and adds that topic to the ROS master topic list. The inputs are set to the `TopicName` and `MessageType` properties of the publisher. If the topic already exists and `msgtype` differs from the topic type on the ROS master topic list, the function displays an error message.

`pub = rospublisher( ____,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments using any of the arguments from previous syntaxes. `Name` is the property name and `Value` is the corresponding value.

`[pub,msg] = rospublisher( ____ )` returns a message, `msg`, that you can send with the publisher, `pub`. The message is initialized with default values. You can also get the ROS message using the `rosmessage` function.

`pub = ros.Publisher(node,topicname)` creates a publisher for a topic with name, `topicname`. `node` is the `robotics.ros.Node` object handle that this publisher attaches to. If `node` is specified as `[]`, the publisher tries to attach to the global node.

`pub = ros.Publisher(node,topicname,type)` creates a publisher with specified message type, `type`. If the topic already exists, MATLAB checks the message type and displays an error if the input type differs. If the ROS master topic list already contains a matching topic, the ROS master adds the MATLAB global node to the list of publishers for that topic.

`pub = ros.Publisher( ____, "IsLatching", value)` specifies if the publisher is latching with a Boolean, `value`. If a publisher is latching, it saves the last sent message and sends it to any new subscribers. By default, `IsLatching` is enabled.

## Properties

### TopicName — Name of the published topic

string scalar | character vector

Name of the published topic, specified as a string scalar or character vector. If the topic does not exist, the object creates the topic using its associated message type.

This property is set at creating by the `TopicName` argument. The value cannot be changed after creation.

Example: `"/chatter"`

Data Types: char

### MessageType — Message type of published messages

string scalar | character vector

Message type of published messages, specified as a string scalar or character vector. This message type remains associated with the topic and must be used for new messages published.

This property is set at creation by the `MessageType` argument. The value cannot be changed after creation.

Example: `"std_msgs/String"`

Data Types: char

### IsLatching — Indicator of whether publisher is latching

true (default) | false

Indicator of whether publisher is latching, specified as `true` or `false`. A publisher that is latching saves the last sent message and resends it to any new subscribers.

This property is set at creating by the `IsLatching` argument. The value cannot be changed after creation.

Data Types: `logical`

### **NumSubscribers — Number of subscribers**

`integer`

Number of subscribers to the published topic, specified as an integer.

This property is set at creating by the `NumSubscribers` argument. The value cannot be changed after creation.

Data Types: `double`

## **Object Functions**

`send`            Publish ROS message to topic  
`rosmesssage`    Create ROS messages

## **Examples**

### **Create ROS Publisher and Send Data**

Start ROS master.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:57316/.
```

```
Initializing global node /matlab_global_node_01158 with NodeURI http://bat5110win64:57323/
```

Create publisher for the `'/chatter'` topic with the `'std_msgs/String'` message type.

```
chatpub = rospublisher('/chatter','std_msgs/String');
```

Create a message to send. Specify the `Data` property.

```
msg = rosmesssage(chatpub);  
msg.Data = 'test phrase';
```

Send the message via the publisher.

```
send(chatpub,msg);
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_01158 with NodeURI http://bat5110win64:57323/
```

```
Shutting down ROS master on http://bat5110win64:57316/.
```



## Create ROS Publisher with rospublisher and View Properties

Create a ROS publisher and view the associated properties for the `rospublisher` object. Add a subscriber and view the updated properties.

Start ROS master.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:56335/.
```

```
Initializing global node /matlab_global_node_57170 with NodeURI http://bat5110win64:56339/
```

Create a publisher and view its properties.

```
pub = rospublisher('/chatter', 'std_msgs/String');
```

```
topic = pub.TopicName
```

```
topic =  
'/chatter'
```

```
subCount = pub.NumSubscribers
```

```
subCount = 0
```

Subscribe to the publisher topic and view the changes in the `NumSubscribers` property.

```
sub = rossubscriber('/chatter');  
pause(1)
```

```
subCount = pub.NumSubscribers
```

```
subCount = 1
```

Shut down the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_57170 with NodeURI http://bat5110win64:56339/  
Shutting down ROS master on http://bat5110win64:56335/.
```

## Publish Data Without ROS Publisher

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:49869/.
```

```
Initializing global node /matlab_global_node_52080 with NodeURI http://bat5110win64:49877/
```

Create a message to send. Specify the `Data` property.

```
msg = rosmessage('std_msgs/String');  
msg.Data = 'test phrase';
```

Send message via the `'/chatter'` topic.

```
rospublisher('/ chatter',msg)
```

```
ans =
```

```
 []
```

Shutdown the ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_52080 with NodeURI http://bat5110win64:49877/  
Shutting down ROS master on http://bat5110win64:49869/.
```

### **Use ROS Publisher Object**

Create a Publisher object using the class constructor.

Start the ROS master.

```
master = ros.Core;
```

Create a ROS node, which connects to the master.

```
node = ros.Node('/ test1');
```

Create a publisher and send string data. The publisher attaches to the node object in the first argument.

```
pub = ros.Publisher(node, '/ robotname', 'std_msgs/String');  
msg = rosmessage('std_msgs/String');  
msg.Data = 'robot1';  
send(pub,msg);
```

Clear the publisher and ROS node. Shut down the ROS master.

```
clear('pub','node')  
clear('master')
```

### **See Also**

#### **Functions**

rosmessage | send

#### **Topics**

“Exchange Data with ROS Publishers and Subscribers”

**Introduced in R2019b**

# ros2publisher

Publish messages on a topic

## Description

Use the `ros2publisher` object to publish messages on a topic. When messages are published on that topic, ROS 2 nodes that subscribe to that topic receive those messages directly.

## Creation

### Syntax

```
pub = ros2publisher(node, topic)
pub = ros2publisher(node, topic, type)
pub = ros2publisher( ____, Name, Value)
[pub, msg] = rospublisher( ____ )
```

### Description

`pub = ros2publisher(node, topic)` creates a publisher, `pub`, for a topic with name `topic` that already exists on the ROS 2 network. `node` is the `ros2node` object handle to which the publisher should attach. The publisher gets the topic message type from the network topic list.

---

**Note** The topic must be on the network topic list.

---

`pub = ros2publisher(node, topic, type)` creates a publisher for a topic and adds that topic to the network topic list. If the topic list already contains a matching topic, `pub` will be added to the list of publishers for that topic.

`pub = ros2publisher( ____, Name, Value)` specifies additional options using one or more name-value pair arguments. Specify name-value pair arguments after all other input arguments.

`[pub, msg] = rospublisher( ____ )` returns a message, `msg`, that you can send with the publisher, `pub`. The message is initialized with default values. You can also get the ROS message using the `ros2message` function.

### Input Arguments

#### **node** — ROS 2 node

node structure

A `ros2node` object on the network.

#### **topic** — Name of the published topic

string scalar | character vector

Name of the published topic, specified as a string scalar or character vector. If the topic does not exist, the object creates the topic using its associated message type.

This property is set at creating by the `TopicName` argument. The value cannot be changed after creation.

Example: `"/chatter"`

Data Types: `char`

### **type — Message type of published messages**

`string scalar | character vector`

Message type of published messages, specified as a string scalar or character vector. This message type remains associated with the topic and must be used for new messages published.

This property is set at creation by the `MessageType` argument. The value cannot be changed after creation.

Example: `"std_msgs/String"`

Data Types: `char`

### **Name-Value Pair Arguments**

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### **History — Mode of storing messages in the queue**

`"keeplast" (default) | "keepall"`

Determines the mode of storing messages in the queue. The queued messages will be sent to late-joining subscribers. If the queue fills with messages waiting to be processed, then old messages will be dropped to make room for new. If set to `"keeplast"`, the queue stores the number of messages set by the `Depth` property. If set to `"keepall"`, the queue stores all messages up to the resource limits of MATLAB.

Data Types: `double`

### **Depth — Size of the message queue**

`positive integer`

Number of messages stored in the message queue when `History` is set to `"keeplast"`.

Example: `42`

Data Types: `double`

### **Reliability — Delivery guarantee of messages**

`"reliable" (default) | "besteffort"`

Affects the guarantee of message delivery. If `"reliable"`, then delivery is guaranteed, but may retry multiple times. If `"besteffort"`, then delivery is attempt, but retried.

Example: `"reliable"`

Data Types: `char | string`

**Durability — Persistence of messages**`"volatile" (default) | "transientlocal"`

Affects persistence of messages in publishers, which allows late-joining subscribers to receive the number of old messages specified by `Depth`. If `"volatile"`, then messages do not persist. If `"transientlocal"`, then publisher will persist most recent messages.

Example: `"volatile"`

Data Types: `char | string`

**Properties****TopicName — Name of the published topic**`string scalar | character vector`

Name of the published topic, specified as a string scalar or character vector. If the topic does not exist, the object creates the topic using its associated message type.

This property is set at creating by the `TopicName` argument. The value cannot be changed after creation.

Example: `"/chatter"`

Data Types: `char`

**MessageType — Message type of published messages**`string scalar | character vector`

Message type of published messages, specified as a string scalar or character vector. This message type remains associated with the topic and must be used for new messages published.

This property is set at creation by the `MessageType` argument. The value cannot be changed after creation.

Example: `"std_msgs/String"`

Data Types: `char`

**History — Message queue mode**`"keeplast" (default) | "keepall"`

This property is read-only.

Determines the mode of storing messages in the queue. The queued messages will be sent to late-joining subscribers. If the queue fills with messages waiting to be processed, then old messages will be dropped to make room for new. When set to `"keeplast"`, the queue stores the number of messages set by the `Depth` property. Otherwise, when set to `"keepall"`, the queue stores all messages up to the resource limits of MATLAB.

Example: `"keeplast"`

Data Types: `char | string`

**Depth — Size of the message queue**`positive integer`

This property is read-only.

Number of messages stored in the message queue when History is set to "keeplast".

Example: 42

Data Types: double

### **Reliability – Delivery guarantee of messages**

"reliable" (default) | "besteffort"

This property is read-only.

Affects the guarantee of message delivery. If "reliable", then delivery is guaranteed, but may retry multiple times. If "besteffort", then delivery is attempt, but retried.

Example: "reliable"

Data Types: char | string

### **Durability – Persistence of messages**

"volatile" (default) | "transientlocal"

This property is read-only.

Affects persistence of messages in publishers, which allows late-joining subscribers to receive the number of old messages specified by Depth. If "volatile", then messages do not persist. If "transientlocal", then publisher will persist most recent messages.

Example: "volatile"

Data Types: char | string

## **Object Functions**

ros2message Create ROS 2 message structures  
send Publish ROS 2 message to topic

## **Examples**

### **Create an Empty ROS 2 Message**

Create a ROS 2 node.

```
node = ros2node("/node_1");
```

Create publisher and message.

```
chatPub = ros2publisher(node, "/chatter", "std_msgs/String")
```

```
chatPub =  
  ros2publisher with properties:  
    TopicName: '/chatter'  
    MessageType: 'std_msgs/String'  
    History: 'keeplast'  
    Depth: 10  
    Reliability: 'reliable'  
    Durability: 'volatile'
```

```
msg = ros2message(chatPub)
```

```
msg = struct with fields:  
  data: ''
```

## See Also

[ros2message](#) | [ros2subscriber](#) | [send](#)

## Topics

“Manage Quality of Service Policies in ROS 2”

## Introduced in R2019b

## ros2subscriber

Subscribe to messages on a topic

### Description

Use the `ros2subscriber` to receive messages on a topic. When ROS 2 nodes publish messages on that topic, MATLAB will receive those message through this subscriber.

### Creation

#### Syntax

```
sub = ros2subscriber(node,topic)
sub = ros2subscriber(node,topic,type)
sub = ros2subscriber(node,topic,callback)
sub = ros2subscriber(node,topic,type,callback)
sub = ros2subscriber( ___,Name,Value)
```

#### Description

`sub = ros2subscriber(node,topic)` creates a subscriber, `sub`, for a topic with name `topic` that already exists on the ROS 2 network. `node` is the `ros2node` object to which this subscriber attaches. The subscriber gets the topic message type from the network topic list.

---

**Note** The topic must be on the network topic list.

---

`sub = ros2subscriber(node,topic,type)` creates a subscriber for a topic and adds that topic to the network topic list. If the topic list already contains a matching topic, `sub` will be added to the list of subscribers for that topic. The `type` must be the same as the topic. Use this syntax to avoid errors when it is possible for the subscriber to subscribe to a topic before a topic has been added to the network.

`sub = ros2subscriber(node,topic,callback)` specifies a callback function, `callback`, and optional data, to run when the subscriber object handle receives a topic message. Use this syntax if action needs to be taken on every message, while not blocking code execution. `callback` can be a single function handle or a cell array. The first element of the cell array needs to be a function handle or a string containing the name of a function. The remaining elements of the cell array can be arbitrary user data that will be passed to the callback function.

---

**Note** The subscriber callback function uses a single input argument, the received message object, `msg`. The function header for the callback is as follows:

```
function subCallback(src, msg)
```

You pass additional parameters to the callback function by including both the callback function and the parameters as elements of a cell array when setting the callback.

---



`sub = ros2subscriber(node, topic, type, callback)` specifies a callback function `callback`, and subscribes to a topic that has the specified name `topic` and message type `type`.

`sub = ros2subscriber( ____, Name, Value)` specifies additional options using one or more name-value pair arguments. Specify name-value pair arguments after all other input arguments.

### Input Arguments

#### **node** — ROS 2 node

`ros2node` structure

A `ros2node` object on the network.

#### **topic** — Name of the published topic

string scalar | char array

Name of the published topic, specified as a string scalar or character vector. If the topic does not exist, the object creates the topic based on the associated message type.

Example: `"/chatter"`

#### **type** — Subscribed message type

string scalar | char array

This property is read-only.

The message type of subscribed messages.

Example: `"std_msgs/String"`

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

#### **History** — Mode of storing messages in the queue

`"keeplast"` (default) | `"keepall"`

Determines the mode of storing messages in the queue. If the queue fills with messages waiting to be processed, then old messages will be dropped to make room for new. If set to `"keeplast"`, the queue stores the number of messages set by the `Depth` property. If set to `"keepall"`, the queue stores all messages up to the MATLAB resource limits.

#### **Depth** — Size of the message queue

non-negative scalar integer (default)

Number of messages stored in the message queue when `History` is set to `"keeplast"`.

Example: 42

Data Types: double

#### **Reliability** — Delivery guarantee of messages

`"reliable"` (default) | `"besteffort"`

Requirement on the guarantee of message delivery. If `"reliable"`, then delivery is guaranteed, but may retry multiple times. If `"besteffort"`, then attempt delivery and do not retry.

Example: "reliable"

Data Types: char | string

### **Durability – Persistence of messages**

"volatile" (default) | "transientlocal"

Requirement on the persistence of messages in connected publishers, which allows late-joining subscribers to receive the number of old messages specified by `Depth`. If "volatile", then message persistence is not required and no messages are requested when the subscriber joins the network. If "transientlocal", then the subscriber will require publishers to persist messages, and will request the number of messages specified by `Depth`.

Example: "volatile"

Data Types: char | string

## **Properties**

### **TopicName – Name of the published topic**

string scalar | character vector

Name of the published topic, specified as a string scalar or character vector. If the topic does not exist, the object creates the topic based on the associated message type.

Example: "/chatter"

Data Types: char

### **MessageType – Subscribed message type**

string scalar | character vector

This property is read-only.

The message type of subscribed messages.

Example: "std\_msgs/String"

Data Types: char | string

### **LatestMessage – Latest received message**

Message object handle

This property is read-only.

The most recently received ROS 2 message, specified as a Message object handle, received.

### **NewMessageFcn – Subscriber callback function**

function handle

This property is read-only.

Callback function for subscriber callbacks.

---

**Note** The subscriber callback function uses a single input argument, the received message object, `msg`. The function header for the callback is as follows:

