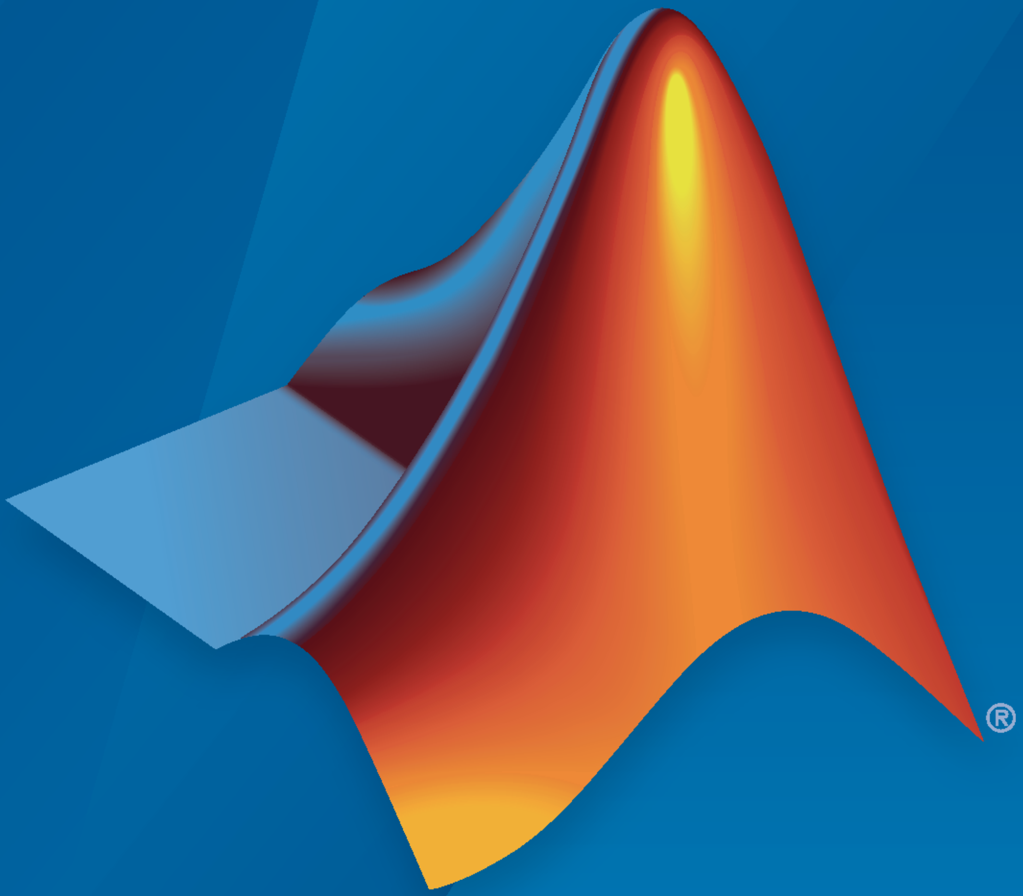
ROS Toolbox

Reference



MATLAB® & SIMULINK®

R2023a

MathWorks®

How to Contact MathWorks

Latest news:	`www.mathworks.com`
Sales and services:	`www.mathworks.com/sales_and_services`
User community:	`www.mathworks.com/matlabcentral`
Technical support:	`www.mathworks.com/support/contact_us`
Phone:	508-647-7000

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

ROS Toolbox Reference

Revision History

September 2019	Online only	New for Version 1.0 (R2019b)
March 2020	Online only	Revised for Version 1.1 (R2020a)
September 2020	Online only	Revised for Version 1.2 (R2020b)
March 2021	Online only	Revised for Version 1.3 (R2021a)
September 2021	Online only	Revised for Version 1.4 (R2021b)
March 2022	Online only	Revised for Version 1.5 (R2022a)
September 2022	Online only	Revised for Version 1.6 (R2022b)
March 2023	Online only	Revised for Version 2.0 (R2023a)

Contents

1

Apps

ROS Bag Viewer

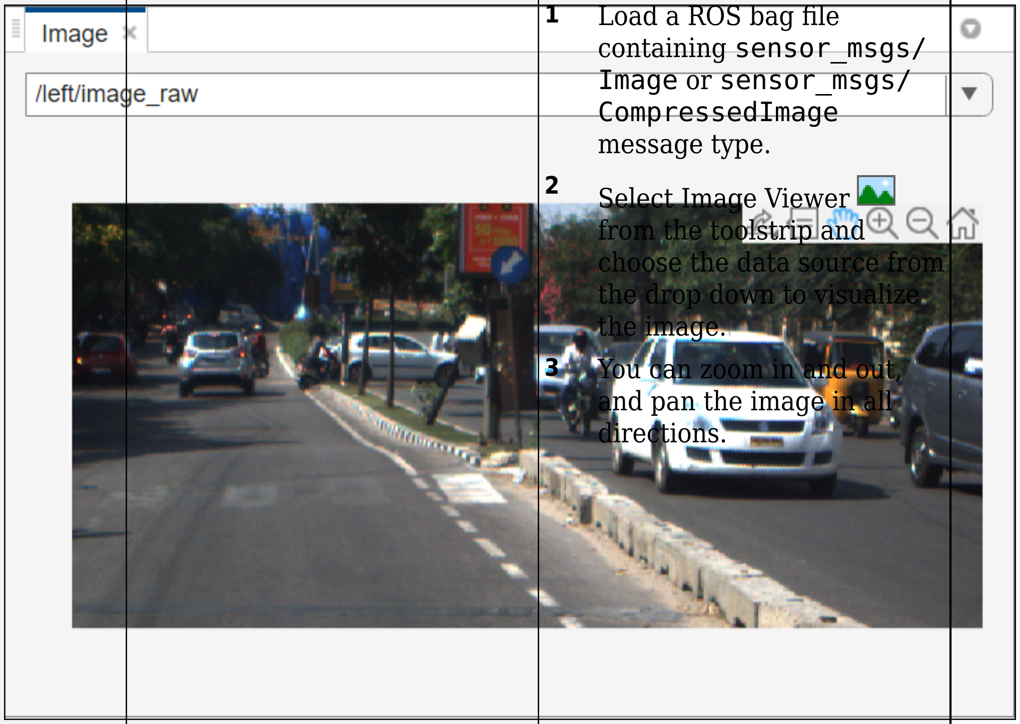
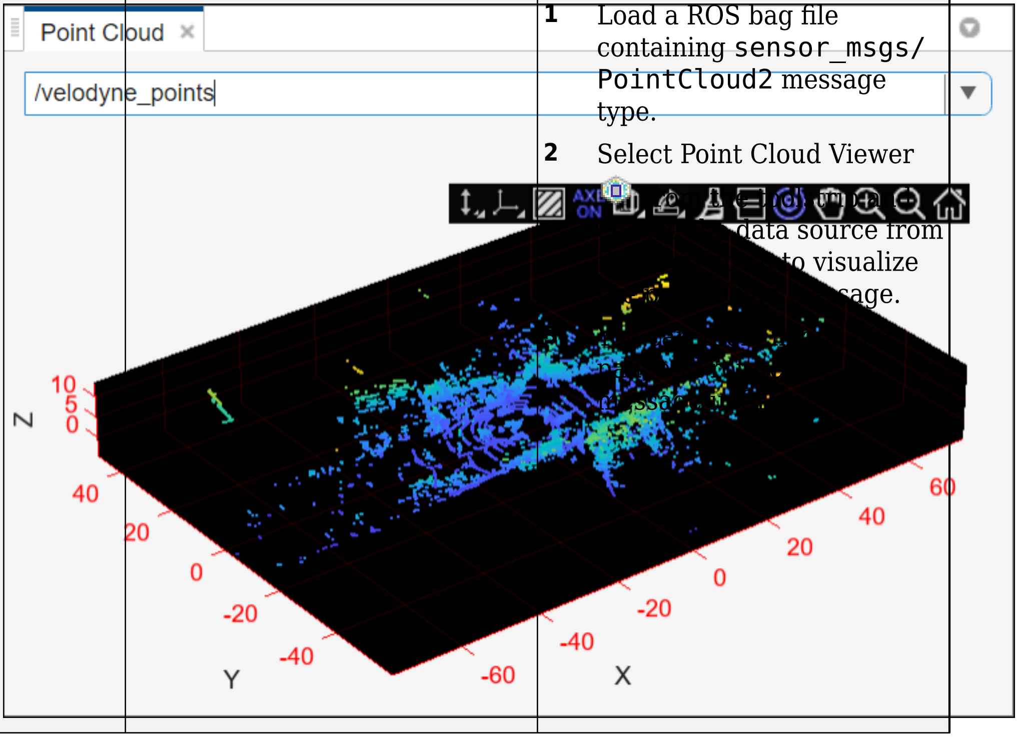
Visualize messages in ROS bag file

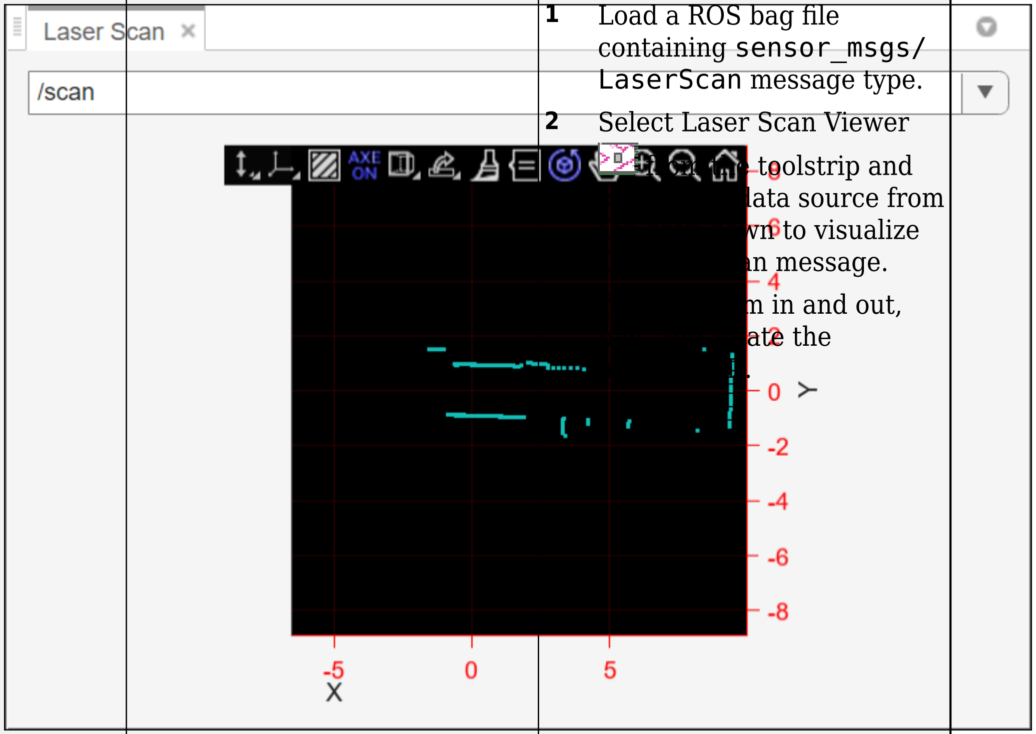
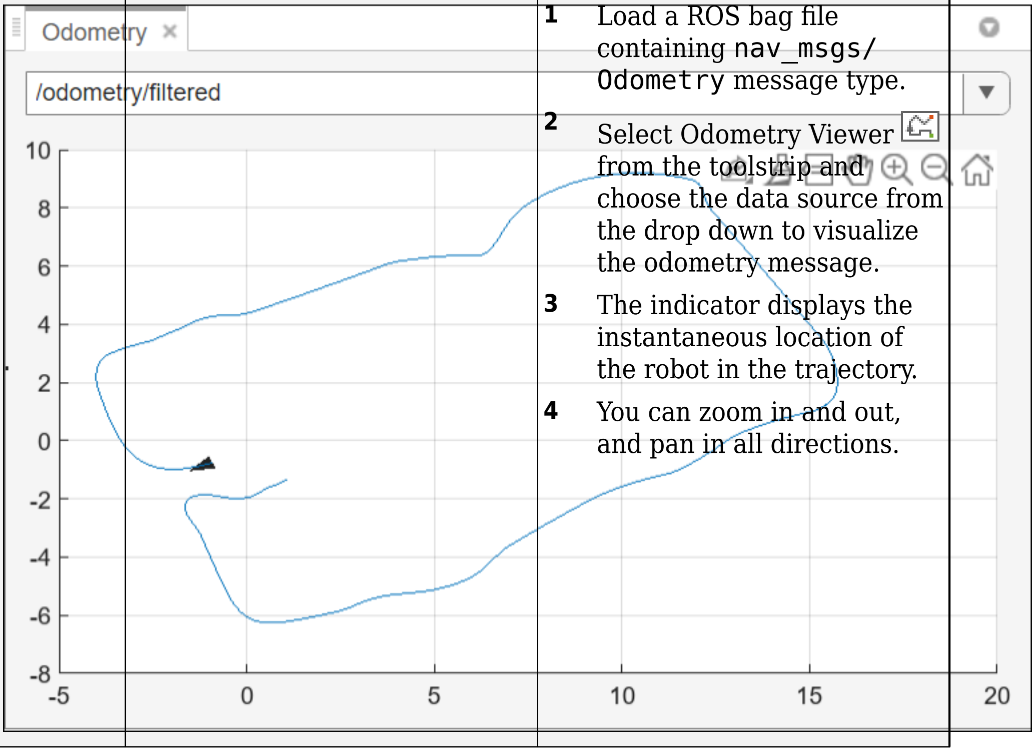
Description

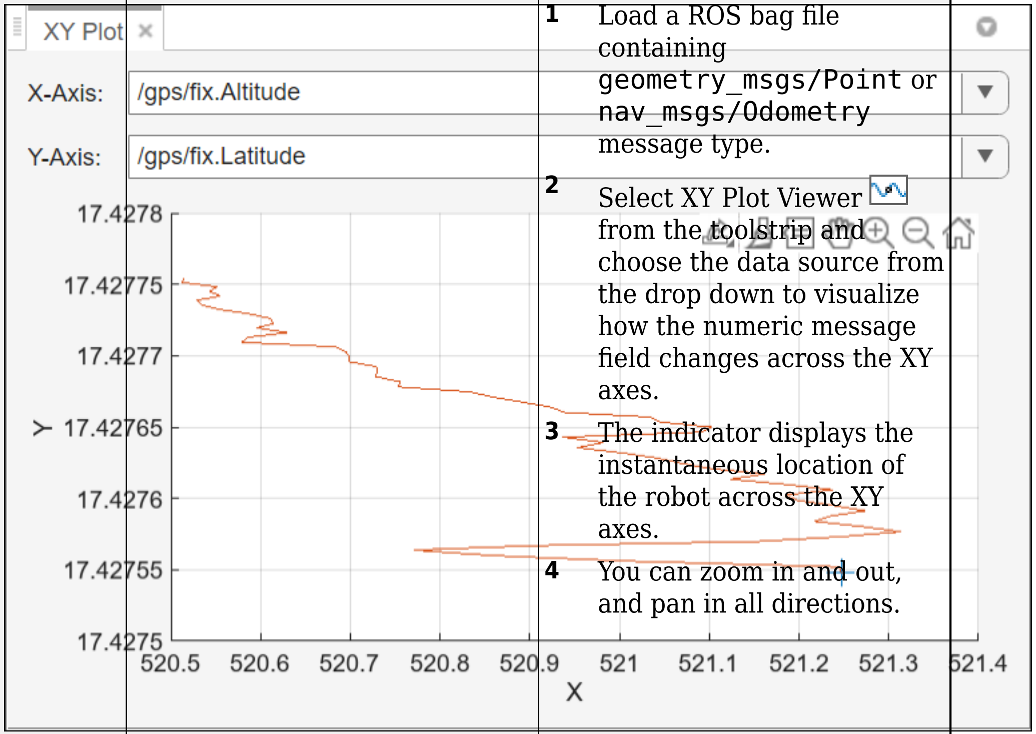
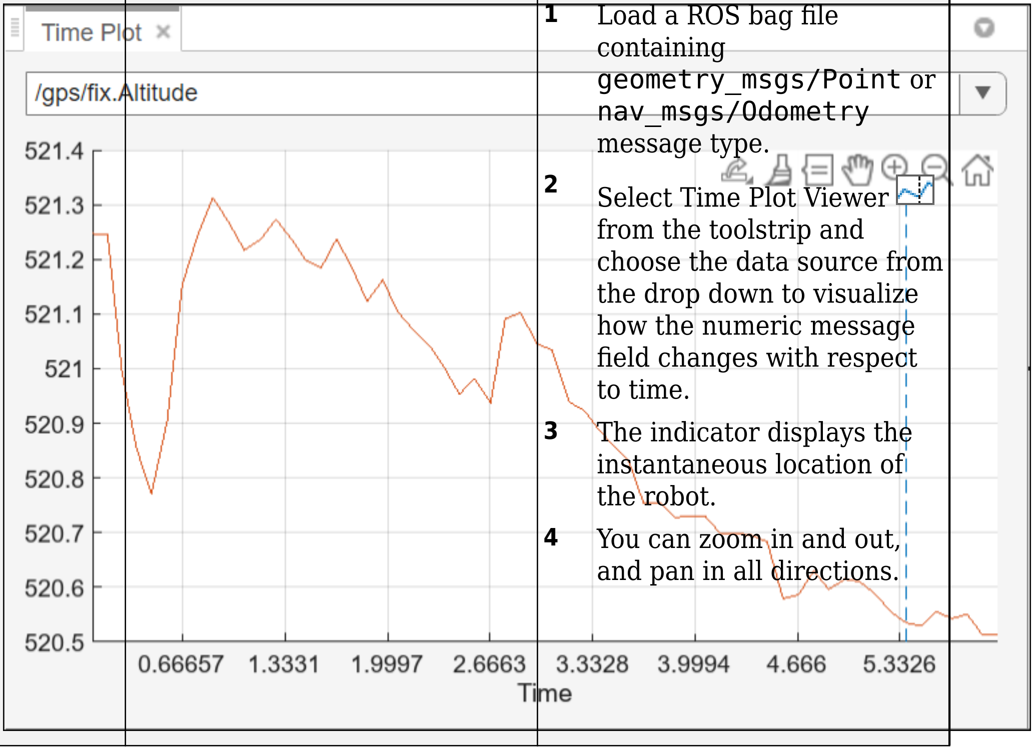
The **ROS Bag Viewer** app enables you to visualize messages in a ROS bag file. You can create multiple viewers within the app and visualize different ROS messages simultaneously.

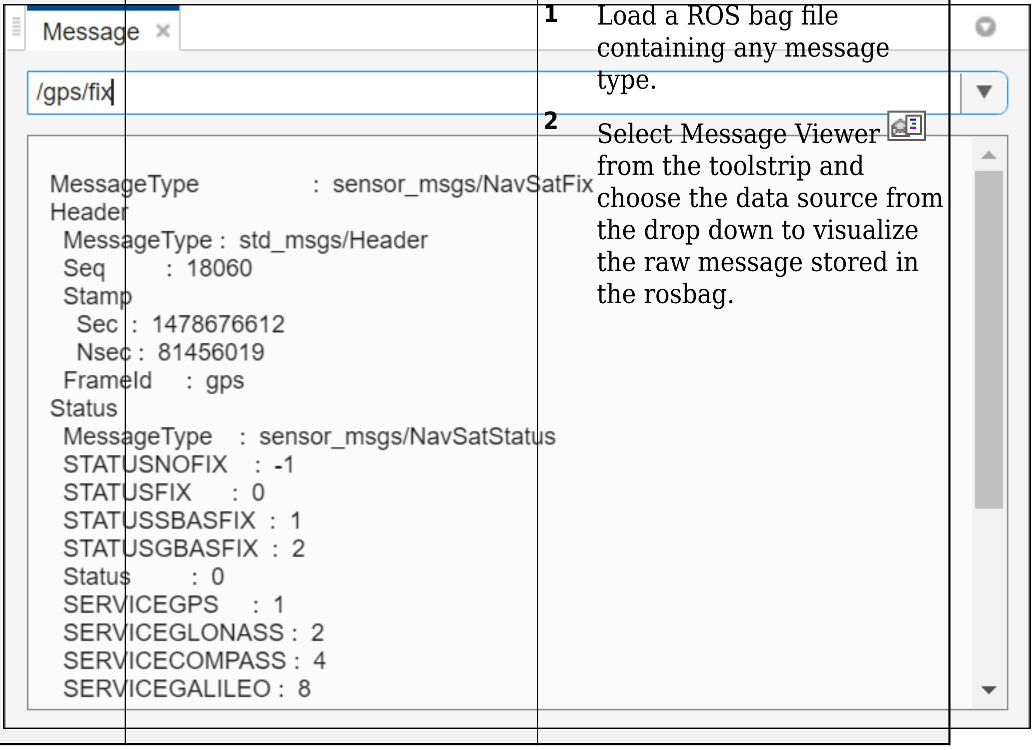
You can open the following viewers using the app, which support the given message types in the table.

Types of Viewer

Viewer	Interface	Description
Image Viewer		1 Load a ROS bag file containing `sensor_msgs/Image` or `sensor_msgs/CompressedImage` message type. 2 Select Image Viewer from the toolstrip and choose the data source from the drop down to visualize the image. 3 You can zoom in and out, and pan the image in all directions.
Point Cloud Viewer	 Point Cloud /velodyne_points Z 10 5 0 40 20 0 -20 -40 Y -60 -40 -20 0 20 40 60 X	1 Load a ROS bag file containing `sensor_msgs/PointCloud2` message type. 2 Select Point Cloud Viewer ... data source from ... to visualize ... age.

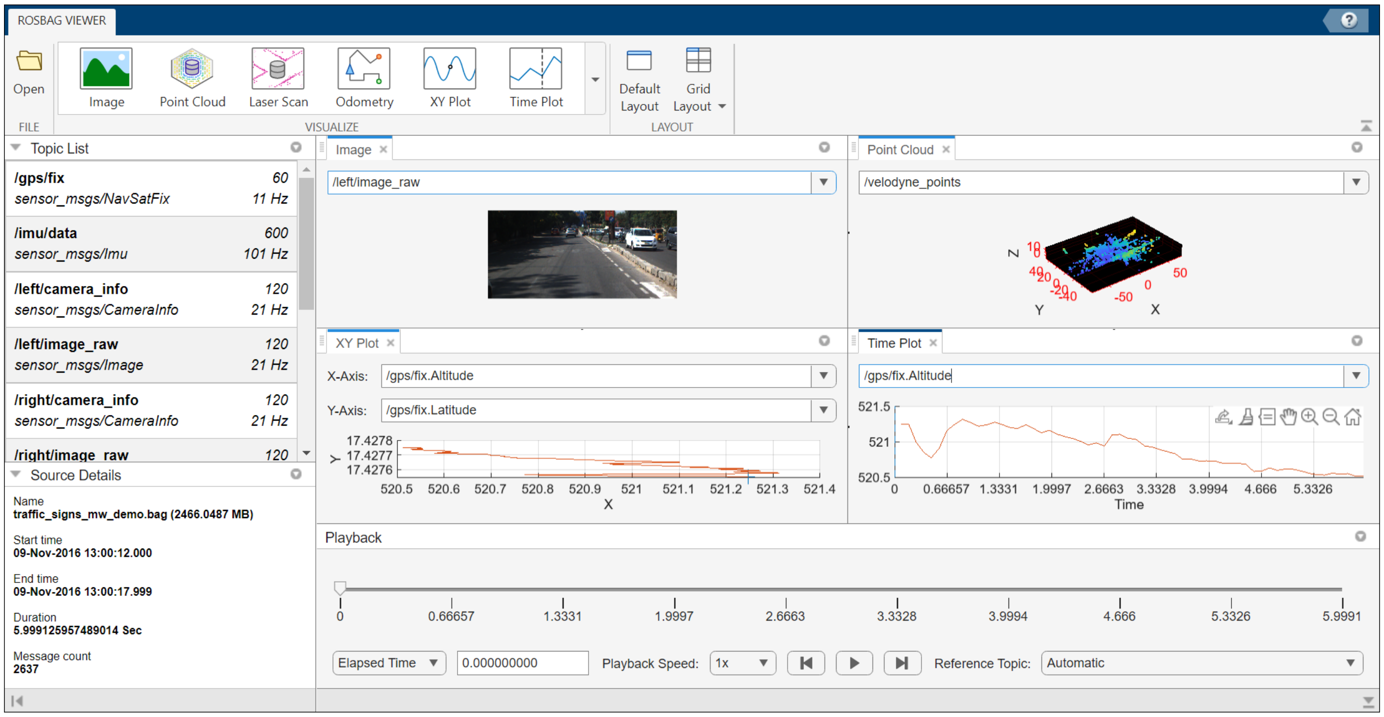
Viewer	Interface	Description
Laser Scan Viewer	 Laser Scan /scan 8 6 4 2 0 -2 -4 -6 -8 Y -5 0 5 X	**1** Load a ROS bag file containing `sensor_msgs/LaserScan` message type. **2** Select Laser Scan Viewer toolstrip and lata source from vn to visualize an message. m in and out, ate the
Odometry Viewer	 Odometry /odometry/filtered 10 8 6 4 2 0 -2 -4 -6 -8 -5 0 5 10 15 20	**1** Load a ROS bag file containing `nav_msgs/Odometry` message type. **2** Select Odometry Viewer from the toolstrip and choose the data source from the drop down to visualize the odometry message. **3** The indicator displays the instantaneous location of the robot in the trajectory. **4** You can zoom in and out, and pan in all directions.

Viewer	Interface	Description
XY Plot Viewer	 XY Plot X-Axis: /gps/fix.Altitude Y-Axis: /gps/fix.Latitude Y X	**1** Load a ROS bag file containing `geometry_msgs/Point` or `nav_msgs/Odometry` message type. **2** Select XY Plot Viewer from the toolstrip and choose the data source from the drop down to visualize how the numeric message field changes across the XY axes. **3** The indicator displays the instantaneous location of the robot across the XY axes. **4** You can zoom in and out, and pan in all directions.
Time Plot Viewer		**1** Load a ROS bag file containing `geometry_msgs/Point` or `nav_msgs/Odometry` message type. **2** Select Time Plot Viewer from the toolstrip and choose the data source from the drop down to visualize how the numeric message field changes with respect to time. **3** The indicator displays the instantaneous location of the robot. **4** You can zoom in and out, and pan in all directions.

Viewer	Interface	Description
Message Viewer		1 Load a ROS bag file containing any message type. 2 Select Message Viewer from the toolstrip and choose the data source from the drop down to visualize the raw message stored in the rosbag.

```
MessageType             :  sensor_msgs/NavSatFix
Header
  MessageType :  std_msgs/Header
  Seq         :  18060
  Stamp
    Sec  :  1478676612
    Nsec :  81456019
  FrameId     :  gps
Status
  MessageType    :  sensor_msgs/NavSatStatus
  STATUSNOFIX    :  -1
  STATUSFIX      :  0
  STATUSSBASFIX  :  1
  STATUSGBASFIX  :  2
  Status         :  0
  SERVICEGPS     :  1
  SERVICEGLONASS :  2
  SERVICECOMPASS :  4
  SERVICEGALILEO :  8
```

For each viewer, you can filter the supported messages in the bag file for visualization. You can fast forward, and rewind based on the message timestamp or elapsed time while playing the bag file. You can also pause, and play the bag frame-by-frame. The app also displays information about the bag file contents after loading the bag file. You can also save a snapshot of the visualization window at any particular instance of time.



/gps/fix sensor_msgs/NavSatFix	60 11 Hz
/imu/data sensor_msgs/Imu	600 101 Hz
/left/camera_info sensor_msgs/CameraInfo	120 21 Hz
/left/image_raw sensor_msgs/Image	120 21 Hz
/right/camera_info sensor_msgs/CameraInfo	120 21 Hz
/right/image_raw	120

Name
traffic_signs_mw_demo.bag (2466.0487 MB)

Start time
09-Nov-2016 13:00:12.000

End time
09-Nov-2016 13:00:17.999

Duration
5.999125957489014 Sec

Message count
2637

/left/image_raw

/velodyne_points

X-Axis: /gps/fix.Altitude

Y-Axis: /gps/fix.Latitude

/gps/fix.Altitude

Elapsed Time 0.000000000 Playback Speed: 1x Reference Topic: Automatic

Open the ROS Bag Viewer App

- MATLAB® Toolstrip: On the **Apps** tab, under **Robotics and Autonomous Systems**, click the app icon .
- MATLAB command prompt: Enter `rosbagViewer`.

Examples

- "Get Started with ROS Bag Viewer App"
- "Sign-following Robot with ROS in MATLAB"
- "Sign Following Robot with ROS in Simulink"

Programmatic Use

`rosbagViewer` opens the **ROS Bag Viewer** app, which enables you to visualize messages in a ROS bag file.

Version History

Introduced in R2023a

See Also

`rosbagwriter` | `rosbag` | `rosbagreader`

Topics

"Get Started with ROS Bag Viewer App"
"Sign-following Robot with ROS in MATLAB"
"Sign Following Robot with ROS in Simulink"

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 apples 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`.

Note apply will be removed. Use `rosApplyTransform` instead. For more information, see "ROS Message Structure Functions" on page 2-3

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 = rosmessage('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/PointCloud2`
- `geometry_msgs/QuaternionStamped`
- `geometry_msgs/Vector3Stamped`

Output Arguments

`tfentity` — Transformed ROS message
`Message` object handle

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

Version History
Introduced in R2019b

R2021a: ROS Message Structure Functions
Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name
Image CompressedImage	readImage writeImage	rosReadImage rosWriteImage
LaserScan	readCartesian readScanAngles lidarScan plot	rosReadCartesian rosReadScanAngles rosReadLidarScan rosPlot
PointCloud2	apply readXYZ readRGB readAllFieldNames readField scatter3	rosApplyTransform rosReadXYZ rosReadRGB rosReadAllFieldNames rosReadField rosPlot
Quaternion	readQuaternion	rosReadQuaternion
OccupancyGrid	readBinaryOccupanyGrid readOccupancyGrid writeBinaryOccupanyGrid writeOccupanyGrid	rosReadOccupancyGrid rosReadBinaryOccupancyGrid rosReadOccupancyGrid rosWriteBinaryOccupancyGrid rosWriteOccupancyGrid
Octomap	readOccupancyMap3D	rosReadOccupancyMap3D
PointStamped PoseStamped QuaternionStamped Vector3Stamped TransformStamped	apply	rosApplyTransform
All messages	showdetails	rosShowDetails

See Also
`rosApplyTransform`

call

Call ROS or ROS 2 service server and receive a response

Syntax

```
response = call(serviceclient)
response = call(serviceclient,requestmsg)
[response,status,statustext] = call(___)
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,status,statustext] = call(___)` returns a `status` indicating whether a `response` has been received successfully, and a `statustext` that captures additional information about the `status`, using any of the arguments from the previous syntaxes. If the call fails, the `status` will be `false` with an empty default `response` message, and this function will not display an error.

`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
```

```
Launching ROS Core...
..Done in 2.6739 seconds.
Initializing ROS master on http://172.30.131.134:59927.
Initializing global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and Ma
```

Set up a service server. Use structures for the ROS message data format.

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

Check whether the service server is available. If it is, wait for the service client to connect to the server.

```
if(isServerAvailable(client))
    [connectionStatus,connectionStatustext] = waitForServer(client)
end
```

```
connectionStatus = logical
   1


connectionStatustext = 
'success'
```

Call service server with default message.

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

```
response = struct with fields:
    MessageType: 'std_srvs/EmptyResponse'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response)
end
```

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

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and M
Shutting down ROS master on http://172.30.131.134:59927.
```

Call for Response Using Specific Request Message

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
...Done in 3.249 seconds.
Initializing ROS master on http://172.30.131.134:55557.
Initializing global node /matlab_global_node_93519 with NodeURI http://bat6234win64:63188/ and Ma
```

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_93519 with NodeURI http://bat6234win64:63188/ and 
Shutting down ROS master on http://172.30.131.134:55557.
```

Call ROS 2 Service Client With a Custom Callback Function

Create a sample ROS 2 network with two nodes.

```
node_1 = ros2node('node_1_service_client');
node_2 = ros2node('node_2_service_client');
```

Set up a service server and attach it to a ROS 2 node. Specify the callback function `flipstring`, which flips the input string. The callback function is defined at the end of this example.

```
server = ros2svcserver(node_1,'/test','test_msgs/BasicTypes',@flipString);
```

Set up a service client of the same service type and attach it to a different node.

```
client = ros2svcclient(node_2,'/test','test_msgs/BasicTypes');
```

Wait for the service client to connect to the server.

```
[connectionStatus,connectionStatustext] = waitForServer(client)
```

```
connectionStatus = logical
   1


connectionStatustext = 
'success'
```

Create a request message based on the client. Assign the string to the corresponding field in the message, `string_value`.

```
request = ros2message(client);
request.string_value = 'hello world';
```

Check whether the service server is available. If it is, send a service request and wait for a response. Specify that the service waits 3 seconds for a response.

```
if(isServerAvailable(client))
    response = call(client,request,'Timeout',3);
end
```

The response is a flipped string from the request message which you see in the `string_value` field.

```
response.string_value
```

```
ans = 
'dlrow olleh'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,request,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response.string_value)
end
```

```
dlrow olleh
```

The callback function used to flip the string is defined below.

```
function resp = flipString(req,resp)
% FLIPSTRING Reverses the order of a string in REQ and returns it in RESP.
resp.string_value = fliplr(req.string_value);
end
```

Input Arguments

`serviceclient` — Service client

`ros.ServiceClient` object handle | `ros2serviclient` object handle

ROS or ROS 2 service client, specified as a `ros.ServiceClient` or `ros2serviceclient` object handle, respectively.

`requestmsg` — Request message

`Message` object handle | structure

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

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see "ROS Message Structures" on page 2-10.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

`DataFormat` — Message format for ROS service clients
`"object"` (default) | `"struct"`

Message format for ROS service clients, specified as `"object"` or `"struct"`. Set this property on creation of the service client using the name-value input. For more information, see "ROS Message Structures" on page 2-10. This argument applies to ROS service clients only.

Output Arguments

`response` — Response message
`Message` object handle | structure

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

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see "ROS Message Structures" on page 2-10.

`status` — Status of the service call
`logical` scalar

Status of the service call, returned as a `logical` scalar. If the call fails, status will be `false`.

Note Use the `status` output argument when you use call for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

statustext — Status text associated with the service call status
character vector

Status text associated with the service call status, returned as one of the following:

- `'success'` — The service response was successfully received.
- `'input'` — The input to the function is invalid.
- `'timeout'` — The service response was not received within the specified timeout.
- `'unknown'` — The service response was not received due to unknown errors.

Tips

- ROS 2 service servers cannot communicate errors in callback execution directly to clients. In such situations, the servers only return the default response without any indication of failure. Hence, it is recommended to use try-catch blocks within the callback function, and set specific fields in the response message to communicate the success/failure of the callback execution on the server side.

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also
`rossvcclient` | `ros2svcclient` | `rosmessage` | `ros2message`

Topics
“Call and Provide ROS Services”
“Call and Provide ROS 2 Services”

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:

```
rosrun 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('192.168.203.133',11311)
```

```
Initializing global node /matlab_global_node_18287 with NodeURI http://192.168.203.1:55284/
```

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

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

```
goalMsg.Order = int32(4);
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg = struct with fields:
    MessageType: 'actionlib_tutorials/FibonacciResult'
       Sequence: [0 1 1 2 3]


resultState = 
'succeeded'
```

```
rosShowDetails(resultMsg)
```

```
ans =
    '
       MessageType :  actionlib_tutorials/FibonacciResult
       Sequence    :  [0, 1, 1, 2, 3]'
```

Send a new goal message without waiting.