```
function subCallback(msg)
```

You pass additional parameters to the callback function by including both the callback function and the parameters as elements of a cell array when setting the callback.

---

### **History — Message queue mode**

"keeplast" (default) | "keepall"

This property is read-only.

Determines the mode of storing messages in the queue. If the queue fills with messages waiting to be processed, then old messages will be dropped to make room for new. When set to "keeplast", the queue stores the number of messages set by the Depth property. Otherwise, when set to "keepall", the queue stores all messages up to the MATLAB resource limits.

Example: "keeplast"

Data Types: char | string

### **Depth — Size of the message queue**

non-negative scalar integer

This property is read-only.

Number of messages stored in the message queue when History is set to "keeplast".

Example: 42

Data Types: double

### **Reliability — Delivery guarantee of messages**

"reliable" (default) | "besteffort"

This property is read-only.

Requirement on the guarantee of message delivery. If "reliable", then delivery is guaranteed, but may retry multiple times. If "besteffort", then attempt delivery and do not retry.

Example: "reliable"

Data Types: char | string

### **Durability — Persistence of messages**

"volatile" (default) | "transientlocal"

This property is read-only.

Requirement on the persistence of messages in connected publishers, which allows late-joining subscribers to receive the number of old messages specified by Depth. If "volatile", then message persistence is not required and no messages are requested when the subscriber joins the network. If "transientlocal", then the subscriber will require publishers to persist messages, and will request the number of messages specified by Depth.

Example: "volatile"

Data Types: char | string

## Object Functions

receive            Wait for new message  
 ros2message    Create ROS 2 message structures

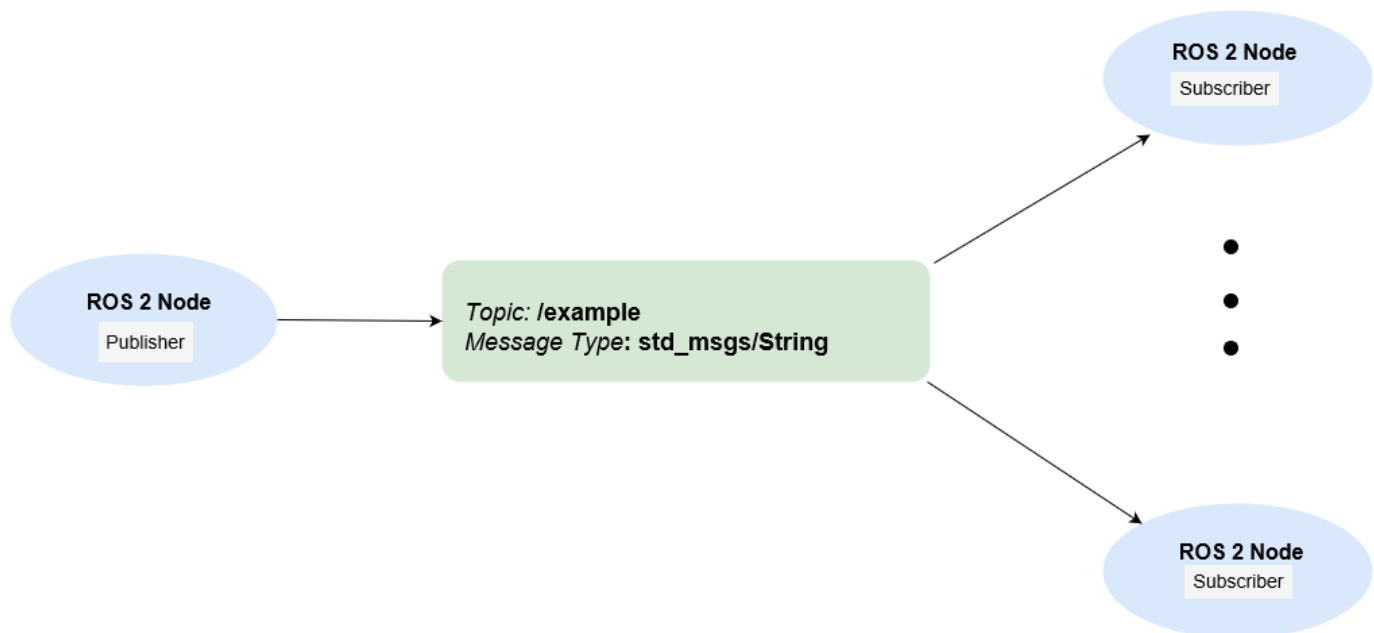
## Examples

### Exchange Data with ROS 2 Publishers and Subscribers

The primary mechanism for ROS 2 nodes to exchange data is to send and receive *messages*. Messages are transmitted on a *topic* and each topic has a unique name in the ROS 2 network. If a node wants to share information, it must use a *publisher* to send data to a topic. A node that wants to receive that information must use a *subscriber* for that same topic. Besides its unique name, each topic also has a *message type*, which determines the type of messages that are allowed to be transmitted in the specific topic.

This publisher-subscriber communication has the following characteristics:

- Topics are used for many-to-many communication. Multiple publishers can send messages to the same topic and multiple subscribers can receive them.
- Publisher and subscribers are decoupled through topics and can be created and destroyed in any order. A message can be published to a topic even if there are no active subscribers.



This example shows how to publish and subscribe to topics in a ROS 2 network. It also shows how to:

- Wait until a new message is received, or
- Use callbacks to process new messages in the background

Prerequisites: “Get Started with ROS 2”, “Connect to a ROS 2 Network”

## Subscribe and Wait for Messages

Create a sample ROS 2 network with several publishers and subscribers.

```
exampleHelperROS2CreateSampleNetwork
```

Use `ros2 topic list` to see which topics are available.

```
ros2 topic list
/parameter_events
/pose
/scan
```

Assume you want to subscribe to the `/scan` topic. Use `ros2subscriber` to subscribe to the `/scan` topic. Specify the name of the node with the subscriber. If the topic already exists in the ROS 2 network, `ros2subscriber` detects its message type automatically, so you do not need to specify it.

```
detectNode = ros2node("/detection");
pause(2)
laserSub = ros2subscriber(detectNode, "/scan");
pause(2)
```

Use `receive` to wait for a new message. Specify a timeout of 10 seconds. The output `scanData` contains the received message data.

```
scanData = receive(laserSub,10);
```

You can now remove the subscriber `laserSub` and the node associated to it.

```
clear laserSub
clear detectNode
```

## Subscribe Using Callback Functions

Instead of using `receive` to get data, you can specify a function to be called when a new message is received. This allows other MATLAB code to execute while the subscriber is waiting for new messages. Callbacks are essential if you want to use multiple subscribers.

Subscribe to the `/pose` topic, using the callback function `exampleHelperROS2PoseCallback`, which takes a received message as the input. One way of sharing data between your main workspace and the callback function is to use global variables. Define two global variables `pos` and `orient`.

```
controlNode = ros2node("/base_station");
poseSub = ros2subscriber(controlNode, "/pose", @exampleHelperROS2PoseCallback);
global pos
global orient
```

The global variables `pos` and `orient` are assigned in the `exampleHelperROS2PoseCallback` function when new message data is received on the `/pose` topic.

```
function exampleHelperROS2PoseCallback(message)
    % Declare global variables to store position and orientation
    global pos
    global orient

    % Extract position and orientation from the ROS message and assign the
    % data to the global variables.
```

```
    pos = [message.linear.x message.linear.y message.linear.z];
    orient = [message.angular.x message.angular.y message.angular.z];
end
```

Wait a moment for the network to publish another `/pose` message. Display the updated values.

```
pause(3)
disp(pos)

    0.0416    -0.0499    -0.0038

disp(orient)

   -0.0076   -0.0039    0.0270
```

If you type in `pos` and `orient` a few times in the command line you can see that the values are continuously updated.

Stop the pose subscriber by clearing the subscriber variable

```
clear poseSub
clear controlNode
```

*Note:* There are other ways to extract information from callback functions besides using globals. For example, you can pass a handle object as additional argument to the callback function. See the “Callback Definition” (MATLAB) documentation for more information about defining callback functions.

### **Publish Messages**

Create a publisher that sends ROS 2 string messages to the `/chatter` topic.

```
chatterPub = ros2publisher(node_1, "/chatter", "std_msgs/String");
```

Create and populate a ROS 2 message to send to the `/chatter` topic.

```
chatterMsg = ros2message(chatterPub);
chatterMsg.data = 'hello world';
```

Use `ros2 topic list` to verify that the `/chatter` topic is available in the ROS 2 network.

```
ros2 topic list

/chatter
/parameter_events
/pose
/scan
```

Define a subscriber for the `/chatter` topic. `exampleHelperROS2ChatterCallback` is called when a new message is received, and displays the string content in the message.

```
chatterSub = ros2subscriber(node_2, "/chatter", @exampleHelperROS2ChatterCallback)
```

```
chatterSub =
  ros2subscriber with properties:
    TopicName: '/chatter'
    LatestMessage: []
    MessageType: 'std_msgs/String'
```

```
NewMessageFcn: @exampleHelperROS2ChatterCallback
  History: 'keeplast'
  Depth: 10
  Reliability: 'reliable'
  Durability: 'volatile'
```

Publish a message to the `/chatter` topic. Observe that the string is displayed by the subscriber callback.

```
send(chatterPub, chatterMsg)
pause(3)

ans =
'hello world'
```

The `exampleHelperROS2ChatterCallback` function was called when the subscriber received the string message.

### Disconnect From ROS 2 Network

Remove the sample nodes, publishers and subscribers from the ROS 2 network. Also clear the global variables `pos` and `orient`

```
clear global pos orient
clear
```

### Next Steps

- “Work with Basic ROS 2 Messages”
- “ROS 2 Custom Message Support”

### See Also

[ros2node](#) | [ros2publisher](#)

### Topics

“Manage Quality of Service Policies in ROS 2”  
“ROS Custom Message Support”

### Introduced in R2019b

## rostrate

Execute loop at fixed frequency

### Description

The `rostrate` object uses the `rateControl` superclass to inherit most of its properties and methods. The main difference is that `rateControl` uses the ROS node as a source for time information. Therefore, it can use the ROS simulation or wall clock time (see the `IsSimulationTime` property).

If `rosinit` creates a ROS master in MATLAB, the global node uses wall clock time.

The performance of the `rostrate` object and the ability to maintain the `DesiredRate` value depends on the publishing of the clock information in ROS.

---

**Tip** The scheduling resolution of your operating system and the level of other system activity can affect rate execution accuracy. As a result, accurate rate timing is limited to 100 Hz for execution of MATLAB code. To improve performance and execution speeds, use code generation.

---

### Creation

#### Syntax

```
rate = rostrate(desiredRate)
rate = ros.Rate(node,desiredRate)
```

#### Description

`rate = rostrate(desiredRate)` creates a `Rate` object, which enables you to execute a loop at a fixed frequency, `DesiredRate`. The time source is linked to the time source of the global ROS node, which requires you to connect MATLAB to a ROS network using `rosinit`.

`rate = ros.Rate(node,desiredRate)` creates a `Rate` object that operates loops at a fixed rate based on the time source linked to the specified ROS node, `node`.

### Properties

#### **DesiredRate — Desired execution rate**

scalar

Desired execution rate of loop, specified as a scalar in hertz. When using `waitfor`, the loop operates every `DesiredRate` seconds, unless the loop takes longer. It then begins the next loop based on the specified `OverRunAction`.

#### **DesiredPeriod — Desired time period between executions**

scalar



Desired time period between executions, specified as a scalar in seconds. This property is equal to the inverse of `DesiredRate`.

**TotalElapsedTime** — Elapsed time since construction or reset

scalar

Elapsed time since construction or reset, specified as a scalar in seconds.

**LastPeriod** — Elapsed time between last two calls to `waitfor`

NaN (default) | scalar

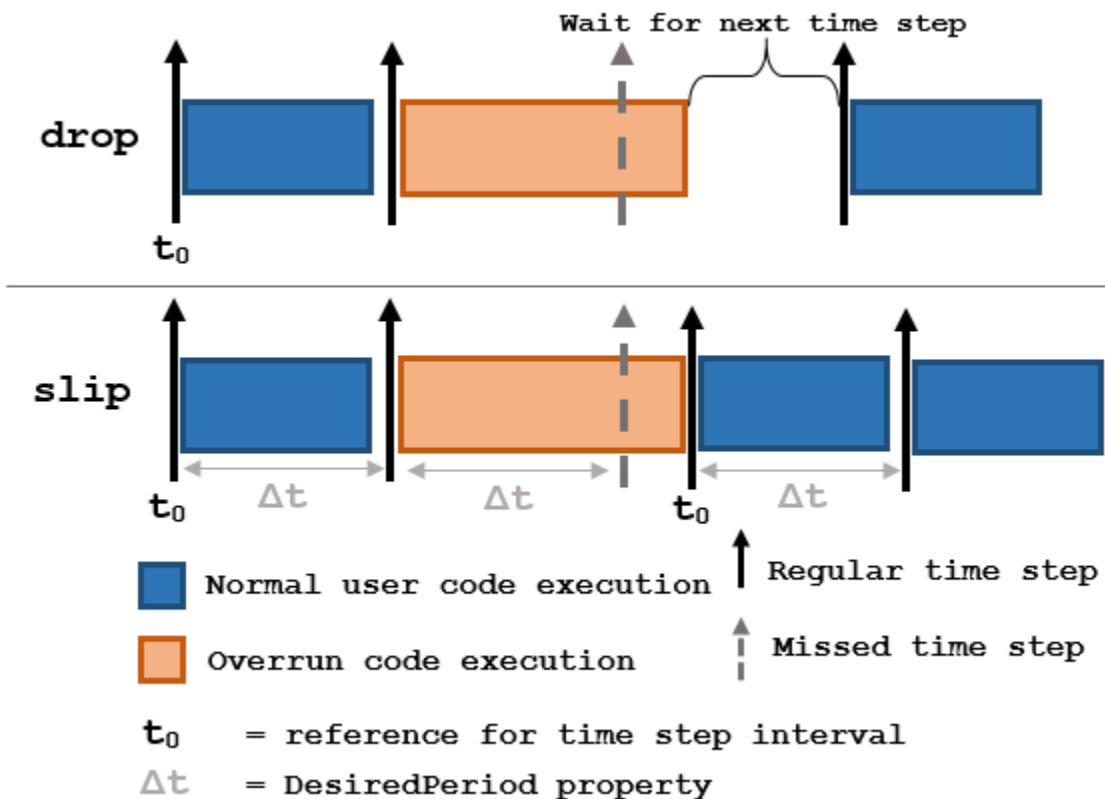
Elapsed time between last two calls to `waitfor`, specified as a scalar. By default, `LastPeriod` is set to NaN until `waitfor` is called for the first time. After the first call, `LastPeriod` equals `TotalElapsedTime`.

**OverrunAction** — Method for handling overruns

'slip' (default) | 'drop'

Method for handling overruns, specified as one of these character vectors:

- 'drop' — waits until the next time interval equal to a multiple of `DesiredPeriod`
- 'slip' — immediately executes the loop again



Each code section calls `waitfor` at the end of execution.

**IsSimulationTime** — Indicator if simulation or wall clock time is used

true | false

Indicator if simulation or wall clock time is used, returned as `true` or `false`. If `true`, the `Rate` object is using the ROS simulation time to regulate the rate of loop execution.

## Object Functions

`waitfor`    Pause code execution to achieve desired execution rate  
`statistics`    Statistics of past execution periods  
`reset`        Reset Rate object

## Examples

### Run Loop At Fixed Rate Using `rosrate`

Initialize the ROS master and node.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:57025/.  
Initializing global node /matlab_global_node_04859 with NodeURI http://bat5110win64:57029/
```

Create a rate object that runs at 1 Hz.

```
r = rosrate(1);
```

Start loop that prints iteration and time elapsed. Use `waitfor` to pause the loop until the next time interval. Reset `r` prior to the loop execution. Notice that each iteration executes at a 1-second interval.

```
reset(r)  
for i = 1:10  
    time = r.TotalElapsedTime;  
    fprintf('Iteration: %d - Time Elapsed: %f\n',i,time)  
    waitfor(r);  
end
```

```
Iteration: 1 - Time Elapsed: 0.011332  
Iteration: 2 - Time Elapsed: 1.007055  
Iteration: 3 - Time Elapsed: 2.005964  
Iteration: 4 - Time Elapsed: 3.005320  
Iteration: 5 - Time Elapsed: 4.004747  
Iteration: 6 - Time Elapsed: 5.012495  
Iteration: 7 - Time Elapsed: 6.000428  
Iteration: 8 - Time Elapsed: 7.000792  
Iteration: 9 - Time Elapsed: 8.000758  
Iteration: 10 - Time Elapsed: 9.000787
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_04859 with NodeURI http://bat5110win64:57029/  
Shutting down ROS master on http://bat5110win64:57025/.
```

## Run Loop At Fixed Rate Using ROS Time

Initialize the ROS master and node.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:56762/.
```

```
Initializing global node /matlab_global_node_99076 with NodeURI http://bat5110win64:56766/
```

```
node = robotics.ros.Node('/testTime');
```

```
Using Master URI http://localhost:56762 from the global node to connect to the ROS master.
```

Create a `ros.Rate` object running at 20 Hz.

```
r = robotics.ros.Rate(node,20);
```

Reset the object to restart the timer and run the loop for 30 iterations. Insert code you want to run in the loop before calling `waitfor`.