```
goalMsg.Order = int32(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_18287 with NodeURI http://192.168.203.1:55284/
```

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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`cancelGoal` | `rosaction` | `sendGoal` | `sendGoalAndWait`

Topics
“ROS Actions Overview”
“Move a Turtlebot Robot Using ROS Actions”

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:

```
rosrun 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('192.168.203.133',11311)
```

```
Initializing global node /matlab_global_node_18287 with NodeURI http://192.168.203.1:55284/
```

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

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

```
goalMsg.Order = int32(4);
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg = struct with fields:
    MessageType: 'actionlib_tutorials/FibonacciResult'
       Sequence: [0 1 1 2 3]
```

```
resultState = 
'succeeded'
```

```
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  actionlib_tutorials/FibonacciResult
       Sequence    :  [0, 1, 1, 2, 3]'
```

Send a new goal message without waiting.

```
goalMsg.Order = int32(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_18287 with NodeURI http://192.168.203.1:55284/
```

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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.

- Usage in MATLAB Function block is not supported.

See Also
`cancelAllGoals` | `rosaction` | `sendGoal` | `sendGoalAndWait`

Topics
"ROS Actions Overview"
"Move a Turtlebot Robot Using ROS Actions"

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)

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

Description

TransformationTree Object

`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`.

BagSelection Object

`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`.

rosbagreader Object

`isAvailable = canTransform(bagreader,targetframe,sourceframe)` verifies if a transformation is available in a rosbag in `bagreader`.

`isAvailable = canTransform(bagreader,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.

```
rosinit;
```

```
Launching ROS Core...
....Done in 4.1192 seconds.
Initializing ROS master on http://192.168.125.1:56090.
Initializing global node /matlab_global_node_16894 with NodeURI http://HYD-KVENNAPU:63122/
```

```
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.X = 0.5;
tform.Transform.Rotation.Y = 0.5;
tform.Transform.Rotation.Z = 0.5;
tform.Transform.Rotation.W = 0.5;
```

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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_16894 with NodeURI http://HYD-KVENNAPU:63122/
Shutting down ROS master on http://192.168.125.1:56090.
```

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 = 36×1 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 = rosmessage('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
```

Get Transformations from rosbag File Using rosbagreader Object

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.

```
bagMsgs = rosbagreader("ros_turtlesim.bag")
```

```
bagMsgs = 
  rosbagreader with properties:

           FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\26\tp4cf343c3\ros-ex81142742\ros_tur
          StartTime: 1.5040e+09
            EndTime: 1.5040e+09
        NumMessages: 6089
    AvailableTopics: [6x3 table]
    AvailableFrames: {2x1 cell}
        MessageList: [6089x4 table]
```

Get a list of available frames.

```
frames = bagMsgs.AvailableFrames
```

```
frames = 2x1 cell
    {'turtle1'}
    {'world'  }
```

Get the latest transformation between two coordinate frames.

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

```
tf = 
  ROS TransformStamped message with properties:

     MessageType: 'geometry_msgs/TransformStamped'
          Header: [1x1 Header]
       Transform: [1x1 Transform]
    ChildFrameId: 'turtle1'

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

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

Input Arguments

`tftree` — ROS transformation tree
`TransformationTree` object

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

`bagSel` — Selection of rosbag messages
`BagSelection` object

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

`bagreader` — Index of messages in rosbag
`rosbagreader` object

Index of the messages in the rosbag, specified as a `rosbagreader` object.

`targetframe` — Target coordinate frame
string scalar | character vector

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

`sourcetime` — ROS or system time
scalar | `Time` object handle

ROS or system time, specified as a scalar or `Time` object handle. The scalar input is converted to a `Time` object using `rostime`.

Output Arguments

`isAvailable` — Indicator if transform exists
boolean

Indicator if transform exists, returned as a boolean. The function returns `false` if:

- `sourcetime` is outside the buffer window for a `tftree` object.
- `sourcetime` is outside the time of the `bagSel` or `bagreader` object.
- `sourcetime` is in the future.
- The transformation is not published yet.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the syntaxes with `TransformationTree` object as input.
- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also
`getTransform` | `transform` | `rosbag` | `rostf` | `waitForTransform` | `rosbagreader`

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.

Version History

Introduced in R2019b

See Also

`rosmessage` | `rosmsg`

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
```

```
Launching ROS Core...
..Done in 2.7086 seconds.
Initializing ROS master on http://172.30.131.134:51867.
Initializing global node /matlab_global_node_00691 with NodeURI http://bat6234win64:57285/ and Ma
```

```
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_00691 with NodeURI http://bat6234win64:57285/ and M
Shutting down ROS master on http://172.30.131.134:51867.
```

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)`.

Version History
Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`set` | `has` | `rosparam`

deleteFile

Delete file from device

Syntax

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

Description

`deleteFile(device,filename)` deletes the specified file from the ROS or ROS 2 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 or ROS 2 device

`rosdevice` object | `ros2device` object

ROS or ROS 2 device, specified as a `rosdevice` or `ros2device` object, respectively.

`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`

Version History

Introduced in R2019b

See Also

`rosdevice` | `ros2device` | `putFile` | `getFile` | `dir` | `openShell` | `system`

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 or ROS 2 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=6×4 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 or ROS 2 device
`rosdevice` object | `ros2device` object

ROS or ROS 2 device, specified as a `rosdevice` or `ros2device` object, respectively.

`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`)

Version History

Introduced in R2019b

See Also

`rosdevice` | `ros2device` | `putFile` | `getFile` | `deleteFile` | `openShell` | `system`

get

Get ROS parameter value

Syntax

```
pvalue = get(ptree)
pvalue = get(ptree,paramname)
pvalue = get(ptree,namespace)
[pvalue,status] = get(ptree,paramname)
pvalue = get(ptree,paramname,"DataType",ptype)
```

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`)

`[pvalue,status] = get(ptree,paramname)` returns the parameter values and the associated status. `status` indicates whether the `pvalue` successfully returned.

`pvalue = get(ptree,paramname,"DataType",ptype)` specifies the ROS parameter data type when generating code. The input parameter type value must match the existing parameter type else the function returns the `pvalue` as empty, `[]`, for the requested type.

Examples

Set and Get Parameter Value

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

```
rosinit
```

```
Launching ROS Core...
..Done in 2.2019 seconds.
Initializing ROS master on http://172.30.131.134:52728.
Initializing global node /matlab_global_node_14820 with NodeURI http://bat6234win64:53364/ and Ma
```

```
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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_14820 with NodeURI http://bat6234win64:53364/ and M
Shutting down ROS master on http://172.30.131.134:52728.
```

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)`.

`ptype` — ROS parameter data type
`'int32'` | `'logical'` | `'double'` | `'char'`

ROS parameter data type, specified as either `'int32'`, `'logical'`, `'double'`, or `'char'`.

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)

status — Status of ROS parameter value
true | false

Status of ROS parameter value, returned as true or false. If the status is false, the pvalue value is returned as empty, [].

Data Types: logical

Limitations

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

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

For code generation, only the following ROS data types are supported as values of parameters,

- 32-bit integer — int32
- Boolean — logical
- double — double
- strings — string scalar, string, or character vector, char

See Also

`set` | `rosparam`

getFile

Get file from device

Syntax

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

Description

`getFile(device,remoteSource)` copies the specified file from the ROS or ROS 2 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 or ROS 2 device

`rosdevice` object | `ros2device` object

ROS or ROS 2 device, specified as a `rosdevice` or `ros2device` object, respectively.

`remoteSource` — Path and name of file on the device

source path

Path and name of the file on the 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`

Version History

Introduced in R2019b

See Also

`rosdevice` | `ros2device` | `putFile` | `deleteFile` | `dir` | `openShell` | `system`

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)
tf = getTransform( ___ ,"DataFormat","struct")

tf = getTransform(bagreader,targetframe,sourceframe)
tf = getTransform(bagreader,targetframe,sourceframe,sourcetime)
tf = getTransform( ___ ,"DataFormat","struct")
```

Description

TransformationTree Object

`tf = getTransform(tftree,targetframe,sourceframe)` returns the latest known transformation between two coordinate frames in `tftree`. Create the `tftree` object using `rostf`, 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 `tftree` at the given source time. If the transformation is not available at that time, the function returns an error.

`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. Use this syntax with any of the input arguments in previous syntaxes.

BagSelection Object

`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 `rosbag`.

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

`tf = getTransform( ___ ,"DataFormat","struct")` returns the ROS `geometry_msgs/TransformStamped` message in the specified format.

rosbagreader Object

`tf = getTransform(bagreader,targetframe,sourceframe)` returns the latest transformation between two frames in the rosbag in `bagreader`.

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

`tf = getTransform( ___ ,"DataFormat","struct")` returns the ROS `geometry_msgs/TransformStamped` message in the specified format.

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 = 36×1 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 = rosmessage('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 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.

```
rosshutdown
```

```
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
```

Get Transformations from rosbag File Using rosbagreader Object

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.

```
bagMsgs = rosbagreader("ros_turtlesim.bag")
```

```
bagMsgs = 
  rosbagreader with properties:

           FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\26\tp4cf343c3\ros-ex81142742\ros_tur
          StartTime: 1.5040e+09
            EndTime: 1.5040e+09
        NumMessages: 6089
    AvailableTopics: [6x3 table]
    AvailableFrames: {2x1 cell}
        MessageList: [6089x4 table]
```

Get a list of available frames.

```
frames = bagMsgs.AvailableFrames
```

```
frames = 2x1 cell
    {'turtle1'}
    {'world'  }
```

Get the latest transformation between two coordinate frames.

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

```
tf = 
  ROS TransformStamped message with properties:

     MessageType: 'geometry_msgs/TransformStamped'
          Header: [1x1 Header]
       Transform: [1x1 Transform]
    ChildFrameId: 'turtle1'

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

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

Input Arguments

`tftree` — ROS transformation tree
`TransformationTree` object

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

`bagSel` — Selection of rosbag messages
`BagSelection` object

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

`bagreader` — Index of messages in rosbag
`rosbagreader` object

Index of the messages in the rosbag, specified as a `rosbagreader` object.

`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 you set the `use_sim_time` ROS parameter 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 the transform, specified as a scalar in seconds. The function returns an error if the timeout is reached and no transform becomes available.

`"DataFormat"` — Transformed ROS messages format
`"object"` (default) | `"struct"`

Transformed ROS messages format, returned as message `"object"` of specific type or message `"struct"` with compatible fields. Using `"struct"` can be faster than using message `"object"`.

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).

Version History

Introduced in R2019b

R2018a: Empty Transforms
Behavior changed in R2018a

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.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the syntaxes with `TransformationTree` object as input.
- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`transform` | `waitForTransform` | `rostf` | `canTransform` | `rosbag` | `rosbagreader`

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
```

```
Launching ROS Core...
..Done in 2.6842 seconds.
Initializing ROS master on http://172.30.131.134:58241.
Initializing global node /matlab_global_node_21986 with NodeURI http://bat6234win64:51299/ and Ma
```

```
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_21986 with NodeURI http://bat6234win64:51299/ and M
Shutting down ROS master on http://172.30.131.134:58241.
```

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`.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`get` | `search` | `set` | `rosparam`

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 = '192.168.203.131';
d = rosdevice(ipaddress,'user','password')
```

```
d = 
  rosdevice with properties:

      DeviceAddress: '192.168.203.131'
           Username: 'user'
          ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws'
     AvailableNodes: {'voxel_grid_filter_sl'}
```

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)
pause(2)
running = isCoreRunning(d)
```

```
running = logical
   0
```

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`.

Version History

Introduced in R2019b

See Also

`rosdevice` | `runCore` | `stopCore`

Topics
"Generate a Standalone ROS Node from Simulink"

isNodeRunning

Determine if ROS or ROS 2 node is running

Syntax

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

Description

`running = isNodeRunning(device,modelName)` determines if the ROS or ROS 2 node associated with the specified Simulink® model is running on the specified `rosdevice` or `ros2device`, `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 = 1×2 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 or ROS 2 device

`rosdevice` object | `ros2device` object

ROS or ROS 2 device, specified as a `rosdevice` or `ros2device` object, respectively.

`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 the ROS or ROS 2 node is running

`true` | `false`

Status of whether the ROS or ROS 2 node is running, returned as `true` or `false`.

Version History

Introduced in R2019b

See Also

`rosdevice` | `ros2device` | `runNode` | `stopNode`

Topics

“Generate a Standalone ROS Node from Simulink”
“Generate a Standalone ROS 2 Node from Simulink”

isServerAvailable

Determine if ROS or ROS 2 service server is available

Syntax

```
status = isServerAvailable(client)
```

Description

`status = isServerAvailable(client)` determines whether a service server with the same service name as `client` is available and returns a `status` accordingly.

Examples

Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6739 seconds.
Initializing ROS master on http://172.30.131.134:59927.
Initializing global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and Ma
```

Set up a service server. Use structures for the ROS message data format.

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

Check whether the service server is available. If it is, wait for the service client to connect to the server.

```
if(isServerAvailable(client))
    [connectionStatus,connectionStatustext] = waitForServer(client)
end
```

```
connectionStatus = logical
   1


connectionStatustext = 
'success'
```

Call service server with default message.

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

```
response = struct with fields:
    MessageType: 'std_srvs/EmptyResponse'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response)
end
```

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

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and M
Shutting down ROS master on http://172.30.131.134:59927.
```

Call ROS 2 Service Client With a Custom Callback Function

Create a sample ROS 2 network with two nodes.

```
node_1 = ros2node('node_1_service_client');
node_2 = ros2node('node_2_service_client');
```

Set up a service server and attach it to a ROS 2 node. Specify the callback function `flipstring`, which flips the input string. The callback function is defined at the end of this example.

```
server = ros2svcserver(node_1,'/test','test_msgs/BasicTypes',@flipString);
```

Set up a service client of the same service type and attach it to a different node.

```
client = ros2svcclient(node_2,'/test','test_msgs/BasicTypes');
```

Wait for the service client to connect to the server.

```
[connectionStatus,connectionStatustext] = waitForServer(client)
```

```
connectionStatus = logical
   1


connectionStatustext = 
'success'
```

Create a request message based on the client. Assign the string to the corresponding field in the message, `string_value`.

```
request = ros2message(client);
request.string_value = 'hello world';
```

Check whether the service server is available. If it is, send a service request and wait for a response. Specify that the service waits 3 seconds for a response.

```
if(isServerAvailable(client))
    response = call(client,request,'Timeout',3);
end
```

The response is a flipped string from the request message which you see in the `string_value` field.

```
response.string_value
```

```
ans = 
'dlrow olleh'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,request,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response.string_value)
end
```

```
dlrow olleh
```

The callback function used to flip the string is defined below.

```
function resp = flipString(req,resp)
% FLIPSTRING Reverses the order of a string in REQ and returns it in RESP.
resp.string_value = fliplr(req.string_value);
end
```

Input Arguments

`client` — ROS service client

`ros.ServiceClient` object handle | `ros2serviceclient` object handle

ROS or ROS 2 service client, specified as a `ros.ServiceClient` or `ros2serviceclient` object handle, respectively. This service client enables you to send requests to the service server.

Output Arguments

`status` — Status of service server availability

`logical` scalar

Status of service server availability, returned as a `logical` scalar. If a server of the same name and type as `client` is not available, `status` will be `false`.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`rossvcclient` | `rossvcserver` | `ros2svcclient` | `ros2svcserver` | `call` | `rosservice`

Topics
"Call and Provide ROS Services"
"Call and Provide ROS 2 Services"

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® command shell on the ROS or ROS 2 device. When prompted, enter a user name and password.

Examples

Open Command Shell on ROS Device

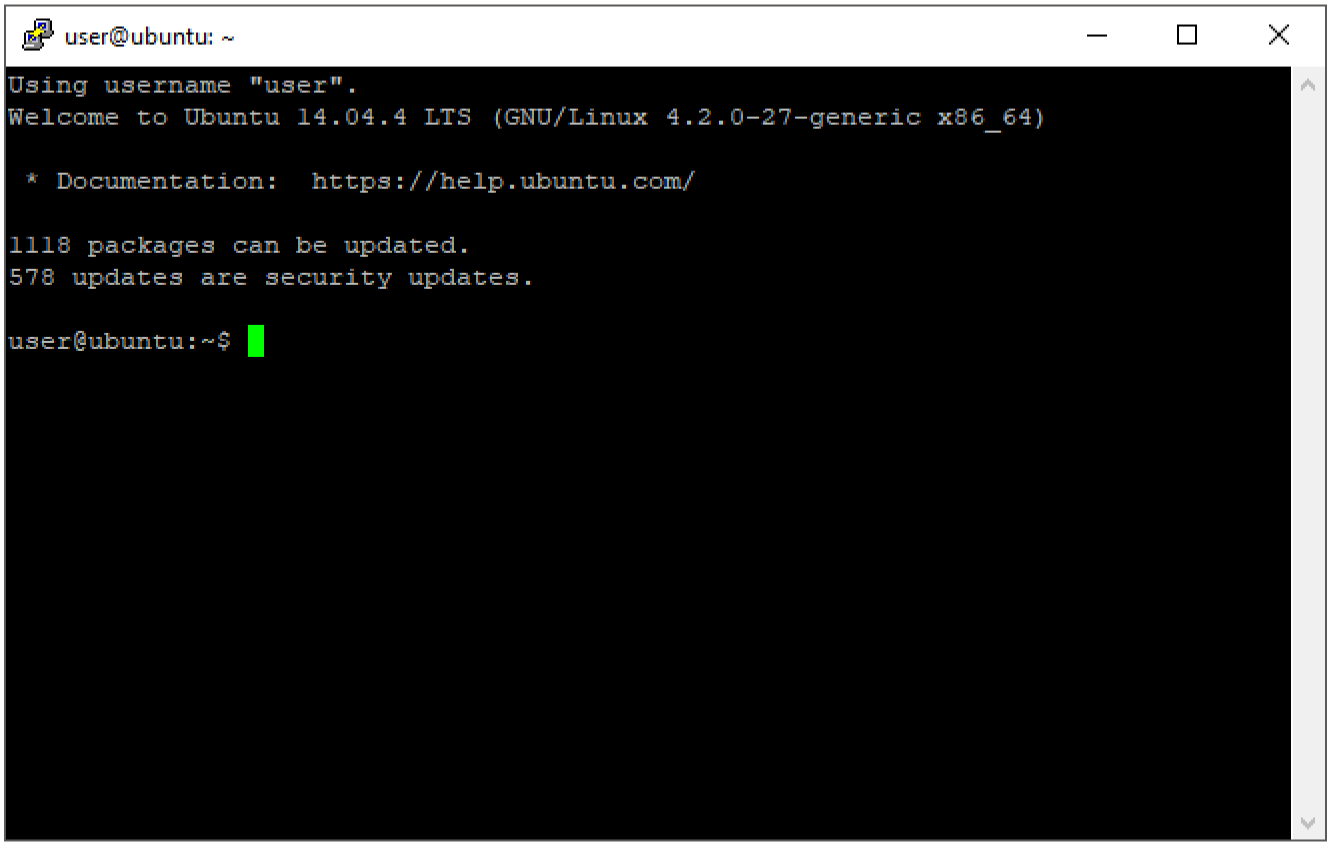
Connect to a ROS device and open the command shell on your host computer.

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);
```



```
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 or ROS 2 device
`rosdevice` object | `ros2device` object

ROS or ROS 2 device, specified as a `rosdevice` or `ros2device` object, respectively.

Version History

Introduced in R2019b

See Also

`rosdevice` | `ros2device` | `putFile` | `getFile` | `deleteFile` | `dir` | `system`

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.

Note plot will be removed. Use `rosPlot` instead. For more information, see “ROS Message Structure Functions” on page 2-63

`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.

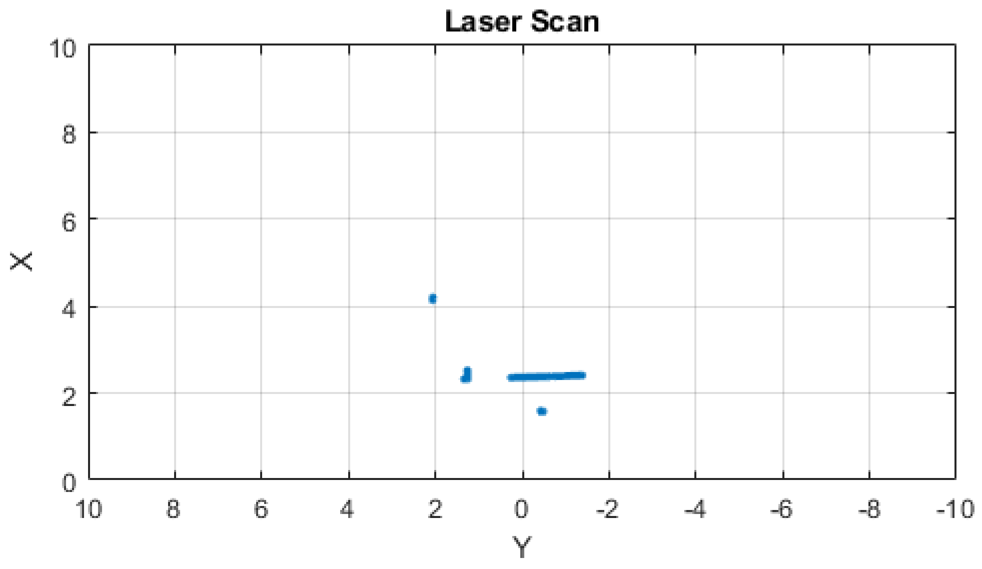
```
rosinit('192.168.17.129')
```

```
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.

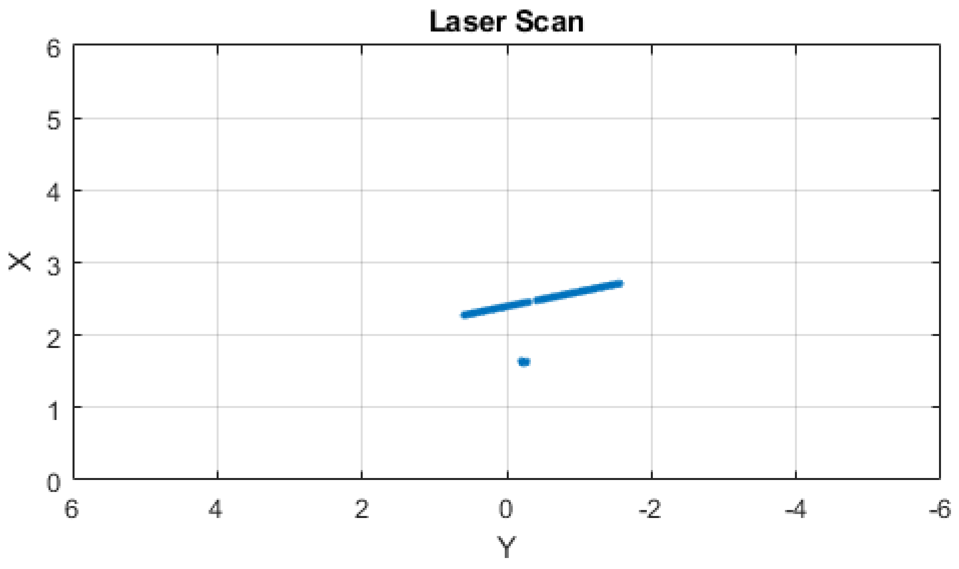
```
rosinit('192.168.17.129')
```

```
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.

```
rosshutdown
```

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

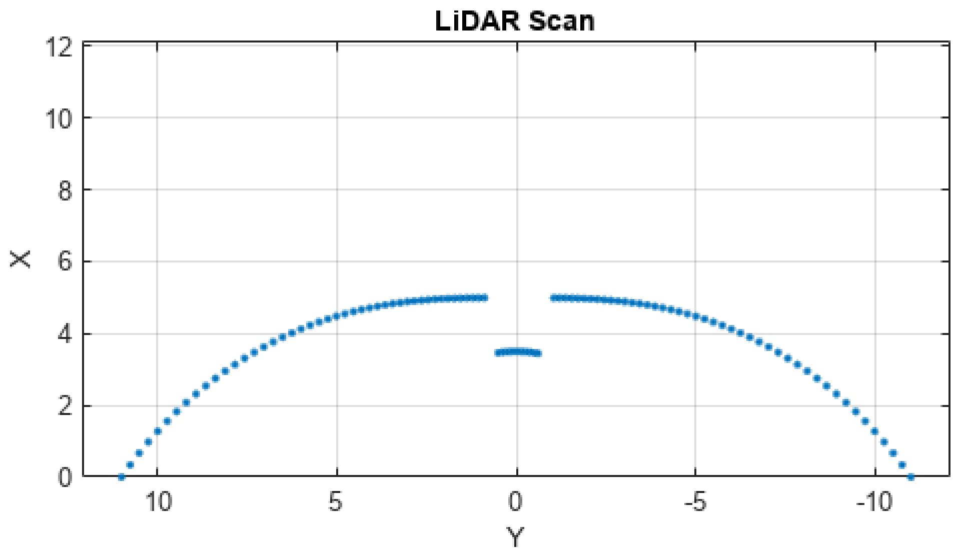
Plot Lidar Scan and Remove Invalid Points

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));
```

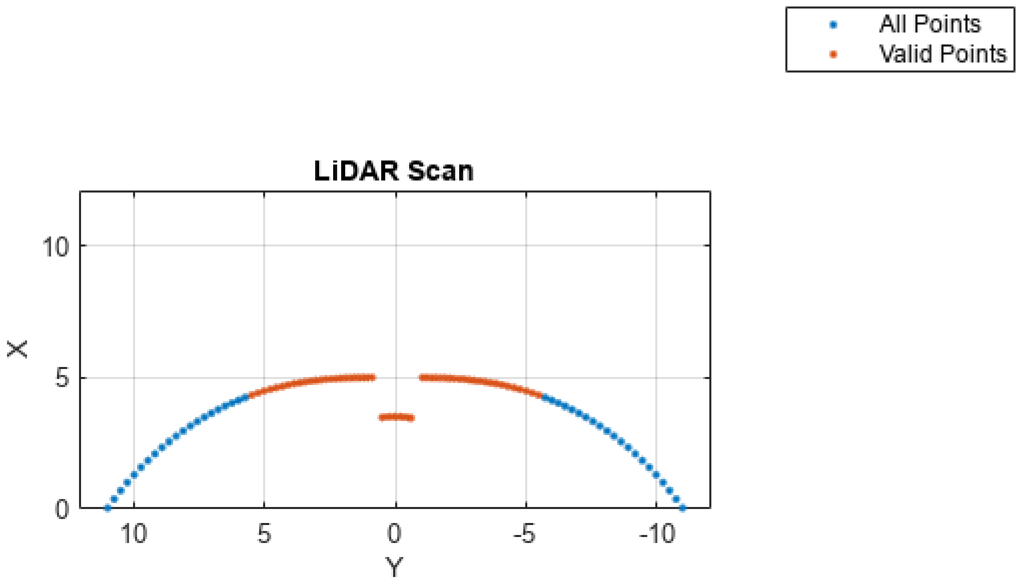
Create a lidar scan by specifying the ranges and angles. Plot all points of the lidar scan.

```
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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions
Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name
`Image` `CompressedImage`	`readImage` `writeImage`	`rosReadImage` `rosWriteImage`

`LaserScan`	`readCartesian`
	`readScanAngles`
	`lidarScan`
	`plot`
		`rosReadCartesian`
		`rosReadScanAngles`
		`rosReadLidarScan`
		`rosPlot`

Message Types	Object Function Name	New Function Name
PointCloud2	apply readXYZ readRGB readAllFieldNames readField scatter3	rosApplyTransform rosReadXYZ rosReadRGB rosReadAllFieldNames rosReadField rosPlot
Quaternion	readQuaternion	rosReadQuaternion
OccupancyGrid	readBinaryOccupanyGrid readOccupancyGrid writeBinaryOccupanyGrid writeOccupanyGrid	rosReadOccupancyGrid rosReadBinaryOccupancyGrid rosReadOccupancyGrid rosWriteBinaryOccupancyGrid rosWriteOccupancyGrid
Octomap	readOccupancyMap3D	rosReadOccupancyMap3D
PointStamped PoseStamped QuaternionStamped Vector3Stamped TransformStamped	apply	rosApplyTransform
All messages	showdetails	rosShowDetails

See Also

rosReadLidarScan | rosReadCartesian | rosPlot

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 or the home directory on the ROS 2 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 or ROS 2 device

`rosdevice` object | `ros2device` object

ROS or ROS 2 device, specified as a `rosdevice` or `ros2device` object, respectively.

`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`

Version History

Introduced in R2019b

See Also

`rosdevice` | `ros2device` | `getFile` | `deleteFile` | `dir` | `openShell` | `system`

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`.

Note readAllFieldNames will be removed. Use `rosReadAllFieldNames` instead. For more information, see "ROS Message Structure Functions" on page 2-68

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.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions
Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name

`Image`
 	
`CompressedImage`
	`readImage`
	
	`writeImage`
		`rosReadImage`
		
		`rosWriteImage`

`LaserScan`	`readCartesian`
	
	`readScanAngles`
	
	`lidarScan`
	
	`plot`
		`rosReadCartesian`
		
		`rosReadScanAngles`
		
		`rosReadLidarScan`
		
		`rosPlot`

`PointCloud2`	`apply`
	
	`readXYZ`
	
	`readRGB`
	
	`readAllFieldNames`
	
	`readField`
	
	`scatter3`
		`rosApplyTransform`
		
		`rosReadXYZ`
		
		`rosReadRGB`
		
		`rosReadAllFieldNames`
		
		`rosReadField`
		
		`rosPlot`

`Quaternion`	`readQuaternion`	`rosReadQuaternion`

`OccupancyGrid`	`readBinaryOccupanyGrid`
	
	`readOccupancyGrid`
	
	`writeBinaryOccupanyGrid`
	
	`writeOccupanyGrid`
		`rosReadOccupancyGrid`
		
		`rosReadBinaryOccupancyGrid`
		
		`rosReadOccupancyGrid`
		
		`rosWriteBinaryOccupancyGrid`
		
		`rosWriteOccupancyGrid`

`Octomap`	`readOccupancyMap3D`	`rosReadOccupancyMap3D`

Message Types	Object Function Name	New Function Name

`PointStamped`
`PoseStamped`
`QuaternionStamped`
`Vector3Stamped`
`TransformStamped`
	`apply`	`rosApplyTransform`

All messages	`showdetails`	`rosShowDetails`

See Also

`rosReadXYZ` | `rosReadField` | `rosReadRGB` | `rosReadCartesian`

readBinaryOccupancyGrid

Read binary occupancy grid

Syntax

```
map = readBinaryOccupancyGrid(msg)
map = readBinaryOccupancyGrid(msg,thresh)
map = readBinaryOccupancyGrid(msg,thresh,val)
```

Description

`map = readBinaryOccupancyGrid(msg)` returns a object by reading the data inside a ROS message, `msg`, which must be a `'nav_msgs/OccupancyGrid'` message. All message data values greater than or equal to the occupancy threshold are set to occupied, `1`, in the map. All other values, including unknown values (`-1`) are set to unoccupied, `0`, in the map.

`map = readBinaryOccupancyGrid(msg,thresh)` specifies a threshold, `thresh`, for occupied values. All values greater than or equal to the threshold are set to occupied, `1`. All other values are set to unoccupied, `0`.

`map = readBinaryOccupancyGrid(msg,thresh,val)` specifies a value to set for unknown values (`-1`). By default, all unknown values are set to unoccupied, `0`.

Input Arguments

`msg` — `'nav_msgs/OccupancyGrid'` ROS message
`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as a `OccupancyGrid` object handle.

`thresh` — Threshold for occupied values
50 (default) | scalar

Threshold for occupied values, specified as a scalar. Any value greater than or equal to the threshold is set to occupied, `1`. All other values are set to unoccupied, `0`.

Data Types: `double`

`val` — Value to replace unknown values
0 (default) | 1

Value to replace unknown values, specified as either `0` or `1`. Unknown message values (`-1`) are set to the given value.

Data Types: `double` | `logical`

Output Arguments

`map` — Binary occupancy grid
`binaryOccupancyMap` object handle

Binary occupancy grid, returned as a object handle. `map` is converted from a `'nav_msgs/OccupancyGrid'` message on the ROS network. The object is a grid of binary values, where `1` indicates an occupied location and `0` indications an unoccupied location.

Version History

Introduced in R2015a

See Also

Objects
`OccupancyGrid` | `occupancyMap`

Functions
`rosReadOccupancyGrid` | `rosWriteBinaryOccupancyGrid` | `rosWriteOccupancyGrid`

readCartesian

Read laser scan ranges in Cartesian coordinates

Syntax

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

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.

Note readCartesian will be removed. Use `rosReadCartesian` instead. For more information, see "ROS Message Structure Functions" on page 2-75

`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( ___ )` 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.