```
reset(r)
for i = 1:30
    % User code goes here.
    waitfor(r);
end
```

Shut down ROS node.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_99076 with NodeURI http://bat5110win64:56766/
```

```
Shutting down ROS master on http://bat5110win64:56762/.
```

## See Also

[rateControl](#) | [waitfor](#)

## Topics

“Execute Code Based on ROS Time”

**Introduced in R2019b**

## rossubscriber

Subscribe to messages on a topic

### Description

Use `rossubscriber` to create a ROS subscriber for receiving messages on the ROS network. To send messages, use `rospublisher`. To wait for a new ROS message, use the `receive` function with your created subscriber.

The `Subscriber` object created by the `rossubscriber` function represents a subscriber on the ROS network. The `Subscriber` object subscribes to an available topic or to a topic that it creates. This topic has an associated message type. Publishers can send messages over the network that the `Subscriber` object receives.

You can create a `Subscriber` object by using the `rossubscriber` function, or by calling `ros.Subscriber`:

- `rossubscriber` works only with the global node using `rosinit`. It does not require a node object handle as an argument.
- `ros.Subscriber` works with additional nodes that are created using `ros.Node`. It requires a node object handle as the first argument.

### Creation

#### Syntax

```
sub = rossubscriber(topicname)
sub = rossubscriber(topicname,msgtype)
sub = rossubscriber(topicname,callback)
sub = rossubscriber(topicname, msgtype,callback)
sub = rossubscriber( ____,Name,Value)

sub = ros.Subscriber(node,topicname)
sub = ros.Subscriber(node,topicname,msgtype)
sub = ros.Subscriber(node,topicname,callback)
sub = ros.Subscriber(node,topicname,type,callback)
sub = ros.Subscriber( ____, "BufferSize",value)
```

#### Description

`sub = rossubscriber(topicname)` subscribes to a topic with the given `TopicName`. The topic must already exist on the ROS master topic list with an established message type. When ROS nodes publish messages on that topic, MATLAB receives those messages through this subscriber.

`sub = rossubscriber(topicname,msgtype)` subscribes to a topic that has the specified name, `TopicName`, and type, `MessageType`. If the topic list on the ROS master does not include a topic with that specified name and type, it is added to the topic list. Use this syntax to avoid errors when subscribing to a topic before a publisher has added the topic to the topic list on the ROS master.

`sub = rossubscriber(topicname, callback)` specifies a callback function, `callback`, that runs when the subscriber object handle receives a topic message. Use this syntax to avoid the blocking receive function. The `callback` function can be a single function handle or a cell array. The first element of the cell array must be a function handle or a string containing the name of a function. The remaining elements of the cell array can be arbitrary user data that is passed to the callback function.

`sub = rossubscriber(topicname, msgtype, callback)` specifies a callback function and subscribes to a topic that has the specified name, `TopicName`, and type, `MessageType`.

`sub = rossubscriber( ___, Name, Value)` provides additional options specified by one or more `Name, Value` pair arguments using any of the arguments from previous syntaxes. `Name` is the property name and `Value` is the corresponding value.

`sub = ros.Subscriber(node, topicname)` subscribes to a topic with name, `TopicName`. The `node` is the `robotics.ros.Node` object handle that this publisher attaches to.

`sub = ros.Subscriber(node, topicname, msgtype)` specifies the message type, `MessageType`, of the topic. If a topic with the same name exists with a different message type, MATLAB creates a new topic with the given message type.

`sub = ros.Subscriber(node, topicname, callback)` specifies a callback function, and optional data, to run when the subscriber object receives a topic message. See `NewMessageFcn` for more information about the callback function.

`sub = ros.Subscriber(node, topicname, type, callback)` specifies the topic name, message type, and callback function for the subscriber.

`sub = ros.Subscriber( ___, "BufferSize", value)` specifies the queue size in `BufferSize` for incoming messages. You can use any combination of previous inputs with this syntax.

## Properties

### TopicName — Name of the subscribed topic

string scalar | character vector

This property is read-only.

Name of the subscribed topic, specified as a string scalar or character vector. If the topic does not exist, the object creates the topic using its associated message type.

Example: `"/chatter"`

Data Types: `char` | `string`

### MessageType — Message type of subscribed messages

string scalar | character vector

This property is read-only.

Message type of subscribed messages, specified as a string scalar or character vector. This message type remains associated with the topic.

Example: `"std_msgs/String"`

Data Types: `char` | `string`

**LatestMessage — Latest message sent to the topic**

Message object

Latest message sent to the topic, specified as a `Message` object. The `Message` object is specific to the given `MessageType`. If the subscriber has not received a message, then the `Message` object is empty.

**BufferSize — Buffer size**

1 (default) | scalar

Buffer size of the incoming message queue, specified as the comma-separated pair consisting of "BufferSize" and a scalar. If messages arrive faster than your callback can process them, they are deleted once the incoming queue is full.

**NewMessageFcn — Callback property**

function handle | cell array

Callback property, specified as a function handle or cell array. In the first element of the cell array, specify either a function handle or a string representing a function name. In subsequent elements, specify user data.

The subscriber callback function requires at least two input arguments. The first argument, `src`, is the associated subscriber object. The second argument, `msg`, is the received message object. The function header for the callback is:

```
function subCallback(src,msg)
```

Specify the `NewMessageFcn` property as:

```
sub.NewMessageFcn = @subCallback;
```

When setting the callback, you pass additional parameters to the callback function by including both the callback function and the parameters as elements of a cell array. The function header for the callback is:

```
function subCallback(src,msg,userData)
```

Specify the `NewMessageFcn` property as:

```
sub.NewMessageFcn = {@subCallback,userData};
```

**Object Functions**

`receive`      Wait for new ROS message  
`rosmessage`   Create ROS messages

**Examples****Create A Subscriber and Get Data From ROS**

Connect to a ROS network. Set up a sample ROS network. The `'/scan'` topic is being published on the network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:55964/.
Initializing global node /matlab_global_node_31941 with NodeURI http://bat5110win64:56008/
```

```
exampleHelperROSCreateSampleNetwork
```

Create a subscriber for the '/scan' topic. Wait for the subscriber to register with the master.

```
sub = rossubscriber('/scan');
pause(1);
```

Receive data from the subscriber as a ROS message. Specify a 10-second timeout.

```
msg2 = receive(sub,10)
```

```
msg2 =
  ROS LaserScan message with properties:

      MessageType: 'sensor_msgs/LaserScan'
      Header: [1x1 Header]
      AngleMin: -0.5216
      AngleMax: 0.5243
      AngleIncrement: 0.0016
      TimeIncrement: 0
      ScanTime: 0.0330
      RangeMin: 0.4500
      RangeMax: 10
      Ranges: [640x1 single]
      Intensities: [0x1 single]
```

Use showdetails to show the contents of the message

Shutdown the timers used by sample network.

```
exampleHelperROSShutDownSampleNetwork
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_31941 with NodeURI http://bat5110win64:56008/
Shutting down ROS master on http://bat5110win64:55964/.
```

## Create A Subscriber That Uses A Callback Function

You can trigger callback functions when subscribers receive messages. Specify the callback when you create it or use the `NewMessageFcn` property.

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:50982/.
Initializing global node /matlab_global_node_65042 with NodeURI http://bat5110win64:50986/
```

Setup a publisher to publish a message to the '/ chatter' topic. This topic is used to trigger the subscriber callback. Specify the Data property of the message. Wait 1 second to allow the publisher to register with the network.

```
pub = rospublisher('/ chatter', 'std_msgs/String');  
msg = rosmessage(pub);  
msg.Data = 'hello world';  
pause(1)
```

Set up a subscriber with a specified callback function. The exampleHelperROSChatterCallback function displays the Data inside the received message.

```
sub = rossubscriber('/ chatter', @exampleHelperROSChatterCallback);  
pause(1)
```

Send the message via the publisher. The subscriber should execute the callback to display the new message. Wait for the message to be received.

```
send(pub, msg);  
pause(1)
```

```
ans =  
'hello world'
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_65042 with NodeURI http://bat5110win64:50986/  
Shutting down ROS master on http://bat5110win64:50982/.
```

### **Use ROS Subscriber Object**

Use a ROS Subscriber object to receive messages over the ROS network.

Start the ROS master and node.

```
master = ros.Core;  
node = ros.Node('/ test');
```

Create a publisher and subscriber to send and receive a message over the ROS network.

```
pub = ros.Publisher(node, '/ chatter', 'std_msgs/String');  
pause(1)  
sub = ros.Subscriber(node, '/ chatter', 'std_msgs/String');
```

Send a message over the network.

```
msg = rosmessage('std_msgs/String');  
msg.Data = 'hello world';  
send(pub, msg)
```

View the message data using the LatestMessage property of the Subscriber object.

```
pause(1)  
sub.LatestMessage
```



```
ans =  
  ROS String message with properties:  
  
    MessageType: 'std_msgs/String'  
    Data: 'hello world'  
  
  Use showdetails to show the contents of the message
```

Clear the publisher, subscriber, and ROS node. Shut down the ROS master.

```
clear('pub', 'sub', 'node')  
clear('master')
```

## See Also

[receive](#) | [rosmesssage](#) | [rospublisher](#)

## Topics

“Exchange Data with ROS Publishers and Subscribers”

**Introduced in R2019b**

## rossvcclient

Connect to ROS service server

### Description

Use `rossvcclient` or `ros.ServiceClient` to create a ROS service client object. This service client uses a persistent connection to send requests to, and receive responses from, a ROS service server. The connection persists until the service client is deleted or the service server becomes unavailable.

Use the `ros.ServiceClient` syntax when connecting to a specific ROS node.

### Creation

#### Syntax

```
client = rossvcclient(servicename)
client = rossvcclient(servicename,Name,Value)
```

```
[client,reqmsg] = rossvcclient( ___ )
```

```
client = ros.ServiceClient(node, name)
client = ros.ServiceClient(node, name,"Timeout",timeout)
```

#### Description

`client = rossvcclient(servicename)` creates a service client with the given `ServiceName` that connects to, and gets its `ServiceType` from, a service server. This command syntax prevents the current MATLAB program from running until it can connect to the service server.

`client = rossvcclient(servicename,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments.

`[client,reqmsg] = rossvcclient( ___ )` returns a new service request message in `reqmsg`, using any of the arguments from previous syntaxes. The message type of `reqmsg` is determined by the service that `client` is connected to. The message is initialized with default values. You can also create the request message using `rosmesssage`.

`client = ros.ServiceClient(node, name)` creates a service client that connects to a service server. The client gets its service type from the server. The service client attaches to the `ros.Node` object handle, `node`.

`client = ros.ServiceClient(node, name,"Timeout",timeout)` specifies a timeout period in seconds for the client to connect the service server.

## Properties

### ServiceName — Name of the service

string scalar | character vector

This property is read-only.

Name of the service, specified as a string scalar or character vector.

Example: `"/gazebo/get_model_state"`

### ServiceType — Type of service

string scalar | character vector

This property is read-only.

Type of service, specified as a string scalar or character vector.

Example: `"gazebo_msgs/GetModelState"`

## Object Functions

`rosmessage` Create ROS messages

`call` Call the ROS service server and receive a response

## Examples

### Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:50514/.
```

```
Initializing global node /matlab_global_node_67124 with NodeURI http://bat5110win64:50518/
```

Set up a service server and client.

```
server = rossvcserver('/test', 'std_srvs/Empty', @exampleHelperROSEmptyCallback);
client = rossvcclient('/test');
```

Call service server with default message.

```
response = call(client)
```

```
response =
  ROS EmptyResponse message with properties:
```

```
    MessageType: 'std_srvs/EmptyResponse'
```

Use `showdetails` to show the contents of the message

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_67124 with NodeURI http://bat5110win64:50518/
Shutting down ROS master on http://bat5110win64:50514/.
```

### Use ROS Service Server with ServiceServer and ServiceClient Objects

Create a ROS service serve by creating a `ServiceServer` object and use `ServiceClient` objects to request information over the network. The callback function used by the server takes a string, reverses it, and returns the reversed string.

Start the ROS master and node.

```
master = ros.Core;
node = ros.Node('/test');
```

Create a service server. This server expects a string as a request and responds with a string based on the callback.

```
server = ros.ServiceServer(node, '/data/string',...
                           'roseus/StringString');
```

Create a callback function. This function takes an input string as the `Str` property of `req` and returns it as the `Str` property of `resp`. The function definition is shown here, but is defined below the example. `req` is a ROS message you create using `rosmessage`.

```
function [resp] = flipString(~,req,resp)
% FLIPSTRING Reverses the order of a string in REQ and returns it in RESP.
resp.Str = fliplr(req.Str);
end
```

Assign the callback function for incoming service calls.

```
server.NewRequestFcn = @flipString;
```

Create a service client and connect to the service server. Create a request message based on the client.

```
client = ros.ServiceClient(node, '/data/string');
request = rosmessage(client);
request.Str = 'hello world';
```

Send a service request and wait for a response. Specify that the service waits 3 seconds for a response.

```
response = call(client,request,'Timeout',3)
```

```
response =
  ROS StringStringResponse message with properties:
```

```
    MessageType: 'roseus/StringStringResponse'
           Str: 'dlrow olleh'
```

```
Use showdetails to show the contents of the message
```

The response is a flipped string from the request message.

Clear the service client, service server, and ROS node. Shut down the ROS master.

```
clear('client', 'server', 'node')
clear('master')

function [resp] = flipString(~,req,resp)
% FLIPSTRING Reverses the order of a string in REQ and returns it in RESP.
resp.Str = fliplr(req.Str);
end
```

## See Also

[call](#) | [rosmesssage](#) | [rosservice](#) | [rossvcserver](#)

## Topics

“Call and Provide ROS Services”

**Introduced in R2019b**

## rossvcserver

Create ROS service server

### Description

Use `rossvcserver` or `ros.ServiceServer` to create a ROS service server that can receive requests from, and send responses to, a ROS service client. You must create the service server before creating the service client (see `ROSSVCCLIENT`).

When you create the service client, it establishes a connection to the server. The connection persists while both client and server exist and can reach each other. When you create the service server, it registers itself with the ROS master. To get a list of services, or to get information about a particular service that is available on the current ROS network, use the `rosservice` function.

The service has an associated message type and contains a pair of messages: one for the request and one for the response. The service server receives a request, constructs an appropriate response based on a call function, and returns it to the client. The behavior of the service server is inherently asynchronous because it becomes active only when a service client connects to the ROS network and issues a call.

Use the `ros.ServiceServer` syntax when connecting to a specific ROS node.

### Creation

#### Syntax

```
server = rossvcserver(servicename,svctype)
server = rossvcserver(servicename,svctype,callback)

server = ros.ServiceServer(node, name,type)
server = ros.ServiceServer(node, name,type,callback)
```

#### Description

`server = rossvcserver(servicename,svctype)` creates a service server object with the specified `ServiceType` available in the ROS network under the name `ServiceName`. The service object cannot respond to service requests until you specify a function handle callback, `NewMessageFcn`.

`server = rossvcserver(servicename,svctype,callback)` specifies the callback function that constructs a response when the server receives a request. The `callback` specifies the `NewMessageFcn` property.

`server = ros.ServiceServer(node, name,type)` creates a service server that attaches to the ROS node, `node`. The server becomes available through the specified service name and type once a callback function handle is specified in `NewMessageFcn`.

`server = ros.ServiceServer(node, name, type, callback)` specifies the callback function, which is set to the `NewMessageFcn` property.

## Properties

### ServiceName — Name of the service

string scalar | character vector

This property is read-only.

Name of the service, specified as a string scalar or character vector.

Example: `"/gazebo/get_model_state"`

Data Types: `char` | `string`

### ServiceType — Type of service

string scalar | character vector

This property is read-only.

Type of service, specified as a string scalar or character vector.

Example: `"gazebo_msgs/GetModelState"`

Data Types: `char` | `string`

### NewMessageFcn — Callback property

function handle | cell array

Callback property, specified as a function handle or cell array. In the first element of the cell array, specify either a function handle, string scalar, or character vector representing a function name. In subsequent elements, specify user data.

The service callback function requires at least three input arguments with one output. The first argument, `src`, is the associated service server object. The second argument, `reqMsg`, is the request message object sent by the service client. The third argument is the default response message object, `defaultRespMsg`. The callback returns a response message, `response`, based on the input request message and sends it back to the service client. Use the default response message as a starting point for constructing the request message. The function header for the callback is:

```
function response = serviceCallback(src, reqMsg, defaultRespMsg)
```

Specify the `NewMessageFcn` property as:

```
server.NewMessageFcn = @serviceCallback;
```

When setting the callback, you pass additional parameters to the callback function by including both the callback function and the parameters as elements of a cell array. The function header for the callback is:

```
function response = serviceCallback(src, reqMsg, defaultRespMsg, userData)
```

Specify the `NewMessageFcn` property as:

```
server.NewMessageFcn = {@serviceCallback, userData};
```

## Object Functions

rosmessage Create ROS messages

## Examples

### Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:50514/.
```

```
Initializing global node /matlab_global_node_67124 with NodeURI http://bat5110win64:50518/
```

Set up a service server and client.