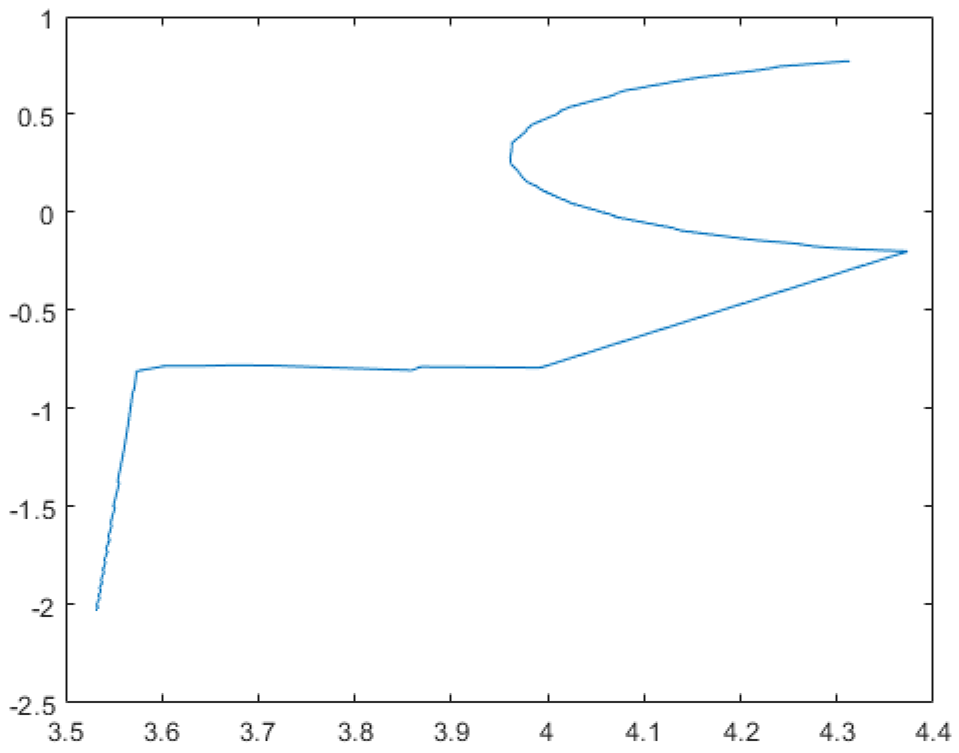
```
rosinit('192.168.17.129')
```

```
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.

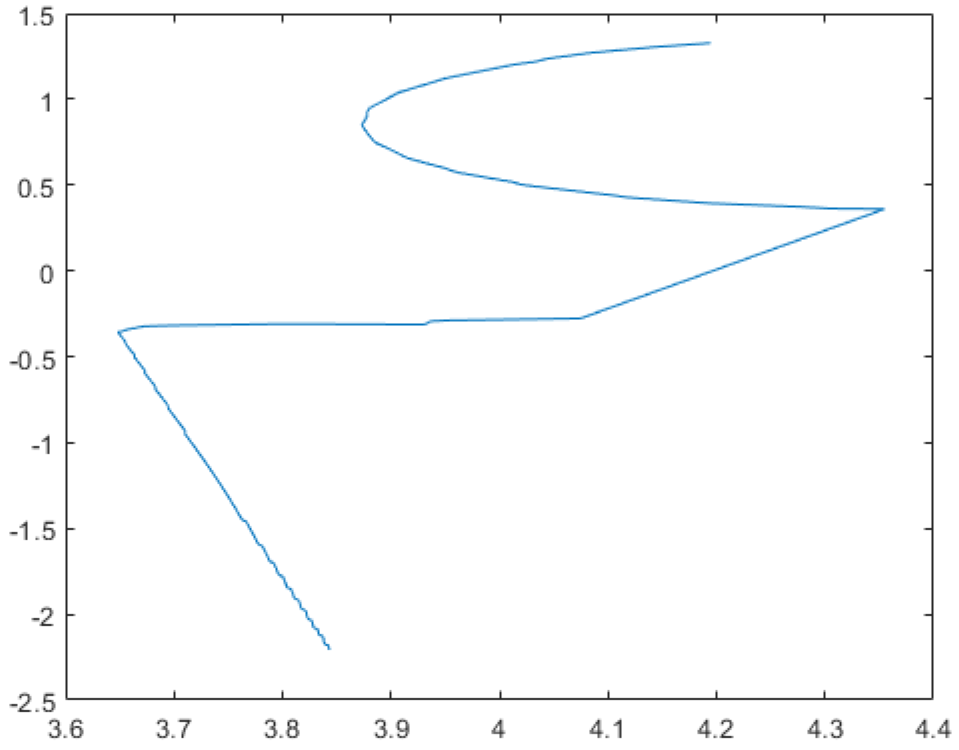
```
rosinit('192.168.17.129')
```

```
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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions
Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name

`Image`
 	
`CompressedImage`
	`readImage`
	
	`writeImage`
		`rosReadImage`
		
		`rosWriteImage`

`LaserScan`	`readCartesian`
	
	`readScanAngles`
	
	`lidarScan`
	
	`plot`
		`rosReadCartesian`
		
		`rosReadScanAngles`
		
		`rosReadLidarScan`
		
		`rosPlot`

Message Types	Object Function Name	New Function Name
PointCloud2	apply readXYZ readRGB readAllFieldNames readField scatter3	rosApplyTransform rosReadXYZ rosReadRGB rosReadAllFieldNames rosReadField rosPlot
Quaternion	readQuaternion	rosReadQuaternion
OccupancyGrid	readBinaryOccupanyGrid readOccupancyGrid writeBinaryOccupanyGrid writeOccupanyGrid	rosReadOccupancyGrid rosReadBinaryOccupancyGrid rosReadOccupancyGrid rosWriteBinaryOccupancyGrid rosWriteOccupancyGrid
Octomap	readOccupancyMap3D	rosReadOccupancyMap3D
PointStamped PoseStamped QuaternionStamped Vector3Stamped TransformStamped	apply	rosApplyTransform
All messages	showdetails	rosShowDetails

See Also

rosReadCartesian | rosReadXYZ | rosPlot

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 2-78.

Note readField will be removed. Use `rosReadField` instead. For more information, see “ROS Message Structure Functions” on page 2-78

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 2-78.

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.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions

Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name

`Image`
 	
`CompressedImage`
	`readImage`
	
	`writeImage`
		`rosReadImage`
		
		`rosWriteImage`

`LaserScan`	`readCartesian`
	
	`readScanAngles`
	
	`lidarScan`
	
	`plot`
		`rosReadCartesian`
		
		`rosReadScanAngles`
		
		`rosReadLidarScan`
		
		`rosPlot`

Message Types	Object Function Name	New Function Name
PointCloud2	apply readXYZ readRGB readAllFieldNames readField scatter3	rosApplyTransform rosReadXYZ rosReadRGB rosReadAllFieldNames rosReadField rosPlot
Quaternion	readQuaternion	rosReadQuaternion
OccupancyGrid	readBinaryOccupanyGrid readOccupancyGrid writeBinaryOccupanyGrid writeOccupanyGrid	rosReadOccupancyGrid rosReadBinaryOccupancyGrid rosReadOccupancyGrid rosWriteBinaryOccupancyGrid rosWriteOccupancyGrid
Octomap	readOccupancyMap3D	rosReadOccupancyMap3D
PointStamped PoseStamped QuaternionStamped Vector3Stamped TransformStamped	apply	rosApplyTransform
All messages	showdetails	rosShowDetails

See Also
PointCloud2 | readAllFieldNames

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`.

Note readImage will be removed. Use `rosReadImage` instead. For more information, see "ROS Message Structure Functions" on page 2-82

`[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`.

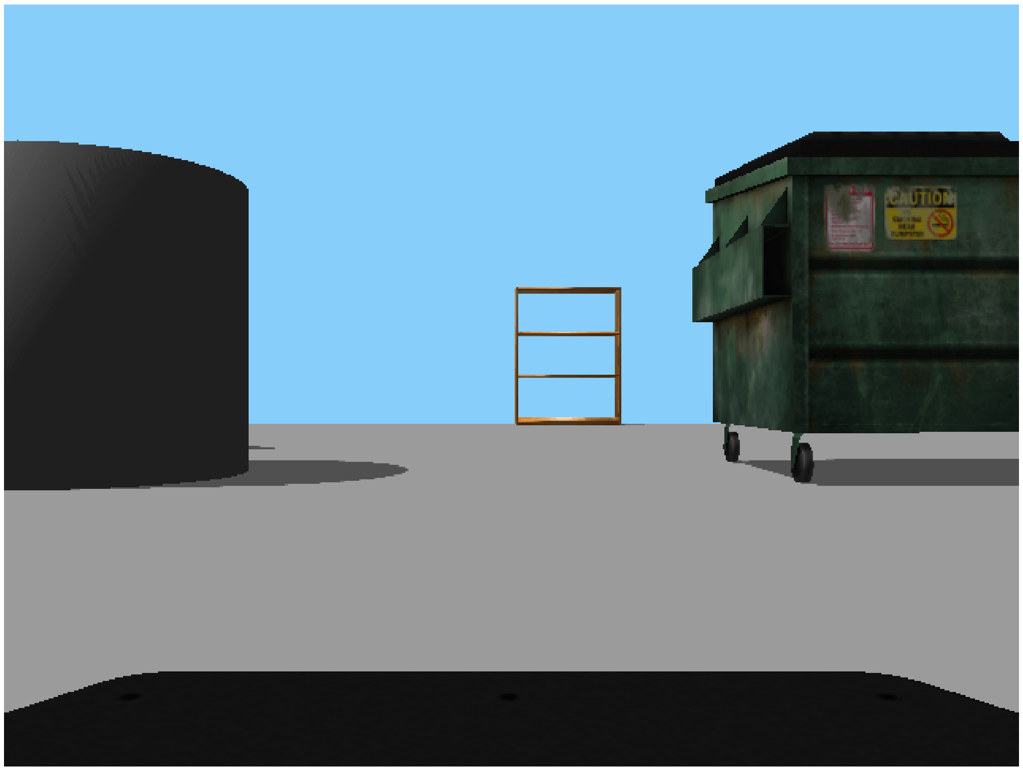
```
exampleHelperROSLoadMessages
```

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`.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions

Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name

`Image`
 	
`CompressedImage`
	`readImage`
	
	`writeImage`
		`rosReadImage`
		
		`rosWriteImage`

`LaserScan`	`readCartesian`
	
	`readScanAngles`
	
	`lidarScan`
	
	`plot`
		`rosReadCartesian`
		
		`rosReadScanAngles`
		
		`rosReadLidarScan`
		
		`rosPlot`

`PointCloud2`	`apply`
	
	`readXYZ`
	
	`readRGB`
	
	`readAllFieldNames`
	
	`readField`
	
	`scatter3`
		`rosApplyTransform`
		
		`rosReadXYZ`
		
		`rosReadRGB`
		
		`rosReadAllFieldNames`
		
		`rosReadField`
		
		`rosPlot`

`Quaternion`	`readQuaternion`	`rosReadQuaternion`

`OccupancyGrid`	`readBinaryOccupanyGrid`
	
	`readOccupancyGrid`
	
	`writeBinaryOccupanyGrid`
	
	`writeOccupanyGrid`
		`rosReadOccupancyGrid`
		
		`rosReadBinaryOccupancyGrid`
		
		`rosReadOccupancyGrid`
		
		`rosWriteBinaryOccupancyGrid`
		
		`rosWriteOccupancyGrid`

`Octomap`	`readOccupancyMap3D`	`rosReadOccupancyMap3D`

Message Types	Object Function Name	New Function Name
PointStamped PoseStamped QuaternionStamped Vector3Stamped TransformStamped	apply	rosApplyTransform
All messages	showdetails	rosShowDetails

See Also
rosReadImage | rosWriteImage | rosReadRGB

readMessages

Read messages from rosbag

Syntax

```
msgs = readMessages(bag)
msgs = readMessages(bag,rows)
msgs = readMessages( ___ ,"DataFormat",Format)
```

Description

`msgs = readMessages(bag)` returns data from all the messages in the `BagSelection` or `rosbagreader` 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",Format)` returns data as a cell array of structures or cell array of message objects using either set of the previous input arguments. Specify `Format` as either `"struct"` or `"object"`.

Using structures can be significantly faster than using message objects, and custom message data can be read directly without loading message definitions using `rosgenmsg`.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see “ROS Message Structures” on page 2-89.

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.

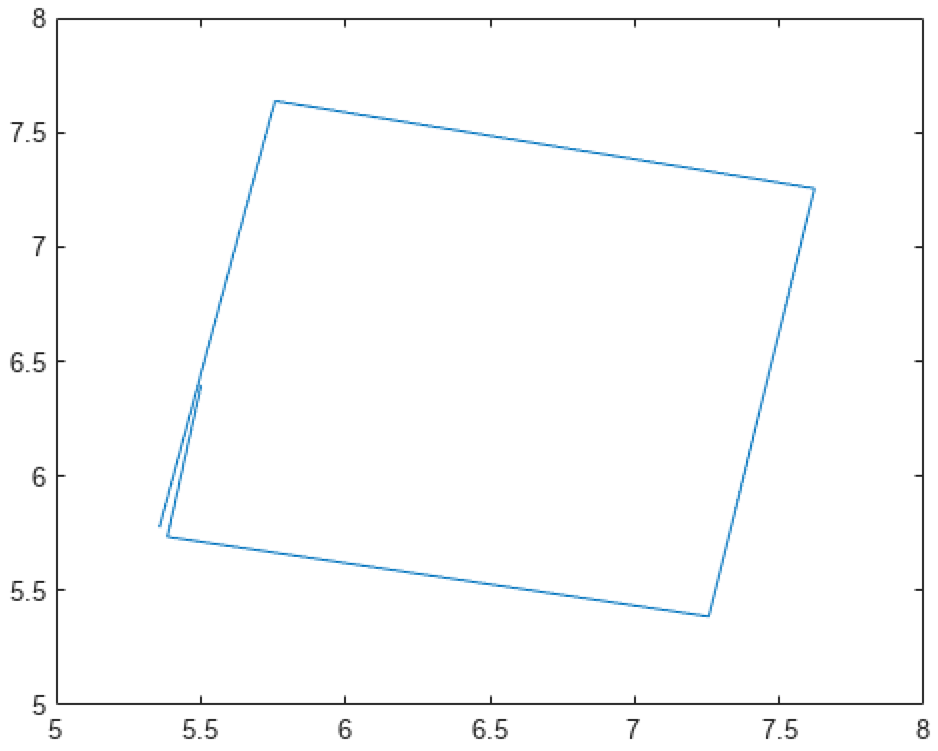
```
msgStructs = readMessages(bSel,'DataFormat','struct');
msgStructs{1}
```

```
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)
```



Create rosbag Selection Using `rosbagreader` Object

Load a rosbag log file and parse out specific messages based on the selected criteria.

Create a `rosbagreader` object of all the messages in the rosbag log file.

```
bagMsgs = rosbagreader("ros_multi_topics.bag")
```

```
bagMsgs = 
  rosbagreader with properties:

           FilePath: 'B:\matlab\toolbox\robotics\robotexamples\ros\data\bags\ros_multi_topics.bag
          StartTime: 201.3400
            EndTime: 321.3400
        NumMessages: 36963
    AvailableTopics: [4x3 table]
    AvailableFrames: {0x1 cell}
        MessageList: [36963x4 table]
```

Select a subset of the messages based on their timestamp and topic.

```
bagMsgs2 = select(bagMsgs,...
    Time=[bagMsgs.StartTime bagMsgs.StartTime + 1],...
    Topic='/odom')
```

```
bagMsgs2 = 
  rosbagreader with properties:

           FilePath: 'B:\matlab\toolbox\robotics\robotexamples\ros\data\bags\ros_multi_topics.bag
          StartTime: 201.3400
            EndTime: 202.3200
        NumMessages: 99
    AvailableTopics: [1x3 table]
    AvailableFrames: {0x1 cell}
        MessageList: [99x4 table]
```

Retrieve the messages in the selection as a cell array.

```
msgs = readMessages(bagMsgs2)
```

```
msgs=99×1 cell array
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
      ⋮
```

Return certain message properties as a time series.

```
ts = timeseries(bagMsgs2,...
    'Pose.Pose.Position.X', ...
    'Twist.Twist.Angular.Y')
```

```
  timeseries

  Timeseries contains duplicate times.

  Common Properties:
            Name: '/odom Properties'
            Time: [99x1 double]
        TimeInfo: tsdata.timemetadata
            Data: [99x2 double]
        DataInfo: tsdata.datametadata
```

Input Arguments

bag — Index of messages in rosbag

`BagSelection` object | `rosbagreader` object

Index of the messages in the rosbag, specified as a `BagSelection` or `rosbagreader` object.

`rows` — Rows of `BagSelection` or `rosbagreader` object
n-element vector

Rows of the `BagSelection` or `rosbagreader` 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 either the `BagSelection` object created using `rosbag` or the `rosbagreader` object.

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`.

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

See Also
`select` | `rosbag` | `timeseries` | `rosbagreader`

readOccupancyGrid

Read occupancy grid message

Syntax

`map = readOccupancyGrid(msg)`

Description

`map = readOccupancyGrid(msg)` returns an `occupancyMap` object by reading the data inside a ROS message, `msg`, which must be a `'nav_msgs/OccupancyGrid'` message. All message data values are converted to probabilities from 0 to 1. The unknown values (-1) in the message are set as 0.5 in the map.

Input Arguments

`msg` — `'nav_msgs/OccupancyGrid'` ROS message
`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as an `OccupancyGrid` ROS message object handle.

Output Arguments

`map` — Occupancy map
`occupancyMap` object handle

Occupancy map, returned as an `occupancyMap` object handle.

Version History

Introduced in R2016b

See Also

Functions
`rosReadBinaryOccupancyGrid` | `rosReadOccupancyMap3D` | `rosWriteBinaryOccupancyGrid` | `rosWriteOccupancyGrid`

readOccupancyMap3D

Read 3-D map from Octomap ROS message

Syntax

`map = readOccupancyMap3D(msg)`

Description

`map = readOccupancyMap3D(msg)` reads the data inside a ROS `'octomap_msgs/Octomap'` message to return an `occupancyMap3D` object. All message data values are converted to probabilities from 0 to 1.

Input Arguments

`msg` — `Octomap` ROS message
structure

`Octomap` ROS message, specified as a structure of message type `'octomap_msgs/Octomap'`. Get this message by subscribing to an `'octomap_msgs/Octomap'` topic using `rossubscriber` on a live ROS network or by creating your own message using `rosmessage`.

Output Arguments

`map` — 3-D occupancy map
`occupancyMap3D` object handle

3-D occupancy map, returned as an `occupancyMap3D` object handle.

Version History

Introduced in R2021a

See Also

`occupancyMap3D` | `rosmessage` | `rossubscriber`

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 matrix 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 2-94.

Note readRGB will be removed. Use `rosReadRGB` instead. For more information, see “ROS Message Structure Functions” on page 2-94

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 2-94.

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.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions

Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name

`Image`
 	
`CompressedImage`
	`readImage`
	
	`writeImage`
		`rosReadImage`
		
		`rosWriteImage`

`LaserScan`	`readCartesian`
	
	`readScanAngles`
	
	`lidarScan`
	
	`plot`
		`rosReadCartesian`
		
		`rosReadScanAngles`
		
		`rosReadLidarScan`
		
		`rosPlot`

Message Types	Object Function Name	New Function Name
PointCloud2	apply readXYZ readRGB readAllFieldNames readField scatter3	rosApplyTransform rosReadXYZ rosReadRGB rosReadAllFieldNames rosReadField rosPlot
Quaternion	readQuaternion	rosReadQuaternion
OccupancyGrid	readBinaryOccupanyGrid readOccupancyGrid writeBinaryOccupanyGrid writeOccupanyGrid	rosReadOccupancyGrid rosReadBinaryOccupancyGrid rosReadOccupancyGrid rosWriteBinaryOccupancyGrid rosWriteOccupancyGrid
Octomap	readOccupancyMap3D	rosReadOccupancyMap3D
PointStamped PoseStamped QuaternionStamped Vector3Stamped TransformStamped	apply	rosApplyTransform
All messages	showdetails	rosShowDetails

See Also

PointCloud2 | readXYZ | PointCloud2

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.

Note readScanAngles will be removed. Use `rosReadScanAngles` instead. For more information, see "ROS Message Structure Functions" on page 2-97

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 = 640×1

   -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.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions
Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name

`Image`
 	
`CompressedImage`
	`readImage`
	
	`writeImage`
		`rosReadImage`
		
		`rosWriteImage`

`LaserScan`	`readCartesian`
	
	`readScanAngles`
	
	`lidarScan`
	
	`plot`
		`rosReadCartesian`
		
		`rosReadScanAngles`
		
		`rosReadLidarScan`
		
		`rosPlot`

`PointCloud2`	`apply`
	
	`readXYZ`
	
	`readRGB`
	
	`readAllFieldNames`
	
	`readField`
	
	`scatter3`
		`rosApplyTransform`
		
		`rosReadXYZ`
		
		`rosReadRGB`
		
		`rosReadAllFieldNames`
		
		`rosReadField`
		
		`rosPlot`

Message Types	Object Function Name	New Function Name
Quaternion	readQuaternion	rosReadQuaternion
OccupancyGrid	readBinaryOccupanyGrid readOccupancyGrid writeBinaryOccupanyGrid writeOccupanyGrid	rosReadOccupancyGrid rosReadBinaryOccupancyGrid rosReadOccupancyGrid rosWriteBinaryOccupancyGrid rosWriteOccupancyGrid
Octomap	readOccupancyMap3D	rosReadOccupancyMap3D
PointStamped PoseStamped QuaternionStamped Vector3Stamped TransformStamped	apply	rosApplyTransform
All messages	showdetails	rosShowDetails

See Also
rosReadCartesian | rosReadXYZ | rosPlot

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 2-100.

Note readXYZ will be removed. Use `rosReadXYZ` instead. For more information, see “ROS Message Structure Functions” on page 2-100

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 2-100.

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.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions

Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name

`Image`
 	
`CompressedImage`
	`readImage`
	
	`writeImage`
		`rosReadImage`
		
		`rosWriteImage`

`LaserScan`	`readCartesian`
	
	`readScanAngles`
	
	`lidarScan`
	
	`plot`
		`rosReadCartesian`
		
		`rosReadScanAngles`
		
		`rosReadLidarScan`
		
		`rosPlot`

Message Types	Object Function Name	New Function Name
PointCloud2	apply readXYZ readRGB readAllFieldNames readField scatter3	rosApplyTransform rosReadXYZ rosReadRGB rosReadAllFieldNames rosReadField rosPlot
Quaternion	readQuaternion	rosReadQuaternion
OccupancyGrid	readBinaryOccupanyGrid readOccupancyGrid writeBinaryOccupanyGrid writeOccupanyGrid	rosReadOccupancyGrid rosReadBinaryOccupancyGrid rosReadOccupancyGrid rosWriteBinaryOccupancyGrid rosWriteOccupancyGrid
Octomap	readOccupancyMap3D	rosReadOccupancyMap3D
PointStamped PoseStamped QuaternionStamped Vector3Stamped TransformStamped	apply	rosApplyTransform
All messages	showdetails	rosShowDetails

See Also

rosReadRGB | rosReadCartesian | rosReadAllFieldNames

receive

Wait for new ROS message

Syntax

```
msg = receive(sub)
msg = receive(sub,timeout)
[msg,status,statustext] = receive( ___ )
```

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, this function will display an error.

`[msg,status,statustext] = receive( ___ )` returns a `status` indicating whether a message has been received successfully, and a `statustext` that captures additional information about the `status`, using any of the arguments from the previous syntaxes. If an error condition occurs, such as no message received within the specified timeout, the `status` will be `false`, and this function will not display 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
```

```
Launching ROS Core...
..Done in 2.6893 seconds.
Initializing ROS master on http://172.30.131.134:57346.
Initializing global node /matlab_global_node_49538 with NodeURI http://bat6234win64:59473/ and Ma
```

```
exampleHelperROSCreateSampleNetwork
```

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

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

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

```
[msg2,status,statustext] = receive(sub,10)
```

```
msg2 = struct with fields:
       MessageType: 'sensor_msgs/LaserScan'
```

```
           Header: [1x1 struct]
         AngleMin: -0.5467
         AngleMax: 0.5467
   AngleIncrement: 0.0017
    TimeIncrement: 0
         ScanTime: 0.0330
         RangeMin: 0.4500
         RangeMax: 10
           Ranges: [640x1 single]
      Intensities: []


status = logical
   1


statustext = 
'success'
```

Shutdown the timers used by sample network.

```
exampleHelperROSShutDownSampleNetwork
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_49538 with NodeURI http://bat6234win64:59473/ and N
Shutting down ROS master on http://172.30.131.134:57346.
```

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 | structure

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

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see “ROS Message Structures” on page 2-104.

`status` — Status of the message reception
`logical` scalar

Status of the message reception, returned as a `logical` scalar. If no message is received, status will be `false`.

Note Use the `status` output argument when you use receive for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

`statustext` — Status text associated with the message reception status
character vector

Status text associated with the message reception, returned as one of the following:

- `'success'` — The message was successfully received.
- `'timeout'` — The message was not received within the specified timeout.
- `'unknown'` — The message was not received due to unknown errors.

Tips

For code generation:

- Use the `status` output argument when you call `receive` in the entry-point function. This will avoid runtime errors and instead, outputs the status of message reception, which can be reacted to in the calling code.

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for `struct` messages.
- To monitor the message reception status and react in the calling code, use the `status` output argument. This will avoid runtime errors when no message is received.

See Also

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

Topics
“Exchange Data with ROS Publishers and Subscribers”

ros2

Retrieve information about ROS 2 network

Syntax

```
ros2 msg list
ros2 msg show msgType
ros2 node list
ros2 topic list
ros2 service list
ros2 service type svcname
ros2 action list
ros2 action type actionname
ros2 bag info folderpath
msgList = ros2("msg","list")
msgInfo = ros2("msg","show",msgType)
nodeList = ros2("node","list")
topicList = ros2("topic","list")
serviceList = ros2("service","list")
serviceTypes = ros2("service","type",svcname)
actionList = ros2("action","list")
actionTypes = ros2("action","type",actionname)
nodeList = ros2("node","list","DomainID",ID)
topicList = ros2("topic","list","DomainID",ID)
bag2info = ros2("bag","info",folderpath)
```

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.

`ros2 service list` lists service names that are registered on the ROS 2 network through either servers or clients.

`ros2 service type svcname` lists service types that are registered on the ROS 2 network for the provided `svcname`.

`ros2 action list` lists action names that are registered on the ROS 2 network through either servers or clients.

`ros2 action type actionname` lists action types that are registered on the ROS 2 network for the provided `actionname`.

`ros2 bag info folderpath` displays the information about the contents of the ros2bag at `folderpath` in the MATLAB Command Window. The information include the contents of the `bag2info` structure.

Note If the ROS 2 bag log file contains custom messages, generate MATLAB interfaces to ROS 2 custom messages using `ros2genmsg` function before using this command.

`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.

`serviceList = ros2("service","list")` lists service names that are registered on the ROS 2 network through either servers or clients.

`serviceTypes = ros2("service","type",svcname)` lists service types that are registered on the ROS 2 network for the provided `svcname`.

`actionList = ros2("action","list")` lists action names that are registered on the ROS 2 network through either servers or clients.

`actionTypes = ros2("action","type",actionname)` lists action types that are registered on the ROS 2 network for the provided `actionname`.

`nodeList = ros2("node","list","DomainID",ID)` lists nodes on the ROS 2 network for the specified network domain ID. By default, the value of `"DomainID"` is `0` unless otherwise specified by the `ROS_DOMAIN_ID` environment variable.

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

Note The `"DomainID"` name-value pair applies only to information gathered from the active network, such as the node and topic list, and not to static ROS 2 data such as message information.

The first time `ros2` is called for a specific domain ID not all information on the network may be immediately available. If incomplete network information is returned from `ros2`, wait for a short time before trying again.

`bag2info = ros2("bag","info",folderpath)` returns information about the contents of the ros2bag as a structure, `bag2info` at `folderpath`.

Note If the ROS 2 bag log file contains custom messages, generate MATLAB interfaces to ROS 2 custom messages using `ros2genmsg` function before using this function.

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
```

```
/image_test
/myNode
```

Remove `myNode` from the network.

```
delete(node)
```

Get List of ROS 2 Topics

List the available ROS 2 topics.

```
ros2 topic list
```

```
/image1
/image2
/parameter_events
/rosout
```

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")`.

Function syntax:

Example: `ros2("msg","show","sensor_msgs/LaserScan")`

Command syntax:

Example: `ros2 msg show sensor_msgs/LaserScan`

Data Types: `char` | `string`

`svcname` — Service name
string scalar | character vector

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

Function syntax:

Example: `ros2("service","type","/example_service")`

Command syntax:

Example: `ros2 service type /example_service`

Data Types: `char` | `string`

`actionname` — Action name
string scalar | character vector

Action name, specified as a string or character vector. The string is case-sensitive and no partial matches are allowed. It must match a service on the list given by calling `ros2("action","list")`.

Function syntax:

Example: `ros2("action","type","/example_action")`

Command syntax:

Example: `ros2 action type /example_action`

Data Types: `char` | `string`

`folderpath` — Path to ros2bag files
string scalar | character vector

Path to the ros2bag files, specified as a string scalar or character vector.

Note `folderpath` location must contain ROS 2 bag file (`.db3`) and `metadata.yaml`, which holds the meta information about the bag file. The folder name must be same as the ROS 2 bag file name.

Function syntax:

Example: `ros2("bag","info","C:\Users\Jack\MATLAB\EM\alltopics")`

Command syntax:

Example: `ros2 bag info C:\Users\Jack\MATLAB\EM\alltopics`

Data Types: `char` | `string`

`ID` — Domain identification of the network
non-negative scalar integer between 0 and 232

The domain identification of the ROS 2 network, specified as a non-negative scalar integer between 0 and 232.

Example: 2

Data Types: `double`

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 on the ROS 2 network
cell array of character vectors

List of topics on the ROS 2 network, returned as a cell array of character vectors.

`serviceList` — List of services on the ROS 2 network
cell array of character vectors

List of services on the ROS 2 network, returned as a cell array of character vectors.

`actionList` — List of actions on the ROS 2 network
cell array of character vectors

List of actions on the ROS 2 network, 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.

`bag2info` — Information about contents of ros2bag
structure

Information about contents of the ros2bag, returned as a structure. This structure contains fields related to the ros2bag log file and its contents. A sample output for a ros2bag as a structure is:

```
         Path: 'C:\Users\Jack\MATLAB\EM\alltopics\alltopics.db3'
      Version: '1'
    StorageId: 'sqlite3'
     Duration: 102.396644003
        Start: [1×1 struct]
          End: [1×1 struct]
         Size: 4965433
     Messages: 36503
        Types: [5×1 struct]
       Topics: [5×1 struct]
```

Data Types: `struct`

Version History

Introduced in R2019b

See Also

`ros2node` | `ros2publisher` | `ros2subscriber` | `ros2message`

ros2genmsg

Generate custom messages from ROS 2 definitions

Syntax

```
ros2genmsg(folderpath)
ros2genmsg(folderpath,Name=Value)
```

Description

`ros2genmsg(folderpath)` generates ROS 2 custom messages by reading ROS 2 custom messages and service definitions in the specified folder path. The function folder must contain one or more ROS 2 package. These packages contain the message definitions in `.msg` files, service definitions in `.srv` files, and action definitions in `.action` files.

After you generate the custom messages, you can send and receive them in MATLAB like all the other supported messages. You can create these messages using `ros2message` or view the list of messages by entering `ros2 msg list` at the MATLAB Command Window.

Note

- To generate custom messages for ROS 2, you must build the ROS 2 packages. This process requires you to have a C++ compiler for your platform. For more information, see “ROS Toolbox System Requirements”.
- With every new release of MATLAB, you must regenerate the custom messages from the ROS 2 definitions.
- Custom messages that you generate in MATLAB now support eProsima Fast DDS and Eclipse Cyclone DDS middleware. For more information on ROS middleware implementations, see “Switching Between ROS Middleware Implementations”.

`ros2genmsg(folderpath,Name=Value)` specifies additional options using one or more name-value arguments.

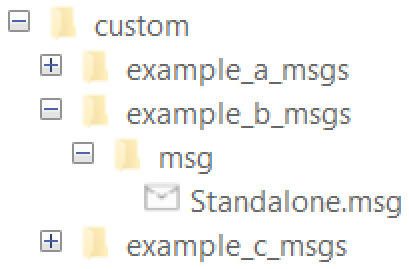
Examples

Generate ROS 2 Custom Messages in MATLAB

Use custom messages to extend the set of message types currently supported in ROS 2. Custom messages are messages that you define. 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, enter `ros2 msg list` in the MATLAB® Command Window. For more information about supported ROS 2 messages, see “Work with Basic ROS 2 Messages”.

If this if your first time working with ROS 2 custom messages, see “ROS Toolbox System Requirements”.

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



The package contains the custom message type `Standalone.msg`. MATLAB uses these files to generate the necessary files for using the custom messages contained in the package.

In this example, you create 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, you specify the path to the parent folder and 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.

Open a new MATLAB session and create a custom message folder in a local folder.

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

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

```
ros2genmsg(folderPath)
```

```
Identifying message files in folder 'C:/Work/custom'.Done.
Removing previous version of Python virtual environment.Done.
Creating a Python virtual environment.Done.
Adding required Python packages to virtual environment.Done.
Copying include folders.Done.
Copying libraries.Done.
Validating message files in folder 'C:/Work/custom'.Done.
[3/3] Generating MATLAB interfaces for custom message packages... Done.
Running colcon build in folder 'C:/Work/custom/matlab_msg_gen/win64'.
Build in progress. This may take several minutes...
Build succeeded.build log
```

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

```
ros2 msg list
```

```
action_msgs/CancelGoalRequest
action_msgs/CancelGoalResponse
action_msgs/GoalInfo
action_msgs/GoalStatus
action_msgs/GoalStatusArray
actionlib_msgs/GoalID
```

```
actionlib_msgs/GoalStatus
actionlib_msgs/GoalStatusArray
builtin_interfaces/Duration
builtin_interfaces/Time
composition_interfaces/ListNodesRequest
composition_interfaces/ListNodesResponse
composition_interfaces/LoadNodeRequest
composition_interfaces/LoadNodeResponse
composition_interfaces/UnloadNodeRequest
composition_interfaces/UnloadNodeResponse
diagnostic_msgs/AddDiagnosticsRequest
diagnostic_msgs/AddDiagnosticsResponse
diagnostic_msgs/DiagnosticArray
diagnostic_msgs/DiagnosticStatus
diagnostic_msgs/KeyValue
diagnostic_msgs/SelfTestRequest
diagnostic_msgs/SelfTestResponse
example_a_msgs/DependsOnB
example_b_msgs/Standalone
example_c_msgs/DependsOnB
example_interfaces/AddTwoIntsRequest
example_interfaces/AddTwoIntsResponse
example_interfaces/Bool
example_interfaces/Byte
example_interfaces/ByteMultiArray
example_interfaces/Char
example_interfaces/Empty
example_interfaces/Float32
example_interfaces/Float32MultiArray
example_interfaces/Float64
example_interfaces/Float64MultiArray
example_interfaces/Int16
example_interfaces/Int16MultiArray
example_interfaces/Int32
example_interfaces/Int32MultiArray
example_interfaces/Int64
example_interfaces/Int64MultiArray
example_interfaces/Int8
example_interfaces/Int8MultiArray
example_interfaces/MultiArrayDimension
example_interfaces/MultiArrayLayout
example_interfaces/SetBoolRequest
example_interfaces/SetBoolResponse
example_interfaces/String
example_interfaces/TriggerRequest
example_interfaces/TriggerResponse
example_interfaces/UInt16
example_interfaces/UInt16MultiArray
example_interfaces/UInt32
example_interfaces/UInt32MultiArray
example_interfaces/UInt64
example_interfaces/UInt64MultiArray
example_interfaces/UInt8
example_interfaces/UInt8MultiArray
example_interfaces/WString
geometry_msgs/Accel
geometry_msgs/AccelStamped
geometry_msgs/AccelWithCovariance
```

```
geometry_msgs/AccelWithCovarianceStamped
geometry_msgs/Inertia
geometry_msgs/InertiaStamped
geometry_msgs/Point
geometry_msgs/Point32
geometry_msgs/PointStamped
geometry_msgs/Polygon
geometry_msgs/PolygonStamped
geometry_msgs/Pose
geometry_msgs/Pose2D
geometry_msgs/PoseArray
geometry_msgs/PoseStamped
geometry_msgs/PoseWithCovariance
geometry_msgs/PoseWithCovarianceStamped
geometry_msgs/Quaternion
geometry_msgs/QuaternionStamped
geometry_msgs/Transform
geometry_msgs/TransformStamped
geometry_msgs/Twist
geometry_msgs/TwistStamped
geometry_msgs/TwistWithCovariance
geometry_msgs/TwistWithCovarianceStamped
geometry_msgs/Vector3
geometry_msgs/Vector3Stamped
geometry_msgs/Wrench
geometry_msgs/WrenchStamped
lifecycle_msgs/ChangeStateRequest
lifecycle_msgs/ChangeStateResponse
lifecycle_msgs/GetAvailableStatesRequest
lifecycle_msgs/GetAvailableStatesResponse
lifecycle_msgs/GetAvailableTransitionsRequest
lifecycle_msgs/GetAvailableTransitionsResponse
lifecycle_msgs/GetStateRequest
lifecycle_msgs/GetStateResponse
lifecycle_msgs/State
lifecycle_msgs/Transition
lifecycle_msgs/TransitionDescription
lifecycle_msgs/TransitionEvent
logging_demo/ConfigLoggerRequest
logging_demo/ConfigLoggerResponse
map_msgs/GetMapROIRequest
map_msgs/GetMapROIResponse
map_msgs/GetPointMapROIRequest
map_msgs/GetPointMapROIResponse
map_msgs/GetPointMapRequest
map_msgs/GetPointMapResponse
map_msgs/OccupancyGridUpdate
map_msgs/PointCloud2Update
map_msgs/ProjectedMap
map_msgs/ProjectedMapInfo
map_msgs/ProjectedMapsInfoRequest
map_msgs/ProjectedMapsInfoResponse
map_msgs/SaveMapRequest
map_msgs/SaveMapResponse
map_msgs/SetMapProjectionsRequest
map_msgs/SetMapProjectionsResponse
nav_msgs/GetMapRequest
nav_msgs/GetMapResponse
```

```
nav_msgs/GetPlanRequest
nav_msgs/GetPlanResponse
nav_msgs/GridCells
nav_msgs/MapMetaData
nav_msgs/OccupancyGrid
nav_msgs/Odometry
nav_msgs/Path
nav_msgs/SetMapRequest
nav_msgs/SetMapResponse
pendulum_msgs/JointCommand
pendulum_msgs/JointState
pendulum_msgs/RttestResults
rcl_interfaces/DescribeParametersRequest
rcl_interfaces/DescribeParametersResponse
rcl_interfaces/FloatingPointRange
rcl_interfaces/GetParameterTypesRequest
rcl_interfaces/GetParameterTypesResponse
rcl_interfaces/GetParametersRequest
rcl_interfaces/GetParametersResponse
rcl_interfaces/IntegerRange
rcl_interfaces/ListParametersRequest
rcl_interfaces/ListParametersResponse
rcl_interfaces/ListParametersResult
rcl_interfaces/Log
rcl_interfaces/Parameter
rcl_interfaces/ParameterDescriptor
rcl_interfaces/ParameterEvent
rcl_interfaces/ParameterEventDescriptors
rcl_interfaces/ParameterType
rcl_interfaces/ParameterValue
rcl_interfaces/SetParametersAtomicallyRequest
rcl_interfaces/SetParametersAtomicallyResponse
rcl_interfaces/SetParametersRequest
rcl_interfaces/SetParametersResponse
rcl_interfaces/SetParametersResult
rosgraph_msgs/Clock
sensor_msgs/BatteryState
sensor_msgs/CameraInfo
sensor_msgs/ChannelFloat32
sensor_msgs/CompressedImage
sensor_msgs/FluidPressure
sensor_msgs/Illuminance
sensor_msgs/Image
sensor_msgs/Imu
sensor_msgs/JointState
sensor_msgs/Joy
sensor_msgs/JoyFeedback
sensor_msgs/JoyFeedbackArray
sensor_msgs/LaserEcho
sensor_msgs/LaserScan
sensor_msgs/MagneticField
sensor_msgs/MultiDOFJointState
sensor_msgs/MultiEchoLaserScan
sensor_msgs/NavSatFix
sensor_msgs/NavSatStatus
sensor_msgs/PointCloud
sensor_msgs/PointCloud2
sensor_msgs/PointField
```

```
    sensor_msgs/Range
    sensor_msgs/RegionOfInterest
    sensor_msgs/RelativeHumidity
    sensor_msgs/SetCameraInfoRequest
    sensor_msgs/SetCameraInfoResponse
    sensor_msgs/Temperature
    sensor_msgs/TimeReference
    shape_msgs/Mesh
    shape_msgs/MeshTriangle
    shape_msgs/Plane
    shape_msgs/SolidPrimitive
    simple_msgs/AddTwoIntsRequest
    simple_msgs/AddTwoIntsResponse
    simple_msgs/Num
    statistics_msgs/MetricsMessage
    statistics_msgs/StatisticDataPoint
    statistics_msgs/StatisticDataType
    std_msgs/Bool
    std_msgs/Byte
    std_msgs/ByteMultiArray
    std_msgs/Char
    std_msgs/ColorRGBA
    std_msgs/Empty
    std_msgs/Float32
    std_msgs/Float32MultiArray
    std_msgs/Float64
    std_msgs/Float64MultiArray
    std_msgs/Header
    std_msgs/Int16
    std_msgs/Int16MultiArray
    std_msgs/Int32
    std_msgs/Int32MultiArray
    std_msgs/Int64
    std_msgs/Int64MultiArray
    std_msgs/Int8
    std_msgs/Int8MultiArray
    std_msgs/MultiArrayDimension
    std_msgs/MultiArrayLayout
    std_msgs/String
    std_msgs/UInt16
    std_msgs/UInt16MultiArray
    std_msgs/UInt32
    std_msgs/UInt32MultiArray
    std_msgs/UInt64
    std_msgs/UInt64MultiArray
    std_msgs/UInt8
    std_msgs/UInt8MultiArray
    std_srvs/EmptyRequest
    std_srvs/EmptyResponse
    std_srvs/SetBoolRequest
    std_srvs/SetBoolResponse
    std_srvs/TriggerRequest
    std_srvs/TriggerResponse
    stereo_msgs/DisparityImage
    test_msgs/Arrays
    test_msgs/ArraysRequest
    test_msgs/ArraysResponse
    test_msgs/BasicTypes
```

```
test_msgs/BasicTypesRequest
test_msgs/BasicTypesResponse
test_msgs/BoundedSequences
test_msgs/Builtins
test_msgs/Constants
test_msgs/Defaults
test_msgs/Empty
test_msgs/EmptyRequest
test_msgs/EmptyResponse
test_msgs/MultiNested
test_msgs/Nested
test_msgs/Strings
test_msgs/UnboundedSequences
test_msgs/WStrings
trajectory_msgs/JointTrajectory
trajectory_msgs/JointTrajectoryPoint
trajectory_msgs/MultiDOFJointTrajectory
trajectory_msgs/MultiDOFJointTrajectoryPoint
unique_identifier_msgs/UUID
visualization_msgs/GetInteractiveMarkersRequest
visualization_msgs/GetInteractiveMarkersResponse
visualization_msgs/ImageMarker
visualization_msgs/InteractiveMarker
visualization_msgs/InteractiveMarkerControl
visualization_msgs/InteractiveMarkerFeedback
visualization_msgs/InteractiveMarkerInit
visualization_msgs/InteractiveMarkerPose
visualization_msgs/InteractiveMarkerUpdate
visualization_msgs/Marker
visualization_msgs/MarkerArray
visualization_msgs/MenuEntry
```

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_b_msgs/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:
        MessageType: 'example_b_msgs/Standalone'
       int_property: 12
```

```
    string_property: 'This is ROS 2 custom message example'
```

Remove the created ROS objects.

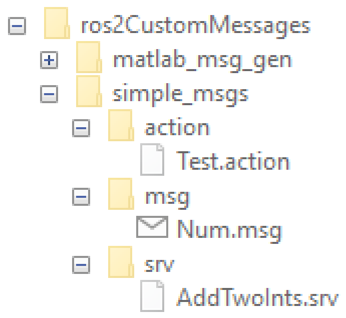
```
clear node pub sub
```

Replacing Definitions of Built-In Messages with Custom Definitions

MATLAB provides a lot of built-in ROS 2 message types. You can replace the definitions of those message types with new definitions using the same custom message creation workflow detailed above. When you are replacing the definitions of a built-in message package, you must ensure that the custom message package folder contains new definitions (`.msg` files) for all the message types in the corresponding built-in message package.

Create Shareable ROS 2 Custom Message Package

In this example, you create a shareable ROS 2 custom message package in MATLAB. You must have a ROS 2 package that contains the required `msg` file. This figure shows an example of an appropriate folder structure.



After you prepare your custom message package folder, you specify the path to the parent folder and call `ros2genmsg` with the specified path.

Open a new MATLAB session and create a custom message package folder in a local folder. Choose a short folder path when you generate custom messages on a Windows machine to avoid limitations on the number of characters in the folder path. For example,

```
genDir = fullfile('C:/test/ros2CustomMessages')
```

```
genDir = fullfile(pwd,'ros2CustomMessages');
packagePath = fullfile(genDir,'simple_msgs');
mkdir(packagePath)
```

Create a folder named `msg` inside the custom message package folder.

```
mkdir(packagePath,'msg')
```

Create a file named `.msg` inside the `msg` folder.

```
messageDefinition = {'int64 num'};

fileID = fopen(fullfile(packagePath,'msg', ...
               'Num.msg'),'w');
```

```
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Create a folder named `srv` inside the custom message package folder.

```
mkdir(packagePath,'srv')
```

Create a file named `.srv` inside the `srv` folder.

```
serviceDefinition = {'int64 a'
                     'int64 b'
                     '---'
                     'int64 sum'};

fileID = fopen(fullfile(packagePath,'srv', ...
               'AddTwoInts.srv'),'w');
fprintf(fileID,'%s\n',serviceDefinition{:});
fclose(fileID);
```

Create a folder named `action` inside the custom message package folder.

```
mkdir(packagePath,'action')
```

Create a file named `.action` inside the `action` folder.

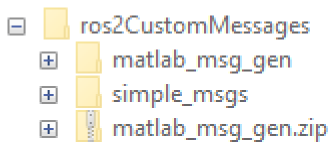
```
actionDefinition = {'int64 goal'
                    '---'
                    'int64 result'
                    '---'
                    'int64 feedback'};

fileID = fopen(fullfile(packagePath,'action', ...
               'Test.action'),'w');
fprintf(fileID,'%s\n',actionDefinition{:});
fclose(fileID);
```

Generate custom messages from ROS 2 definitions in `.msg`, `.srv` and `.action` files. Use the `CreateShareableFile` name-value argument to create a shareable ZIP archive of the generated custom messages.

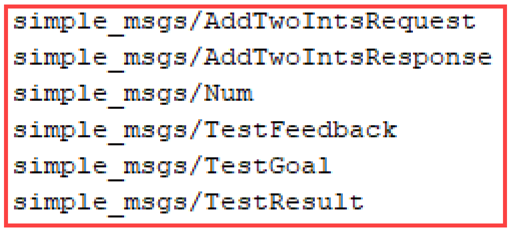
For information about how to use use this ZIP archive to register the custom messages in another machine, see `ros2RegisterMessages`.

```
ros2genmsg(genDir,CreateShareableFile=true);
```

```
Identifying message files in folder 'C:/Users/echakrab/OneDrive - MathWorks/Documents/MATLAB/Exa
Creating a Python virtual environment.Done.
Adding required Python packages to virtual environment.Done.
Copying include folders.Done.
Copying libraries.Done.
Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running colcon build in folder 'C:/Users/echakrab/OneDrive - MathWorks/Documents/MATLAB/ExampleMa
Build in progress. This may take several minutes...
```



Verify creation of the new custom messages by entering `ros2 msg list` in the Command Window.



```
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
simple_msgs/TestFeedback
simple_msgs/TestGoal
simple_msgs/TestResult
```

Input Arguments

`folderpath` — Path to ROS interfaces folder
string scalar | character vector

Path to the ROS interfaces folder, which is the parent folder of ROS message packages, specified as a string scalar or character vector. The parent folder must contain a package `.xml` and package folders. These folders contain a folder named `/msg` with `.msg` files for message definitions, a folder named `/srv` with `.srv` files for service definitions, and a folder named `/action` with `.action` files for action definitions. For more information, see About ROS 2 Interfaces.

Example: `'C:/test/ros2CustomMessages'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BuildConfiguration='fasterruns'`

`BuildConfiguration` — Allows for selecting different compiler optimizations when building the message libraries
`'fasterbuilds'` (default) | `'fasterruns'`

Build configuration, specified as the comma-separated pair consisting of `BuildConfiguration` and a character vector or string scalar containing `'fasterbuilds'` or `'fasterruns'`.

- `'fasterbuilds'` — Build the message libraries with compiler optimizations for shorter build times.
- `'fasterruns'` — Build the message libraries with compiler optimizations for faster execution.

Example: `ros2genmsg('C:/test/ros2CustomMessages',BuildConfiguration='fasterruns')`

Data Types: `char` | `string`

`CreateShareableFile` — Option to generate shareable ZIP archive
`false` or `0` (default) | `true` or `1`

Option to create a sharable ZIP archive, specified as a numeric or logical `1 (true)` or `0 (false)`.

When you specify this argument as `1 (true)`, the function creates a ZIP archive be compressing the install folder in the `matlab_msg_gen` folder. You can use this file with another machine running on the same platform and using the same MATLAB version.

When you specify this argument as `0 (false)`, the function does not create a ZIP archive.

Example: `ros2genmsg('C:/test/ros2CustomMessages',CreateShareableFile=true)`

Data Types: `logical`

Limitations

Restart Nodes

- After you generate 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 TGZ archive includes definitions for the custom messages, but not the ROS 2 custom message packages. When the function builds the generated code in the destination, the custom message packages must be available in the colcon workspace. Set this workspace as your current working directory. Install or copy the custom message package to your system before building the generated code.

MATLAB Compiler

- MATLAB Compiler™ software do not support ROS custom messages and the `ros2genmsg` function.

Version History

Introduced in R2019b

R2022a: Support for ROS Middleware Implementations

MATLAB generated custom messages now support eProsima Fast DDS and Eclipse Cyclone DDS middleware. For more information on ROS middleware implementations see “Switching Between ROS Middleware Implementations”.

See Also

`ros2message` | `ros2`

Topics

“Generate ROS Custom Messages in MATLAB” on page 2-145

External Websites

About ROS 2 Interfaces
Download Python

ros2message

Create ROS 2 message structures

Syntax

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

Description

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

`msg = ros2message(client)` creates an empty message determined by the action associated with the ROS 2 action client, `client`.

Examples

Create a String Message

Create a ROS 2 string message.

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

```
strMsg = struct with fields:
    MessageType: 'std_msgs/String'
           data: ''
```

Create an empty laser scan message

Create an empty ROS 2 laser scan message.

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

```
scanMsg = struct with fields:
        MessageType: 'sensor_msgs/LaserScan'
             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.

`client` — ROS 2 action client
`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

Output Arguments

`msg` — ROS 2 message for a given topic
structure

ROS 2 message for a given topic, returned as a message structure.

Version History

Introduced in R2019b

R2022a: Deprecation of Messages
Behavior changed in R2022a

The following messages in `std_msgs` is deprecated in ROS 2 Foxy Fitzroy.

Messages
`std_msgs/msg/Bool`
`std_msgs/msg/Byte`
`std_msgs/msg/ByteMultiArray`
`std_msgs/msg/Char`
`std_msgs/msg/Float32`
`std_msgs/msg/Float32MultiArray`
`std_msgs/msg/Float64`
`std_msgs/msg/Float64MultiArray`
`std_msgs/msg/Int16`
`std_msgs/msg/Int16MultiArray`
`std_msgs/msg/Int32`
`std_msgs/msg/Int32MultiArray`
`std_msgs/msg/Int64`
`std_msgs/msg/Int64MultiArray`
`std_msgs/msg/Int8`

Messages
`std_msgs/msg/Int8MultiArray`
`std_msgs/msg/MultiArrayDimension`
`std_msgs/msg/MultiArrayLayout`
`std_msgs/msg/String`
`std_msgs/msg/UInt16`
`std_msgs/msg/UInt16MultiArray`
`std_msgs/msg/UInt32`
`std_msgs/msg/UInt32MultiArray`
`std_msgs/msg/UInt64`
`std_msgs/msg/UInt64MultiArray`
`std_msgs/msg/UInt8`
`std_msgs/msg/UInt8MultiArray`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

- For messages with fields containing cell array of strings, such as `sensor_msgs/JointState`, accessing those fields in the MATLAB function is not supported.

See Also

`ros2node` | `ros2publisher` | `ros2subscriber` | `ros2actionclient` | `ros2`

ros2duration

Create a ROS 2 duration message

Syntax

```
dur = ros2duration
dur = ros2duration(totalSecs)
dur = ros2duration(secs,nsecs)
```

Description

`dur = ros2duration` returns a `builtin_interfaces/Duration` ROS 2 message structure, `dur`, with seconds and nanoseconds set to `0`.

`dur = ros2duration(totalSecs)` initializes the duration values for seconds and nanoseconds based on the specified time, in seconds, `totalSecs`

`dur = ros2duration(secs,nsecs)` initializes the duration values for seconds and nanoseconds individually. The function automatically wraps large values of `nsecs`, and adds the remainder to the seconds value of the message, `secs`.

Examples

Work with ROS 2 Duration Messages

Create a ROS 2 duration message using seconds and nanoseconds.

```
dur1 = ros2duration(100,2000000)
```

```
dur1 = struct with fields:
    MessageType: 'builtin_interfaces/Duration'
            sec: 100
        nanosec: 2000000
```

Create a ROS 2 duration message using a floating-point value. This sets the seconds using the integer portion and nanoseconds with the remainder.

```
dur2 = ros2duration(20.5)
```

```
dur2 = struct with fields:
    MessageType: 'builtin_interfaces/Duration'
            sec: 20
        nanosec: 500000000
```

Add a ROS 2 duration mesage to a ROS 2 time message.

```
node = ros2node("/test");
t1 = ros2time(node,"now");
t2 = ros2time(t1.sec+dur1.sec,t1.nanosec+dur1.nanosec)
```

```
t2 = struct with fields:
    MessageType: 'builtin_interfaces/Time'
            sec: 1677880410
        nanosec: 44948200
```

Input Arguments

`totalSecs` — Total time
0 (default) | scalar

Total time, specified as a floating-point scalar. The integer portion sets the `sec` field, and the remainder sets the `nanosec` field of the duration message `dur`.

`secs` — Whole seconds
0 (default) | integer

Whole seconds, specified as an integer. This value directly sets to the `sec` field of the duration message `dur`.

Note The maximum and minimum values for `secs` are `2147483648` and `2147483647`, respectively.

`nsecs` — Nanoseconds
0 (default) | positive integer

Nanoseconds, specified as a positive integer. This value directly sets the `nanoSec` field of the duration message `dur`, unless it is greater than or equal to 10^9. If so, the function wraps the value and adds the remainder to the value of `secs`.

Output Arguments

`dur` — ROS 2 duration
`builtin_interfaces/Duration` message structure

ROS 2 duration, returned as a `builtin_interfaces/Duration` message structure.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ros2time` | `ros2message`

ros2time

Access ROS 2 time functionality

Syntax

```
time = ros2time
time = ros2time(totalSecs)
time = ros2time(secs,nsecs)

time = ros2time(node,"now")
[time,issimtime] = ros2time(node,"now")
time = ros2time(node,"now","system")
```

Description

`time = ros2time` returns a `builtin_interfaces/Time` ROS 2 message structure, `time`, with seconds and nanoseconds set to `0`.

`time = ros2time(totalSecs)` initializes the time values for seconds and nanoseconds based on the specified time, in seconds, `totalSecs`.

`time = ros2time(secs,nsecs)` initializes the time values for seconds and nanoseconds individually. The function automatically wraps large values of `nsecs`, and adds the remainder to the seconds value of the message, `secs`.

`time = ros2time(node,"now")` returns the current ROS 2 time `time` within the specified `ros2node` object `node`. If the `use_sim_time` ROS 2 parameter is set to `true`, then `ros2time` returns the simulation time published on the `clock` topic. Otherwise, the function returns the system time of your machine. If you do not specify an output argument, the function prints the current time (in seconds) to the screen.

You can use `ros2time` to timestamp messages or to measure time in the ROS 2 network.

`[time,issimtime] = ros2time(node,"now")` also returns a logical scalar `issimtime`, that indicates if `time` is in simulation time (`true`) or system time (`false`).

`time = ros2time(node,"now","system")` returns the system time of your machine, even if ROS publishes simulation time on the `clock` topic. If you do not specify an output argument, the function prints the system time (in seconds) to the screen.

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), January 1 1970, not counting leap seconds.

Examples

Get Current ROS 2 Time and Timestamp ROS 2 Message Data

Create a ROS 2 node.

```
node = ros2node("/my_node");
```

Get current ROS 2 time based on the source used by the ROS 2 node.

```
t = ros2time(node,"now")
```

```
t = struct with fields:
    MessageType: 'builtin_interfaces/Time'
            sec: 1677880390
        nanosec: 359746500
```

Create a stamped ROS 2 point message. Specify the `header.stamp` property with the current system time.

```
point = ros2message("geometry_msgs/PointStamped");
point.header.stamp = t;
point.point.x = 5;
```

Convert ROS 2 Time to the specified MATLAB format, `datetime.`

```
time = datetime(t.sec + 10^-9*int32(t.nanosec),'ConvertFrom','posixtime')
```

```
time = datetime
   03-Mar-2023 21:53:10
```

Input Arguments

`totalSecs` — Total time

`0` (default) | scalar

Total time, specified as a floating-point scalar. The integer portion sets the `sec` field, and the remainder sets the `nanosec` field the time message `time`.

`secs` — Whole seconds

`0` (default) | positive integer

Whole seconds, specified as a positive integer.

Note The maximum and minimum values for `secs` are `0` and `4294967294`.

`nsecs` — Nanoseconds

`0` (default) | positive integer

Nanoseconds, specified as a positive integer. If this value is greater than or equal to 10^9, then the function wraps the value and adds the remainder to the value of `secs`.

`node` — ROS 2 node on network

`ros2node` object

ROS 2 node on the network, specified as a `ros2node` object.

Output Arguments

`time` — ROS 2 time
`builtin_interfaces/Time` message structure

ROS 2 time, returned as a `builtin_interfaces/Time` message structure.

`issimtime` — Indicator whether `time` is simulation time
logical scalar

Indicator whether `time` is simulation time, returned as a logical scalar.

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ros2duration` | `ros2message`

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:

```
rosrun 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.

```
rosaction list
```

```
/fibonacci
```

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

```
rosaction 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.

```
rosshutdown
```

```
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 `rosaction("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.

Version History

Introduced in R2019b

See Also

`sendGoal` | `cancelGoal` | `waitForServer` | `rosmessage` | `rostopic`

Topics
“ROS Actions Overview”
“Move a Turtlebot Robot Using ROS Actions”

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-on Explorer to browse the available add-ons.

Examples

Install Add-ons for ROS Toolbox™

```
rosAddons
```

Version History

Introduced in R2019b

See Also

Topics
“ROS Custom Message Support”
“Get and Manage Add-Ons”
“Manage Add-Ons”

rosApplyTransform

Transform message entities into target frame

Syntax

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

Description

`tfentity = rosApplyTransform(tfmsg,entity)` applies the transformation represented by the `'TransformStamped'` ROS or ROS 2 message to the input message object `entity`.

This function determines the message type of `entity` and apples the appropriate transformation method to it.

Input Arguments

`tfmsg` — Transformation message
`TransformStamped` ROS or ROS 2 message structure

Transformation message, specified as a `TransformStamped` ROS or ROS 2 message handle. The `tfmsg` is a ROS or ROS 2 message of type: `'geometry_msgs/TransformStamped'`.

`entity` — ROS or ROS 2 message
message structure

ROS or ROS 2 message, specified as a message structure.

Supported messages are:

- `geometry_msgs/PointStamped`
- `sensor_msgs/PointCloud2`
- `geometry_msgs/PoseStamped`
- `geometry_msgs/QuaternionStamped`
- `geometry_msgs/Vector3Stamped`

Output Arguments

`tfentity` — Transformed ROS or ROS 2 message
message structure

Transformed ROS or ROS 2 message, returned as a message structure.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

`rostf` | `rosbag`

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\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex32890909\ex_multiple_topics.b
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.

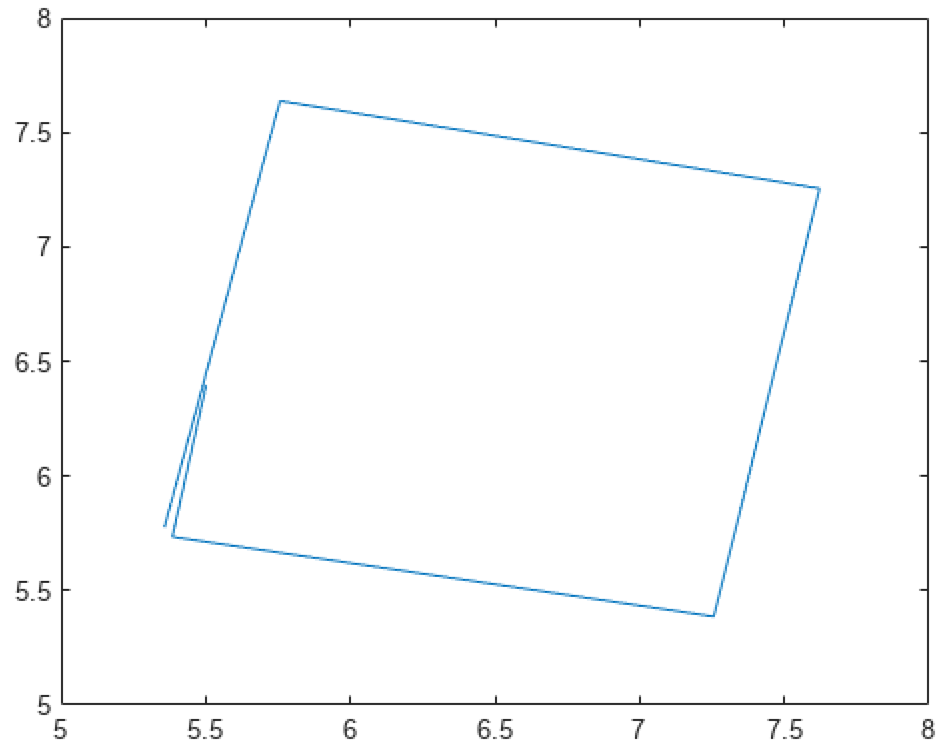
```
msgStructs = readMessages(bSel,'DataFormat','struct');
msgStructs{1}
```

```
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
```

Version History

Introduced in R2019b

See Also

`select` | `readMessages` | `canTransform` | `getTransform` | `timeseries` | `BagSelection`

rosduration

Create a ROS duration object

Syntax

```
dur = rosduration
dur = rosduration(totalSecs)
dur = rosduration(secs,nsecs)
dur = rosduration (___,"DataFormat","struct")
```

Description

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

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see "ROS Message Structures" on page 2-144.

`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`.

`dur = rosduration (___,"DataFormat","struct")` uses message structures instead of objects with any of the arguments in previous syntaxes. For more information, see "ROS Message Structures" on page 2-144.

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
```

Initialize a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.1491 seconds.
```

```
Initializing ROS master on http://172.30.131.134:58487.
Initializing global node /matlab_global_node_59956 with NodeURI http://bat6234win64:64593/ and Ma
```

Add a duration to a ROS `Time` object.

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

```
time = 
  ROS Time with properties:

     Sec: 1.6779e+09
    Nsec: 321308300
```

```
timeFuture = time + dur3
```

```
timeFuture = 
  ROS Time with properties:

     Sec: 1.6779e+09
    Nsec: 823308300
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_59956 with NodeURI http://bat6234win64:64593/ and M
Shutting down ROS master on http://172.30.131.134:58487.
```

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 | structure

Duration, returned as a ROS `Duration` object or message structure with fields `Sec` and `NSec`

Version History

Introduced in R2019b

R2021a: ROS Message Structures
Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rostime` | `rosmessage`

rosgenmsg

Generate custom messages from ROS definitions

Syntax

```
rosgenmsg(folderpath)
rosgenmsg(folderpath,Name=Value)
```

Description

`rosgenmsg(folderpath)` generates ROS custom messages by reading ROS custom messages, service definitions, and action definitions in the specified folder path. The function folder must contain one or more ROS package folders. These packages contain the message definitions in `.msg` files, service definitions in `.srv` files, and action definitions in `.action` files.

After you generate the custom messages, you can send and receive them in MATLAB like all the other supported messages. You can create these messages using `rosmessage` or view the list of messages by entering `rosmsg list` at the MATLAB Command Window.

Note

- To generate custom messages for ROS, you must build the ROS packages. This process requires you to have access to CMake software and a C++ compiler for your platform. For more information, see “ROS Toolbox System Requirements”
- With every new release of MATLAB, you must regenerate the custom messages from the ROS definitions.

`rosgenmsg(folderpath,Name=Value)` specifies additional options using one or more name-value arguments.

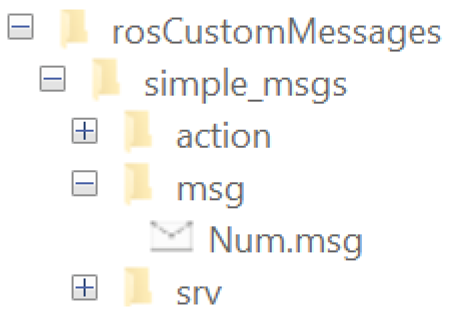
Examples

Generate ROS Custom Messages in MATLAB

Use custom messages to extend the set of message types currently supported in ROS. Custom messages are messages that you define. If you are sending and receiving supported message types, you do not need to use custom messages. To see the list of supported message types, enter `rosmsg list` in the MATLAB Command Window. For more information about supported ROS messages, see “Work with Basic ROS Messages”.

If this is your first time working with ROS custom messages, see “ROS Toolbox System Requirements”.

ROS custom messages are specified in ROS package folders that contain a folder named `msg`. The `msg` folder contains all your custom message type definitions. For example, the `simple_msgs` package in the `rosCustomMessages` folder, has this folder and file structure.



The package contains the custom message type `Num.msg`. MATLAB uses these files to generate the necessary files for using the custom messages contained in the package.

In this example, you create ROS custom messages in MATLAB and compress them in a shareable ZIP archive. You must have a ROS package that contains the required `msg` file.

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

Open a new MATLAB session and create a custom message package folder in a local folder. Choose a short folder path when you generate custom messages on a Windows machine to avoid limitations on the number of characters in the folder path. For example,

`genDir = fullfile('C:/test/rosCustomMessages')`

```
genDir = fullfile(pwd,'rosCustomMessages');
packagePath = fullfile(genDir,'simple_msgs');
mkdir(packagePath)
```

Create a folder named `msg` inside the custom message package folder.

```
mkdir(packagePath,'msg')
```

Create a file named `.msg` inside the `msg` folder.

```
messageDefinition = {'int64 num'};

fileID = fopen(fullfile(packagePath,'msg', ...
               'Num.msg'),'w');
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Create a folder named `srv` inside the custom message package folder.

```
mkdir(packagePath,'srv')
```

Create a file named `.srv` inside the `srv` folder.

```
serviceDefinition = {'int64 a'
                     'int64 b'
                     '---'
                     'int64 sum'};

fileID = fopen(fullfile(packagePath,'srv', ...
               'AddTwoInts.srv'),'w');
fprintf(fileID,'%s\n',serviceDefinition{:});
fclose(fileID);
```

Create a folder named `action` inside the custom message package folder.

```
mkdir(packagePath,'action')
```

Create a file named `.action` inside the `action` folder.

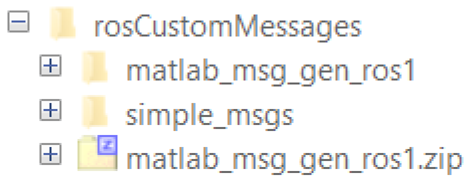
```
actionDefinition = {'int64 goal'
                    '---'
                    'int64 result'
                    '---'
                    'int64 feedback'};

fileID = fopen(fullfile(packagePath,'action', ...
               'Test.action'),'w');
fprintf(fileID,'%s\n',actionDefinition{:});
fclose(fileID);
```

Generate custom messages from ROS definitions in `.msg`, `.srv`, and `.action` files. Use the `CreateShareableFile` name-value argument to create a shareable ZIP archive of the generated custom messages.

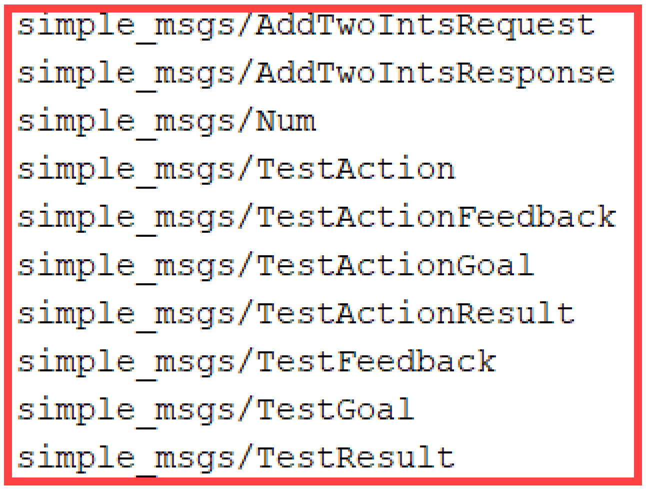
For information about how to use use this ZIP archive to register the custom messages in another machine, see `rosRegisterMessages`.

```
rosgenmsg(genDir,CreateShareableFile=true)
```

```
Identifying message files in folder 'C:/test/rosCustomMessages'.Done.
Creating a Python virtual environment.Done.
Adding required Python packages to virtual environment.Done.
Copying include folders.Done.
Copying libraries.Done.
Validating message files in folder 'C:/test/rosCustomMessages'.Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running catkin build in folder 'C:/test/rosCustomMessages/matlab_msg_gen_ros1/win64'.
Build in progress. This may take several minutes...
Build succeeded.build log
Generating zip file in the folder 'C:/test/rosCustomMessages'.Done.

To use the custom messages, follow these steps:

1. Add the custom message folder to the MATLAB path by executing:

addpath('C:\test\rosCustomMessages\matlab_msg_gen_ros1\win64\install\m')
savepath

2. Refresh all message class definitions, which requires clearing the workspace, by executing:

clear classes
rehash toolboxcache

3. Verify that you can use the custom messages.
   Enter "rosmsg list" and ensure that the output contains the generated
   custom message types.
```



Verify creation of the new custom messages by entering `rosmsg list`.



```
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
simple_msgs/TestAction
simple_msgs/TestActionFeedback
simple_msgs/TestActionGoal
simple_msgs/TestActionResult
simple_msgs/TestFeedback
simple_msgs/TestGoal
simple_msgs/TestResult
```

Input Arguments

`folderpath` — Path to ROS package folders
string scalar | character vector

Path to the ROS message packages, specified as a string scalar or character vector. The parent folder must contain package folders. These folders contain a folder named `/msg` with `.msg` files for message definitions, a folder named `/srv` with `.srv` files for service definitions, and a folder named `/action` with `.action` files for action definitions.

Example: `'C:/test/rosCustomMessages'`

Data Types: `char` | `string`

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `BuildConfiguration='fasterruns'`

`BuildConfiguration` — Allows for selecting different compiler optimizations when building the message libraries
`'fasterbuilds'` | `'fasterruns'`

Build configuration, specified as one of these values.

- `'fasterbuilds'` — Build the message libraries with compiler optimizations for shorter build times. This is the default configuration.
- `'fasterruns'` — Build the message libraries with compiler optimizations for faster execution.

Example: `rosgenmsg('C:/test/rosCustomMessages',BuildConfiguration='fasterruns')`

Data Types: `char` | `string`

`CreateShareableFile` — Option to generate shareable ZIP archive
`false` or `0` (default) | `true` or `1`

Option to create a sharable ZIP archive, specified as a numeric or logical `1 (true)` or `0 (false)`.

When you specify this argument as `1 (true)`, the function creates a ZIP archive be compressing the install folder in the `matlab_msg_gen_ros1` folder. You can use this file with another machine running on the same platform and using the same MATLAB version.

When you specify this argument as `0 (false)`, the function does not create a ZIP archive.

Example: `rosgenmsg('C:/test/rosCustomMessages',CreateShareableFile=true)`

Data Types: `logical`

Limitations

MATLAB Compiler software do not support ROS custom messages and the `rosgenmsg` function.

Version History

Introduced in R2019b

See Also

`rosmessage` | `rosmsg`

Topics
"Create Custom Messages from ROS Package"
"Generate ROS 2 Custom Messages in MATLAB" on page 2-112

External Websites
ROS Tutorials: Defining Custom Messages
ROS Tutorials: Creating a ROS msg and srv

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.

Note The first time you connect to a ROS network, you must install and setup Python.

- From R2020b to R2021b, install Python 2.7.
- From R2022a or later, install Python 3.9.

To check your Python version in MATLAB, use the `pyenv` function. For more information, see “ROS Toolbox System Requirements”.

`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
```

```
Launching ROS Core...
..Done in 2.7022 seconds.
Initializing ROS master on http://172.30.131.134:54795.
Initializing global node /matlab_global_node_90971 with NodeURI http://bat6234win64:58551/ and Ma
```

When you are finished, shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_90971 with NodeURI http://bat6234win64:58551/ and M
Shutting down ROS master on http://172.30.131.134:54795.
```

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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 `rosinit` is registered on the ROS network with this name.

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

Version History

Introduced in R2019b

See Also

`Node` | `rosshutdown`

Topics
"Connect to a ROS Network"
"ROS Toolbox System Requirements"

rosmessage

Create ROS messages

Syntax

```
msg = rosmessage(messagetype)
```

```
msg = rosmessage(pub)
msg = rosmessage(sub)
msg = rosmessage(client)
msg = rosmessage(server)
msg = rosmessage( ___ ,"DataFormat","struct")
```

Description

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

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see “ROS Message Structures” on page 2-156.

`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 or action type of `server`.

`msg = rosmessage( ___ ,"DataFormat","struct")` creates an empty message as a message structure with any of the arguments in previous syntaxes. For more information, see “ROS Message Structures” on page 2-156.

Examples

Create Empty String Message

Create a ROS message as a structure with the `std_msgs/String` message type.