```
server = rossvcserver('/test', 'std_srvs/Empty', @exampleHelperROSEmptyCallback);  
client = rossvcclient('/test');
```

Call service server with default message.

```
response = call(client)
```

```
response =  
  ROS EmptyResponse message with properties:
```

```
    MessageType: 'std_srvs/EmptyResponse'
```

```
    Use showdetails to show the contents of the message
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_67124 with NodeURI http://bat5110win64:50518/  
Shutting down ROS master on http://bat5110win64:50514/.
```

## See Also

[call](#) | [rosmessage](#) | [rossvcclient](#)

## Topics

“Call and Provide ROS Services”

**Introduced in R2019b**



# rostdf

Receive, send, and apply ROS transformations

## Description

Calling the `rostdf` function creates a ROS `TransformationTree` object, which allows you to access the `tf` coordinate transformations that are shared on the ROS network. You can receive transformations and apply them to different entities. You can also send transformations and share them with the rest of the ROS network.

ROS uses the `tf` transform library to keep track of the relationship between multiple coordinate frames. The relative transformations between these coordinate frames are maintained in a tree structure. Querying this tree lets you transform entities like poses and points between any two coordinate frames. To access available frames, use the syntax:

```
tfTree.AvailableFrames
```

Use the `ros.TransformationTree` syntax when connecting to a specific ROS node, otherwise use `rostdf` to create the transformation tree.

## Creation

### Syntax

```
tfTree = rostdf
```

```
trtree = ros.TransformationTree(node)
```

### Description

`tfTree = rostdf` creates a ROS `TransformationTree` object.

`trtree = ros.TransformationTree(node)` creates a ROS transformation tree object handle that the transformation tree is attached to. The `node` is the node connected to the ROS network that publishes transformations.

## Properties

### AvailableFrames — List of all available coordinate frames

cell array

This property is read-only.

List of all available coordinate frames, specified as a cell array. This list of available frames updates if new transformations are received by the transformation tree object.

Example: `{'camera_center'; 'mounting_point'; 'robot_base'}`

Data Types: `cell`

**LastUpdateTime — Time when the last transform was received**

ROS Time object

This property is read-only.

Time when the last transform was received, specified as a ROS Time object.

**BufferTime — Length of time transformations are buffered**

10 (default) | scalar

This property is read-only.

Length of time transformations are buffered, specified as a scalar in seconds. If you change the buffer time from the current value, the transformation tree and all transformations are reinitialized. You must wait for the entire buffer time to be completed to get a fully buffered transformation tree.

**Object Functions**

waitForTransform	Wait until a transformation is available
getTransform	Retrieve transformation between two coordinate frames
transform	Transform message entities into target coordinate frame
sendTransform	Send transformation to ROS network

**Examples****Create a ROS Transformation Tree**

Connect to a ROS network and create a transformation tree.

Connect to a ROS network. Specify the IP address.

```
rosinit('192.168.17.129')
```

```
Initializing global node /matlab_global_node_20832 with NodeURI http://192.168.17.1:55526/
```

Create a transformation tree. Use the AvailableFrames property to see the transformation frames available. These transformations were specified separately prior to connecting to the network.

```
tree = rostf;
pause(1);
tree.AvailableFrames

ans = 36x1 cell
    {'base_footprint'      }
    {'base_link'          }
    {'camera_depth_frame' }
    {'camera_depth_optical_frame'}
    {'camera_link'        }
    {'camera_rgb_frame'   }
    {'camera_rgb_optical_frame' }
    {'caster_back_link'   }
    {'caster_front_link'  }
    {'cliff_sensor_front_link'}
    {'cliff_sensor_left_link'}
    {'cliff_sensor_right_link' }
```

```

{'gyro_link'          }
{'mount_asus_xtion_pro_link' }
{'odom'              }
{'plate_bottom_link'  }
{'plate_middle_link'  }
{'plate_top_link'     }
{'pole_bottom_0_link' }
{'pole_bottom_1_link' }
{'pole_bottom_2_link' }
{'pole_bottom_3_link' }
{'pole_bottom_4_link' }
{'pole_bottom_5_link' }
{'pole_kinect_0_link'  }
{'pole_kinect_1_link'  }
{'pole_middle_0_link'  }
{'pole_middle_1_link'  }
{'pole_middle_2_link'  }
{'pole_middle_3_link'  }
:

```

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_20832 with NodeURI http://192.168.17.1:55526/
```

### Use TransformationTree Object

Create a ROS transformation tree. You can then view or use transformation information for different coordinate frames setup in the ROS network.

Start ROS network and broadcast sample transformation data.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:50077/.
```

```
Initializing global node /matlab_global_node_12266 with NodeURI http://bat5110win64:50081/
```

```
node = ros.Node('/testTf');
```

```
Using Master URI http://localhost:50077 from the global node to connect to the ROS master.
```

```
exampleHelperROSstartTfPublisher
```

Retrieve the TransformationTree object. Pause to wait for tftree to update.

```
tftree = ros.TransformationTree(node);
pause(1)
```

View available coordinate frames and the time when they were last received.

```
frames = tftree.AvailableFrames
```

```
frames = 3x1 cell
    {'camera_center' }
    {'mounting_point'}
```

```
    {'robot_base'    }
```

```
updateTime = tftree.LastUpdateTime
```

```
updateTime =  
  ROS Time with properties:
```

```
    Sec: 1.5830e+09  
    Nsec: 926000000
```

Wait for the transform between two frames, 'camera\_center' and 'robot\_base'. This will wait until the transformation is valid and block all other operations. A time out of 5 seconds is also given.

```
waitForTransform(tftree, 'robot_base', 'camera_center', 5)
```

Define a point in the camera's coordinate frame.

```
pt = rosmesssage('geometry_msgs/PointStamped');  
pt.Header.FrameId = 'camera_center';  
pt.Point.X = 3;  
pt.Point.Y = 1.5;  
pt.Point.Z = 0.2;
```

Transform the point into the 'base\_link' frame.

```
tfpt = transform(tftree, 'robot_base', pt)
```

```
tfpt =  
  ROS PointStamped message with properties:  
  
    MessageType: 'geometry_msgs/PointStamped'  
    Header: [1x1 Header]  
    Point: [1x1 Point]
```

Use showdetails to show the contents of the message

Display the transformed point coordinates.

```
tfpt.Point
```

```
ans =  
  ROS Point message with properties:  
  
    MessageType: 'geometry_msgs/Point'  
    X: 1.2000  
    Y: 1.5000  
    Z: -2.5000
```

Use showdetails to show the contents of the message

Clear ROS node. Shut down ROS master.

```
clear('node')  
roshutdowm
```

Shutting down global node /matlab\_global\_node\_12266 with NodeURI http://bat5110win64:50081/  
Shutting down ROS master on http://bat5110win64:50077/.

## **See Also**

[getTransform](#) | [sendTransform](#) | [transform](#) | [waitForTransform](#)

## **Topics**

“Access the tf Transformation Tree in ROS”

## **Introduced in R2019b**

# rostime

Access ROS time functionality

## Description

A ROS Time object representing an instance of time in seconds and nanoseconds. This time can be based on your system time, the ROS simulation time, or an arbitrary time.

## Creation

### Syntax

```
time = rostime(totalSecs)
time = rostime(secs,nsecs)

time = rostime("now")
[time,issimtime] = rostime("now")
time = rostime("now","system")
```

### Description

`time = rostime(totalSecs)` initializes the time values for seconds and nanoseconds based on `totalSecs`, which represents the time in seconds as a floating-point number.

`time = rostime(secs,nsecs)` initializes the time values for seconds and nanoseconds individually. Both inputs must be integers. Large values for `nsecs` are wrapped automatically with the remainder added to `secs`.

`time = rostime("now")` returns the current ROS time. If the `use_sim_time` ROS parameter is set to `true`, the `rostime` returns the simulation time published on the `clock` topic. Otherwise, the function returns the system time of your machine. The `time` is a ROS Time object. If no output argument is given, the current time (in seconds) is printed to the screen.

The `rostime` can be used to timestamp messages or to measure time in the ROS network.

`[time,issimtime] = rostime("now")` also returns a Boolean that indicates if `time` is in simulation time (`true`) or system time (`false`).

`time = rostime("now","system")` always returns the system time of your machine, even if ROS publishes simulation time on the `clock` topic. If no output argument is given, the system time (in seconds) is printed to the screen.

The system time in ROS follows the UNIX or POSIX time standard. POSIX time is defined as the time that has elapsed since 00:00:00 Coordinated Universal Time (UTC), 1 January 1970, not counting leap seconds.

## Properties

### totalSecs — Total time

0 (default) | scalar

Total time, specified as a floating-point scalar. The integer portion is set to the Sec property with the remainder applied to Nsec property of the Time object.

### Sec — Whole seconds

0 (default) | positive integer

Whole seconds, specified as a positive integer.

---

**Note** The maximum and minimum values for secs are [0, 4294967294].

---

### Nsec — Nanoseconds

0 (default) | positive integer

Nanoseconds, specified as a positive integer. If this value is greater than or equal to  $10^9$ , then the value is wrapped and the remainders are added to the value of Sec.

## Examples

### Get Current ROS Time

Connect to a ROS network.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:57657/.
```

```
Initializing global node /matlab_global_node_70134 with NodeURI http://bat5110win64:57661/
```

Get the current ROS Time. You can also check whether is it system time by getting the `issim` output.

```
[t,issim] = rostime('now')
```

```
t =
```

```
ROS Time with properties:
```

```
    Sec: 1.5830e+09
```

```
    Nsec: 280000000
```

```
issim = logical
```

```
    0
```

### Timestamp ROS Message Data

Create a stamped ROS message. Specify the Header .Stamp property with the current system time.

```
point = rosmesssage('geometry_msgs/PointStamped');
point.Header.Stamp = rostime('now', 'system');
```

### ROS Time to MATLAB Time Example

This example shows how to convert current ROS time into a MATLAB® standard time. The ROS Time object is first converted to a double in seconds, then to the specified MATLAB time.

Set up ROS network and store ROS time.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:50216/.
Initializing global node /matlab_global_node_50000 with NodeURI http://bat5110win64:50223/
```

```
t = rostime('now');
```

Convert ROS time to a double in seconds.

```
secondtime = double(t.Sec)+double(t.Nsec)*10^-9;
```

Set time to a specified MATLAB format.

```
time = datetime(secondtime, 'ConvertFrom', 'posixtime')
```

```
time = datetime
      29-Feb-2020 06:06:33
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_50000 with NodeURI http://bat5110win64:50223/
Shutting down ROS master on http://bat5110win64:50216/.
```

### Get Seconds From A Time Object

Use the `seconds` function to get the total seconds of a `Time` object from its `Secs` and `Nsecs` properties.

Create a `Time` object.

```
time = rostime(1,860000000)
```

```
time =
  ROS Time with properties:
```

```
    Sec: 1
   Nsec: 860000000
```

Get the total seconds from the time object.

```
secs = seconds(time)
```



```
secs = 1.8600
```

**See Also**

[rostduration](#) | [rosmesssage](#) | [seconds](#)

**Introduced in R2019b**

# TransformStamped

Create transformation message

## Description

The TransformStamped object is an implementation of the `geometry_msgs/TransformStamped` message type in ROS. The object contains meta-information about the message itself and the transformation. The transformation has a translational and rotational component.

## Creation

### Syntax

```
tform = getTransform(tftree, targetframe, sourceframe)
```

### Description

`tform = getTransform(tftree, targetframe, sourceframe)` returns the latest known transformation between two coordinate frames. Transformations are structured as a 3-D translation (3-element vector) and a 3-D rotation (quaternion).

## Properties

### MessageType — Message type of ROS message

character vector

This property is read-only.

Message type of ROS message, returned as a character vector.

Data Types: char

### Header — ROS Header message

Header object

This property is read-only.

ROS Header message, returned as a Header object. This header message contains the MessageType, sequence (Seq), timestamp (Stamp), and FrameId.

### ChildFrameID — Second coordinate frame to transform point into

character vector

Second coordinate frame to transform point into, specified as a character vector.

### Transform — Transformation message

Transform object

This property is read-only.

Transformation message, specified as a `Transform` object. The object contains the `MessageType` with a `Translation` vector and `Rotation` quaternion.

## Object Functions

`apply` Transform message entities into target frame

## Examples

### Inspect Sample TransformStamped Object

This example looks at the `TransformStamped` object to show the underlying structure of a `TransformStamped` ROS message. After setting up a network and transformations, you can create a transformation tree and get transformations between specific coordinate systems. Using `showdetails` lets you inspect the information in the transformation. It contains the `ChildFrameId`, `Header`, and `Transform`.

Start ROS network and setup transformations.

```
rosinit
```

```
Initializing ROS master on http://bat5110win64:56701/.
Initializing global node /matlab_global_node_77543 with NodeURI http://bat5110win64:56707/
```

```
exampleHelperROSStartTfPublisher
```

Create transformation tree and wait for tree to update. Get the transform between the robot base and its camera center.

```
tftree = rostf;
waitForTransform(tftree, 'camera_center', 'robot_base');
tform = getTransform(tftree, 'camera_center', 'robot_base');
```

Inspect the `TransformStamped` object.

```
showdetails(tform)
```

```
ChildFrameId : robot_base
Header
  Seq      : 155
  FrameId  : camera_center
  Stamp
    Sec    : 1582956093
    Nsec   : 910000128
Transform
  Translation
    X : 0.4999999999999998
    Y : 0
    Z : -1
  Rotation
    X : 0
    Y : -0.7071067811865475
    Z : 0
    W : 0.7071067811865476
```

Access the `Translation` vector inside the `Transform` property.

```
trans = tform.Transform.Translation
trans =
  ROS Vector3 message with properties:
    MessageType: 'geometry_msgs/Vector3'
      X: 0.5000
      Y: 0
      Z: -1
  Use showdetails to show the contents of the message
```

Stop the example transformation publisher.

```
exampleHelperROSStopTfPublisher
```

Shutdown ROS network.

```
roshutdown
```

```
Shutting down global node /matlab_global_node_77543 with NodeURI http://bat5110win64:56707/
Shutting down ROS master on http://bat5110win64:56701/.
```

### **Apply Transformation using TransformStamped Object**

Apply a transformation from a TransformStamped object to a PointStamped message.

Start ROS network and setup transformations.

```
rosinit
```

```
Initializing ROS master on http://ah-sradford:51698/.
Initializing global node /matlab_global_node_81716 with NodeURI http://ah-sradford:51702/
```

```
exampleHelperROSStartTfPublisher
```

Create transformation tree and wait for tree to update. Get the transform between the robot base and its camera center. Inspect the transformation.

```
tftree = rostf;
waitForTransform(tftree, 'camera_center', 'robot_base');
tform = getTransform(tftree, 'camera_center', 'robot_base');
showdetails(tform)
```

```
ChildFrameId : robot_base
Header
  Seq      : 156
  FrameId  : camera_center
  Stamp
    Sec    : 1562097411
    Nsec   : 644999936
Transform
  Translation
    X : 0.4999999999999998
    Y : 0
```

```
Z : -1
Rotation
X : 0
Y : -0.7071067811865475
Z : 0
W : 0.7071067811865476
```

Create point to transform. You could also get this point message off the ROS network.

```
pt = rosmesssage('geometry_msgs/PointStamped');
pt.Header.FrameId = 'camera_center';
pt.Point.X = 3;
pt.Point.Y = 1.5;
pt.Point.Z = 0.2;
```

Apply the transformation to the point.

```
tfpt = apply(tform,pt);
```

Shutdown ROS network.

```
roshuttdown
```

```
Shutting down global node /matlab_global_node_81716 with NodeURI http://ah-sradford:51702/
Shutting down ROS master on http://ah-sradford:51698/.
```

## See Also

### Functions

`apply` | `getTransform` | `rostf` | `transform` | `waitForTransform`

### Topics

“Access the tf Transformation Tree in ROS”

### Introduced in R2019b



# Methods

---

## reset

Reset Rate object

### Syntax

```
reset(rate)
```

### Description

`reset(rate)` resets the state of the Rate object, including the elapsed time and all statistics about previous periods. `reset` is useful if you want to run multiple successive loops at the same rate, or if the object is created before the loop is executed.

### Input Arguments

**rate — rateControl object**

handle

`rateControl` object, specified as a handle. This object contains the information for the desired rate and other information about the execution. See `rateControl` for more information.

### Examples

#### Run Loop At Fixed Rate and Reset Rate Object

Create a Rate object for running at 20 Hz.

```
r = robotics.Rate(2);
```

Start a loop and control operation using the Rate object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Display the Rate object properties after loop operation.