```
strMsg = rosmessage("std_msgs/String","DataFormat","struct")
```

```
strMsg = struct with fields:
    MessageType: 'std_msgs/String'
```

```
           Data: ''
```

Create ROS Publisher and Send Data

Start ROS master.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.705 seconds.
Initializing ROS master on http://172.30.131.134:53000.
Initializing global node /matlab_global_node_37957 with NodeURI http://bat6234win64:63529/ and Ma
```

Create publisher for the `/chatter` topic with the `std_msgs/String` message type. Set the `"DataFormat"` name-value argument to structure ROS messages.

```
chatpub = rospublisher("/chatter","std_msgs/String","DataFormat","struct");
```

Create a message to send. Specify the `Data` property with a character vector.

```
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_37957 with NodeURI http://bat6234win64:63529/ and M
Shutting down ROS master on http://172.30.131.134:53000.
```

Create and Access Array of ROS Messages

You can create an structure 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 an array of two messages. Specify the `DataFormat` name-value argument to use structures for ROS messages.

```
blankMsg = rosmessage("std_msgs/String","DataFormat","struct")
```

```
blankMsg = struct with fields:
    MessageType: 'std_msgs/String'
           Data: ''
```

```
msgArray = [blankMsg blankMsg]
```

```
msgArray=1×2 struct array with fields:
    MessageType
    Data
```

Assign data to individual message fields in the array.

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

Read all the `Data` fields from the messages 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 as objects, use the `arrayfun` or `cellfun` functions instead of `repmat`. These functions properly create object or cell arrays for handle classes.

Note: *In a future release, ROS message objects will be removed. To use ROS messages as structures and utilize structure arrays, specify the* `DataFormat` *name-value pair when calling the* `rosmessage` *function.*

Preallocate an object array of ROS messages.

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

Preallocate a cell array of ROS messages.

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

Input Arguments

`messagetype` — 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 `rosmsg("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 | structure

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

Version History

Introduced in R2019b

R2021a: ROS Message Structures
Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

R2022a: Unsupported ROS Message Types
Behavior changed in R2022a

Message Types
adhoc_communication/ExpAuction

Message Types
adhoc_communication/ExpCluster
adhoc_communication/ExpFrontier
adhoc_communication/ExpFrontierElement
adhoc_communication/SendExpAuctionRequest
adhoc_communication/SendExpClusterRequest
adhoc_communication/SendExpFrontierRequest
arbotix_msgs/Analog
baxter_core_msgs/AssemblyState
baxter_core_msgs/AssemblyStates
baxter_core_msgs/EndEffectorProperties
baxter_core_msgs/HeadPanCommand
baxter_core_msgs/HeadState
baxter_core_msgs/NavigatorState
baxter_core_msgs/NavigatorStates
baxter_core_msgs/SEAJointState
baxter_maintenance_msgs/CalibrateArmData
baxter_maintenance_msgs/CalibrateArmEnable
capabilities/StartCapabilityResponse
cmvision/Blob
cmvision/Blobs
cob_grasp_generation/GenerateGraspsAction
cob_grasp_generation/GenerateGraspsActionGoal
cob_grasp_generation/GenerateGraspsGoal
cob_grasp_generation/QueryGraspsAction
cob_grasp_generation/QueryGraspsActionGoal
cob_grasp_generation/QueryGraspsGoal
cob_grasp_generation/ShowGraspsAction
cob_grasp_generation/ShowGraspsActionGoal
cob_grasp_generation/ShowGraspsGoal
cob_light/LightMode
cob_light/SetLightModeAction
cob_light/SetLightModeActionGoal
cob_light/SetLightModeActionResult
cob_light/SetLightModeGoal
cob_light/SetLightModeRequest
cob_light/SetLightModeResponse

Message Types
cob_light/SetLightModeResult
cob_lookat_action/LookAtAction
cob_lookat_action/LookAtActionFeedback
cob_lookat_action/LookAtActionGoal
cob_lookat_action/LookAtActionResult
cob_lookat_action/LookAtFeedback
cob_lookat_action/LookAtGoal
cob_lookat_action/LookAtResult
cob_pick_place_action/CobPickAction
cob_pick_place_action/CobPickActionGoal
cob_pick_place_action/CobPickGoal
cob_script_server/ScriptState
cob_script_server/StateAction
cob_script_server/StateActionGoal
cob_script_server/StateGoal
cob_sound/SayAction
cob_sound/SayActionGoal
cob_sound/SayActionResult
cob_sound/SayGoal
cob_sound/SayResult
cob_srvs/SetFloatResponse
cob_srvs/SetIntResponse
cob_srvs/SetStringResponse
control_toolbox/SetPidGainsRequest
controller_manager_msgs/ControllerState
controller_manager_msgs/ListControllersResponse
controller_manager_msgs/SwitchControllerRequest
data_vis_msgs/DataVis
data_vis_msgs/ValueList
designator_integration_msgs/Designator
designator_integration_msgs/DesignatorCommunicationRequest
designator_integration_msgs/DesignatorCommunicationResponse
designator_integration_msgs/DesignatorRequest
designator_integration_msgs/DesignatorResponse
designator_integration_msgs/KeyValuePair
gateway_msgs/GatewayInfo

Message Types
gateway_msgs/RemoteGateway
gateway_msgs/RemoteGatewayInfoResponse
graph_msgs/GeometryGraph
grizzly_msgs/Ambience
hector_nav_msgs/GetNormalResponse
image_view2/ImageMarker2
jsk_rviz_plugins/OverlayMenu
jsk_topic_tools/UpdateRequest
kobuki_msgs/ButtonEvent
kobuki_msgs/KeyboardInput
leap_motion/leapros
moveit_msgs/AttachedCollisionObject
moveit_msgs/CollisionObject
moveit_msgs/Constraints
moveit_msgs/DisplayRobotState
moveit_msgs/DisplayTrajectory
moveit_msgs/ExecuteKnownTrajectoryResponse
moveit_msgs/GetCartesianPathRequest
moveit_msgs/GetCartesianPathResponse
moveit_msgs/GetMotionPlanRequest
moveit_msgs/GetMotionPlanResponse
moveit_msgs/GetPlanningSceneResponse
moveit_msgs/GetPositionFKRequest
moveit_msgs/GetPositionFKResponse
moveit_msgs/GetPositionIKRequest
moveit_msgs/GetPositionIKResponse
moveit_msgs/GetStateValidityRequest
moveit_msgs/MotionPlanDetailedResponse
moveit_msgs/MotionPlanRequest
moveit_msgs/MotionPlanResponse
moveit_msgs/MoveGroupAction
moveit_msgs/MoveGroupActionGoal
moveit_msgs/MoveGroupActionResult
moveit_msgs/MoveGroupGoal
moveit_msgs/MoveGroupResult
moveit_msgs/MoveItErrorCodes

Message Types
moveit_msgs/OrientationConstraint
moveit_msgs/PickupAction
moveit_msgs/PickupActionGoal
moveit_msgs/PickupActionResult
moveit_msgs/PickupGoal
moveit_msgs/PickupResult
moveit_msgs/PlaceAction
moveit_msgs/PlaceActionGoal
moveit_msgs/PlaceActionResult
moveit_msgs/PlaceGoal
moveit_msgs/PlaceLocation
moveit_msgs/PlaceResult
moveit_msgs/PlannerInterfaceDescription
moveit_msgs/PlanningOptions
moveit_msgs/PlanningScene
moveit_msgs/PlanningSceneWorld
moveit_msgs/PositionIKRequest
moveit_msgs/QueryPlannerInterfacesResponse
moveit_msgs/RobotState
moveit_msgs/TrajectoryConstraints
pddl_msgs/PDDLAction
pddl_msgs/PDDLActionArray
pddl_msgs/PDDLDomain
pddl_msgs/PDDLPlannerAction
pddl_msgs/PDDLPlannerActionGoal
pddl_msgs/PDDLPlannerActionResult
pddl_msgs/PDDLPlannerGoal
pddl_msgs/PDDLPlannerResult
pddl_msgs/PDDLStep
posedetection_msgs/DetectResponse
posedetection_msgs/Object6DPose
posedetection_msgs/ObjectDetection
roboteq_msgs/Command
robotnik_msgs/Axis
robotnik_msgs/MotorStatus
robotnik_msgs/MotorsStatus

Message Types
rocon_std_msgs/MasterInfo
rocon_std_msgs/Strings
scheduler_msgs/CurrentStatus
scheduler_msgs/KnownResources
scheduler_msgs/Request
scheduler_msgs/Resource
scheduler_msgs/SchedulerRequests
sound_play/SoundRequest
stdr_msgs/KinematicMsg
stdr_msgs/RegisterGuiResponse
stdr_msgs/RegisterRobotAction
stdr_msgs/RegisterRobotActionResult
stdr_msgs/RegisterRobotResult
stdr_msgs/RobotIndexedMsg
stdr_msgs/RobotIndexedVectorMsg
stdr_msgs/RobotMsg
stdr_msgs/SpawnRobotAction
stdr_msgs/SpawnRobotActionGoal
stdr_msgs/SpawnRobotActionResult
stdr_msgs/SpawnRobotGoal
stdr_msgs/SpawnRobotResult
visp_tracker/InitRequest
visp_tracker/KltSettings
visp_tracker/MovingEdgeSettings
wireless_msgs/Connection

R2022a: Newly Added ROS Message Types
Behavior changed in R2022a

Message Types
adhoc_communication/ExpAuctionElement
adhoc_communication/ExpClusterElement
audio_common_msgs/AudioInfo
baxter_core_msgs/URDFConfiguration
clearpath_base/GPADCOutput
clearpath_base/GPIO
clearpath_base/Joy
clearpath_base/JoySwitch

Message Types
clearpath_base/Magnetometer
clearpath_base/Orientation
clearpath_base/RotateRate
clearpath_base/StateChange
cob_light/ColorRGBAArray
cob_light/LightModes
cob_light/Sequence
cob_light/StopLightModeRequest
cob_light/StopLightModeResponse
cob_perception_msgs/ColorDepthImage
cob_perception_msgs/ColorDepthImageArray
cob_perception_msgs/Detection
cob_perception_msgs/DetectionArray
cob_perception_msgs/Float64ArrayStamped
cob_perception_msgs/Mask
cob_perception_msgs/People
cob_perception_msgs/Person
cob_perception_msgs/PersonStamped
cob_perception_msgs/PositionMeasurement
cob_perception_msgs/PositionMeasurementArray
cob_perception_msgs/Rect
cob_perception_msgs/Skeleton
cob_script_server/ComposeTrajectoryRequest
cob_script_server/ComposeTrajectoryResponse
cob_sound/PlayAction
cob_sound/PlayActionFeedback
cob_sound/PlayActionGoal
cob_sound/PlayActionResult
cob_sound/PlayFeedback
cob_sound/PlayGoal
cob_sound/PlayResult
cob_srvs/DockRequest
cob_srvs/DockResponse
controller_manager_msgs/HardwareInterfaceResources
data_vis_msgs/Speech
data_vis_msgs/Task

Message Types
data_vis_msgs/TaskTree
fkie_multimaster_msgs/DiscoverMastersRequest
fkie_multimaster_msgs/DiscoverMastersResponse
fkie_multimaster_msgs/GetSyncInfoRequest
fkie_multimaster_msgs/GetSyncInfoResponse
fkie_multimaster_msgs/LinkState
fkie_multimaster_msgs/LinkStatesStamped
fkie_multimaster_msgs/LoadLaunchRequest
fkie_multimaster_msgs/LoadLaunchResponse
fkie_multimaster_msgs/MasterState
fkie_multimaster_msgs/ROSMaster
fkie_multimaster_msgs/SyncMasterInfo
fkie_multimaster_msgs/SyncServiceInfo
fkie_multimaster_msgs/SyncTopicInfo
fkie_multimaster_msgs/TaskRequest
fkie_multimaster_msgs/TaskResponse
gateway_msgs/ConnectionStatistics
gateway_msgs/RemoteRuleWithStatus
gazebo_msgs/DeleteLightRequest
gazebo_msgs/DeleteLightResponse
gazebo_msgs/GetLightPropertiesRequest
gazebo_msgs/GetLightPropertiesResponse
gazebo_msgs/PerformanceMetrics
gazebo_msgs/SensorPerformanceMetric
gazebo_msgs/SetLightPropertiesRequest
gazebo_msgs/SetLightPropertiesResponse
geographic_msgs/GeoPath
geographic_msgs/GeoPointStamped
geographic_msgs/GeoPoseStamped
geographic_msgs/GetGeoPathRequest
geographic_msgs/GetGeoPathResponse
grizzly_msgs/Indicators
grizzly_msgs/Status
hector_mapping/ResetMappingRequest
hector_mapping/ResetMappingResponse
image_view2/ChangeModeRequest

Message Types
image_view2/ChangeModeResponse
image_view2/MouseEvent
jsk_footstep_controller/FootCoordsLowLevelInfo
jsk_footstep_controller/GoPosAction
jsk_footstep_controller/GoPosActionFeedback
jsk_footstep_controller/GoPosActionGoal
jsk_footstep_controller/GoPosActionResult
jsk_footstep_controller/GoPosFeedback
jsk_footstep_controller/GoPosGoal
jsk_footstep_controller/GoPosResult
jsk_footstep_controller/GroundContactState
jsk_footstep_controller/LookAroundGroundAction
jsk_footstep_controller/LookAroundGroundActionFeedback
jsk_footstep_controller/LookAroundGroundActionGoal
jsk_footstep_controller/LookAroundGroundActionResult
jsk_footstep_controller/LookAroundGroundFeedback
jsk_footstep_controller/LookAroundGroundGoal
jsk_footstep_controller/LookAroundGroundResult
jsk_footstep_controller/RequireMonitorStatusRequest
jsk_footstep_controller/RequireMonitorStatusResponse
jsk_footstep_controller/SynchronizedForces
jsk_gui_msgs/SlackMessage
jsk_gui_msgs/YesNoRequest
jsk_gui_msgs/YesNoResponse
jsk_network_tools/AllTypeTest
jsk_network_tools/CompressedAngleVectorPR2
jsk_network_tools/FC2OCS
jsk_network_tools/FC2OCSLargeData
jsk_network_tools/OCS2FC
jsk_network_tools/OpenNISample
jsk_network_tools/SetSendRateRequest
jsk_network_tools/SetSendRateResponse
jsk_network_tools/SilverhammerInternalBuffer
jsk_network_tools/WifiStatus
jsk_rviz_plugins/EusCommandRequest
jsk_rviz_plugins/EusCommandResponse

Message Types
jsk_rviz_plugins/ObjectFitCommand
jsk_rviz_plugins/Pictogram
jsk_rviz_plugins/PictogramArray
jsk_rviz_plugins/RecordCommand
jsk_rviz_plugins/RequestMarkerOperateRequest
jsk_rviz_plugins/RequestMarkerOperateResponse
jsk_rviz_plugins/ScreenshotRequest
jsk_rviz_plugins/ScreenshotResponse
jsk_rviz_plugins/StringStamped
jsk_rviz_plugins/TransformableMarkerOperate
jsk_topic_tools/ChangeTopicRequest
jsk_topic_tools/ChangeTopicResponse
jsk_topic_tools/PassthroughDurationRequest
jsk_topic_tools/PassthroughDurationResponse
kingfisher_msgs/Power
kobuki_msgs/ScanAngle
leap_motion/Bone
leap_motion/Finger
leap_motion/Gesture
leap_motion/Hand
leap_motion/Human
move_base_msgs/RecoveryStatus
moveit_msgs/ApplyPlanningSceneRequest
moveit_msgs/ApplyPlanningSceneResponse
moveit_msgs/CartesianPoint
moveit_msgs/CartesianTrajectory
moveit_msgs/CartesianTrajectoryPoint
moveit_msgs/ChangeControlDimensionsRequest
moveit_msgs/ChangeControlDimensionsResponse
moveit_msgs/ChangeDriftDimensionsRequest
moveit_msgs/ChangeDriftDimensionsResponse
moveit_msgs/CheckIfRobotStateExistsInWarehouseRequest
moveit_msgs/CheckIfRobotStateExistsInWarehouseResponse
moveit_msgs/DeleteRobotStateFromWarehouseRequest
moveit_msgs/DeleteRobotStateFromWarehouseResponse
moveit_msgs/ExecuteTrajectoryAction

Message Types
moveit_msgs/ExecuteTrajectoryActionFeedback
moveit_msgs/ExecuteTrajectoryActionGoal
moveit_msgs/ExecuteTrajectoryActionResult
moveit_msgs/ExecuteTrajectoryFeedback
moveit_msgs/ExecuteTrajectoryGoal
moveit_msgs/ExecuteTrajectoryResult
moveit_msgs/GenericTrajectory
moveit_msgs/GetMotionSequenceRequest
moveit_msgs/GetMotionSequenceResponse
moveit_msgs/GetPlannerParamsRequest
moveit_msgs/GetPlannerParamsResponse
moveit_msgs/GetRobotStateFromWarehouseRequest
moveit_msgs/GetRobotStateFromWarehouseResponse
moveit_msgs/GraspPlanningRequest
moveit_msgs/GraspPlanningResponse
moveit_msgs/ListRobotStatesInWarehouseRequest
moveit_msgs/ListRobotStatesInWarehouseResponse
moveit_msgs/MotionSequenceItem
moveit_msgs/MotionSequenceRequest
moveit_msgs/MotionSequenceResponse
moveit_msgs/MoveGroupSequenceAction
moveit_msgs/MoveGroupSequenceActionFeedback
moveit_msgs/MoveGroupSequenceActionGoal
moveit_msgs/MoveGroupSequenceActionResult
moveit_msgs/MoveGroupSequenceFeedback
moveit_msgs/MoveGroupSequenceGoal
moveit_msgs/MoveGroupSequenceResult
moveit_msgs/PlannerParams
moveit_msgs/RenameRobotStateInWarehouseRequest
moveit_msgs/RenameRobotStateInWarehouseResponse
moveit_msgs/SaveRobotStateToWarehouseRequest
moveit_msgs/SaveRobotStateToWarehouseResponse
moveit_msgs/SetPlannerParamsRequest
moveit_msgs/SetPlannerParamsResponse
moveit_msgs/UpdatePointcloudOctomapRequest
moveit_msgs/UpdatePointcloudOctomapResponse

Message Types
multisense_ros/DeviceStatus
posedetection_msgs/TargetObjRequest
posedetection_msgs/TargetObjResponse
rmp_msgs/RMPBatteryStatus
rmp_msgs/RMPCommand
rmp_msgs/RMPFeedback
robotnik_msgs/BatteryDockingStatus
robotnik_msgs/BatteryDockingStatusStamped
robotnik_msgs/BatteryStatus
robotnik_msgs/BatteryStatusStamped
robotnik_msgs/BoolArray
robotnik_msgs/Cartesian_Euler_pose
robotnik_msgs/ElevatorAction
robotnik_msgs/ElevatorStatus
robotnik_msgs/GetBoolRequest
robotnik_msgs/GetBoolResponse
robotnik_msgs/GetMotorsHeadingOffsetRequest
robotnik_msgs/GetMotorsHeadingOffsetResponse
robotnik_msgs/InsertTaskRequest
robotnik_msgs/InsertTaskResponse
robotnik_msgs/InverterStatus
robotnik_msgs/LaserMode
robotnik_msgs/LaserStatus
robotnik_msgs/MotorHeadingOffset
robotnik_msgs/MotorPID
robotnik_msgs/MotorsStatusDifferential
robotnik_msgs/Pose2DArray
robotnik_msgs/Pose2DStamped
robotnik_msgs/PresenceSensor
robotnik_msgs/PresenceSensorArray
robotnik_msgs/QueryAlarm
robotnik_msgs/QueryAlarmsRequest
robotnik_msgs/QueryAlarmsResponse
robotnik_msgs/Register
robotnik_msgs/Registers
robotnik_msgs/ResetFromSubStateRequest

Message Types
robotnik_msgs/ResetFromSubStateResponse
robotnik_msgs/ReturnMessage
robotnik_msgs/RobotnikMotorsStatus
robotnik_msgs/SafetyModuleStatus
robotnik_msgs/SetBuzzerRequest
robotnik_msgs/SetBuzzerResponse
robotnik_msgs/SetByteRequest
robotnik_msgs/SetByteResponse
robotnik_msgs/SetElevatorAction
robotnik_msgs/SetElevatorActionFeedback
robotnik_msgs/SetElevatorActionGoal
robotnik_msgs/SetElevatorActionResult
robotnik_msgs/SetElevatorFeedback
robotnik_msgs/SetElevatorGoal
robotnik_msgs/SetElevatorRequest
robotnik_msgs/SetElevatorResponse
robotnik_msgs/SetElevatorResult
robotnik_msgs/SetEncoderTurnsRequest
robotnik_msgs/SetEncoderTurnsResponse
robotnik_msgs/SetLaserModeRequest
robotnik_msgs/SetLaserModeResponse
robotnik_msgs/SetMotorModeRequest
robotnik_msgs/SetMotorModeResponse
robotnik_msgs/SetMotorPIDRequest
robotnik_msgs/SetMotorPIDResponse
robotnik_msgs/SetMotorStatusRequest
robotnik_msgs/SetMotorStatusResponse
robotnik_msgs/SetNamedDigitalOutputRequest
robotnik_msgs/SetNamedDigitalOutputResponse
robotnik_msgs/SetTransformRequest
robotnik_msgs/SetTransformResponse
robotnik_msgs/State
robotnik_msgs/StringArray
robotnik_msgs/SubState
robotnik_msgs/ack_alarmRequest
robotnik_msgs/ack_alarmResponse

Message Types
robotnik_msgs/alarmmonitor
robotnik_msgs/alarmsmonitor
robotnik_msgs/get_alarmsRequest
robotnik_msgs/get_alarmsResponse
robotnik_msgs/get_modbus_registerRequest
robotnik_msgs/get_modbus_registerResponse
robotnik_msgs/named_input_output
robotnik_msgs/named_inputs_outputs
robotnik_msgs/set_CartesianEuler_poseRequest
robotnik_msgs/set_CartesianEuler_poseResponse
robotnik_msgs/set_modbus_registerRequest
robotnik_msgs/set_modbus_registerResponse
robotnik_msgs/set_named_digital_outputRequest
robotnik_msgs/set_named_digital_outputResponse
rocon_std_msgs/Connection
rocon_std_msgs/ConnectionCacheSpin
rocon_std_msgs/ConnectionsDiff
rocon_std_msgs/ConnectionsList
rocon_std_msgs/EmptyStringRequest
rocon_std_msgs/EmptyStringResponse
rocon_std_msgs/Float32Stamped
roseus/FixedArray
roseus/TestName
roseus/VariableArray
rospy_message_converter/NestedUint8ArrayTestMessage
rospy_message_converter/NestedUint8ArrayTestServiceRequest
rospy_message_converter/NestedUint8ArrayTestServiceResponse
rospy_message_converter/Uint8Array3TestMessage
rospy_message_converter/Uint8ArrayTestMessage
rtt_ros_msgs/EvalRequest
rtt_ros_msgs/EvalResponse
schunk_sdh/PressureArray
schunk_sdh/PressureArrayList
schunk_sdh/TemperatureArray
sound_play/SoundRequestAction
sound_play/SoundRequestActionFeedback

Message Types
sound_play/SoundRequestActionGoal
sound_play/SoundRequestActionResult
sound_play/SoundRequestFeedback
sound_play/SoundRequestGoal
sound_play/SoundRequestResult
speech_recognition_msgs/Grammar
speech_recognition_msgs/PhraseRule
speech_recognition_msgs/SpeechRecognitionRequest
speech_recognition_msgs/SpeechRecognitionResponse
speech_recognition_msgs/Vocabulary
visp_tracker/TrackerSettings

R2022a: Deleted ROS Message Types
Behavior changed in R2022a

Message Types
baxter_core_msgs/ITBState
baxter_core_msgs/ITBStates
cob_relayboard/EmergencyStopState
cob_sound/SayTextRequest
cob_sound/SayTextResponse
cob_srvs/GetPoseStampedTransformedRequest
cob_srvs/GetPoseStampedTransformedResponse
cob_srvs/SetDefaultVelRequest
cob_srvs/SetDefaultVelResponse
cob_srvs/SetJointStiffnessRequest
cob_srvs/SetJointStiffnessResponse
cob_srvs/SetJointTrajectoryRequest
cob_srvs/SetJointTrajectoryResponse
cob_srvs/SetMaxVelRequest
cob_srvs/SetMaxVelResponse
cob_srvs/SetOperationModeRequest
cob_srvs/SetOperationModeResponse
cob_srvs/TriggerRequest
cob_srvs/TriggerResponse
grizzly_msgs/Drive
grizzly_msgs/RawStatus
jsk_gui_msgs/DeviceSensorALL

Message Types
jsk_gui_msgs/Imu
moveit_msgs/GetConstraintAwarePositionIKRequest
moveit_msgs/GetConstraintAwarePositionIKResponse
moveit_msgs/GetKinematicSolverInfoRequest
moveit_msgs/GetKinematicSolverInfoResponse
rmp_msgs/AudioCommand
rmp_msgs/Battery
rmp_msgs/BoolStamped
rmp_msgs/FaultStatus
rmp_msgs/MotorStatus
rocon_std_msgs/GetPlatformInfoRequest
rocon_std_msgs/GetPlatformInfoResponse
rocon_std_msgs/PlatformInfo
rosserial_msgs/RequestMessageInfoRequest
rosserial_msgs/RequestMessageInfoResponse

R2022a: Message Packages Deprecated in ROS Noetic
Behavior changed in R2022a

Message Packages
cob_camera_sensors
cob_kinematics
cob_relayboard
cob_trajectory_controller
hrpsys_gazebo_msgs
iai_pancake_perception_action
jaco_msgs
linux_hardware
lizi
mln_robosherlock_msgs
mongodb_store_msgs
monocam_settler
nao_interaction_msgs
nao_msgs
network_monitor_udp
nmea_msgs
p2os_driver
pano_ros

Message Packages
pcl_msgs
play_motion_msgs
program_queue
rosauth
saphari_msgs
scanning_table_msgs
segbot_gui
sherlock_sim_msgs
simple_robot_control
sr_ronex_msgs
statistics_msgs
underwater_sensor_msgs
uuid_msgs
yocs_msgs

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for `struct` messages.
- For messages with fields containing cell array of strings, such as `sensor_msgs/JointState`, accessing those fields in the MATLAB function is not supported.

See Also

Functions
`rostopic` | `rosmsg`

Objects
`rospublisher` | `rossubscriber`

Topics
"Work with Basic ROS Messages"
"Built-In Message Support"

rosmsg

Retrieve information about ROS messages and message types

Syntax

```
rosmsg show msgtype
rosmsg md5 msgtype
rosmsg list

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

Description

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

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

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

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

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

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

Examples

Retrieve Message Type Definition

```
msgInfo = rosmsg('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 = rosmsg('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.

Version History

Introduced in R2019b

rosnode

Retrieve information about ROS network nodes

Syntax

```
rosnode list
rosnode info nodename
rosnode ping nodename

nodelist = 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.

`nodelist = 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.

```
rosshutdown
```

```
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.

```
rosshutdown
```

```
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`.

`nodelist` — List of node names available
cell array of character vectors

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

Version History
Introduced in R2019b

See Also
`rosinit` | `rostopic`

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 2-183 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
```

```
Launching ROS Core...
..Done in 2.6882 seconds.
Initializing ROS master on http://172.30.131.134:57364.
Initializing global node /matlab_global_node_00921 with NodeURI http://bat6234win64:64435/ and Ma
```

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 specific 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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_00921 with NodeURI http://bat6234win64:64435/ and M
Shutting down ROS master on http://172.30.131.134:57364.
```

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 = 33×1 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'     }
    {'/rosdistro'                                   }
    {'/roslaunch/uris/host_192_168_17_128__34863'   }
    {'/rosversion'                                  }
    {'/run_id'                                      }
    {'/use_sim_time'                                }
```

Shut down the ROS network.

```
rosshutdown
```

```
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
```

```
Launching ROS Core...
..Done in 2.7008 seconds.
Initializing ROS master on http://172.30.131.134:54064.
Initializing global node /matlab_global_node_04167 with NodeURI http://bat6234win64:65339/ and Ma
```

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 parameter 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_04167 with NodeURI http://bat6234win64:65339/ and N
Shutting down ROS master on http://172.30.131.134:54064.
```

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	`int32`
Boolean	`logical`
double	`double`
string	character vector (`char`)
list	cell array (`cell`)
dictionary	structure (`struct`)

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

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

For code generation, only the following ROS data types are supported as values of parameters,

- 32-bit integer — `int32`
- Boolean — `logical`
- double — `double`
- strings — string scalar, `string`, or character vector, `char`

See Also

Functions
`get` | `set` | `has` | `search` | `del`

Objects
`ParameterTree`

rosReadAllFieldNames

Get all available field names from ROS or ROS 2 point cloud message structure

Syntax

`names = rosReadAllFieldNames(pcloud)`

Description

`names = rosReadAllFieldNames(pcloud)` returns all the fields that are stored in the ROS or ROS 2 `'sensor_msgs/PointCloud2'` message structure, `pcloud`, and returns them in `names`.

Input Arguments

`pcloud` — Point cloud
`'sensor_msgs/PointCloud2'` message structure

Point cloud, specified as a message structure for ROS or ROS 2 `'sensor_msgs/PointCloud2'` message.

Output Arguments

`names` — List of field names in `PointCloud2` object
cell array of character vectors

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

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

`rosReadXYZ` | `rosReadField` | `rosReadRGB` | `rosReadCartesian`

Topics
"Work with Specialized ROS Messages"
"Improve Performance of ROS Using Message Structures"

rosReadBinaryOccupancyGrid

Read binary occupancy grid data from ROS or ROS 2 message structure

Syntax

```
map = rosReadBinaryOccupancyGrid(msg)
map = rosReadBinaryOccupancyGrid(msg,thresh)
map = rosReadBinaryOccupancyGrid(msg,thresh,val)
```

Description

`map = rosReadBinaryOccupancyGrid(msg)` returns a `binaryOccupancyGrid` object by reading the data inside a ROS or ROS 2 message structure, `msg`, which must be a `'nav_msgs/OccupancyGrid'` message. All message data values greater than or equal to the occupancy threshold are set to occupied, `1`, in the map. All other values, including unknown values (`-1`) are set to unoccupied, `0`, in the map.

`map = rosReadBinaryOccupancyGrid(msg,thresh)` specifies a threshold, `thresh`, for occupied values. All values greater than or equal to the threshold are set to occupied, `1`. All other values are set to unoccupied, `0`.

`map = rosReadBinaryOccupancyGrid(msg,thresh,val)` specifies a value to set for unknown values (`-1`). By default, all unknown values are set to unoccupied, `0`.

Input Arguments

`msg` — ROS or ROS 2 occupancy grid message
`'nav_msgs/OccupancyGrid'` message structure

ROS or ROS 2 `'nav_msgs/OccupancyGrid'` message, specified as a message structure.

`thresh` — Threshold for occupied values
50 (default) | scalar

Threshold for occupied values, specified as a scalar. Any value greater than or equal to the threshold is set to occupied, `1`. All other values are set to unoccupied, `0`.

Data Types: `double`

`val` — Value to replace unknown values
0 (default) | 1

Value to replace unknown values, specified as either `0` or `1`. Unknown message values (`-1`) are set to the given value.

Data Types: `double` | `logical`

Output Arguments

`map` — Binary occupancy grid
`binaryOccupancyMap` object handle

Binary occupancy grid, returned as a `binaryOccupancyMap` object handle. `map` contains the occupancy data from a `'nav_msgs/OccupancyGrid'` message converted to binary values. The object stores a grid of binary values, where `1` indicates an occupied location and `0` indications an unoccupied location.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

Functions

`rosReadOccupancyGrid` | `rosReadOccupancyMap3D` | `rosWriteBinaryOccupancyGrid` | `rosWriteOccupancyGrid`

Topics

"Work with Specialized ROS Messages"
"Improve Performance of ROS Using Message Structures"

rosReadCartesian

Read laser scan ranges in Cartesian coordinates from ROS or ROS 2 message structure

Syntax

```
cart = rosReadCartesian(scan)
cart = rosReadCartesian( ___ ,Name,Value)
[cart,angles] = rosReadCartesian( ___ )
```

Description

`cart = rosReadCartesian(scan)` converts the polar measurements of the ROS or ROS 2 laser scan message structure, `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 = rosReadCartesian( ___ ,Name,Value)` provides additional options specified by one or more `Name,Value` pair arguments. You can specify several name-value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

`[cart,angles] = rosReadCartesian( ___ )` 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.

Input Arguments

`scan` — Laser scan message
`LaserScan` structure

ROS or ROS 2 laser scan message of type `'sensor_msgs/LaserScan'`, specified as a message structure.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'RangeLimits',[0.05 2]` sets the range limits for the scan in meters

`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.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rosReadScanAngles` | `rosReadXYZ` | `rosPlot`

Topics
"Work with Specialized ROS Messages"
"Improve Performance of ROS Using Message Structures"

rosReadField

Read point cloud data from ROS or ROS 2 message structure based on field name

Syntax

```
fielddata = rosReadField(pcloud,fieldname)
fielddata = rosReadField(pcloud,fieldname,"PreserveStructureOnRead",true)
fielddata = rosReadField(pcloud,fieldname,"Datatype","double")
```

Description

`fielddata = rosReadField(pcloud,fieldname)` reads the point field from the ROS or ROS 2 `'sensor_msgs/PointCloud2'` message structure, `pcloud`, specified by `fieldname` and returns it in `fielddata`.

`fielddata = rosReadField(pcloud,fieldname,"PreserveStructureOnRead",true)` preserves the organizational structure of the point cloud field data returned in the `fielddata` output. For more information, see “Preserving Point Cloud Structure” on page 2-191.

`fielddata = rosReadField(pcloud,fieldname,"Datatype","double")` reads the point field data in double precision during code generation. If you use this syntax for MATLAB execution, the function always reads the data in the precision specified by the corresponding field in the input message structure, `pcloud`.

Input Arguments

`pcloud` — Point cloud
`'sensor_msgs/PointCloud2'` message structure

Point cloud, specified as a message structure for ROS or ROS 2 `'sensor_msgs/PointCloud2'` 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 `PreserveStructureOnRead` name-value pair argument is set to `true`, the points are returned as an *h*-by-*w*-by-*c* matrix.

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 (`rosReadXYZ`, `rosReadRGB`, or `rosReadField`) 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.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

`PointCloud2` | `rosReadAllFieldNames`

Topics

“Work with Specialized ROS Messages”
“Improve Performance of ROS Using Message Structures”

rosReadImage

Convert ROS or ROS 2 image data into MATLAB image

Syntax

```
img = rosReadImage(msg)
msgOut = rosReadImage( ___ ,"Encoding",encodingParam)
[img,alpha] = rosReadImage( ___ )
```

Description

`img = rosReadImage(msg)` converts the raw image data in the ROS or ROS 2 message structure, `msg`, into an image matrix, `img`. You can call `rosReadImage` using either `'sensor_msgs/Image'` or `'sensor_msgs/CompressedImage'` messages.

ROS or ROS 2 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`.

`msgOut = rosReadImage( ___ ,"Encoding",encodingParam)` specifies the encoding of the image message as a name-value argument using any of the previous input arguments. If the `Encoding` field of the message is not set, use this syntax to set the field.

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

Input Arguments

`msg` — ROS or ROS 2 image message
`'sensor_msgs/Image'` message structure | `'sensor_msgs/CompressedImage'` message structure

ROS or ROS 2 `'sensor_msgs/Image'` or `'sensor_msgs/CompressedImage'` message, specified as a message structure.

`encodingParam` — Encoding of image message
`"rgb8"` | `"rgba8"` | `"rgb16"` | string scalar

Encoding of image message, specified as a string scalar. Using this input argument overwrites the `Encoding` field of the input `msg`. For more information, see “Supported Image Encodings” on page 2-193.

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 or ROS 2 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`.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Not supported for `CompressedImage` messages.
- Specify the `"Encoding"`,`encodParam` name-value argument when generating code.

See Also

`rosWriteImage` | `rosReadRGB`

Topics
“Work with Specialized ROS Messages”
“Improve Performance of ROS Using Message Structures”

rosReadLidarScan

Display lidar scan or point cloud from ROS or ROS 2 message structure

Syntax

```
scans = rosReadLidarScan(scanMsg)
```

Description

`scans = rosReadLidarScan(scanMsg)` creates a `lidarScan` object from a ROS or ROS 2 `sensor_msgs/LaserScan` message structure.

Input Arguments

`scanMsg` — ROS or ROS 2 laser scan message
`sensor_msgs/LaserScan` message structure

ROS or ROS 2 laser scan message of type `sensor_msgs/LaserScan`, specified as a message structure.

Outputs

`scans` — Lidar scan readings
`lidarScan` object

Lidar scan readings, returned as a `lidarScan` object.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rosReadCartesian`

Topics
"Work with Specialized ROS Messages"
"Improve Performance of ROS Using Message Structures"

rosReadOccupancyGrid

Read occupancy grid data from ROS or ROS 2 message structure

Syntax

```
map = rosReadOccupancyGrid(msg)
```

Description

`map = rosReadOccupancyGrid(msg)` returns an `occupancyMap` object by reading the data inside a ROS or ROS 2 message structure, `msg`, which must be a `'nav_msgs/OccupancyGrid'` message. All message data values are converted to probabilities from 0 to 1. The unknown values (-1) in the message are set as 0.5 in the map.

Input Arguments

`msg` — ROS or ROS 2 occupancy grid message
`'nav_msgs/OccupancyGrid'` message structure

ROS or ROS 2 `'nav_msgs/OccupancyGrid'` message, specified as a message structure.

Output Arguments

`map` — Occupancy map
`occupancyMap` object handle

Occupancy map, returned as an `occupancyMap` object handle.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

Functions
`rosReadBinaryOccupancyGrid` | `rosReadOccupancyMap3D` | `rosWriteBinaryOccupancyGrid` | `rosWriteOccupancyGrid`

Topics
“Work with Specialized ROS Messages”
“Improve Performance of ROS Using Message Structures”

rosReadOccupancyMap3D

Read 3-D map from ROS or ROS 2 `Octomap` message structure

Syntax

```
map = rosReadOccupancyMap3D(msg)
```

Description

`map = rosReadOccupancyMap3D(msg)` reads the data inside a ROS or ROS 2 `'octomap_msgs/Octomap'` message to return an `occupancyMap3D` object. All message data values are converted to probabilities from 0 to 1.

Input Arguments

`msg` — ROS or ROS 2 `Octomap` message
`'octomap_msgs/Octomap'` message structure

ROS or ROS 2 `'octomap_msgs/Octomap'` message, specified as a message structure. Get this message by subscribing to an `'octomap_msgs/Octomap'` topic using `rossubscriber` or `ros2subscriber` on a live ROS or ROS 2 network, respectively. You can also create your own message using `rosmessage` or `ros2message`.

Output Arguments

`map` — 3-D occupancy map
`occupancyMap3D` object handle

3-D occupancy map, returned as an `occupancyMap3D` object handle.

Version History

Introduced in R2021a

See Also

`occupancyMap3D` | `rosmessage` | `rossubscriber` | `rosReadOccupancyGrid` | `rosWriteOccupancyGrid`

Topics

"Work with Specialized ROS Messages"
"Improve Performance of ROS Using Message Structures"

rosRegisterMessages

Register ROS custom messages with MATLAB

Syntax

```
rosRegisterMessages(genDir)
```

Description

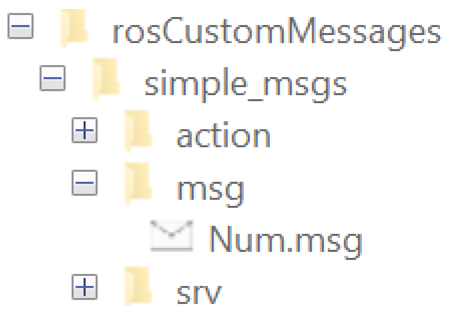
`rosRegisterMessages(genDir)` registers ROS custom messages with MATLAB. Use this function to register the custom messages generated on another computer running on the same platform and same version of MATLAB.

Note The `rosRegisterMessages` function allows sharing of the custom messages on different machines running on the same platform only. Use this `rosgenmsg` function on the new platform if the machines are running on different platforms.

Examples

Register ROS Custom Messages to MATLAB

In this example, you create ROS custom messages in MATLAB® to share them on another machine. This other machine must run on the same platform and use the same version of MATLAB®. You must have a ROS package that contains the required `msg` file, as this figure shows.



Open a new MATLAB session and create a custom message package folder in a local folder. Specify a short folder path when you generate custom messages on a Windows machine to avoid limitations on the number of characters in the folder path. For example,

```
genDir = fullfile('C:/Work/rosCustomMessages')
```

```
genDir = fullfile(pwd,'rosCustomMessages');
packagePath = fullfile(genDir,'simple_msgs');
mkdir(packagePath)
```

Create a folder named `msg` inside the custom message package folder.

```
mkdir(packagePath,'msg')
```

Create a file named `.msg` inside the `msg` folder.

```
messageDefinition = {'int64 num'};

fileID = fopen(fullfile(packagePath,'msg', ...
               'Num.msg'),'w');
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Create a folder named `srv` inside the custom message package folder.

```
mkdir(packagePath,'srv')
```

Create a file named `.srv` inside the `srv` folder.

```
serviceDefinition = {'int64 a'
                     'int64 b'
                     '---'
                     'int64 sum'};

fileID = fopen(fullfile(packagePath,'srv', ...
               'AddTwoInts.srv'),'w');
fprintf(fileID,'%s\n',serviceDefinition{:});
fclose(fileID);
```

Create a folder named `action` inside the custom message package folder.

```
mkdir(packagePath,'action')
```

Create a file named `.action` inside the `action` folder.

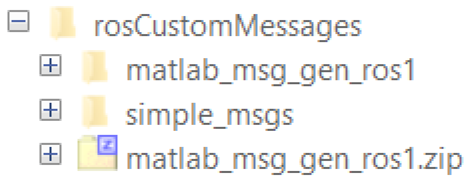
```
actionDefinition = {'int64 goal'
                    '---'
                    'int64 result'
                    '---'
                    'int64 feedback'};

fileID = fopen(fullfile(packagePath,'action', ...
               'Test.action'),'w');
fprintf(fileID,'%s\n',actionDefinition{:});
fclose(fileID);
```

Generate the custom messages from the ROS definitions in `.msg`, `.srv`, and `.action` files as a shareable ZIP archive.

```
rosgenmsg(genDir,CreateShareableFile=true)
```

```
Identifying message files in folder 'C:/Work/rosCustomMessages'.Done.
Validating message files in folder 'C:/Work/rosCustomMessages'.Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running catkin build in folder 'C:/Work/rosCustomMessages/matlab_msg_gen_ros1/win64'.
Build in progress. This may take several minutes...
Build succeeded.build log
Generating zip file in the folder 'C:/Work/rosCustomMessages'.Done.

To use the custom messages, follow these steps:

1. Add the custom message folder to the MATLAB path by executing:
```

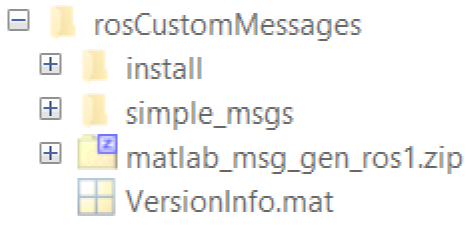
```
addpath('C:\Work\rosCustomMessages\matlab_msg_gen_ros1\win64\install\m')
savepath

2. Refresh all message class definitions, which requires clearing the workspace, by executing:

clear classes
rehash toolboxcache

3. Verify that you can use the custom messages.
   Enter "rosmsg list" and ensure that the output contains the generated
   custom message types.
```



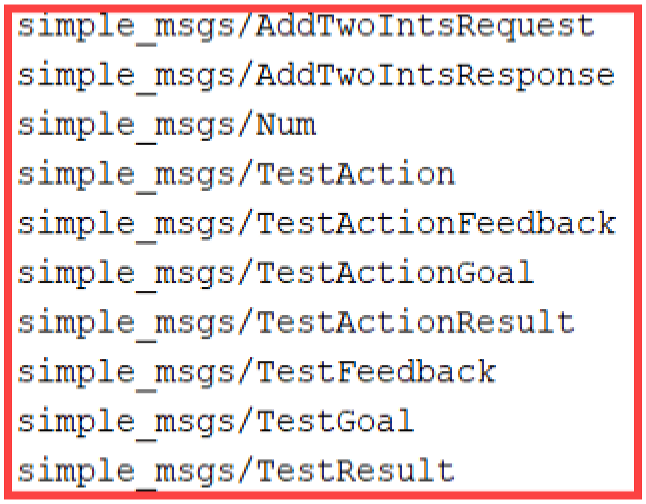
Copy the generated custom messages in the ZIP archive to the target computer and register it using the `rosRegisterMessages` function.

```
rosRegisterMessages(genDir)
```

```
To use the custom messages, follow these steps:

1. Add the custom message folder to the MATLAB path by executing:

addpath('C:\Work\rosCustomMessages\install\m')
savepath

2. Refresh all message class definitions, which requires clearing the workspace, by executing:

clear classes
rehash toolboxcache

3. Verify that you can use the custom messages.
   Enter "rosmsg list" and ensure that the output contains the generated
   custom message types.
```



Run `rosmsg list` on the target computer to view the custom messages for using in the workflow.



```
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
simple_msgs/TestAction
simple_msgs/TestActionFeedback
simple_msgs/TestActionGoal
simple_msgs/TestActionResult
simple_msgs/TestFeedback
simple_msgs/TestGoal
simple_msgs/TestResult
```

Input Arguments

`genDir` — Path to the folder that contains `matlab_msg_gen_ros1.zip` file
string scalar | character vector

Path to the folder that contains `matlab_msg_gen_ros1.zip` file, specified as a string scalar or a character vector. These folders contain a folder named `/msg` with `.msg` files for message definitions, a folder named `/srv` with `.srv` files for service definitions, and a folder named `/action` with `.action` files for action definitions.

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`rosgenmsg`

ros2RegisterMessages

Register ROS 2 custom messages with MATLAB

Syntax

`ros2RegisterMessages(genDir)`

Description

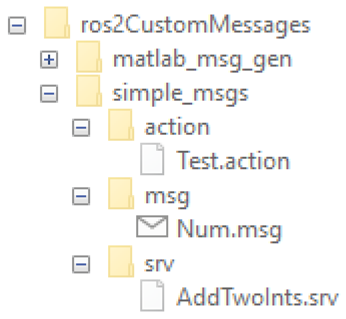
`ros2RegisterMessages(genDir)` registers the custom messages with MATLAB. Use this function to register the custom messages generated on another computer running on the same platform and same version of MATLAB.

Note The `ros2RegisterMessages` function allows sharing of the custom messages on different machines running on the same platform only. Use the `ros2genmsg` function on the new platform if the machines are running on different platforms.

Examples

Register ROS 2 Custom Messages to MATLAB

In this example, you create ROS 2 custom messages in MATLAB® to share them on another machine. This other machine must run on the same platform and same version of MATLAB®. You must have a ROS 2 package that contains the required `msg` file, as this figure shows.



Open a new MATLAB session and create a custom message package folder in a local folder. Specify a short folder path when you generate custom messages on a Windows machine to avoid limitations on the number of characters in the folder path. For example,

`genDir = fullfile('C:/Work/ros2CustomMessages')`.

```
genDir = fullfile(pwd,'ros2CustomMessages');
packagePath = fullfile(genDir,'simple_msgs');
mkdir(packagePath)
```

Create a folder named `msg` inside the custom message package folder.

```
mkdir(packagePath,'msg')
```

Create a file named `.msg` inside the `msg` folder.

```
messageDefinition = {'int64 num'};

fileID = fopen(fullfile(packagePath,'msg', ...
               'Num.msg'),'w');
fprintf(fileID,'%s\n',messageDefinition{:});
fclose(fileID);
```

Create a folder named `srv` inside the custom message package folder.

```
mkdir(packagePath,'srv')
```

Create a file named `.srv` inside the `srv` folder.

```
serviceDefinition = {'int64 a'
                     'int64 b'
                     '---'
                     'int64 sum'};

fileID = fopen(fullfile(packagePath,'srv', ...
               'AddTwoInts.srv'),'w');
fprintf(fileID,'%s\n',serviceDefinition{:});
fclose(fileID);
```

Create a folder named `action` inside the custom message package folder.

```
mkdir(packagePath,'action')
```

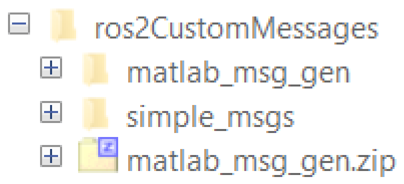
Create a file named `.action` inside the `action` folder.

```
actionDefinition = {'int64 goal'
                    '---'
                    'int64 result'
                    '---'
                    'int64 feedback'};

fileID = fopen(fullfile(packagePath,'action', ...
               'Test.action'),'w');
fprintf(fileID,'%s\n',actionDefinition{:});
fclose(fileID);
```

Generate the custom messages from the ROS 2 definitions in `.msg`, `.srv` and `.action` files as a shareable ZIP archive.

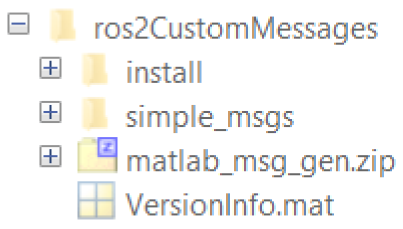
```
ros2genmsg(genDir,CreateShareableFile=true)
```

```
Identifying message files in folder 'C:/Work/ros2CustomMessages'.Done.
Validating message files in folder 'C:/Work/ros2CustomMessages'.Done.
[1/1] Generating MATLAB interfaces for custom message packages... Done.
Running colcon build in folder 'C:/Work/ros2CustomMessages/matlab_msg_gen/win64'.
Build in progress. This may take several minutes...
Build succeeded.build log
Generating zip file in the folder 'C:/Work/ros2CustomMessages'.Done.
```

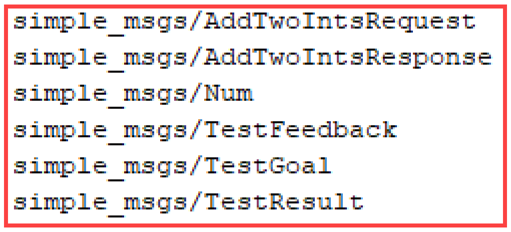


Copy the generated custom messages in the ZIP archive to the target computer and register it using the `ros2RegisterMessages` function.

```
ros2RegisterMessages(genDir)
```



Run `ros2 msg list` on the target computer to view the custom messages for using in the workflow.



```
simple_msgs/AddTwoIntsRequest
simple_msgs/AddTwoIntsResponse
simple_msgs/Num
simple_msgs/TestFeedback
simple_msgs/TestGoal
simple_msgs/TestResult
```

Input Arguments

`genDir` — Path to the folder that contains `matlab_msg_gen.zip` file
string scalar | character vector

Path to the folder that contains `matlab_msg_gen.zip` file, specified as a string scalar or a character vector. These folders contain a folder named `/msg` with `.msg` files for message definitions, a folder named `/srv` with `.srv` files for service definitions, and a folder named `/action` with `.action` files for action definitions.

Data Types: `char` | `string`

Version History

Introduced in R2022b

See Also

`ros2genmsg`

Topics
"Create Shareable ROS 2 Custom Message Package" on page 2-119

"Add Input Layer to dlnetwork" (Deep Learning Toolbox)

rosPlot

Display lidar scan or point cloud from ROS or ROS 2 message structures

Syntax

```
rosPlot(scanMsg)
rosPlot(ptcloudMsg)
rosPlot( ___ ,Name,Value)
linehandle = plot( ___ )
```

Description

`rosPlot(scanMsg)` plots the laser scan readings specified in the input ROS or ROS 2 `sensor_msgs/LaserScan` message structure. Axes are automatically scaled to the maximum range of the sensor.

`rosPlot(ptcloudMsg)` plots the point cloud readings specified in the input `sensor_msgs/PointCloud2` message structure.

`rosPlot( ___ ,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.

Input Arguments

`scanMsg` — Laser scan message
`LaserScan` message structure

ROS or ROS 2 message of type `sensor_msgs/LaserScan`, specified as a message structure.

`ptcloudMsg` — Point cloud message
message structure

ROS or ROS 2 message of type `sensor_msgs/PointCloud2`, specified as a message structure.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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.

Version History
Introduced in R2021a

See Also
`rosReadCartesian`

Topics
“Work with Specialized ROS Messages”
“Improve Performance of ROS Using Message Structures”

rosReadQuaternion

Create MATLAB quaternion object from ROS or ROS 2 message structure

Syntax

`q = rosReadQuaternion(quatMsg)`

Description

`q = rosReadQuaternion(quatMsg)` creates a `quaternion` object from a ROS or ROS 2 `geometry_msgs/Quaternion` message structure.

Input Arguments

`quatMsg` — ROS or ROS 2 quaternion message
`geometry_msgs/Quaternion` message structure

ROS or ROS 2 message of type `geometry_msgs/Quaternion`, specified as a message structure.

Outputs

q — Quaternion
`quaternion` object

Quaternion, returned as a `quaternion` object.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rosReadCartesian` | `quaternion`

Topics
"Work with Specialized ROS Messages"
"Improve Performance of ROS Using Message Structures"

rosReadRGB

Extract RGB color values from ROS or ROS 2 point cloud message structure

Syntax

```
rgb = rosReadRGB(pcloud)
rgb = rosReadRGB(pcloud,"PreserveStructureOnRead",true)
fielddata = rosReadRGB(pcloud,"Datatype","double")
```

Description

`rgb = rosReadRGB(pcloud)` extracts the `[r g b]` values from all points in the ROS or ROS 2 `'sensor_msgs/PointCloud2'` message structure, `pcloud`, and returns them as an *n*-by-3 matrix of *n* 3-D point coordinates.

`rgb = rosReadRGB(pcloud,"PreserveStructureOnRead",true)` preserves the organizational structure of the point cloud returned in the `rgb` output. For more information, see "Preserving Point Cloud Structure" on page 2-209.

`fielddata = rosReadRGB(pcloud,"Datatype","double")` reads the `[r g b]` data in double precision during code generation. If you use this syntax for MATLAB execution, the function always reads the data in the precision specified by the corresponding field in the input message structure, `pcloud`.

Input Arguments

`pcloud` — Point cloud
`'sensor_msgs/PointCloud2'` message structure

Point cloud, specified as a message structure for ROS or ROS 2 `'sensor_msgs/PointCloud2'` 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 a *n*-by-3 matrix.

If the `PreserveStructureOnRead` name-value pair argument is set to `true`, the points are returned as an *h*-by-*w*-by-3 matrix.

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 (`rosReadXYZ`, `rosReadRGB`, or `rosReadField`) 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.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

`PointCloud2` | `rosReadXYZ`

Topics
“Work with Specialized ROS Messages”
“Improve Performance of ROS Using Message Structures”

rosWriteRGB

Write RGB color information to a ROS or ROS 2 PointCloud2 message structure

Syntax

```
msgOut = rosWriteRGB(msgIn,rgb)
msgOut = rosWriteRGB(msgIn,rgb,Name=Value)
```

Description

`msgOut = rosWriteRGB(msgIn,rgb)` writes the `[r g b]` values from *m*-by-3 matrix or *m*-by-*n*-by-3 matrix of 3-D point to a ROS or ROS 2 `sensor_msgs/PointCloud2` message `msgIn` and stores the data points in the message `msgOut`.

`msgOut = rosWriteRGB(msgIn,rgb,Name=Value)` specifies additional options using one or more name-value arguments.

Examples

Write RGB Color Information to a ROS or ROS 2 PointCloud2 Message

This example shows how to write RGB color information to a ROS or ROS 2 PointCloud2 message structure. To write RGB data points to a ROS or ROS 2 PointCloud2 message, you must write x,y and z data points first.

Create a random m-by-n-by-3 matrix with x, y and z coordinate points.

```
xyzPoints = single(10*rand(128,128,3));
```

Create a `sensor_msgs/PointCloud2` message in ROS network.

```
rosMsg = rosmessage("sensor_msgs/PointCloud2","DataFormat","struct")
```

```
rosMsg = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
         Header: [1x1 struct]
         Height: 0
          Width: 0
         Fields: [0x1 struct]
    IsBigendian: 0
      PointStep: 0
        RowStep: 0
           Data: [0x1 uint8]
        IsDense: 0
```

Write the x, y and z coordinate points to the ROS message and set `PointStep` of `sensor_msgs/PointCloud2` to 32 to store the RGB data points.

```
rosMsg = rosWriteXYZ(rosMsg,xyzPoints,"PointStep",32)
rosMsg = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
         Header: [1x1 struct]
         Height: 128
          Width: 128
         Fields: [3x1 struct]
    IsBigendian: 0
      PointStep: 32
        RowStep: 4096
           Data: [524288x1 uint8]
        IsDense: 1
```

Create a random m-by-n-by-3 matrix with RGB values.

```
rgb = single(10*rand(128,128,3));
```

Write the RGB color information to the ROS message and set the offset of the RGB field in `sensor_msgs/PointField` to 16. This means that RGB field begins to be stored from 16th byte for each `PointStep`.

```
rosMsg = rosWriteRGB(rosMsg,rgb,"FieldOffset",16)
```

```
rosMsg = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
         Header: [1x1 struct]
         Height: 128
          Width: 128
         Fields: [4x1 struct]
    IsBigendian: 0
      PointStep: 32
        RowStep: 4096
           Data: [524288x1 uint8]
        IsDense: 1
```

You can also create a `sensor_msgs/PointCloud2` message in ROS 2 network.

```
ros2Msg = ros2message("sensor_msgs/PointCloud2");
```

Write the x, y and z coordinate points to the ROS 2 message. Set `PointStep` to 16.

```
ros2Msg = rosWriteXYZ(ros2Msg,xyzPoints,"PointStep",16)
```

```
ros2Msg = struct with fields:
     MessageType: 'sensor_msgs/PointCloud2'
          header: [1x1 struct]
          height: 128
           width: 128
          fields: [3x1 struct]
    is_bigendian: 0
      point_step: 16
        row_step: 2048
            data: [262144x1 uint8]
        is_dense: 1
```

Write the RGB color information to the ROS 2 message and set `FieldOffset` to 8.

```
ros2Msg = rosWriteRGB(ros2Msg,rgb,"FieldOffset",8)
```

```
ros2Msg = struct with fields:
     MessageType: 'sensor_msgs/PointCloud2'
          header: [1x1 struct]
          height: 128
           width: 128
          fields: [4x1 struct]
    is_bigendian: 0
      point_step: 16
        row_step: 2048
            data: [262144x1 uint8]
        is_dense: 1
```

Input Arguments

msgIn — PointCloud2 message
"struct"

PointCloud2, specified as a structure for ROS or ROS 2 sensor_msgs/PointCloud2 message.

Data Types: struct

rgb — List of RGB values
m-by-3 matrix | *m*-by-*n*-by-3 matrix

List of RGB values, specified as a *m*-by-3 or *m*-by-*n*-by-3 matrix.

Data Types: single | double | int32 | uint8 | uint16 | uint32

Name-Value Pair Arguments

Example: PointStep=pointstep

PointStep — Provides optional parameter for setting up the point step value of the input sensor_msgs/PointCloud2 message
uint32(0) (default)

Point step is number of bytes or data entries for one point. If the PointStep field is not set in the input 'sensor_msgs/PointCloud2' message, you can use this parameter to manually set the PointStep information.

Example: msgOut = rosWriteIntensity(msgIn,rgb,PointStep=pointstep)

Data Types: uint32

FieldOffset — provides optional parameter for setting up the offset of a PointField of the input sensor_msgs/PointCloud2 message
uint32(0) (default)

Field Offset is number of bytes from the start of the point to the byte in which the field begins to be stored. If the offset field is not set for a PointField in the input 'sensor_msgs/PointCloud2' message, you can use this parameter to manually set the offset information.

Example: msgOut = rosWriteRGB(msgIn,rgb,FieldOffset=fieldoffset)

Data Types: uint32

Output Arguments

`msgOut` — `PointCloud2` message
`"struct"`

`PointCloud2`, specified as a structure for ROS or ROS 2 `sensor_msgs/PointCloud2` message.

Data Types: `struct`

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

`rosWriteXYZ` | `rosWriteIntensity`

rosReadScanAngles

Return scan angles from ROS or ROS 2 message structure

Syntax

```
angles = rosReadScanAngles(scan)
```

Description

`angles = rosReadScanAngles(scan)` calculates the scan angles, `angles`, corresponding to the range readings in the ROS or ROS 2 laser scan message structure, `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.

Input Arguments

`scan` — ROS or ROS 2 laser scan message
`'sensor_msgs/LaserScan'` message structure

ROS or ROS 2 laser scan message of type `'sensor_msgs/LaserScan'`, specified as a message structure.

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.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rosReadCartesian` | `rosReadXYZ` | `rosPlot`

Topics
"Work with Specialized ROS Messages"
"Improve Performance of ROS Using Message Structures"

rosReadXYZ

Extract XYZ coordinates from ROS or ROS 2 point cloud message structure

Syntax

```
xyz = rosReadXYZ(pcloud)
xyz = rosReadXYZ(pcloud,"PreserveStructureOnRead",true)
fielddata = rosReadXYZ(pcloud,"Datatype","double")
```

Description

`xyz = rosReadXYZ(pcloud)` extracts the `[x y z]` coordinates from all points in the ROS or ROS 2 `'sensor_msgs/PointCloud2'` message structure, `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.

`xyz = rosReadXYZ(pcloud,"PreserveStructureOnRead",true)` preserves the organizational structure of the point cloud returned in the `xyz` output. For more information, see “Preserving Point Cloud Structure” on page 2-216.

`fielddata = rosReadXYZ(pcloud,"Datatype","double")` reads the `[x y z]` data in double precision during code generation. If you use this syntax for MATLAB execution, the function always reads the data in the precision specified by the corresponding field in the input message structure, `pcloud`.

Input Arguments

pcloud — Point cloud
`PointCloud2` message structure

Point cloud, specified as a message structure for ROS or ROS 2 `'sensor_msgs/PointCloud2'` message.

Output Arguments

xyz — List of XYZ values from point cloud
n-by-3 matrix | *h*-by-*w*-by-3 matrix

List of XYZ values from point cloud, returned as a matrix. By default, this is a *n*-by-3 matrix.

If the `PreserveStructureOnRead` name-value pair argument is set to `true`, the points are returned as an *h*-by-*w*-by-3 matrix.

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 (`rosReadXYZ`, `rosReadRGB`, or `rosReadField`) 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.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

`rosReadRGB` | `rosReadCartesian` | `rosReadAllFieldNames`

Topics
“Work with Specialized ROS Messages”
“Improve Performance of ROS Using Message Structures”

rosWriteXYZ

Write data points in x, y and z coordinates to a ROS or ROS 2 PointCloud2 message structure

Syntax

```
msgOut = rosWriteXYZ(msgIn,xyz)
msgOut = rosWriteXYZ(msgIn,xyz,Name=Value)
```

Description

`msgOut = rosWriteXYZ(msgIn,xyz)` writes the `[x y z]` coordinates from *m*-by-3 matrix or *m*-by-*n*-by-3 matrix of 3-D point to a ROS or ROS 2 `sensor_msgs/PointCloud2` message `msgIn` and stores the data points in the message `msgOut`.

`msgOut = rosWriteXYZ(msgIn,xyz,Name=Value)` specifies additional options using one or more name-value arguments.

Examples

Write Data Points in x, y and z Coordinates to a ROS or ROS 2 PointCloud2 Message

This example shows how to write data points in x,y and z coordinates to a ROS or ROS 2 PointCloud2 message structure.

Create a random m-by-n-by-3 matrix with x, y and z coordinate points.

```
xyz = uint8(10*rand(128,128,3));
```

Create a `sensor_msgs/PointCloud2` message in ROS network.

```
rosMsg = rosmessage("sensor_msgs/PointCloud2","DataFormat","struct")
```

```
rosMsg = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
         Header: [1x1 struct]
         Height: 0
          Width: 0
         Fields: [0x1 struct]
    IsBigendian: 0
      PointStep: 0
        RowStep: 0
           Data: [0x1 uint8]
        IsDense: 0
```

Write the x, y and z coordinate points to the ROS message. As x, y and z are of data type uint8, the `PointStep` is 3 bytes.

```
rosMsg = rosWriteXYZ(rosMsg,xyz)
```

```
rosMsg = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
```

```
          Header: [1x1 struct]
          Height: 128
           Width: 128
          Fields: [3x1 struct]
     IsBigendian: 0
       PointStep: 3
         RowStep: 384
            Data: [49152x1 uint8]
         IsDense: 1
```

Now create a `sensor_msgs/PointCloud2` message in ROS network to set the `PointStep` to the desired value.

```
emptyRosMsg = rosmessage("sensor_msgs/PointCloud2","DataFormat","struct");
```

Set the `PointStep` in `sensor_msgs/PointCloud2` message to 32 to store remaining bytes with RGB or intensity data or both.

```
rosMsg = rosWriteXYZ(emptyRosMsg,xyz,"PointStep",32)
```

```
rosMsg = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
         Header: [1x1 struct]
         Height: 128
          Width: 128
         Fields: [3x1 struct]
    IsBigendian: 0
      PointStep: 32
        RowStep: 4096
           Data: [524288x1 uint8]
        IsDense: 1
```

You can also create a `sensor_msgs/PointCloud2` message in ROS 2 network.

```
ros2Msg = ros2message("sensor_msgs/PointCloud2")
```

```
ros2Msg = struct with fields:
     MessageType: 'sensor_msgs/PointCloud2'
          header: [1x1 struct]
          height: 0
           width: 0
          fields: [1x1 struct]
    is_bigendian: 0
      point_step: 0
        row_step: 0
            data: 0
        is_dense: 0
```

Write the x, y and z coordinate points to the ROS 2 message. Set `PointStep` to 32.

```
ros2Msg = rosWriteXYZ(ros2Msg,xyz,"PointStep",32)
```

```
ros2Msg = struct with fields:
     MessageType: 'sensor_msgs/PointCloud2'
          header: [1x1 struct]
          height: 128
          width: 128
         fields: [3x1 struct]
    is_bigendian: 0
      point_step: 32
        row_step: 4096
            data: [524288x1 uint8]
        is_dense: 1
```

Input Arguments

`msgIn` — `PointCloud2` message
`"struct"`

`PointCloud2`, specified as a structure for ROS or ROS 2 `sensor_msgs/PointCloud2` message.

Data Types: `struct`

`xyz` — List of x, y and z coordinate values
m-by-3 matrix | *m*-by-*n*-by-3 matrix

List of XYZ values, specified as a *m*-by-3 or *m*-by-*n*-by-3 matrix.

Data Types: `int8` | `uint8` | `int16` | `uint16` | `int32` | `uint32` | `single` | `double`

Name-Value Pair Arguments

Example: `PointStep=pointstep`

`PointStep` — Provides optional parameter for setting up the point step value of the input `sensor_msgs/PointCloud2` message
uint32(0) (default)

Point step is number of bytes or data entries for one point. If the `PointStep` field is not set in the input `sensor_msgs/PointCloud2` message, you can use this parameter to manually set the `PointStep` information.

Example: `msgOut = rosWriteXYZ(msgIn,xyz,PointStep=pointstep)`

Data Types: `uint32`

`FieldOffset` — provides optional parameter for setting up the offset of a `PointField` of the input `sensor_msgs/PointCloud2` message
uint32(0) (default)

Field Offset is number of bytes from the start of the point to the byte in which the field begins to be stored. If the `offset` field is not set for a `PointField` in the input `sensor_msgs/PointCloud2` message, you can use this parameter to manually set the `offset` information.

Example: `msgOut = rosWriteXYZ(msgIn,xyz,FieldOffset=fieldoffset)`

Data Types: `uint32`

Output Arguments

`msgOut` — `PointCloud2` message
`"struct"`

`PointCloud2`, specified as a structure for ROS or ROS 2 `sensor_msgs/PointCloud2` message.

Data Types: `struct`

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

`rosWriteRGB` | `rosWriteIntensity`

rosWriteIntensity

Write intensity data points to a ROS or ROS 2 PointCloud2 message structure

Syntax

```
msgOut = rosWriteIntensity(msgIn,intensity)
msgOut = rosWriteIntensity(msgIn,intensity,Name=Value)
```

Description

`msgOut = rosWriteIntensity(msgIn,intensity)` writes the intensity values from *m*-by-1 matrix or *m*-by-*n* matrix of 3-D point to a ROS or ROS 2 `sensor_msgs/PointCloud2` message `msgIn` and stores the data points in the message `msgOut`.

`msgOut = rosWriteIntensity(msgIn,intensity,Name=Value)` specifies additional options using one or more name-value arguments.

Examples

Write Intensity Data Points to a ROS or ROS 2 PointCloud2 Message

This example shows how to write intensity data points to a ROS or ROS 2 PointCloud2 message structure. To write intensity data points to a ROS or ROS 2 PointCloud2 message, you must write x,y and z data points first.