```
disp(r)

rateControl with properties:

    DesiredRate: 2
    DesiredPeriod: 0.5000
    OverrunAction: 'slip'
    TotalElapsedTime: 15.0196
    LastPeriod: 0.5000
```

Reset the object to restart the time statistics.



```
reset(r);  
disp(r)
```

```
rateControl with properties:
```

```
    DesiredRate: 2  
    DesiredPeriod: 0.5000  
    OverrunAction: 'slip'  
    TotalElapsedTime: 0.0058  
    LastPeriod: NaN
```

## See Also

[rateControl](#) | [rosclock](#) | [waitfor](#)

## Topics

“Execute Code at a Fixed-Rate” (Robotics System Toolbox)

**Introduced in R2019b**

## statistics

Statistics of past execution periods

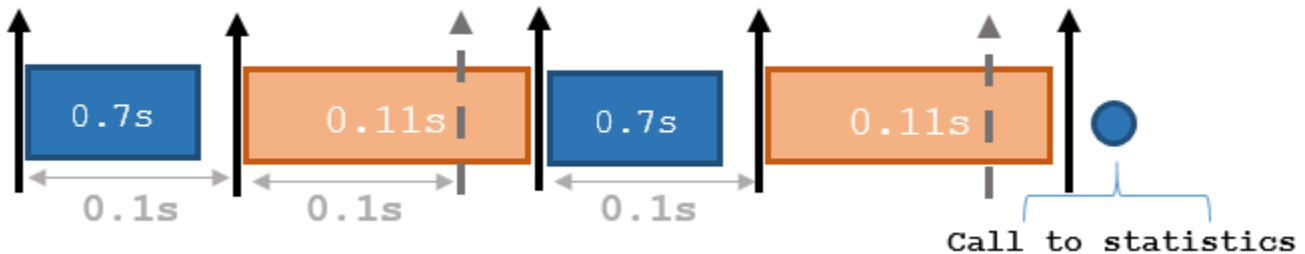
### Syntax

```
stats = statistics(rate)
```

### Description

`stats = statistics(rate)` returns statistics of previous periods of code execution. `stats` is a struct with these fields: `Periods`, `NumPeriods`, `AveragePeriod`, `StandardDeviation`, and `NumOverruns`.

Here is a sample execution graphic using the default setting, 'slip', for the `OverrunAction` property in the `Rate` object. See `OverrunAction` for more information on overrun code execution.



The output of `statistics` is:

```
stats =
    Periods: [0.7 0.11 0.7 0.11]
    NumPeriods: 4
    AveragePeriod: 0.09
    StandardDeviation: 0.0231
    NumOverruns: 2
```

### Input Arguments

**rate** — Rate object

handle

Rate object, specified as an object handle. This object contains the information for the `DesiredRate` and other info about the execution. See `rateControl` for more information.

### Output Arguments

**stats** — Time execution statistics

structure

Time execution statistics, returned as a structure. This structure contains the following fields:

- **Period** — All time periods (returned in seconds) used to calculate statistics as an indexed array. `stats.Period(end)` is the most recent period.
- **NumPeriods** — Number of elements in **Periods**
- **AveragePeriod** — Average time in seconds
- **StandardDeviation** — Standard deviation of all periods in seconds, centered around the mean stored in **AveragePeriod**
- **NumOverruns** — Number of periods with overrun

## Examples

### Get Statistics From Rate Object Execution

Create a `Rate` object for running at 20 Hz.

```
r = robotics.Rate(2);
```

Start a loop and control operation using the `Rate` object.

```
for i = 1:30
    % Your code goes here
    waitfor(r);
end
```

Get `Rate` object statistics after loop operation.

```
stats = statistics(r)

stats = struct with fields:
    Periods: [1x30 double]
    NumPeriods: 30
    AveragePeriod: 0.5000
    StandardDeviation: 0.0011
    NumOverruns: 0
```

## See Also

`rateControl` | `rosrate` | `waitfor`

## Topics

“Execute Code at a Fixed-Rate” (Robotics System Toolbox)

## Introduced in R2019b

## waitfor

**Package:** ros

Pause code execution to achieve desired execution rate

### Syntax

```
waitfor(rate)
numMisses = waitfor(rate)
```

### Description

`waitfor(rate)` pauses execution until the code reaches the desired execution rate. The function accounts for the time that is spent executing code between `waitfor` calls.

`numMisses = waitfor(rate)` returns the number of iterations missed while executing code between calls.

### Examples

#### Run Loop at Fixed Rate

Create a rate object that runs at 1 Hz.

```
r = robotics.Rate(1);
```

Start a loop using the `Rate` object inside to control the loop execution. Reset the object prior to the loop execution to reset timer. Print the iteration and time elapsed.

```
reset(r)
for i = 1:10
    time = r.TotalElapsedTime;
    fprintf('Iteration: %d - Time Elapsed: %f\n',i,time)
    waitfor(r);
end
```

```
Iteration: 1 - Time Elapsed: 0.003078
Iteration: 2 - Time Elapsed: 1.000985
Iteration: 3 - Time Elapsed: 2.001207
Iteration: 4 - Time Elapsed: 3.000967
Iteration: 5 - Time Elapsed: 4.000802
Iteration: 6 - Time Elapsed: 5.001010
Iteration: 7 - Time Elapsed: 6.000952
Iteration: 8 - Time Elapsed: 7.000241
Iteration: 9 - Time Elapsed: 8.000404
Iteration: 10 - Time Elapsed: 9.000650
```

Each iteration executes at a 1-second interval.

## Input Arguments

### **rate** — Rate object

handle

Rate object, specified as a handle. This object contains the information for the desired rate and other information about the execution. See `rateControl` for more information.

## Output Arguments

### **numMisses** — Number of missed task executions

scalar

Number of missed task executions, returned as a scalar. `waitfor` returns the number of times the task was missed in the `Rate` object based on the `LastPeriod` time. For example, if the desired rate is 1 Hz and the last period was 3.2 seconds, `numMisses` returns 3.

## See Also

`rateControl` | `rosrate` | `waitfor`

## Topics

“Execute Code at a Fixed-Rate” (Robotics System Toolbox)

**Introduced in R2019b**

## send

Publish ROS 2 message to topic

### Syntax

```
send(pub, msg)
```

### Description

`send(pub, msg)` publishes a message to the topic specified by the publisher, `pub`. This message can be received by all subscribers in the ROS 2 network that are subscribed to the topic specified by `pub`.

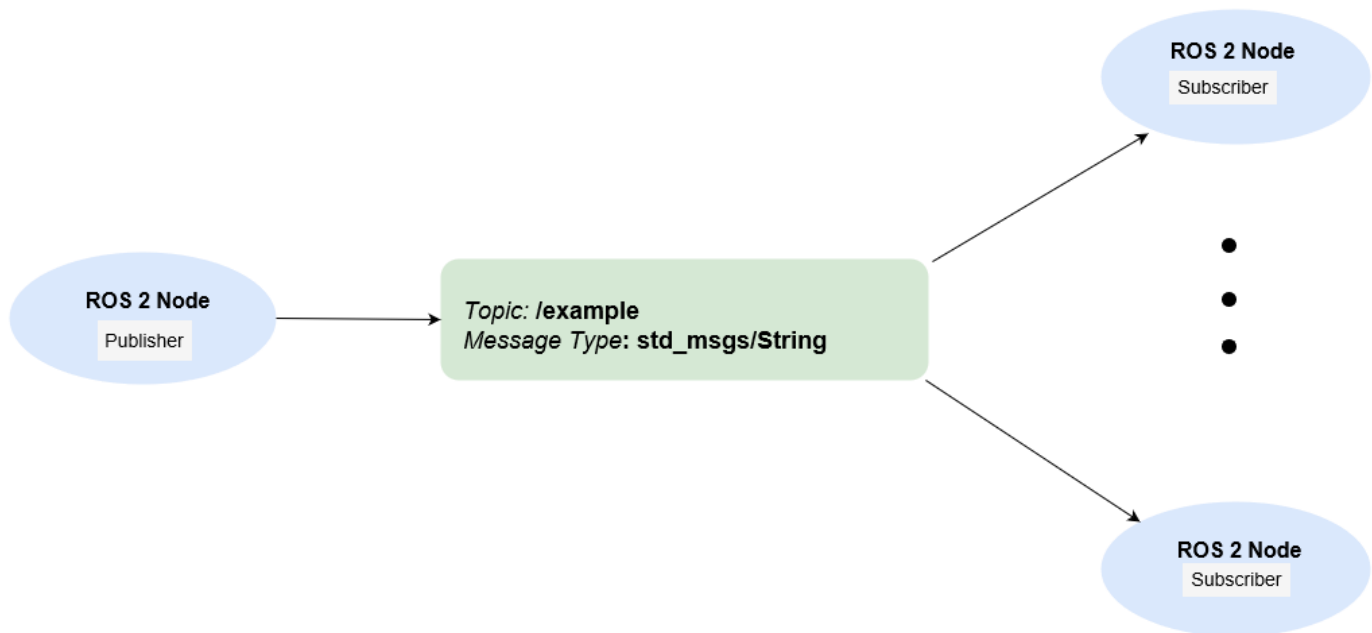
### Examples

#### Exchange Data with ROS 2 Publishers and Subscribers

The primary mechanism for ROS 2 nodes to exchange data is to send and receive *messages*. Messages are transmitted on a *topic* and each topic has a unique name in the ROS 2 network. If a node wants to share information, it must use a *publisher* to send data to a topic. A node that wants to receive that information must use a *subscriber* for that same topic. Besides its unique name, each topic also has a *message type*, which determines the type of messages that are allowed to be transmitted in the specific topic.

This publisher-subscriber communication has the following characteristics:

- Topics are used for many-to-many communication. Multiple publishers can send messages to the same topic and multiple subscribers can receive them.
- Publisher and subscribers are decoupled through topics and can be created and destroyed in any order. A message can be published to a topic even if there are no active subscribers.



This example shows how to publish and subscribe to topics in a ROS 2 network. It also shows how to:

- Wait until a new message is received, or
- Use callbacks to process new messages in the background

Prerequisites: “Get Started with ROS 2”, “Connect to a ROS 2 Network”

### Subscribe and Wait for Messages

Create a sample ROS 2 network with several publishers and subscribers.

```
exampleHelperROS2CreateSampleNetwork
```

Use `ros2 topic list` to see which topics are available.

```
ros2 topic list
```

```
/parameter_events
/pose
/scan
```

Assume you want to subscribe to the `/scan` topic. Use `ros2subscriber` to subscribe to the `/scan` topic. Specify the name of the node with the subscriber. If the topic already exists in the ROS 2 network, `ros2subscriber` detects its message type automatically, so you do not need to specify it.

```
detectNode = ros2node("/detection");
pause(2)
laserSub = ros2subscriber(detectNode, "/scan");
pause(2)
```

Use `receive` to wait for a new message. Specify a timeout of 10 seconds. The output `scanData` contains the received message data.

```
scanData = receive(laserSub,10);
```

You can now remove the subscriber `laserSub` and the node associated to it.

```
clear laserSub
clear detectNode
```

### Subscribe Using Callback Functions

Instead of using `receive` to get data, you can specify a function to be called when a new message is received. This allows other MATLAB code to execute while the subscriber is waiting for new messages. Callbacks are essential if you want to use multiple subscribers.

Subscribe to the `/pose` topic, using the callback function `exampleHelperROS2PoseCallback`, which takes a received message as the input. One way of sharing data between your main workspace and the callback function is to use global variables. Define two global variables `pos` and `orient`.

```
controlNode = ros2node("/base_station");
poseSub = ros2subscriber(controlNode, "/pose", @exampleHelperROS2PoseCallback);
global pos
global orient
```

The global variables `pos` and `orient` are assigned in the `exampleHelperROS2PoseCallback` function when new message data is received on the `/pose` topic.

```
function exampleHelperROS2PoseCallback(message)
    % Declare global variables to store position and orientation
    global pos
    global orient

    % Extract position and orientation from the ROS message and assign the
    % data to the global variables.
    pos = [message.linear.x message.linear.y message.linear.z];
    orient = [message.angular.x message.angular.y message.angular.z];
end
```

Wait a moment for the network to publish another `/pose` message. Display the updated values.

```
pause(3)
disp(pos)

    0.0416    -0.0499    -0.0038

disp(orient)

   -0.0076   -0.0039    0.0270
```

If you type in `pos` and `orient` a few times in the command line you can see that the values are continuously updated.

Stop the pose subscriber by clearing the subscriber variable

```
clear poseSub
clear controlNode
```

*Note:* There are other ways to extract information from callback functions besides using globals. For example, you can pass a handle object as additional argument to the callback function. See the “Callback Definition” (MATLAB) documentation for more information about defining callback functions.



## Publish Messages

Create a publisher that sends ROS 2 string messages to the `/chatter` topic.

```
chatterPub = ros2publisher(node_1, "/chatter", "std_msgs/String");
```

Create and populate a ROS 2 message to send to the `/chatter` topic.

```
chatterMsg = ros2message(chatterPub);
chatterMsg.data = 'hello world';
```

Use `ros2 topic list` to verify that the `/chatter` topic is available in the ROS 2 network.

```
ros2 topic list
```

```
/chatter
/parameter_events
/pose
/scan
```

Define a subscriber for the `/chatter` topic. `exampleHelperROS2ChatterCallback` is called when a new message is received, and displays the string content in the message.

```
chatterSub = ros2subscriber(node_2, "/chatter", @exampleHelperROS2ChatterCallback)
```

```
chatterSub =
  ros2subscriber with properties:
    TopicName: '/chatter'
    LatestMessage: []
    MessageType: 'std_msgs/String'
    NewMessageFcn: @exampleHelperROS2ChatterCallback
    History: 'keeplast'
    Depth: 10
    Reliability: 'reliable'
    Durability: 'volatile'
```

Publish a message to the `/chatter` topic. Observe that the string is displayed by the subscriber callback.

```
send(chatterPub, chatterMsg)
pause(3)
```

```
ans =
'hello world'
```

The `exampleHelperROS2ChatterCallback` function was called when the subscriber received the string message.

## Disconnect From ROS 2 Network

Remove the sample nodes, publishers and subscribers from the ROS 2 network. Also clear the global variables `pos` and `orient`

```
clear global pos orient
clear
```

**Next Steps**

- “Work with Basic ROS 2 Messages”
- “ROS 2 Custom Message Support”

**Input Arguments****pub — ros2publisher object**`ros2publisher`

`ros2publisher` object, specified as a handle, that publishes the specified topic.

**msg — ROS 2 message**`Message structure`

ROS 2 message, specified as a structure, with compatible fields for that message type.

**See Also**`ros2publisher`**Topics**

“Exchange Data with ROS 2 Publishers and Subscribers”

“Manage Quality of Service Policies in ROS 2”

**Introduced in R2019b**

# delete

Remove reference to ROS 2 node

## Syntax

```
delete(node)
```

## Description

`delete(node)` removes the reference in `node` to the ROS 2 node on the network. If no further references to the node exist, such as would be in publishers and subscribers, the node is shut down.

## Input Arguments

**node** — ROS 2 node on network

`handle` (default)

A `ros2node` object on the network.

## See Also

`ros2node`

**Introduced in R2019b**

## receive

Wait for new message

### Syntax