Create a random m-by-n-by-3 matrix with x, y and z coordinate points.

```
xyzPoints = single(10*rand(480,640,3));
```

Create a `sensor_msgs/PointCloud2` message in ROS network.

```
rosMsg = rosmessage("sensor_msgs/PointCloud2","DataFormat","struct")
```

```
rosMsg = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
         Header: [1x1 struct]
         Height: 0
          Width: 0
         Fields: [0x1 struct]
    IsBigendian: 0
      PointStep: 0
        RowStep: 0
           Data: [0x1 uint8]
        IsDense: 0
```

Write the x, y and z coordinate points to the ROS message and set `PointStep` of `sensor_msgs/PointCloud2` to 16 to store the intensity data points.

```
rosMsg = rosWriteXYZ(rosMsg,xyzPoints,"PointStep",16)
rosMsg = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
         Header: [1x1 struct]
         Height: 480
          Width: 640
         Fields: [3x1 struct]
    IsBigendian: 0
      PointStep: 16
        RowStep: 10240
           Data: [4915200x1 uint8]
        IsDense: 1
```

Create a random m-by-n matrix with intensity values.

```
intensity = uint8(10*rand(480,640));
```

Write the intensity information to the ROS message and set the offset of the intensity field in `sensor_msgs/PointField` to 8. This means that intensity field begins to be stored from 16th byte for each `PointStep`.

```
rosMsg = rosWriteIntensity(rosMsg,intensity,"FieldOffset",8)
```

```
rosMsg = struct with fields:
    MessageType: 'sensor_msgs/PointCloud2'
         Header: [1x1 struct]
         Height: 480
          Width: 640
         Fields: [4x1 struct]
    IsBigendian: 0
      PointStep: 16
        RowStep: 10240
           Data: [4915200x1 uint8]
        IsDense: 1
```

You can also create a `sensor_msgs/PointCloud2` message in ROS 2 network.

```
ros2Msg = ros2message("sensor_msgs/PointCloud2");
```

Write the x, y and z coordinate points to the ROS 2 message. Set `PointStep` to 16.

```
ros2Msg = rosWriteXYZ(ros2Msg,xyzPoints,"PointStep",16)
```

```
ros2Msg = struct with fields:
     MessageType: 'sensor_msgs/PointCloud2'
          header: [1x1 struct]
          height: 480
           width: 640
          fields: [3x1 struct]
    is_bigendian: 0
      point_step: 16
        row_step: 10240
            data: [4915200x1 uint8]
        is_dense: 1
```

Write the intensity information to the ROS 2 message and set `FieldOffset` to 8.

```
ros2Msg = rosWriteIntensity(ros2Msg,intensity,"FieldOffset",8)
```

```
ros2Msg = struct with fields:
     MessageType: 'sensor_msgs/PointCloud2'
          header: [1x1 struct]
          height: 480
           width: 640
          fields: [4x1 struct]
    is_bigendian: 0
      point_step: 16
        row_step: 10240
            data: [4915200x1 uint8]
        is_dense: 1
```

Input Arguments

msgIn — PointCloud2 message
"struct"

PointCloud2, specified as a structure for ROS or ROS 2 sensor_msgs/PointCloud2 message.

Data Types: struct

intensity — List of intensity values
m-by-1 vector | *m*-by-*n* matrix

List of intensity values, specified as a *m*-by-1 vector or *m*-by-*n* matrix.

Data Types: single | double | int32 | uint8 | uint16 | uint32

Name-Value Pair Arguments

Example: PointStep=pointstep

PointStep — Provides optional parameter for setting up the point step value of the input sensor_msgs/PointCloud2 message
uint32(0) (default)

Point step is number of bytes or data entries for one point. If the PointStep field is not set in the input sensor_msgs/PointCloud2 message, you can use this parameter to manually set the PointStep information.

Example: msgOut = rosWriteIntensity(msgIn,intensity,PointStep=pointstep)

Data Types: uint32

FieldOffset — provides optional parameter for setting up the offset of a PointField of the input sensor_msgs/PointCloud2 message
uint32(0) (default)

Field Offset is number of bytes from the start of the point to the byte in which the field begins to be stored. If the offset field is not set for a PointField in the input sensor_msgs/PointCloud2 message, you can use this parameter to manually set the offset information.

Example: msgOut = rosWriteIntensity(msgIn,intensity,FieldOffset=fieldoffset)

Data Types: uint32

Output Arguments

`msgOut` — `PointCloud2` message
`"struct"`

`PointCloud2`, specified as a structure for ROS or ROS 2 `'sensor_msgs/PointCloud2'` message.

Data Types: `struct`

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

`rosWriteRGB` | `rosWriteXYZ`

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.

Version History

Introduced in R2019b

See Also

`rosinit` | `rosparam`

rosShowDetails

Display all ROS message contents

Syntax

`details = rosShowDetails(msg)`

Description

`details = rosShowDetails(msg)` gets all data contents of the ROS message structure `msg`. The details are stored in `details` or if not specified, are displayed on the command line.

Input Arguments

`msg` — ROS message
structure

ROS message, specified as a ROS message structure.

Output Arguments

`details` — Details of ROS message
character vector

Details of a ROS message, returned as a character vector.

Version History

Introduced in R2021a

See Also

`rosmessage`

rosshutdown

Shut down ROS system

Syntax

```
rosshutdown
```

Description

`rosshutdown` shuts down the global node and, if it is running, the ROS master. When you finish working with the ROS network, use `rosshutdown` 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 `rosshutdown`, any ROS entities (objects) that depend on the global node like subscribers created with `rossubscriber`, are deleted and become unstable.

Prior to calling `rosshutdown`, call `clear` on these objects for a clean removal of ROS entities.

Examples

Start ROS Core and Global Node

```
rosinit
```

```
Launching ROS Core...
..Done in 2.7022 seconds.
Initializing ROS master on http://172.30.131.134:54795.
Initializing global node /matlab_global_node_90971 with NodeURI http://bat6234win64:58551/ and Ma
```

When you are finished, shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_90971 with NodeURI http://bat6234win64:58551/ and M
Shutting down ROS master on http://172.30.131.134:54795.
```

Version History

Introduced in R2019b

See Also

`rosinit`

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 = rosmessage(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.

Version History

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
```

Version History

Introduced in R2019b

See Also

`rosmessage` | `rostopic`

Topics

“Built-In Message Support”

"Work with Basic ROS Messages"

rosWriteBinaryOccupancyGrid

Write values from binary occupancy grid to ROS or ROS 2 message structure

Syntax

```
msgOut = rosWriteBinaryOccupancyGrid(msg,map)
```

Description

`msgOut = rosWriteBinaryOccupancyGrid(msg,map)` writes occupancy values from the occupancy grid `map` and other information from the ROS or ROS 2 message structure `msg` to an output message `msgOut`.

Input Arguments

`msg` — ROS or ROS 2 occupancy grid message
`'nav_msgs/OccupancyGrid'` message structure

ROS or ROS 2 `'nav_msgs/OccupancyGrid'` message, specified as a message structure.

`map` — Binary occupancy grid
`binaryOccupancyMap` object handle

Binary occupancy grid, specified as a `binaryOccupancyMap` object handle. The object stores a grid of binary values, where `1` indicates an occupied location and `0` indications an unoccupied location.

Outputs

`msgOut` — ROS or ROS 2 occupancy grid message
`'nav_msgs/OccupancyGrid'` message structure

ROS or ROS 2 `'nav_msgs/OccupancyGrid'` message, specified as a message structure.

You can use the same variable for the input and output argument to directly assign to the existing message.

```
map = occupancyMap(rand(10));
msg = rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");
msg = rosWriteBinaryOccupancyGrid(msg,map)
```

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

Functions
`rosReadBinaryOccupancyGrid` | `rosReadOccupancyMap3D` | `rosReadOccupancyGrid` | `rosWriteOccupancyGrid`

Topics
"Improve Performance of ROS Using Message Structures"

rosWriteCameraInfo

Write monocular or stereo camera parameters to ROS or ROS 2 message structure

Syntax

```
msgOut = rosWriteCameraInfo(msg,cameraParams)
[msgOut1,msgOut2] = rosWriteCameraInfo(msg,stereoParams)
```

Description

`msgOut = rosWriteCameraInfo(msg,cameraParams)` writes data from the monocular camera parameters structure, `cameraParams`, to a `sensor_msgs/CameraInfo` message structure, `msg`, and returns the output message, `msgOut`.

Use `rosWriteCameraInfo` to write the camera parameters obtained after the calibration process. For more information on performing camera calibration using Computer Vision Toolbox™, see "Camera Calibration" (Computer Vision Toolbox).

`[msgOut1,msgOut2] = rosWriteCameraInfo(msg,stereoParams)` writes data from the stereo camera parameters structure, `stereoParams`, to two `sensor_msgs/CameraInfo` message structures, `msgOut1` and `msgOut2`.

Examples

Write Camera Parameters to ROS Message

Create a set of calibration images.

```
images = imageDatastore(fullfile(toolboxdir("vision"),"visiondata", ...
      "calibration","mono"));
imageFileNames = images.Files;
```

Detect calibration pattern.

```
[imagePoints,boardSize] = detectCheckerboardPoints(imageFileNames);
```

Generate world coordinates of the corners of the squares.

```
squareSize = 29; % millimeters
worldPoints = generateCheckerboardPoints(boardSize,squareSize);
```

Calibrate the camera.

```
I = readimage(images,1);
imageSize = [size(I,1),size(I,2)];
params = estimateCameraParameters(imagePoints,worldPoints, ...
                                  ImageSize=imageSize);
```

Create a ROS `sensor_msgs/CameraInfo` message structure.

```
msg = rosmessage("sensor_msgs/CameraInfo","DataFormat","struct");
```

Write the camera parameters obtained after calibration to the ROS message. Use the `toStruct` function to convert the `cameraParameters` object to a structure.

```
msg = rosWriteCameraInfo(msg,toStruct(params))
```

```
msg = struct with fields:
        MessageType: 'sensor_msgs/CameraInfo'
             Header: [1x1 struct]
             Height: 712
              Width: 1072
    DistortionModel: 'plumb_bob'
                  D: [5x1 double]
                  K: [9x1 double]
                  R: [9x1 double]
                  P: [12x1 double]
           BinningX: 0
           BinningY: 0
                Roi: [1x1 struct]
```

Write Stereo Camera Parameters to ROS Messages

Specify images containing a checkerboard for calibration.

```
imageDir = fullfile(toolboxdir("vision"),"visiondata","calibration","stereo");
leftImages = imageDatastore(fullfile(imageDir,"left"));
rightImages = imageDatastore(fullfile(imageDir,"right"));
```

Detect the checkerboards.

```
[imagePoints,boardSize] = detectCheckerboardPoints(leftImages.Files,rightImages.Files);
```

Specify world coordinates of checkerboard keypoints.

```
squareSizeInMillimeters = 108;
worldPoints = generateCheckerboardPoints(boardSize,squareSizeInMillimeters);
```

Read in the images.

```
I1 = readimage(leftImages,1);
I2 = readimage(rightImages,1);
imageSize = [size(I1, 1),size(I1, 2)];
```

Calibrate the stereo camera system.

```
stereoParams = estimateCameraParameters(imagePoints,worldPoints,ImageSize=imageSize);
```

Rectify the images using `full` output view.

```
[J1_full,J2_full] = rectifyStereoImages(I1,I2,stereoParams,OutputView="full");
```

Create a ROS `sensor_msgs/CameraInfo` message structure.

```
msg = rosmessage("sensor_msgs/CameraInfo","DataFormat","struct");
```

Write the stereo parameters obtained after calibration to two ROS messages. Use the `toStruct` function to convert the `stereoParameters` object to a structure.

```
[msg1,msg2] = rosWriteCameraInfo(msg,toStruct(stereoParams))

msg1 = struct with fields:
        MessageType: 'sensor_msgs/CameraInfo'
             Header: [1x1 struct]
             Height: 960
              Width: 1280
    DistortionModel: 'plumb_bob'
                  D: [5x1 double]
                  K: [9x1 double]
                  R: [9x1 double]
                  P: [12x1 double]
           BinningX: 0
           BinningY: 0
                Roi: [1x1 struct]


msg2 = struct with fields:
        MessageType: 'sensor_msgs/CameraInfo'
             Header: [1x1 struct]
             Height: 960
              Width: 1280
    DistortionModel: 'plumb_bob'
                  D: [5x1 double]
                  K: [9x1 double]
                  R: [9x1 double]
                  P: [12x1 double]
           BinningX: 0
           BinningY: 0
                Roi: [1x1 struct]
```

Input Arguments

`msg` — ROS or ROS 2 camera info message
`sensor_msgs/CameraInfo` message structure

ROS or ROS 2 `sensor_msgs/CameraInfo` message, specified as a message structure.

Data Types: `struct`

`cameraParams` — Monocular camera parameters structure
`CameraParameters` structure

Monocular camera parameters structure, specified as a `cameraParameters` structure. To obtain the structure from a `cameraParameters` object, use the `toStruct` function.

Data Types: `struct`

`stereoParams` — Stereo camera parameters structure
`StereoParameters` structure

Stereo camera parameters structure, specified as a `stereoParameters` structure. To obtain the structure from a `stereoParameters` object, use the `toStruct` function.

Data Types: `struct`

Output Arguments

`msgOut` — ROS or ROS 2 camera info message for a monocular camera
`sensor_msgs/CameraInfo` message structure

ROS or ROS 2 camera info message for a monocular camera, returned as a `sensor_msgs/CameraInfo` message structure.

`msgOut1` — ROS or ROS 2 camera info message for camera 1 in a stereo pair
`sensor_msgs/CameraInfo` message structure

ROS or ROS 2 camera info message for camera 1 in a stereo pair, returned as a `sensor_msgs/CameraInfo` message structure.

`msgOut2` — ROS or ROS 2 camera info message for camera 2 in a stereo pair
`sensor_msgs/CameraInfo` message structure

ROS or ROS 2 camera info message for camera 2 in a stereo pair, returned as a `sensor_msgs/CameraInfo` message structure.

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rosReadImage` | `rosWriteImage`

Topics
“Work with Specialized ROS Messages”
“Improve Performance of ROS Using Message Structures”

rosWriteImage

Write MATLAB image to ROS or ROS 2 image message

Syntax

```
msgOut = rosWriteImage(msg,img)
msgOut = rosWriteImage(msg,img,alpha)
msgOut = rosWriteImage( ___ ,"Encoding",encodingParam)
```

Description

`msgOut = rosWriteImage(msg,img)` converts the MATLAB image, `img`, to a message structure and stores the ROS or ROS 2 compatible image data in the message structure, `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.

`msgOut = rosWriteImage(msg,img,alpha)` converts the MATLAB image, `img`, to a message structure. 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.

`msgOut = rosWriteImage( ___ ,"Encoding",encodingParam)` specifies the encoding of the image message as a name-value argument using any of the previous input arguments. If the `Encoding` field of the message is not set, use this syntax to set the field.

Input Arguments

`msg` — ROS or ROS 2 image message
`'sensor_msgs/Image'` message structure

ROS or ROS 2 `'sensor_msgs/Image'` message, specified as a message structure.

`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`.

`encodingParam` — Encoding of image message
`"rgb8"` | `"rgba8"` | `"rgb16"` | string scalar

Encoding of image message, specified as a string scalar. Using this input argument overwrites the `Encoding` field of the input `msg`. For more information, see “ROS Image Encoding” on page 2-245.

Outputs

`msgOut` — ROS or ROS 2 image message
`'sensor_msgs/Image'` structure

ROS or ROS 2 `'sensor_msgs/Image'` image message, specified as a message structure. `'sensor_msgs/CompressedImage'` messages are not supported.

You can use the same variable for the input and output argument to directly assign to the existing message.

```
img = uint8(10*rand(128,128,3));
msg = rosmessage("sensor_msgs/Image","DataFormat","struct");
msg = rosWriteImage(msg,img,"Encoding","rgb8");
```

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 `rosReadImage` are also supported in this function. For more information on supported encoding types and their representations in MATLAB, see `rosReadImage`.

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.

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Specify the `"Encoding",encodParam` name-value argument when generating code.
- Usage in MATLAB Function block is not supported.

See Also

`rosReadImage` | `rosReadRGB`

Topics
"Work with Specialized ROS Messages"
"Improve Performance of ROS Using Message Structures"

rosWriteOccupancyGrid

Write values from occupancy grid to ROS or ROS 2 message structure

Syntax

`msgOut = rosWriteOccupancyGrid(msg,map)`

Description

`msgOut = rosWriteOccupancyGrid(msg,map)` writes occupancy values from the occupancy grid `map` and other information from the ROS or ROS 2 message structure, `msg`, to an output message `msgOut`.

Input Arguments

`msg` — ROS or ROS 2 occupancy grid message
`'nav_msgs/OccupancyGrid'` message structure

ROS or ROS 2 `'nav_msgs/OccupancyGrid'` message, specified as a message structure.

`map` — Occupancy map
`occupancyMap` object handle

Occupancy map, specified as an `occupancyMap` object handle.

Outputs

`msgOut` — ROS or ROS 2 occupancy grid message
`'nav_msgs/OccupancyGrid'` message structure

ROS or ROS 2 `'nav_msgs/OccupancyGrid'` message, specified as a message structure.

You can use the same variable for the input and output argument to directly assign to the existing message.

```
map = occupancyMap(rand(10));
msg = rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");
msg = rosWriteOccupancyGrid(msg,map)
```

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Usage in MATLAB Function block is not supported.

See Also

Functions
`rosReadBinaryOccupancyGrid` | `rosReadOccupancyGrid` | `rosReadOccupancyMap3D` | `rosWriteBinaryOccupancyGrid`

Topics
"Work with Specialized ROS Messages"
"Improve Performance of ROS Using Message Structures"

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 = '192.168.203.131';
d = rosdevice(ipaddress,'user','password')
```

```
d = 
  rosdevice with properties:

      DeviceAddress: '192.168.203.131'
           Username: 'user'
          ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws'
     AvailableNodes: {'voxel_grid_filter_sl'}
```

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)
pause(2)
running = isCoreRunning(d)

running = logical
   0
```

Input Arguments

`device` — ROS device
`rosdevice` object

ROS device, specified as a `rosdevice` object.

Version History

Introduced in R2019b

See Also

`rosdevice` | `stopCore` | `isCoreRunning`

Topics
"Generate a Standalone ROS Node from Simulink"

runNode

Start ROS or ROS 2 node

Syntax

```
runNode(device,modelName)
runNode(device,modelName,masterURI)
runNode(device,modelName,masterURI,nodeHost)
```

Description

`runNode(device,modelName)` starts the ROS or ROS 2 node associated with the deployed Simulink model named `modelName`. The node must be deployed in the workspace specified by the `CatkinWorkspace` property of the input `rosdevice` object or the `ROS2Workspace` property of the input `ros2device` 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. This syntax is applicable only when `device` is a `rosdevice` object.

`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`. This syntax is applicable only when `device` is a `rosdevice` object.

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 = 1×2 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: {0×1 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/
```

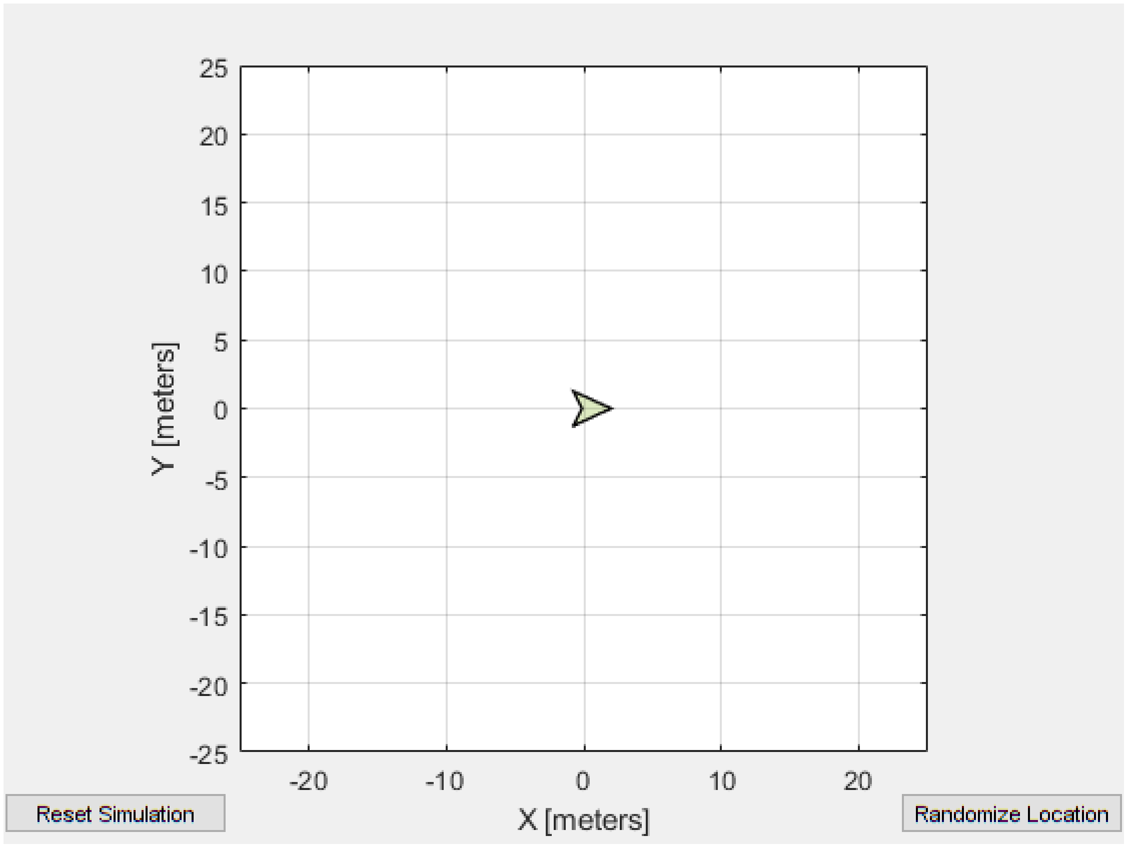
Check the available ROS nodes on the connected ROS device. The nodes listed in this example were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink” example. Two separate nodes are generated, `'robotcontroller'` and `'robotcontroller2'`, which have the linear velocity set to 1 and 2 in the model respectively.

```
d.AvailableNodes
```

```
ans = 1×2 cell
    {'robotcontroller'}    {'robotcontroller2'}
```

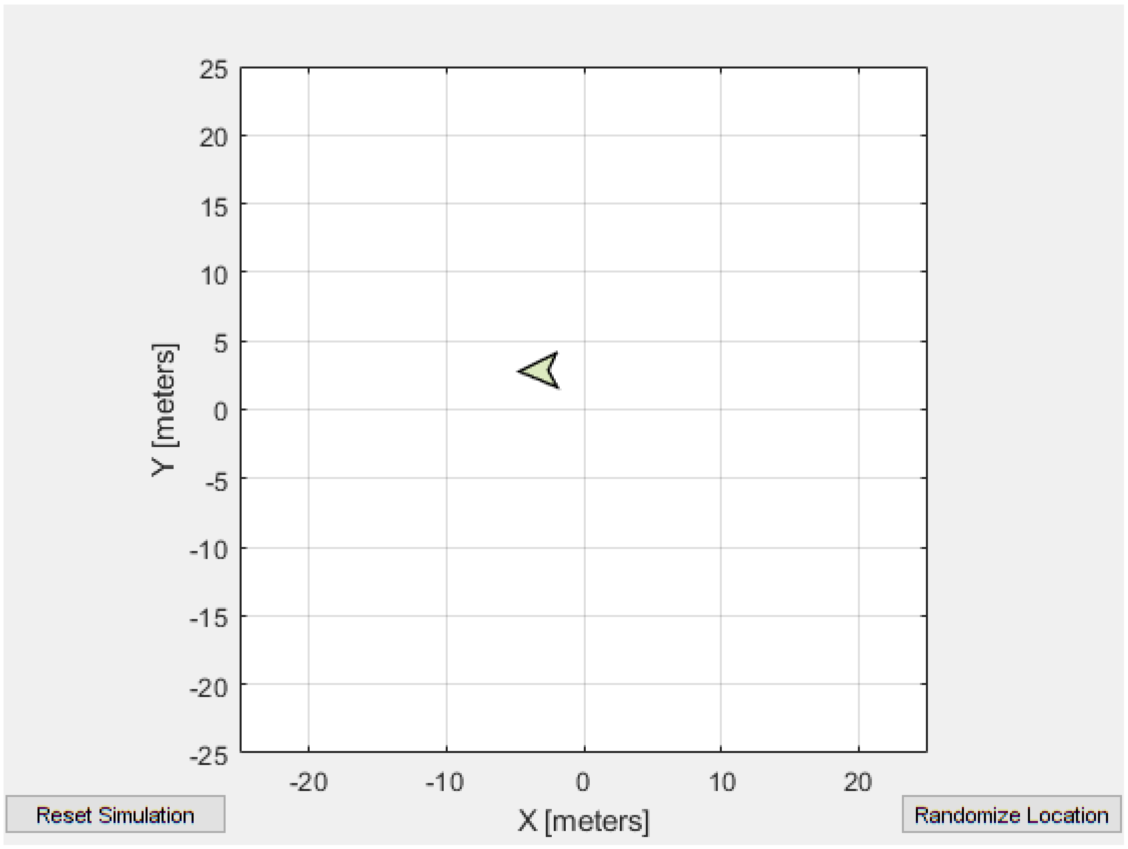
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



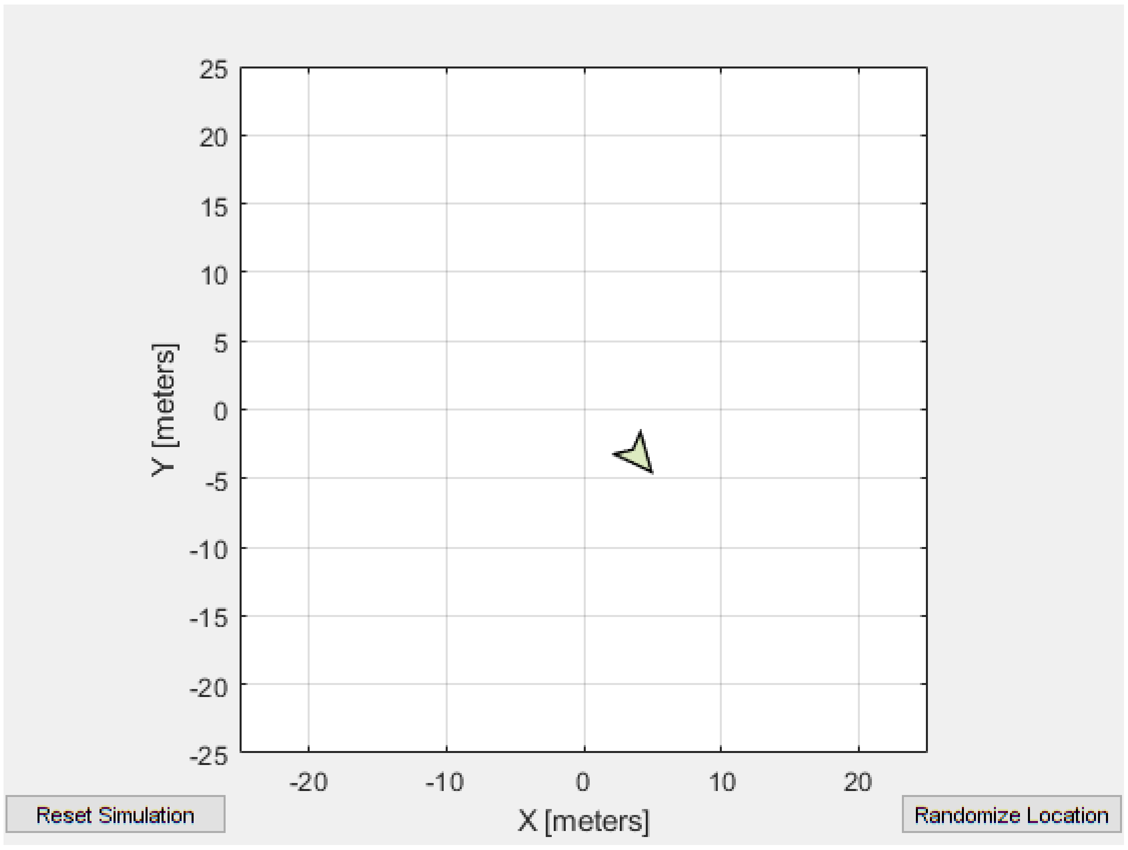
Run a ROS node, specifying the node name. The `'robotcontroller'` node commands the robot to a specific location (`[-10 10]`). Wait to see the robot drive.

```
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.

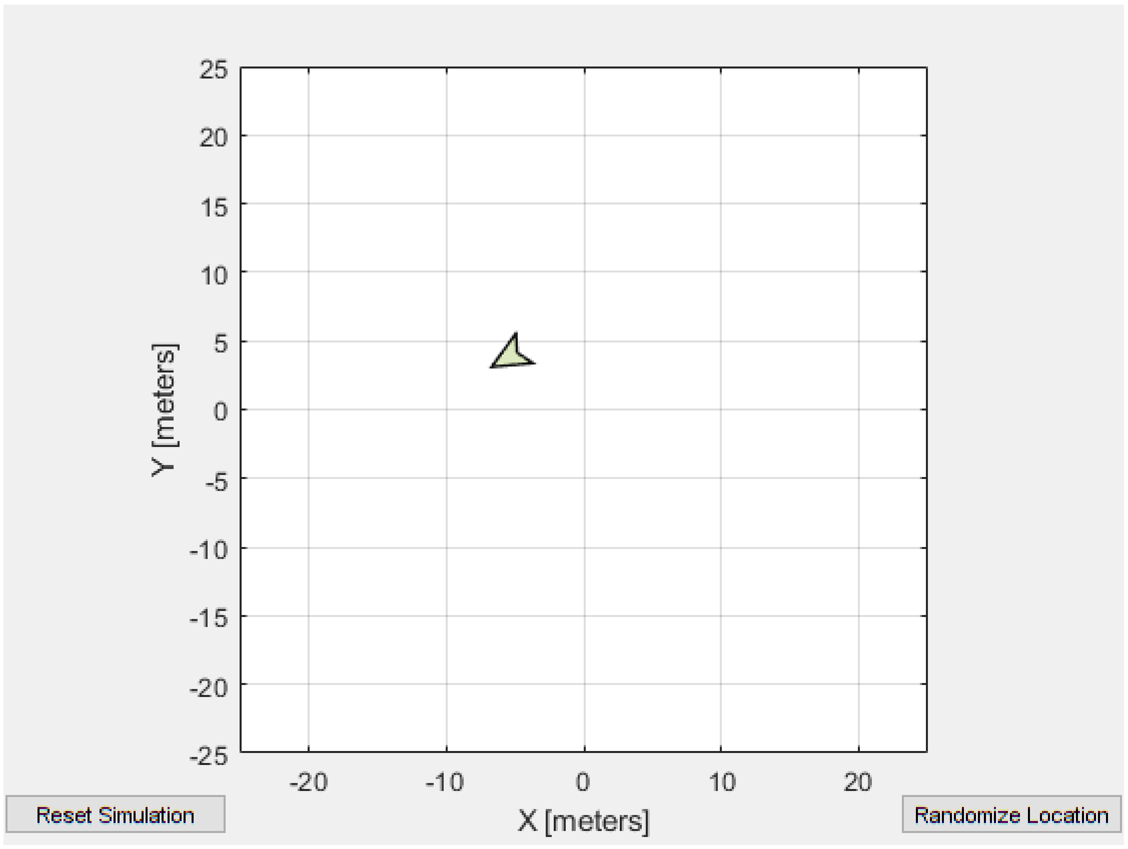
```
resetSimulation(sim.Simulator)
pause(5)
```



```
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')
rosshutdown
```

```
Shutting down global node /matlab_global_node_66434 with NodeURI http://192.168.203.1:59395/
```

```
stopCore(d)
```

Input Arguments

device — ROS or ROS 2 device
`rosdevice` object | `ros2device` object

ROS or ROS 2 device, specified as a `rosdevice` or `ros2device` object, respectively.

`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.

Version History

Introduced in R2019b

See Also

`rosdevice` | `ros2device` | `stopNode` | `isNodeRunning`

Topics
"Connect to a ROS Network"
"Generate a Standalone ROS Node from Simulink"
"Generate a Standalone ROS 2 Node from Simulink"

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.