```
msg = receive(sub)
msg = receive(sub, timeout)
```

### Description

`msg = receive(sub)` blocks code execution until a new message is received by the subscriber, `sub`, for the specific topic.

`msg = receive(sub, timeout)` specifies a `timeout` period, in seconds. If the subscriber does not receive a topic message and the timeout period elapses, the function displays an error message.

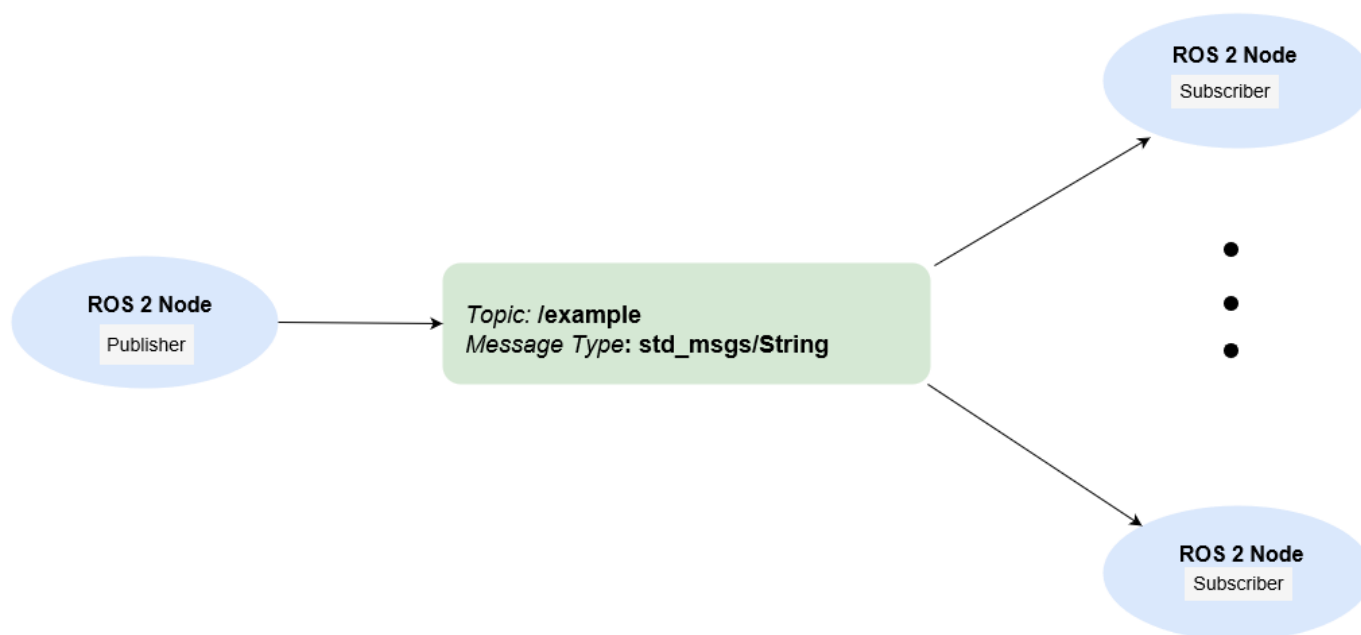
### Examples

#### Exchange Data with ROS 2 Publishers and Subscribers

The primary mechanism for ROS 2 nodes to exchange data is to send and receive *messages*. Messages are transmitted on a *topic* and each topic has a unique name in the ROS 2 network. If a node wants to share information, it must use a *publisher* to send data to a topic. A node that wants to receive that information must use a *subscriber* for that same topic. Besides its unique name, each topic also has a *message type*, which determines the type of messages that are allowed to be transmitted in the specific topic.

This publisher-subscriber communication has the following characteristics:

- Topics are used for many-to-many communication. Multiple publishers can send messages to the same topic and multiple subscribers can receive them.
- Publisher and subscribers are decoupled through topics and can be created and destroyed in any order. A message can be published to a topic even if there are no active subscribers.



This example shows how to publish and subscribe to topics in a ROS 2 network. It also shows how to:

- Wait until a new message is received, or
- Use callbacks to process new messages in the background

Prerequisites: “Get Started with ROS 2”, “Connect to a ROS 2 Network”

### Subscribe and Wait for Messages

Create a sample ROS 2 network with several publishers and subscribers.

```
exampleHelperROS2CreateSampleNetwork
```

Use `ros2 topic list` to see which topics are available.

```
ros2 topic list
```

```
/parameter_events
/pose
/scan
```

Assume you want to subscribe to the `/scan` topic. Use `ros2subscriber` to subscribe to the `/scan` topic. Specify the name of the node with the subscriber. If the topic already exists in the ROS 2 network, `ros2subscriber` detects its message type automatically, so you do not need to specify it.

```
detectNode = ros2node("/detection");
pause(2)
laserSub = ros2subscriber(detectNode, "/scan");
pause(2)
```

Use `receive` to wait for a new message. Specify a timeout of 10 seconds. The output `scanData` contains the received message data.

```
scanData = receive(laserSub,10);
```

You can now remove the subscriber `laserSub` and the node associated to it.

```
clear laserSub
clear detectNode
```

### Subscribe Using Callback Functions

Instead of using `receive` to get data, you can specify a function to be called when a new message is received. This allows other MATLAB code to execute while the subscriber is waiting for new messages. Callbacks are essential if you want to use multiple subscribers.

Subscribe to the `/pose` topic, using the callback function `exampleHelperROS2PoseCallback`, which takes a received message as the input. One way of sharing data between your main workspace and the callback function is to use global variables. Define two global variables `pos` and `orient`.

```
controlNode = ros2node("/base_station");
poseSub = ros2subscriber(controlNode, "/pose", @exampleHelperROS2PoseCallback);
global pos
global orient
```

The global variables `pos` and `orient` are assigned in the `exampleHelperROS2PoseCallback` function when new message data is received on the `/pose` topic.

```
function exampleHelperROS2PoseCallback(message)
    % Declare global variables to store position and orientation
    global pos
    global orient

    % Extract position and orientation from the ROS message and assign the
    % data to the global variables.
    pos = [message.linear.x message.linear.y message.linear.z];
    orient = [message.angular.x message.angular.y message.angular.z];
end
```

Wait a moment for the network to publish another `/pose` message. Display the updated values.

```
pause(3)
disp(pos)

    0.0416   -0.0499   -0.0038

disp(orient)

   -0.0076   -0.0039    0.0270
```

If you type in `pos` and `orient` a few times in the command line you can see that the values are continuously updated.

Stop the pose subscriber by clearing the subscriber variable

```
clear poseSub
clear controlNode
```

*Note:* There are other ways to extract information from callback functions besides using globals. For example, you can pass a handle object as additional argument to the callback function. See the “Callback Definition” (MATLAB) documentation for more information about defining callback functions.

## Publish Messages

Create a publisher that sends ROS 2 string messages to the `/chatter` topic.

```
chatterPub = ros2publisher(node_1, "/chatter", "std_msgs/String");
```

Create and populate a ROS 2 message to send to the `/chatter` topic.

```
chatterMsg = ros2message(chatterPub);
chatterMsg.data = 'hello world';
```

Use `ros2 topic list` to verify that the `/chatter` topic is available in the ROS 2 network.

```
ros2 topic list
```

```
/chatter
/parameter_events
/pose
/scan
```

Define a subscriber for the `/chatter` topic. `exampleHelperROS2ChatterCallback` is called when a new message is received, and displays the string content in the message.

```
chatterSub = ros2subscriber(node_2, "/chatter", @exampleHelperROS2ChatterCallback)
```

```
chatterSub =
  ros2subscriber with properties:
    TopicName: '/chatter'
    LatestMessage: []
    MessageType: 'std_msgs/String'
    NewMessageFcn: @exampleHelperROS2ChatterCallback
    History: 'keeplast'
    Depth: 10
    Reliability: 'reliable'
    Durability: 'volatile'
```

Publish a message to the `/chatter` topic. Observe that the string is displayed by the subscriber callback.

```
send(chatterPub, chatterMsg)
pause(3)
```

```
ans =
'hello world'
```

The `exampleHelperROS2ChatterCallback` function was called when the subscriber received the string message.

## Disconnect From ROS 2 Network

Remove the sample nodes, publishers and subscribers from the ROS 2 network. Also clear the global variables `pos` and `orient`

```
clear global pos orient
clear
```

### Next Steps

- “Work with Basic ROS 2 Messages”
- “ROS 2 Custom Message Support”

## Input Arguments

### **sub** — **ros2subscriber** object

handle (default)

ros2subscriber object, specified as a handle, that subscribes to a specific topic.

### **timeout** — **Timeout period**

positive scalar

The amount of time before the `receiver` function will error out if a message is not received.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64`

## Output Arguments

### **msg** — **ROS 2 message**

Message object handle

ROS 2 message, specified as a `Message` object handle.

## Tips

Choosing between `receive` and using a callback:

- Use `receive` when your program should wait until the next message is received on the topic and no other processing should happen in the meantime.
- If you want your program to keep running and be notified whenever a new message arrives, consider using a callback instead of `receive`.
- If you want your program to periodically use the most recent data received by the subscriber, consider accessing the `LatestMessage` property instead of using `receive` or a callback.

## See Also

ros2subscriber

**Introduced in R2019b**



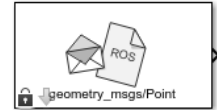
# Blocks

---

## Blank Message

Create blank message using specified message type

**Library:** ROS Toolbox / ROS



### Description

The Blank Message block creates a Simulink nonvirtual bus corresponding to the selected ROS message type. The block creates ROS message buses that work with Publish, Subscribe, or Call Service blocks. On each sample hit, the block outputs a blank or “zero” signal for the designated message type. All elements of the bus are initialized to 0. The lengths of the variable-length arrays are also initialized to 0.

### Limitations

Before R2016b, models using ROS message types with certain reserved property names could not generate code. In 2016b, this limitation has been removed. The property names are now appended with an underscore (e.g. `Vector3Stamped_`). If you use models created with a pre-R2016b release, update the ROS message types using the new names with an underscore. Redefine custom maximum sizes for variable length arrays.

The affected message types are:

- 'geometry\_msgs/Vector3Stamped'
- 'jsk\_pcl\_ros/TransformScreenpointResponse'
- 'pddl\_msgs/PDDLAction'
- 'rocon\_interaction\_msgs/Interaction'
- 'capabilities/GetRemappingsResponse'
- 'dynamic\_reconfigure/Group'

### Input/Output Ports

#### Output

##### Msg — Blank ROS message

nonvirtual bus

Blank ROS message, returned as a nonvirtual bus. To specify the type of ROS message, use the **Message type** parameter. All elements of the bus are initialized to 0. The lengths of the variable-length arrays are also initialized to 0.

Data Types: bus

## Parameters

### Class — Class of ROS message

Message (default) | Service Request | Service Response

Class of ROS message, specified as Message, Service Request, or Service Response. For basic publishing and subscribing, use the Message class.

### Type — ROS message type

'geometry\_msgs/Point' (default) | character vector | dialog box selection

ROS message type, specified as a character vector or a dialog box selection. Use **Select** to select from a list of supported ROS messages. The list of messages given depends on the **Class** of message you select.

### Sample time — Interval between outputs

Inf (default) | numeric scalar

Interval between outputs, specified as a numeric scalar. The default value indicates that the block output never changes. Using this value speeds simulation and code generation by eliminating the need to recompute the block output. Otherwise, the block outputs a new blank message at each interval of **Sample time**.

For more information, see “Specify Sample Time” (Simulink).

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## See Also

Call Service | Publish | Subscribe

### Topics

“Get Started with ROS in Simulink®”

“Work with ROS Messages in Simulink®”

“Connect to a ROS-enabled Robot from Simulink®”

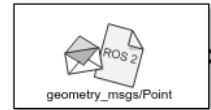
“Types of Composite Signals” (Simulink)

### Introduced in R2019b

## Blank Message

Create blank ROS 2 message using specified message type

**Library:** ROS Toolbox / ROS 2



### Description

Create a blank ROS 2 message with the specified message type.

### Ports

#### Output

##### Msg — Blank ROS 2 message

non-virtual bus

Blank ROS 2 message, returned as a non-virtual bus. To specify the type of ROS message, use the **Message type** parameter. All elements of the bus are initialized to 0. The lengths of the variable-length arrays are also initialized to 0.

Data Types: bus

### Parameters

#### Message type — ROS 2 message type

'geometry\_msgs/Point' (default) | character vector | dialog box selection

ROS 2 message type, specified as a character vector or a dialog box selection. Use **Select** to select from a list of supported ROS messages. The list of messages given depends on the **Class** of message you select.

#### Sample time — Interval between outputs

Inf (default) | positive numeric scalar

Interval between outputs, specified as a numeric scalar. The default value indicates that the block output never changes. Using this value speeds simulation and code generation by eliminating the need to recompute the block output. Otherwise, the block outputs a new blank message at each interval of **Sample time**.

For more information, see “Specify Sample Time” (Simulink).

### Extended Capabilities

#### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

## **See Also**

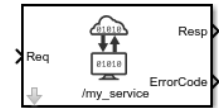
Publish | [Subscribe](#)

**Introduced in R2019b**

## Call Service

Call service in ROS network

**Library:** ROS Toolbox / ROS



### Description

The Call Service block takes a ROS service request message, sends it to the ROS service server, and waits for a response. Connect to a ROS network using `rosinit`. A ROS server should be set up somewhere on the network before using this block. Check the available services on a ROS network using `rosservice`. Use `rossvcserver` to set up a service server in MATLAB.

Specify the name for your ROS service and the service type in the block mask. If connected to a ROS network, you can select from a list of available services. You can create a blank service request or response message to populate with data using the Blank Message block.

### Ports

#### Input

##### Req — Request message

nonvirtual bus

Request message, specified as a nonvirtual bus. The request message type corresponds to your service type. To generate an empty request message bus to populate with data, use the Blank Message block.

Data Types: bus

#### Output

##### Resp — Response message

nonvirtual bus

Response message, returned as a nonvirtual bus. The response is based on the input **Req** message. The response message type corresponds to your service type. To generate an empty response message bus to populate with data, use the Blank Message block.

Data Types: bus

##### ErrorCode — Error conditions for service call

integer

Error conditions for service call, specified as an integer. Each integer corresponds to a different error condition for the service connection or the status of the service call. If an error condition occurs, **Resp** outputs the last response message or a blank message if a response was not previously received.

**Error Codes:**

Error Code	Condition
0	The service response was successfully retrieved and is available in the Resp output.
1	The connection was not established within the specified Connection timeout.
2	The response from the server was not received within the specified Call timeout
3	The service call failed for unknown reasons.

**Dependencies**

This output is enabled when the **Show ErrorCode output port** check box is on.

Data Types: uint8

**Parameters****Source — Source for specifying service name**

Select from ROS network | Specify your own

Source for specifying the service name:

- **Select from ROS network** — Use **Select** to select a service name. The **Name** and **Type** parameters are set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a service name in **Name** and specify its service type in **Type**. You must match a service name exactly.

**Name — Service name**

character vector

Service name, specified as a character vector. The service name must match a service name available on the ROS service server. To see a list of valid services in a ROS network, see `rosservice`.

**Type — Service type**

character vector

Service type, specified as a character vector. Each service name has a corresponding type.

**Connection timeout — Timeout for service server connection**

5 (default) | positive numeric scalar

Timeout for service server connection, specified as a positive numeric scalar in seconds. If a connection cannot be established with the ROS service server in this time, then **ErrorCode** outputs 1.

**Keep persistent connection — Keep connection to service server**

off (default) | on

Check this box to maintain a persistent connection with the ROS service server. When `off`, the block creates a service client every time a request message is input into **Req**.

**Show ErrorCode output port — Enable error code output port**

on (default) | off

Check this box to output the **ErrorCode** output. If an error condition occurs, **Resp** outputs the last response message or a blank message if response was not previously received.

**See Also****Blocks**

Blank Message | Publish | Subscribe

**Functions**

rosservice | rossvcclient | rossvcserver

**Topics**

“Call and Provide ROS Services”

“Publish and Subscribe to ROS Messages in Simulink”

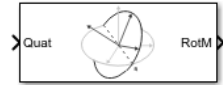
**Introduced in R2019b**



# Coordinate Transformation Conversion

Convert to a specified coordinate transformation representation

**Library:**           Robotics System Toolbox / Utilities  
                   Navigation Toolbox / Utilities  
                   ROS Toolbox / Utilities



## Description

The Coordinate Transformation Conversion block converts a coordinate transformation from the input representation to a specified output representation. The input and output representations use the following forms:

- Axis-Angle (AxAng) - [x y z theta]
- Euler Angles (Eul) - [z y x], [z y z], or [x y z]
- Homogeneous Transformation (TForm) - 4-by-4 matrix
- Quaternion (Quat) - [w x y z]
- Rotation Matrix (RotM) - 3-by-3 matrix
- Translation Vector (TrVec) - [x y z]

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (TrVec or Eul, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional Show TrVec input/output port parameter can be selected on the block mask to toggle the multiple ports.

## Ports

### Input

#### Input transformation — Coordinate transformation

column vector | 3-by-3 matrix | 4-by-4 matrix

Input transformation, specified as a coordinate transformation. The following representations are supported:

- Axis-Angle (AxAng) - [x y z theta]
- Euler Angles (Eul) - [z y x], [z y z], or [x y z]
- Homogeneous Transformation (TForm) - 4-by-4 matrix
- Quaternion (Quat) - [w x y z]
- Rotation Matrix (RotM) - 3-by-3 matrix
- Translation Vector (TrVec) - [x y z]

All vectors must be **column vectors**.

To accommodate representations that only contain position or orientation information (TrVec or Eul, for example), you can specify two inputs or outputs to handle all transformation information. When

you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

### **TrVec — Translation vector**

3-element column vector

Translation vector, specified as a 3-element column vector,  $[x \ y \ z]$ , which corresponds to a translation in the  $x$ ,  $y$ , and  $z$  axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

#### **Dependencies**

You must select Homogeneous Transformation (TForm) for the opposite transformation port to get the option to show the additional TrVec port. Enable the port by clicking `Show TrVec input/output port`.

#### **Output Arguments**

### **Output transformation — Coordinate transformation**

column vector | 3-by-3 matrix | 4-by-4 matrix

Output transformation, specified as a coordinate transformation with the specified representation. The following representations are supported:

- Axis-Angle (AxAng) -  $[x \ y \ z \ \text{theta}]$
- Euler Angles (Eul) -  $[z \ y \ x]$ ,  $[z \ y \ z]$ , or  $[x \ y \ z]$
- Homogeneous Transformation (TForm) - 4-by-4 matrix
- Quaternion (Quat) -  $[w \ x \ y \ z]$
- Rotation Matrix (RotM) - 3-by-3 matrix
- Translation Vector (TrVec) -  $[x \ y \ z]$

To accommodate representations that only contain position or orientation information (TrVec or Eul, for example), you can specify two inputs or outputs to handle all transformation information. When you select the Homogeneous Transformation as an input or output, an optional `Show TrVec input/output port` parameter can be selected on the block mask to toggle the multiple ports.

### **TrVec — Translation vector**

three-element column vector

Translation vector, specified as a three-element column vector,  $[x \ y \ z]$ , which corresponds to a translation in the  $x$ ,  $y$ , and  $z$  axes respectively. This port can be used to input or output the translation information separately from the rotation vector.

#### **Dependencies**

You must select Homogeneous Transformation (TForm) for the opposite transformation port to get the option to show the additional TrVec port. Enable the port by clicking `Show TrVec input/output port`.

## Parameters

### Representation — Input or output representation

Axis-Angle | Euler Angles | Homogeneous Transformation | Rotation Matrix | Translation Vector | Quaternion

Select the representation for both the input and output port for the block. If you are using a transformation with only orientation information, you can also select the Show TrVec input/output port when converting to or from a homogeneous transformation.

### Show TrVec input/output port — Toggle TrVec port

off (default) | on

Toggle the TrVec input or output port when you want to specify or receive a separate translation vector for position information along with an orientation representation.

### Dependencies

You must select Homogeneous Transformation (TForm) for the opposite transformation port to get the option to show the additional TrVec port.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

rosmesssage

### Introduced in R2017b

## Current Time

Retrieve current ROS time or system time

**Library:** ROS Toolbox / ROS

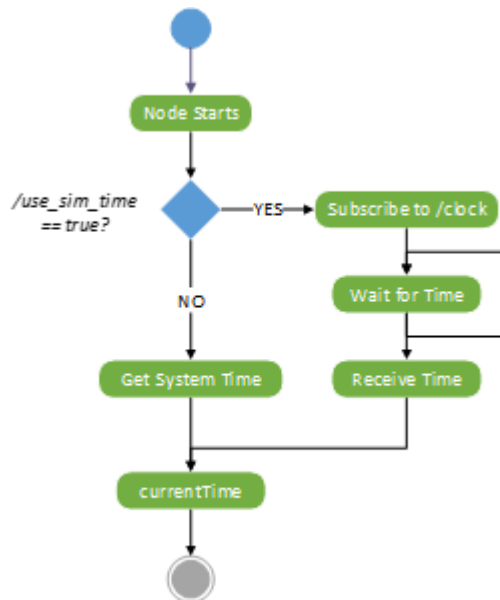


### Description

The Current Time block outputs the current ROS or system time. ROS Time is based on the system clock of your computer or the `/clock` topic being published on the ROS node.

Use this block to synchronize your simulation time with your connected ROS node.

If the `use_sim_time` ROS parameter is set to `true`, the block returns the simulation time published on the `/clock` topic. Otherwise, the block returns the system time of your machine.



### Ports

#### Output

##### Time – ROS time

bus | scalar

ROS time, returned as a bus signal or a scalar. The bus represents a `rosgraph_msgs/Clock` ROS message with `Sec` and `NSec` elements. The scalar is the ROS time in seconds. If no time has been received on the `/clock` topic, the block outputs `0`.

Data Types: bus | double

## Parameters

### Output format — Format of ROS time

bus (default) | double

Format of ROS Time output, specified as either bus or double.

### Sample time — Interval between outputs

-1 (default) | numeric scalar

Interval between outputs, specified as a numeric scalar.

For more information, see “Specify Sample Time” (Simulink).

## Tips

- To set the `use_sim_time` parameters and get time from a `/clock` topic:

Connect to a ROS network, then use the Set Parameter block or set the parameter in the MATLAB command window:

```
ptree = rosparam;  
set(ptree, '/use_sim_time', true)
```

Usually, the ROS node that publishes on the `/clock` topic sets up the parameter.

## See Also

### Blocks

Get Parameter | Publish | Set Parameter

### Functions

get | rosparam | rospublisher | rostime | set

### External Websites

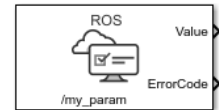
ROS Time

### Introduced in R2019b

## Get Parameter

Get values from ROS parameter server

**Library:** ROS Toolbox / ROS



### Description

The Get Parameter block outputs the value of the specified ROS parameter. The block uses the ROS node of the Simulink model to connect to the ROS network. This node is created when you run the model and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each sample hit, the block checks the ROS parameter server for the specified ROS parameter and outputs its value.

### Input/Output Ports

#### Output

##### Value — Parameter value

scalar | logical | uint8 array

Parameter value from the ROS network. The value depends on the **Data type** parameter.

##### ErrorCode — Status of ROS parameter

0 | 1 | 2 | 3

Status of ROS parameter, specified as one of the following:

- **0** — ROS parameter retrieved successfully. The retrieved value is output in the **Value** port.
- **1** — No ROS parameter with specified name found. If there is no known value, **Value** is set to the last received value or to **Initial value**.
- **2** — ROS parameter retrieved, but its type is different than the specified **Data type**. If there is no known value, Value is set to the last received value or to **Initial value**.
- **3** — For string parameters, the incoming string has been truncated based on the specified length.

##### Length — Length of string parameter

integer

Length of the string parameter, returned as an integer. This length is the number of elements of the uint8 array or the number of characters in the string that you cast to uint8.

---

**Note** When getting string parameters from the ROS network, an ASCII value of 13 returns an error due to its incompatible character type.

---

**Dependencies**

To enable this port, set the **Data type** to `uint8[] (string)`.

**Parameters****Source — Source for specifying the parameter name**

Select from ROS network | Specify your own

Source for specifying the parameter name as one of the following:

- **Select from ROS network** — Use **Select** to select a parameter name. The **Data type** parameter is set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a parameter name in **Name** and specify its data type in **Data type**. You must match a parameter name exactly.

**Name — Parameter name**

string

Parameter name to get from the ROS network, specified as a string. When **Source** is set to **Select from ROS network**, use **Select** to select an existing parameter. You must be connected to a ROS network to get a list of parameters. Otherwise, specify the parameter and data type.

Parameter name strings must follow the rules of ROS graph names. Valid names have these characteristics:

- The first character is an alpha character ([a-z|A-Z]), tilde (~), or forward slash (/).
- Subsequent characters are alphanumeric ([0-9|a-z|A-Z]), underscores(\_), or forward slashes (/).

**Data type — Data type of your parameter**

double | int32 | boolean | uint8[] (string)

Data type of your parameter, specified as a string. The `uint8[] (string)` enables the **Maximum length** parameter.

---

**Note** The `uint8[] (string)` data type is an array of ASCII values corresponding to the characters in a string. When getting string parameters, you can create a MATLAB Function block to compare the string to a desired parameter value. For more information, see “ROS Parameters in Simulink”.

---

Data Types: double | int32 | Boolean | uint8

**Maximum length — Maximum length of the uint8 array**

scalar

Maximum length of the `uint8` array, specified as a scalar. If the parameter string has a length greater than **Maximum length**, the **ErrorCode** output is set to 3.

**Dependencies**

To enable this port, set the **Data type** to `uint8[] (string)`.

**Initial value — Default parameter value output**

double | int32 | boolean | uint8

Default parameter value output from when an error occurs and no valid value has been received from the parameter server. The data type must match the specified **Data type**.

**Sample time – Interval between outputs**

inf (default) | scalar

Interval between outputs, specified as a scalar. This default value indicates that the block output never changes. Using this value speeds simulation and code generation by eliminating the need to recompute the block output. Otherwise, the block outputs a new blank message at each interval of **Sample time**.

For more information, see “Specify Sample Time” (Simulink).

**Show ErrorCode output port – Display error code output**

on | off

To enable error code output, select this parameter. When you clear this parameter, the **ErrorCode** output port is removed from the block. The status options are:

- **0** – ROS parameter retrieved successfully. The retrieved value is output in the **Value** port.
- **1** – No ROS parameter with specified name found. If there is no known value, **Value** is set to the last received value or to **Initial value**.
- **2** – ROS parameter retrieved, but its type is different than the specified **Data type**. If there is no known value, Value is set to the last received value or to **Initial value**.
- **3** – For string parameters, the incoming string has been truncated based on the specified length.

**See Also**

Set Parameter

**Topics**

“ROS Parameters in Simulink”

**External Websites**

ROS Parameter Server

ROS Graph Names

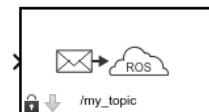
**Introduced in R2019b**



# Publish

Send messages to ROS network

**Library:** ROS Toolbox / ROS



## Description

The Publish block takes in as its input a Simulink nonvirtual bus that corresponds to the specified ROS message type and publishes it to the ROS network. It uses the node of the Simulink model to create a ROS publisher for a specific topic. This node is created when the model runs and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each sample hit, the block converts the **Msg** input from a Simulink bus signal to a ROS message and publishes it. The block does not distinguish whether the input is a new message but merely publishes it on every sample hit. For simulation, this input is a MATLAB ROS message. In code generation, it is a C++ ROS message.

## Input/Output Ports

### Input

#### Msg — ROS message

nonvirtual bus

ROS message, specified as a nonvirtual bus. To specify the type of ROS message, use the **Message type** parameter.

Data Types: bus

## Parameters

#### Topic source — Source for specifying topic name

Select from ROS network | Specify your own

Source for specifying the topic name, specified as one of the following:

- Select from ROS network — Use **Select** to select a topic name. The **Topic** and **Message type** parameters are set automatically. You must be connected to a ROS network.
- Specify your own — Enter a topic name in **Topic** and specify its message type in **Message type**. You must match a topic name exactly.

#### Topic — Topic name to publish to

string

Topic name to publish to, specified as a string. When **Topic source** is set to **Select from ROS network**, use **Select** to select a topic from the ROS network. You must be connected to a ROS network to get a list of topics. Otherwise, set **Topic source** to **Specify your own** and specify the topic you want.

**Message type — ROS message type**

string

ROS message type, specified as a string. Use **Select** to select from a full list of supported ROS messages. Service message types are not supported and are not included in the list.

**Length of publish queue — Message queue length**

1 (default) | integer

Message queue length in code generation, specified as an integer. In simulation, the message queue is always 1 and cannot be adjusted. To ensure each message is processed, use a smaller model step or only execute the model when publishing a new message.

**Tips**

You can also set the addresses for the ROS master and node host by clicking the **Configure network addresses** link in the block.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**See Also**

[Blank Message](#) | [Subscribe](#)

**Topics**

“Types of Composite Signals” (Simulink)

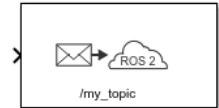
“ROS Simulink Interaction”

**Introduced in R2019b**

# Publish

Send messages to ROS 2 network

**Library:** ROS Toolbox / ROS 2



## Description

The Publish ROS 2 block takes in as its input a Simulink non-virtual bus that corresponds to the specified ROS 2 message type and publishes it to the ROS 2 network. It uses the node of the Simulink model to create a ROS 2 publisher for a specific topic. This node is created when the model runs and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each sample hit, the block converts the **Msg** input from a Simulink bus signal to a ROS 2 message and publishes it. The block does not distinguish whether the input is a new message but instead publishes it on every sample hit. For simulation, this input is a MATLAB ROS 2 message. In code generation, it is a C++ ROS 2 message.

## Ports

### Input

#### Msg — ROS message

non-virtual bus

ROS message, specified as a nonvirtual bus. To specify the type of ROS message, use the **Message type** parameter.

Data Types: bus

## Parameters

### Main

#### Topic source — Source for specifying topic name

Select from ROS network | Specify your own

Source for specifying the topic name, specified as one of the following:

- **Select from ROS network** — Use **Select** to select a topic name. The **Topic** and **Message type** parameters are set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a topic name in **Topic** and specify its message type in **Message type**. You must match a topic name exactly.

#### Topic — Topic name to publish to

string

Topic name to publish to, specified as a string. When **Topic source** is set to **Select from ROS network**, use **Select** to select a topic from the ROS network. You must be connected to a ROS 2

network to get a list of topics. Otherwise, set **Topic source** to `Specify your own` and specify the topic you want.

### **Message type – ROS message type**

`string`

ROS message type, specified as a string. Use **Select** to select from a full list of supported ROS messages. Service message types are not supported and are not included in the list.

### **Quality of Service (QoS)**

#### **History – Mode of storing messages in the queue**

`Keep last (default) | Keep all`

Determines the mode of storing messages in the queue. The queued messages will be sent to late-joining subscribers. If the queue fills with messages waiting to be processed, then old messages will be dropped to make room for new. If set to `'keeplast'`, the queue stores the number of messages set by the `Depth` parameter. If set to `'keepall'`, the queue stores all messages up to the MATLAB resource limits.

#### **Depth – Size of the message queue**

`1 (default) | positive scalar`

Number of messages stored in the message queue when `History` is set to `Keep last`.

#### **Reliability – Delivery guarantee of messages**

`Reliable (default) | Best effort`

Affects the guarantee of message delivery. If `Reliable`, then delivery is guaranteed, but may retry multiple times. If `Best effort`, then attempt delivery and do not retry.

#### **Durability – Persistence of messages**

`Volatile (default) | Transient local`

Affects persistence of messages in publishers, which allows late-joining subscribers to receive the number of old messages specified by `Depth`. If `Volatile`, then messages do not persist. If `Transient local`, then publisher will persist most recent messages.

## **Extended Capabilities**

### **C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

### **See Also**

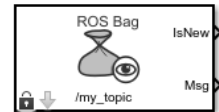
[Blank Message](#) | [Subscribe](#)

### **Introduced in R2019b**

# Read Data

Play back data from log file

**Library:** ROS Toolbox / ROS



## Description

The Read Data block plays back rosbag logfiles by outputting the most recent message from the log file based on the current simulation time. You must load a rosbag log file (.bag) and specify the **Topic** in the block mask to get a stream of messages from the file. Messages on this topic are output from the file in sync with the simulation time.

In the Read Data block mask, click **Load log file data** to specify a rosbag log file (.bag) to load. In the **Load Log File** window, specify a **Start time offset**, in seconds, to start playback at a certain point in the file. **Duration** specifies how long the block should play back this file in seconds. By default, the block outputs all messages for the specific **Topic** in the file.

## Ports

### Output

#### IsNew — New message indicator

0 | 1

New message indicator, returned as a logical. If the output is 1, then a new message was loaded from the rosbag file at that time. This output can be used to trigger subsystems for processing new messages received.

#### Msg — ROS message

nonvirtual bus

ROS message, returned as a nonvirtual bus. Messages are output in the order they are stored in the rosbag and synced with the simulation time.

Data Types: bus

## Parameters

#### Topic — Topic name to extract from log file

string

Topic name to extract from log file, specified as a string. This topic must exist in the loaded rosbag. Click the **Load rosbag file** Use **Select ...** to inspect the topics available and select a specific topic.

#### Sample time — Interval between outputs

-1 (default) | scalar

Interval between outputs, specified as a scalar. In simulation, the sample time follows simulation time and not actual wall-clock time.

This default value indicates that the block sample time is *inherited*.

For more information about the inherited sample time type, see “Specify Sample Time” (Simulink).

## **See Also**

### **Blocks**

[Publish](#) | [Read Image](#) | [Read Point Cloud](#) | [Subscribe](#)

### **Functions**

[readMessages](#) | [rosbag](#) | [select](#)

### **Topics**

“[Work with ROS Messages in Simulink®](#)”

“[Work with rosbag Logfiles](#)”

### **Introduced in R2019b**

# Read Image

Extract image from ROS Image message

**Library:** ROS Toolbox / ROS



## Description

The Read Image block extracts an image from a ROS Image or CompressedImage message. You can select the ROS message parameters of a topic active on a live ROS network or specify the message parameters separately. The ROS messages are specified as a nonvirtual bus. Use the Subscribe block output to receive a message from a ROS network and input the message to the Read Image block.

---

**Note** When reading ROS image messages from the network, the **Data** property of the message can exceed the maximum array length set in Simulink. To increase the maximum array length, click **Tools > Manage Array Lengths > Robot Operating System**, select the **Data** array, and increase the size based on the number of points in the image.

---

## Ports

### Input

#### Msg — ROS Image or CompressedImage message

nonvirtual bus

ROS Image or CompressedImage message, specified as a nonvirtual bus. You can use the Subscribe block to get a message from an active ROS network.

Data Types: bus

### Output

#### Image — Extracted image signal

$M$ -by- $N$ -by-3 matrix |  $M$ -by- $N$  matrix

Extracted image signal from ROS message, returned as an  $M$ -by- $N$ -by-3 matrix for color images, and an  $M$ -by- $N$  matrix for grayscale images. The matrix contains the pixel data from the **Data** property of the ROS message.

Data Types: single | double | int8 | int16 | int32 | uint8 | uint16

#### AlphaChannel — Alpha channel for image

$M$ -by- $N$  matrix

Alpha channel for image, returned as an  $M$ -by- $N$  matrix. This matrix is the same height and width as the image output and has values [0 1] to indicate the opacity of each corresponding pixel, with a value of 0 being completely transparent.

---

**Note** For `CompressedImage` messages, the Alpha channel returns all zeros if the `Show Alpha output port` is enabled.

---

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16`

### **ErrorCode — Error code for image conversion**

scalar

Error code for image conversion, returned as a scalar. The error code values are:

- 0 - Successfully converted the image message.
- 1 - Incorrect image encoding. Check that the incoming message encoding matches the `ImageEncoding` parameter.
- 2 - The dimensions of the image message exceed the limits specified in the `Maximum Image Size` parameter.
- 3 - The `Data` field of the image message was truncated. See “Manage Array Sizes for ROS Messages in Simulink” to increase the maximum length of the array.
- 4 - Image decompression failed.

Data Types: `uint8`

## **Parameters**

### **Maximum Image Size — Maximum image size**

[2000 2000] (default) | two-element vector

Maximum image size, specified as a two-element [height width] vector.

Click **Configure using ROS ...** to set this parameter automatically using an active topic on a ROS network. You must be connected to the ROS network.

### **Image Encoding — Image encoding**

`rgb8` (default) | `rgba8` | ...

Image encoding for the input `ImageMsg`. Select the supported encoding type which matches the `Encoding` property of the message. For more information about encoding types, see `readImage`.

### **Show Alpha output port — Toggle AlphaChannel port**

`off` (default) | `on`

Toggle Alpha channel output port if your encoding supports an Alpha channel.

### **Dependencies**

Only certain encoding types support alpha channels. The `ImageEncoding` parameter determines if this parameter appears in the block mask.

### **Show ErrorCode output port — Toggle ErrorCode port**

`on` (default) | `off`

Toggle the `ErrorCode` port to monitor errors.



**Output variable-size signals — Toggle variable-size signal output**

off (default) | on

Toggle variable-size signal output. Variable-sized signals should only be used if the image size is expected to change over time. For more information about variable sized signals, see “Variable-Size Signal Basics” (Simulink).

**See Also**

[Blank Message](#) | [CompressedImage](#) | [Image](#) | [Subscribe](#) | [readImage](#)

**Topics**

[“Manage Array Sizes for ROS Messages in Simulink”](#)

[“Variable-Size Signal Basics” \(Simulink\)](#)

**Introduced in R2019b**

## Read Point Cloud

Extract point cloud from ROS PointCloud2 message

**Library:** ROS Toolbox / ROS



### Description

The Read Point Cloud block extracts a point cloud from a ROS `PointCloud2` message. You can select the ROS message parameters of a topic active on a live ROS network or specify the message parameters separately. The ROS messages are specified as a nonvirtual bus. Use the Subscribe block to receive a message from a ROS network and input the message to the Read Point Cloud block.

---

**Note** When reading ROS point cloud messages from the network, the Data property of the message can exceed the maximum array length set in Simulink. To increase the maximum array length, click **Tools > Manage Array Lengths > Robot Operating System**, select the **Data** array, and increase the size based on the number of points in the point cloud.

---

### Ports

#### Input

##### Msg — ROS PointCloud2 message

nonvirtual bus

ROS `PointCloud2` message, specified as a nonvirtual bus. You can use the Subscribe block to get a message from the ROS network.

Data Types: bus

#### Output

##### XYZ — XYZ coordinates

matrix | multidimensional array

$x$ ,  $y$ , and  $z$  coordinates of the point cloud data, output as either an  $N$ -by-3 matrix or  $h$ -by- $w$ -by-3 multidimensional array.  $N$  is the number of points.  $h$  and  $w$  are the height and width of the image in pixels. To get the  $x$ ,  $y$ , and  $z$  coordinates as a multidimensional array, select the **Preserve point cloud structure** check box in the block mask parameters.

Data Types: single

##### RGB — RGB values for each point

matrix | multidimensional array

RGB values for each point of the point cloud data, output as either an  $N$ -by-3 matrix or  $h$ -by- $w$ -by-3 multidimensional array.  $N$  is the number of points.  $h$  and  $w$  are the height and width of the image in pixels. The RGB values correspond to the red, green, and blue color intensities with a range of [0

1]. To get the RGB values as a multidimensional array, select the `Preserve point cloud structure` check box in the block mask parameters.

Data Types: `double`

### **Intensity — Intensity values for each point**

`array` | `matrix`

Intensity values for each point of the point cloud data, output as either an array or a  $h$ -by- $w$  matrix.  $h$  and  $w$  are the height and width of the image in pixels. To get the intensity values as a matrix, select the `Preserve point cloud structure` check box in the block mask parameters.

Data Types: `single`

### **ErrorCode — Error code for image conversion**

`scalar`

Error code for image conversion, returned as a scalar. The error code values are:

- 0 - Successfully converted the point cloud message.
- 1 - The dimensions of the incoming point cloud exceed the limits set in `Maximum point cloud size`.
- 2 - One of the variable-length arrays in the incoming message was truncated. See “Manage Array Sizes for ROS Messages in Simulink” to increase the maximum length of the array.
- 3 - The X, Y, or Z field of the point cloud message is missing.
- 4 -The point cloud does not contain any RGB color data. You must have toggled `Show RGB output port` to on to get this error .
- 5 -The point cloud does not contain any intensity data. You must have toggled `Show Intensity output port` to on to get this error.
- 6 - The X, Y, or Z field of the point cloud message does not have the correct data type (`float32`).
- 7 - The RGB field of the point cloud message does not have the correct data type (`float32`).
- 8 - The `Intensity` field of the point cloud message does not have the correct data type (`float32`).

For certain error codes, data is truncated or populated with NaN values where appropriate.

Data Types: `uint8`

## **Parameters**

### **Maximum point cloud size — Maximum point cloud image size**

`[480 640]` (default) | two-element vector

Maximum point cloud image size, specified as a two-element `[height width]` vector.

Click **Configure using ROS ...** to set this parameter automatically using an active topic on a ROS network. You must be connected to the ROS network.

### **Preserve point cloud structure — Preserve point cloud data output shape**

`off` (default) | `on`

When this check box is selected, the cloud data output shape for `XYZ`, `RGB`, and `Intensity` are preserved. The outputs maintain the structure of the original image. Therefore, `XYZ` and `RGB` are output as multidimensional arrays, and `Intensity` is output as a matrix.

**Show RGB output port — Toggle RGB port**

off (default) | on

Select this check box to get RGB values for each point of the point cloud message from the RGB port. The RGB data must be supplied by the message.

**Show Intensity output port — Toggle Intensity port**

off (default) | on

Select this check box to get intensity values for each point of the point cloud message from the `Intensity` port. The intensity data must be supplied by the message.

**Show ErrorCode output port — Toggle ErrorCode port**

on (default) | off

Select this check box to monitor errors with the `ErrorCode` port.

**Output variable-size signals — Toggle variable-size signal output**

off (default) | on

Select this check box to output variable-size signals. Variable-sized signals should only be used if the image size is expected to change over time. For more information about variable sized signals, see “Variable-Size Signal Basics” (Simulink).

**See Also**

[Blank Message](#) | [PointCloud2](#) | [Subscribe](#)

**Topics**

“[Manage Array Sizes for ROS Messages in Simulink](#)”

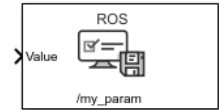
“[Variable-Size Signal Basics](#)” (Simulink)

**Introduced in R2019b**

# Set Parameter

Set values on ROS parameter server

**Library:** ROS Toolbox / ROS



## Description

The Set Parameter block sets the **Value** input to the specified name on the ROS parameter server. The block uses the ROS node of the Simulink model to connect to the ROS network. This node is created when you run the model and is deleted when the model terminates. If the model does not have a node, the block creates one.

## Input/Output Ports

### Input

#### Value — Parameter value

scalar | logical | uint8 array

Parameter value from the ROS network. The value depends on the **Data type** parameter.

#### Length — Length of string parameter

integer

Length of the string parameter, specified as an integer. This length is the number of elements of the uint8 array or the number of characters in the string that you cast to uint8.

---

**Note** When casting your string parameters to uint8, ASCII values 0-31 (control characters) return an error due to their incompatible character type.

---

### Dependencies

To enable this port, set the **Data type** to uint8[] (string).

## Parameters

#### Source — Source for specifying the parameter name

Select from ROS network | Specify your own

Source for specifying the parameter name as one of the following:

- **Select from ROS network** — Use **Select** to select a parameter name. The **Data type** parameter is set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a parameter name in **Name** and specify its data type in **Data type**. You must match a parameter name exactly.

**Name — Parameter name**

string

Parameter name to get from the ROS network, specified as a string. When **Source** is set to **Select from ROS network**, use **Select** to select an existing parameter. You must be connected to a ROS network to get a list of parameters. Otherwise, specify the parameter and data type.

Parameter name strings must follow the rules of ROS graph names. Valid names have these characteristics:

- The first character is an alpha character ([a-z|A-Z]), tilde (~), or forward slash (/).
- Subsequent characters are alphanumeric ([0-9|a-z|A-Z]), underscores(\_), or forward slashes (/).

**Data type — Data type of your parameter**

double | int32 | boolean | uint8[] (string)

Data type of your parameter, specified as a string.

---

**Note** The uint8[] (string) data type is an array of ASCII values corresponding to the characters in a string. When getting string parameters, you can create a MATLAB Function block to compare the string to a desired parameter value. For more information, see “ROS Parameters in Simulink”.

---

Data Types: double | int32 | Boolean | uint8

**See Also**

Get Parameter

**Topics**

“ROS Parameters in Simulink”

**External Websites**

ROS Parameter Servers

ROS Graph Names

**Introduced in R2019b**

# Subscribe

Receive messages from ROS network

**Library:** ROS Toolbox / ROS



## Description

The Subscribe block creates a Simulink nonvirtual bus that corresponds to the specified ROS message type. The block uses the node of the Simulink model to create a ROS subscriber for a specific topic. This node is created when the model runs and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each simulation step, the block checks if a new message is available on the specific topic. If a new message is available, the block retrieves the message and converts it to a Simulink bus signal. The **Msg** port outputs this new message. If a new message is not available, **Msg** outputs the last received ROS message. If a message has not been received since the start of the simulation, **Msg** outputs a blank message.

## Input/Output Ports

### Output

#### IsNew — New message indicator

0 | 1

New message indicator, returned as a logical. If the output is 1, then a new message was received since the last sample hit. This output can be used to trigger subsystems for processing new messages received in the ROS network.

#### Msg — ROS message

nonvirtual bus

ROS message, returned as a nonvirtual bus. The type of ROS message is specified in the **Message type** parameter. The Subscribe block outputs blank messages until it receives a message on the topic name you specify. These blank messages allow you to create and test full models before the rest of the network has been setup.

Data Types: bus

## Parameters

#### Topic source — Source for specifying topic name

Select from ROS network | Specify your own

Source for specifying the topic name, specified as one of the following:

- Select from ROS network — Use **Select** to select a topic name. The **Topic** and **Message type** parameters are set automatically. You must be connected to a ROS network.

- **Specify your own** — Enter a topic name in **Topic** and specify its message type in **Message type**. You must match a topic name exactly.

**Topic — Topic name to subscribe to**

string

Topic name to subscribe to, specified as a string. When **Topic source** is set to **Select** from ROS network, use **Select** to select a topic from the ROS network. You must be connected to a ROS network to get a list of topics. Otherwise, set **Topic source** to **Specify your own** and specify the topic you want.

**Message type — ROS message type**

string

ROS message type, specified as a string. Use **Select** to select from a full list of supported ROS messages. Service message types are not supported and are not included in the list.

**Sample time — Interval between outputs**

-1 (default) | scalar

Interval between outputs, specified as a scalar. In simulation, the sample time follows simulation time and not actual clock time.

This default value indicates that the block sample time is *inherited*.

For more information about the inherited sample time type, see “Specify Sample Time” (Simulink).

**Length of subscribe callback queue — Message queue length**

1 (default) | integer

Message queue length in code generation, specified as an integer. In simulation, the message queue is always 1 and cannot be adjusted. To ensure each message is caught, use a smaller model step or only execute the model if `IsNew` returns 1.

**Tips**

You can also set the addresses for the ROS master and node host by clicking the **Configure network addresses** link in the block.

**Extended Capabilities****C/C++ Code Generation**

Generate C and C++ code using Simulink® Coder™.

**See Also**

Blank Message | Publish

**Topics**

“Types of Composite Signals” (Simulink)

“ROS Simulink Interaction”

**Introduced in R2019b**



# Subscribe

Receive messages from ROS 2 network

**Library:** ROS Toolbox / ROS 2



## Description

The Subscribe block creates a Simulink non-virtual bus that corresponds to the specified ROS 2 message type. The block uses the node of the Simulink model to create a ROS 2 subscriber for a specific topic. This node is created when the model runs and is deleted when the model terminates. If the model does not have a node, the block creates one.

On each simulation step, the block checks if a new message is available on the specific topic. If a new message is available, the block retrieves the message and converts it to a Simulink bus signal. The **Msg** port outputs this new message. If a new message is not available, **Msg** outputs the last received ROS 2 message. If a message has not been received since the start of the simulation, **Msg** outputs a blank message.

## Ports

### Output

#### IsNew — New message indicator

0 | 1

New message indicator, returned as a logical. If the output is 1, then a new message was received since the last sample hit. This output can be used to trigger subsystems for processing new messages received in the ROS 2 network.

#### Msg — ROS 2 message

non-virtual bus

ROS 2 message, returned as a non-virtual bus. The type of ROS message is specified in the **Message type** parameter. The Subscribe ROS 2 block outputs blank messages until it receives a message on the topic name you specify. These blank messages allow you to create and test full models before the rest of the network has been setup.

Data Types: bus

## Parameters

### Main

#### Topic source — Source for specifying topic name

Select from ROS network | Specify your own

Source for specifying the topic name, specified as one of the following:

- **Select from ROS network** — Use **Select** to select a topic name. The **Topic** and **Message type** parameters are set automatically. You must be connected to a ROS network.
- **Specify your own** — Enter a topic name in **Topic** and specify its message type in **Message type**. You must match a topic name exactly.

#### **Topic — Topic name to subscribe to**

string

Topic name to subscribe to, specified as a string. When **Topic source** is set to **Select from ROS network**, use **Select** to select a topic from the ROS network. You must be connected to a ROS 2 network to get a list of topics. Otherwise, set **Topic source** to **Specify your own** and specify the topic you want.

#### **Message type — ROS 2 message type**

string

ROS 2 message type, specified as a string. Use **Select** to select from a full list of supported ROS 2 messages. Service message types are not supported and are not included in the list.

#### **Sample time — Interval between outputs**

-1 (default) | scalar

Interval between outputs, specified as a scalar. In simulation, the sample time follows simulation time and not actual wall-clock time.

This default value indicates that the block sample time is *inherited*.

For more information about the inherited sample time type, see “Specify Sample Time” (Simulink).

#### **Quality of Service (QoS)**

##### **History — Mode of storing messages in the queue**

Keep last (default) | Keep all

Determines the mode of storing messages in the queue. The queued messages will be sent to late-joining subscribers. If the queue fills with messages waiting to be processed, then old messages will be dropped to make room for new. When set to **Keep last**, the queue stores the number of messages set by the **Depth** property. Otherwise, when set to **Keep all**, the queue stores all messages up to the MATLAB resource limits.

##### **Depth — Size of the message queue**

1 (default) | positive scalar

Number of messages stored in the message queue when **History** is set to **Keep last**.

##### **Reliability — Delivery guarantee of messages**

Reliable (default) | Best effort

Affects the guarantee of message delivery. If **Reliable**, then delivery is guaranteed, but may retry multiple times. If **Best effort**, then attempt delivery and do not retry.

##### **Durability — Persistence of messages**

Volatile (default) | Transient local

Affects persistence of messages in publishers, which allows late-joining subscribers to receive the number of old messages specified by `Depth`. If `Volatile`, then messages do not persist. If `Transient local`, then publisher will retain the most recent messages.

## Extended Capabilities

### C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

### See Also

[Blank Message](#) | [Publish](#)

**Introduced in R2019b**