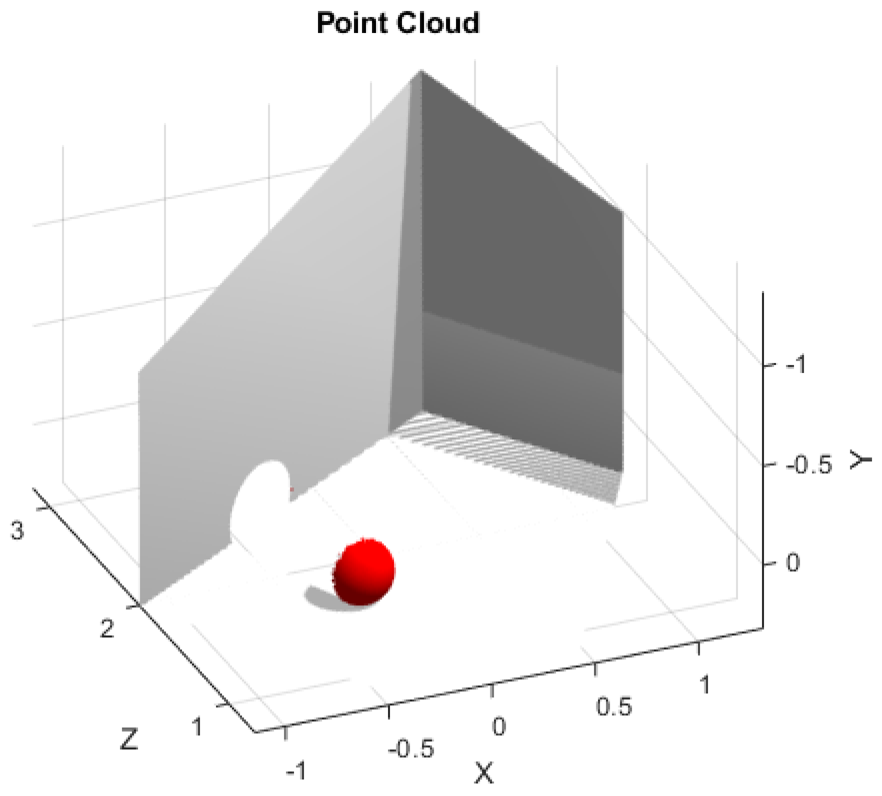
```
rosinit('192.168.17.129')
```

```
Initializing global node /matlab_global_node_65972 with NodeURI http://192.168.17.1:51971/
```

```
sub = rossubscriber('/camera/depth/points');
pause(1)
```

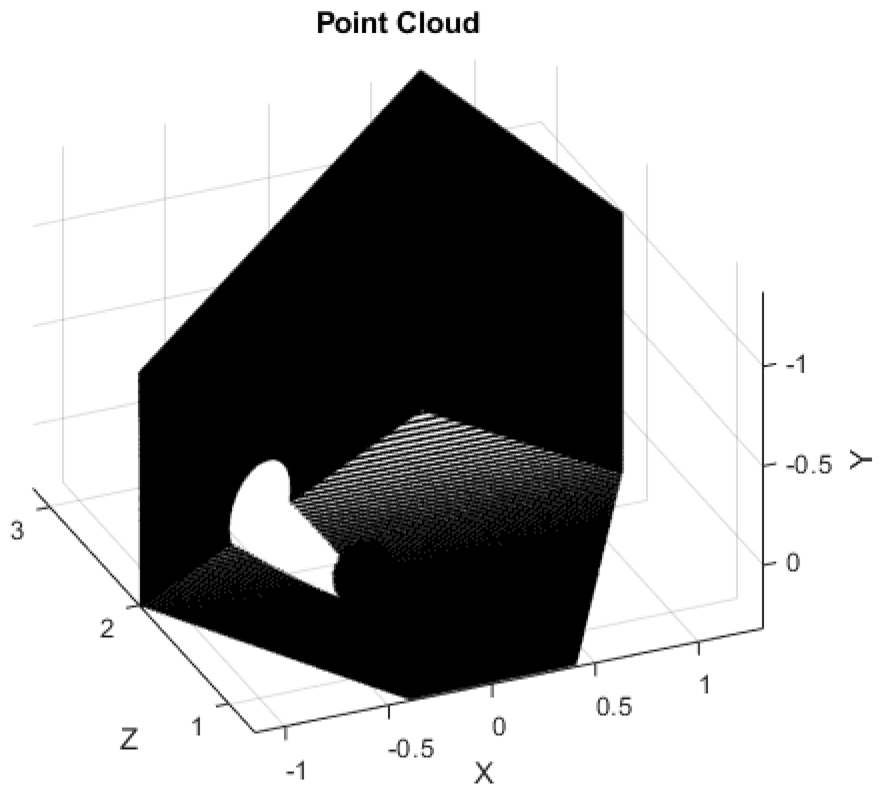
Get the latest point cloud message. Plot the point cloud.

```
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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

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 string scalar or character vector 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. Therefore, 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.

Version History

Introduced in R2019b

See Also

`readXYZ` | `readRGB`

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 = 1×3 cell array
    {'/gazebo/gravity_x'}    {'/gazebo/gravity_y'}    {'/gazebo/gravity_z'}


pvalues = 3×1 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 iso 8601 data from ROS are not supported.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

This function supports C/C++ code generation with the limitations:

- Retrieving values is not supported.

See Also

`get` | `rosparam`

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`.

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.

Version History
Introduced in R2019b

See Also
`rosduration` | `rostime`

select

Select subset of messages in rosbag

Syntax

```
bagsel = select(bag)
bagsel = select(bag,Name,Value)
```

Description

`bagsel = select(bag)` returns a `BagSelection` object, `bagsel`, that contains all of the messages in the `BagSelection` object, `bag`.

This function creates a copy of the `BagSelection` object or returns a new `BagSelection` object that contains the specified message selection.

`bagsel = select(bag,Name,Value)` provides additional options specified by one or more name-value pair arguments. For example, `"Topic","/odom"` selects a subset of the messages, filtered by the topic `/odom`.

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 in rosbag
`BagSelection` object

Messages in a rosbag, specified as a `BagSelection` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `"Topic","/odom"` selects a subset of the messages, filtered by the topic `/odom`.

`MessageType` — ROS message type

string scalar | character vector | cell array of string scalars | cell array of character vectors

ROS message type, specified as a string scalar, character vector, cell array of string scalars, or cell array of character vectors. Multiple message types can be specified with a cell array.

Example: `select(bag,"MessageType",{"nav_msgs/Odometry","rosgraph_msgs/Clock"})`

Data Types: `char` | `string` | `cell`

`Time` — Start and end times of rosbag selection

n-by-2 vector

Start and end times of the rosbag selection, specified as an *n*-by-2 vector.

Example: `select(bag,"Time",[bag.StartTime,bag.StartTime+1])`

Data Types: `double`

`Topic` — ROS topic name

string scalar | character vector | cell array of string scalars | cell array of character vectors

ROS topic name, specified as a string scalar, character vector, cell array of string scalars, or cell array of character vectors. Multiple topic names can be specified with a cell array.

Example: `select(bag,"Topic",{"/odom","/clock"})`

Data Types: `char` | `string` | `cell`

Output Arguments

`bagsel` — Copy or subset of rosbag messages

`BagSelection` object

Copy or subset of rosbag messages, returned as a `BagSelection` object.

Version History

Introduced in R2019b

See Also

`readMessages` | `rosbag` | `timeseries`

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
```

```
Launching ROS Core...
.Done in 1.6447 seconds.
Initializing ROS master on http://172.30.131.134:52608.
Initializing global node /matlab_global_node_30321 with NodeURI http://bat6234win64:64847/ and Ma
```

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'
    LatestMessage: [1x1 Point]
      MessageType: 'geometry_msgs/Point'
       BufferSize: 1
    NewMessageFcn: []
       DataFormat: 'object'
```

```
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_30321 with NodeURI http://bat6234win64:64847/ and N
Shutting down ROS master on http://172.30.131.134:52608.
```

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 | structure

ROS message, specified as a `Message` object handle or structure. You can create object using `rosmessage`.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see "ROS Message Structures" on page 2-270.

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for `struct` messages.

See Also

`receive` | `rosmessage` | `rostopic` | `rossubscriber` | `rospublisher`

Topics
“Exchange Data with ROS Publishers and Subscribers”

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 ROS Action Goal Message

This example shows how to create goal messages and send them to an active ROS action server on a ROS network. You must create a ROS action client to connect to this server.

Start ROS-Enabled Virtual Machine

- Download and install the virtual machine (VM) using the instructions in “Get Started with Gazebo and Simulated TurtleBot” example.
- Start the Ubuntu® VM desktop.
- On the Ubuntu desktop, click **ROS Noetic Terminal**.

Launch ROS Action Server in ROS-Enabled VM

Source the appropriate ROS environment setup script in the ROS noetic terminal before running any ROS commands.

```
source ~/Documents/mw_catkin_ws/devel/setup.bash
```

Run the action server in the ROS noetic terminal.

```
roslaunch turtlebot_actions server_turtlebot_move.launch
```

Connect to ROS from MATLAB

Connect to the ROS node using `rosinit` with the IP address of the ROS-enabled VM.

```
rosIP = "192.168.198.128"; % IP address
rosinit(rosIP,11311) % Initialize ROS connection
```

```
Initializing global node /matlab_global_node_19677 with NodeURI http://192.168.198.1:61572/ and N
```

Create ROS Action Client

Create a ROS action client using `rosactionclient` and get a goal message. The action client object `actClient` 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] = rosactionclient("/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
```

Create ROS Message Using ROS Action Client

Create a message using `rosmessage` function 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
```

Send Goal Message to Action Server

Modify the goal message parameters and send the goal to the action server.

```
goalMsg.ForwardDistance = 2; % in meters
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:

```
rosrun 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('192.168.203.133',11311)
```

```
Initializing global node /matlab_global_node_18287 with NodeURI http://192.168.203.1:55284/
```

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

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

```
goalMsg.Order = int32(4);
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg = struct with fields:
    MessageType: 'actionlib_tutorials/FibonacciResult'
       Sequence: [0 1 1 2 3]


resultState = 
'succeeded'
```

```
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  actionlib_tutorials/FibonacciResult
       Sequence    :  [0, 1, 1, 2, 3]'
```

Send a new goal message without waiting.

```
goalMsg.Order = int32(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_18287 with NodeURI http://192.168.203.1:55284/
```

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 | structure

ROS action goal message, specified as a `Message` object handle or structure. Update this message with your goal details and send it to the ROS action client using `sendGoal` or `sendGoalAndWait`.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see "ROS Message Structures" on page 2-275.

Version History

Introduced in R2019b

R2021a: ROS Message Structures
Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see "Improve Performance of ROS Using Message Structures".

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.

- Usage in MATLAB Function block is not supported.

See Also

`sendGoalAndWait` | `cancelGoal` | `rosactionclient` | `rosaction`

Topics

"ROS Actions Overview"
"Move a Turtlebot Robot Using ROS Actions"

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.

Note In a future release, this syntax will not display an error if the server does not return the result in the timeout period. Instead, it will return the `state` as `'timeout'`, which can be reacted to in the calling code.

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:

```
rosrun 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('192.168.203.133',11311)
```

```
Initializing global node /matlab_global_node_18287 with NodeURI http://192.168.203.1:55284/
```

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

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

```
goalMsg.Order = int32(4);
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg = struct with fields:
    MessageType: 'actionlib_tutorials/FibonacciResult'
       Sequence: [0 1 1 2 3]


resultState = 
'succeeded'
```

```
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  actionlib_tutorials/FibonacciResult
       Sequence    :  [0, 1, 1, 2, 3]'
```

Send a new goal message without waiting.

```
goalMsg.Order = int32(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_18287 with NodeURI http://192.168.203.1:55284/
```

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 | structure

ROS action goal message, specified as a `Message` object handle or structure. Update this message with your goal details and send it to the ROS action client using `sendGoal` or `sendGoalAndWait`.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see “ROS Message Structures” on page 2-280.

`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 | structure

Result message, returned as a ROS `Message` object or structure. The result message contains the result data sent by the action server. This data depends on the action type.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see “ROS Message Structures” on page 2-280.

`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.

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`sendGoal` | `cancelGoal` | `rosactionclient` | `rosaction`

Topics

“ROS Actions Overview”
“Move a Turtlebot Robot Using ROS Actions”

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.

```
rosinit;
```

```
Launching ROS Core...
....Done in 4.1192 seconds.
Initializing ROS master on http://192.168.125.1:56090.
Initializing global node /matlab_global_node_16894 with NodeURI http://HYD-KVENNAPU:63122/
```

```
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.X = 0.5;
tform.Transform.Rotation.Y = 0.5;
tform.Transform.Rotation.Z = 0.5;
tform.Transform.Rotation.W = 0.5;
```

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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_16894 with NodeURI http://HYD-KVENNAPU:63122/
Shutting down ROS master on http://192.168.125.1:56090.
```

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 `rostf` 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).

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`transform` | `getTransform`

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
```

```
Launching ROS Core...
..Done in 2.1888 seconds.
Initializing ROS master on http://172.30.131.134:49954.
Initializing global node /matlab_global_node_42357 with NodeURI http://bat6234win64:65482/ and Ma
```

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_42357 with NodeURI http://bat6234win64:65482/ and M
Shutting down ROS master on http://172.30.131.134:49954.
```

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
```

```
Launching ROS Core...
..Done in 2.7008 seconds.
Initializing ROS master on http://172.30.131.134:54064.
Initializing global node /matlab_global_node_04167 with NodeURI http://bat6234win64:65339/ and Ma
```

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 parameter 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_04167 with NodeURI http://bat6234win64:65339/ and M
Shutting down ROS master on http://172.30.131.134:54064.
```

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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Setting parameter values as heterogeneous cell arrays and structures are not supported.

See Also

`get` | `rosparam`

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.

Note showdetails will be removed. Use `rosShowDetails` instead. For more information, see "ROS Message Structure Functions" on page 2-288

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.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions

Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name

`Image`
 	
`CompressedImage`
	`readImage`
	
	`writeImage`
		`rosReadImage`
		
		`rosWriteImage`

`LaserScan`	`readCartesian`
	
	`readScanAngles`
	
	`lidarScan`
	
	`plot`
		`rosReadCartesian`
		
		`rosReadScanAngles`
		
		`rosReadLidarScan`
		
		`rosPlot`

`PointCloud2`	`apply`
	
	`readXYZ`
	
	`readRGB`
	
	`readAllFieldNames`
	
	`readField`
	
	`scatter3`
		`rosApplyTransform`
		
		`rosReadXYZ`
		
		`rosReadRGB`
		
		`rosReadAllFieldNames`
		
		`rosReadField`
		
		`rosPlot`

`Quaternion`	`readQuaternion`	`rosReadQuaternion`

`OccupancyGrid`	`readBinaryOccupanyGrid`
	
	`readOccupancyGrid`
	
	`writeBinaryOccupanyGrid`
	
	`writeOccupanyGrid`
		`rosReadOccupancyGrid`
		
		`rosReadBinaryOccupancyGrid`
		
		`rosReadOccupancyGrid`
		
		`rosWriteBinaryOccupancyGrid`
		
		`rosWriteOccupancyGrid`

`Octomap`	`readOccupancyMap3D`	`rosReadOccupancyMap3D`

Message Types	Object Function Name	New Function Name

`PointStamped`
`PoseStamped`
`QuaternionStamped`
`Vector3Stamped`
`TransformStamped`
	`apply`	`rosApplyTransform`

All messages	`showdetails`	`rosShowDetails`

See Also
`rosmessage`

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 = '192.168.203.131';
d = rosdevice(ipaddress,'user','password')
```

```
d = 
  rosdevice with properties:

      DeviceAddress: '192.168.203.131'
           Username: 'user'
          ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws'
     AvailableNodes: {'voxel_grid_filter_sl'}
```

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)
pause(2)
running = isCoreRunning(d)

running = logical
   0
```

Input Arguments

device — ROS device
`rosdevice` object

ROS device, specified as a `rosdevice` object.

Version History

Introduced in R2019b

See Also

`rosdevice` | `runCore` | `isCoreRunning`

Topics
"Generate a Standalone ROS Node from Simulink"

stopNode

Stop ROS or ROS 2 node

Syntax

`stopNode(device,modelName)`

Description

`stopNode(device,modelName)` stops a running ROS or ROS 2 node that was deployed from a Simulink model named `modelName`. The node is running on the specified `rosdevice` or `ros2device` 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 = 1×2 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: {0×1 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/
```

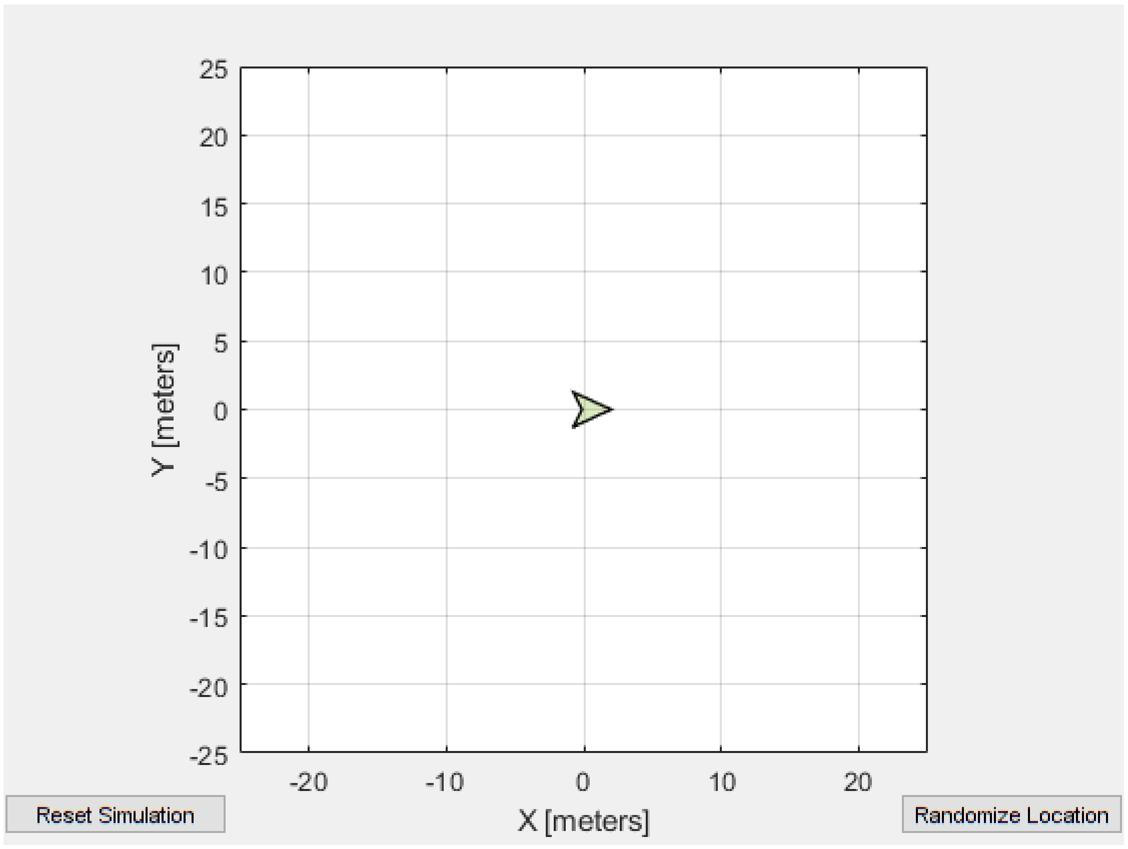
Check the available ROS nodes on the connected ROS device. The nodes listed in this example were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink” example. Two separate nodes are generated, `'robotcontroller'` and `'robotcontroller2'`, which have the linear velocity set to 1 and 2 in the model respectively.

```
d.AvailableNodes
```

```
ans = 1×2 cell
    {'robotcontroller'}    {'robotcontroller2'}
```

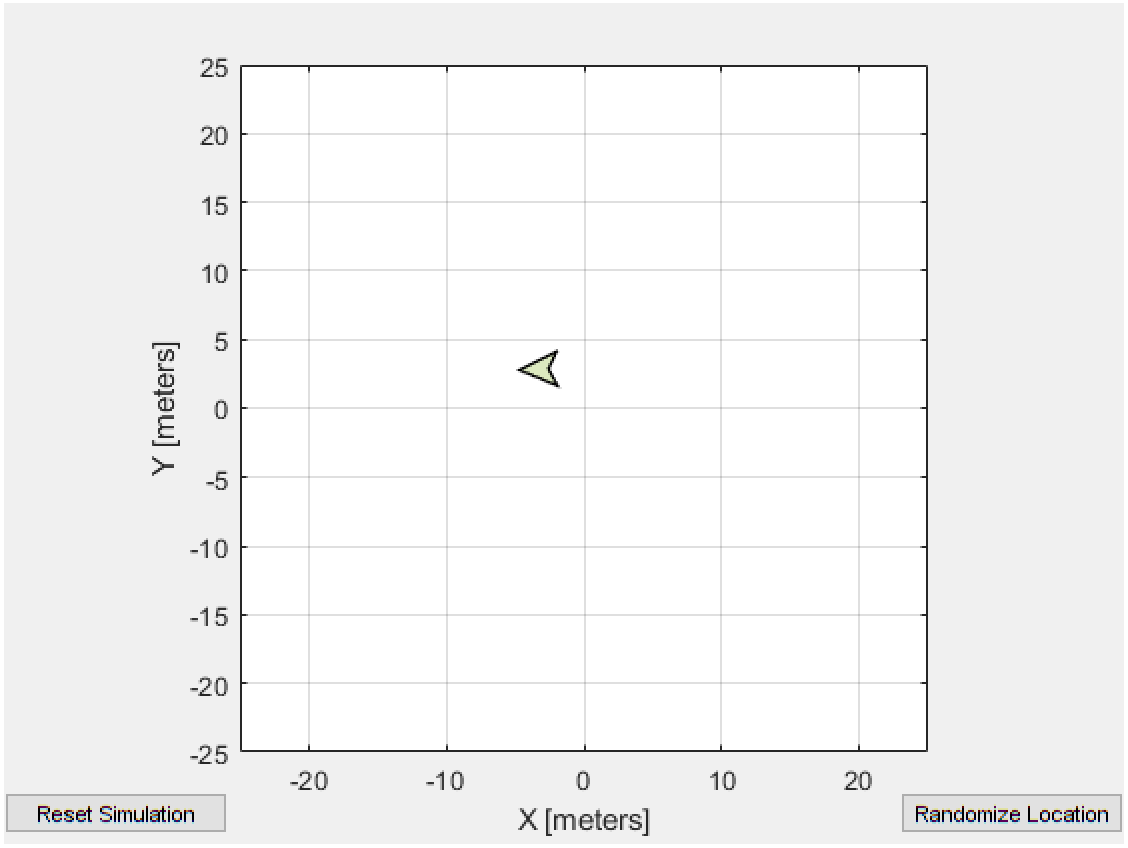
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



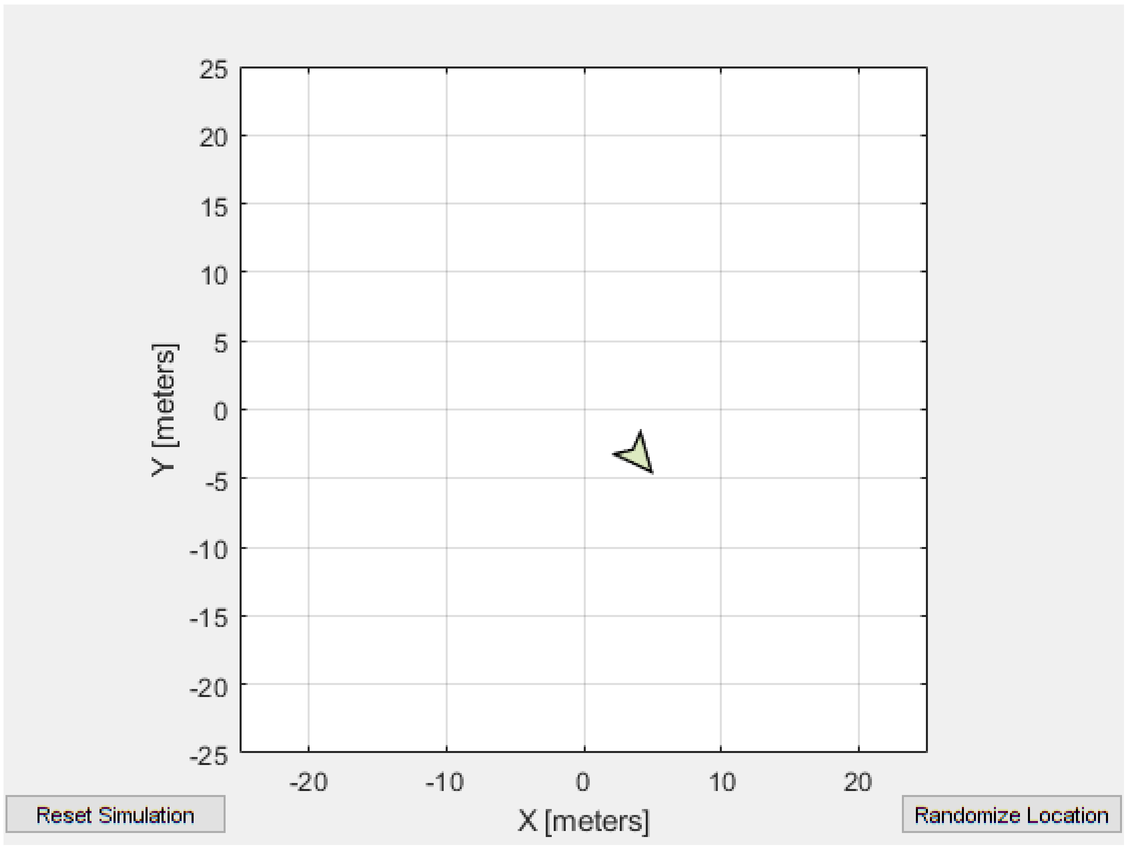
Run a ROS node, specifying the node name. The `'robotcontroller'` node commands the robot to a specific location (`[-10 10]`). Wait to see the robot drive.

```
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.

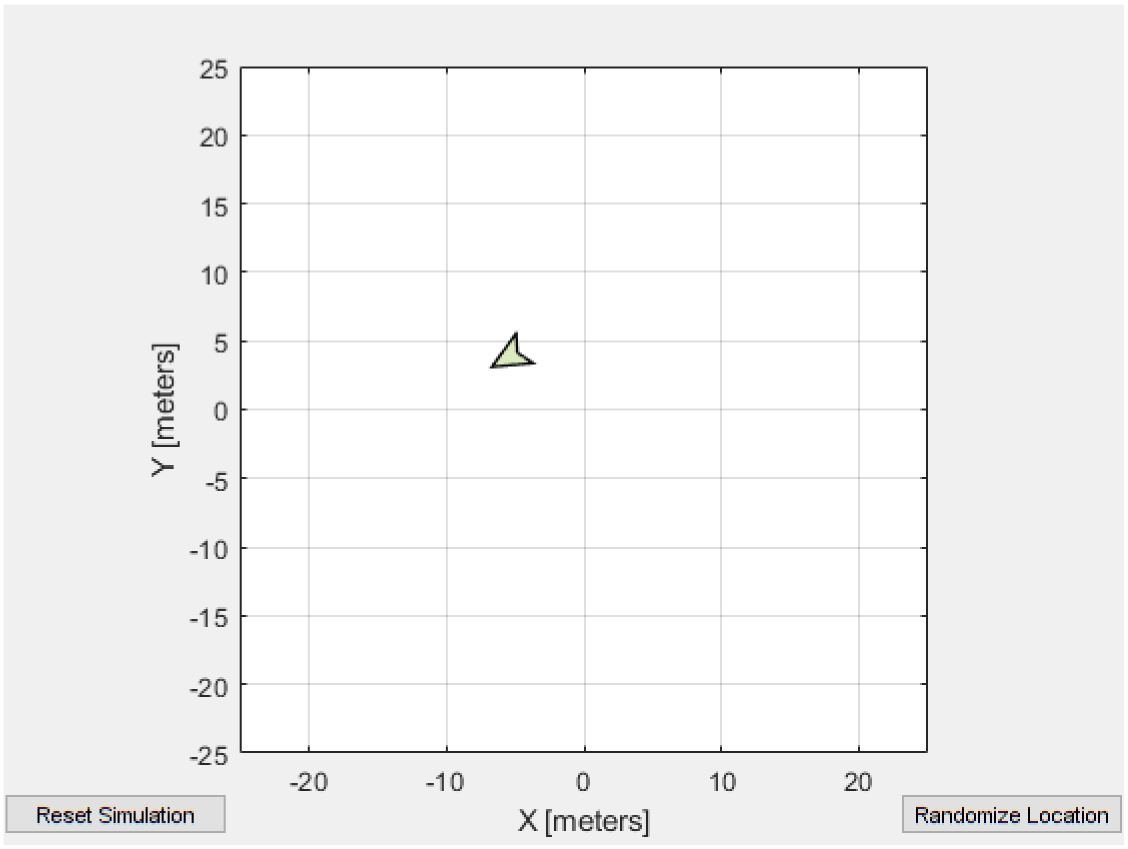
```
resetSimulation(sim.Simulator)
pause(5)
```



```
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')
rosshutdown
```

```
Shutting down global node /matlab_global_node_66434 with NodeURI http://192.168.203.1:59395/
```

```
stopCore(d)
```

Input Arguments

device — ROS or ROS2 device

`rosdevice` object | `ros2device` object

ROS or ROS 2 device, specified as a `rosdevice` or `ros2device` object, respectively.

`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.

Version History

Introduced in R2019b

See Also

`rosdevice` | `ros2device` | `runNode` | `isNodeRunning`

Topics

"Generate a Standalone ROS Node from Simulink"
"Generate a Standalone ROS 2 Node from Simulink"

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 or ROS 2 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 or ROS 2 device
`rosdevice` object | `ros2device` object

ROS or ROS 2 device, specified as a `rosdevice` or `ros2device` object, respectively.

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.

Version History

Introduced in R2019b

See Also

`rosdevice` | `ros2device` | `putFile` | `getFile` | `deleteFile` | `dir` | `openShell`

timeseries

Create 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` or `rosbagreader` 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 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');
```

Create rosbag Selection Using `rosbagreader` Object

Load a rosbag log file and parse out specific messages based on the selected criteria.

Create a `rosbagreader` object of all the messages in the rosbag log file.

```
bagMsgs = rosbagreader("ros_multi_topics.bag")
```

```
bagMsgs = 
  rosbagreader with properties:

           FilePath: 'B:\matlab\toolbox\robotics\robotexamples\ros\data\bags\ros_multi_topics.bag
          StartTime: 201.3400
            EndTime: 321.3400
        NumMessages: 36963
    AvailableTopics: [4x3 table]
    AvailableFrames: {0x1 cell}
        MessageList: [36963x4 table]
```

Select a subset of the messages based on their timestamp and topic.

```
bagMsgs2 = select(bagMsgs,...
    Time=[bagMsgs.StartTime bagMsgs.StartTime + 1],...
    Topic='/odom')
```

```
bagMsgs2 = 
  rosbagreader with properties:

           FilePath: 'B:\matlab\toolbox\robotics\robotexamples\ros\data\bags\ros_multi_topics.bag
          StartTime: 201.3400
```

```
              EndTime: 202.3200
          NumMessages: 99
      AvailableTopics: [1x3 table]
      AvailableFrames: {0x1 cell}
          MessageList: [99x4 table]
```

Retrieve the messages in the selection as a cell array.

```
msgs = readMessages(bagMsgs2)
```

```
msgs=99×1 cell array
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
      ⋮
```

Return certain message properties as a time series.

```
ts = timeseries(bagMsgs2,...
    'Pose.Pose.Position.X', ...
    'Twist.Twist.Angular.Y')
```

```
  timeseries

  Timeseries contains duplicate times.

  Common Properties:
            Name: '/odom Properties'
            Time: [99x1 double]
        TimeInfo: tsdata.timemetadata
            Data: [99x2 double]
        DataInfo: tsdata.datametadata
```

Input Arguments

bag — Index of messages in rosbag
`BagSelection` object | `rosbagreader` object

Index of the messages in the rosbag, specified as a `BagSelection` or `rosbagreader` object. You can get a `BagSelection` object by calling `rosbag`.

`property` — Property names
string scalar | character vector

Property names, specified as a string scalar or character vector. You can specify multiple properties. 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.

Version History

Introduced in R2019b

See Also

`rosbag` | `select` | `readMessages` | `rosbagreader`

Topics
"Time Series"

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 = 36×1 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 = rosmessage('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 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.

```
rosshutdown
```

```
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 `rostf` 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 `rostime`.

Output Arguments

`tfentity` — Transformed entity
`Message` object handle

Transformed entity, returned as a `Message` object handle.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also
`getTransform` | `canTransform`

waitForServer

Wait for action server to start

Syntax

```
waitForServer(client)
waitForServer(client,timeout)
status = waitForServer( ___ )
```

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 action server does not start up in the timeout period, this function displays an error.

`status = waitForServer( ___ )` returns a `status` indicating whether the action server is available, using any of the arguments from the previous syntaxes. If the server is not available within the `timeout`, `status` will be `false`, and this function will not display 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:

```
rosrun 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 = '192.168.203.133';
rosinit(ipaddress,11311)
```

```
Initializing global node /matlab_global_node_81947 with NodeURI http://192.168.203.1:54283/
```

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 by specifying the action name. Use structures for ROS messages.

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

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. Set the order to an `int32` value of 8.

```
goalMsg.Order = int32(8);
```

Send the goal and wait for its completion. Specify a timeout of 10 seconds to complete the action.

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,10);
```

```
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  actionlib_tutorials/FibonacciResult
       Sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_81947 with NodeURI http://192.168.203.1:54283/
```

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:

```
rosrun 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('192.168.203.133',11311)
```

```
Initializing global node /matlab_global_node_18287 with NodeURI http://192.168.203.1:55284/
```

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

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

```
goalMsg.Order = int32(4);
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg = struct with fields:
    MessageType: 'actionlib_tutorials/FibonacciResult'
       Sequence: [0 1 1 2 3]


resultState =
'succeeded'
```

```
rosShowDetails(resultMsg)
```

```
ans =
    '
       MessageType :  actionlib_tutorials/FibonacciResult
       Sequence    :  [0, 1, 1, 2, 3]'
```

Send a new goal message without waiting.

```
goalMsg.Order = int32(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_18287 with NodeURI http://192.168.203.1:55284/
```

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.

Output Arguments

`status` — Status of the action server start up
`logical` scalar

Status of the action server start up, returned as a `logical` scalar. If the server is not available within the timeout period, `status` will be `false`.

Note Use the `status` output argument when you use waitForServer for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`rosactionclient` | `sendGoalAndWait` | `cancelGoal` | `rosaction`

Topics
“ROS Actions Overview”
“Move a Turtlebot Robot Using ROS Actions”

waitForServer

Wait for ROS or ROS 2 service server to start

Syntax

```
waitForServer(client)
waitForServer(client,Timeout=timeoutperiod)
[status,statustext] = waitForServer( ___ )
```

Description

`waitForServer(client)` waits until the service server is started up and available to receive requests. Press **Ctrl+C** to cancel the wait.

`waitForServer(client,Timeout=timeoutperiod)` specifies a timeout period in seconds using the name-value pair `Timeout=timeoutperiod`. If the service server does not start up in the timeout period, this function displays an error 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.

`[status,statustext] = waitForServer( ___ )` returns a `status` indicating whether the service server is available, and a `statustext` that captures additional information about the `status`, using any of the arguments from the previous syntaxes. If the server is not available within the `Timeout`, `status` will be `false`, and this function will not display an error.

Examples

Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6739 seconds.
Initializing ROS master on http://172.30.131.134:59927.
Initializing global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and Ma
```

Set up a service server. Use structures for the ROS message data format.

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

Check whether the service server is available. If it is, wait for the service client to connect to the server.

```
if(isServerAvailable(client))
    [connectionStatus,connectionStatustext] = waitForServer(client)
end
```

```
connectionStatus = logical
   1


connectionStatustext = 
'success'
```

Call service server with default message.

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

```
response = struct with fields:
    MessageType: 'std_srvs/EmptyResponse'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response)
end
```

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

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and M
Shutting down ROS master on http://172.30.131.134:59927.
```

Call ROS 2 Service Client With a Custom Callback Function

Create a sample ROS 2 network with two nodes.

```
node_1 = ros2node('node_1_service_client');
node_2 = ros2node('node_2_service_client');
```

Set up a service server and attach it to a ROS 2 node. Specify the callback function `flipstring`, which flips the input string. The callback function is defined at the end of this example.

```
server = ros2svcserver(node_1,'/test','test_msgs/BasicTypes',@flipString);
```

Set up a service client of the same service type and attach it to a different node.

```
client = ros2svcclient(node_2,'/test','test_msgs/BasicTypes');
```

Wait for the service client to connect to the server.

```
[connectionStatus,connectionStatustext] = waitForServer(client)
```

```
connectionStatus = logical
   1
```

```
connectionStatustext = 
'success'
```

Create a request message based on the client. Assign the string to the corresponding field in the message, `string_value`.

```
request = ros2message(client);
request.string_value = 'hello world';
```

Check whether the service server is available. If it is, send a service request and wait for a response. Specify that the service waits 3 seconds for a response.

```
if(isServerAvailable(client))
    response = call(client,request,'Timeout',3);
end
```

The response is a flipped string from the request message which you see in the `string_value` field.

```
response.string_value
```

```
ans = 
'dlrow olleh'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,request,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response.string_value)
end
```

```
dlrow olleh
```

The callback function used to flip the string is defined below.

```
function resp = flipString(req,resp)
% FLIPSTRING Reverses the order of a string in REQ and returns it in RESP.
resp.string_value = fliplr(req.string_value);
end
```

Input Arguments

client — ROS or ROS 2 service client

`ros.ServiceClient` object handle | `ros2serviceclient` object handle

ROS or ROS 2 service client, specified as a `ros.ServiceClient` or `ros2serviceclient` object handle, respectively. This service client enables you to send requests to the service server.

Output Arguments

`status` — Status of the service server start up
`logical` scalar

Status of the service server start up, returned as a `logical` scalar. If the server is not available within the timeout period, `status` will be `false`.

Note Use the `status` output argument when you use waitForServer for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

`statustext` — Status text associated with the service call status
character vector

Status text associated with the service call status, returned as one of the following:

- `'success'` — The server is available.
- `'input'` — The input to the function is invalid.
- `'timeout'` — The server did not become available before the timeout period expired.

Version History
Introduced in R2021b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also
`rossvcclient` | `rossvcserver` | `ros2svcclient` | `ros2svcserver` | `call` | `rosservice`

Topics
“Call and Provide ROS Services”
“Call and Provide ROS 2 Services”

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 functions 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 `rostf` 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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`transform` | `getTransform` | `receive`

writeBinaryOccupancyGrid

Write values from grid to ROS message

Syntax

`writeBinaryOccupancyGrid(msg,map)`

Description

`writeBinaryOccupancyGrid(msg,map)` writes occupancy values and other information to the ROS message, `msg`, from the binary occupancy grid, `map`.

Input Arguments

`map` — Binary occupancy grid
`binaryOccupancyMap` object handle

Binary occupancy grid, specified as a object handle. `map` is converted to a `'nav_msgs/OccupancyGrid'` message on the ROS network. `map` is an object with a grid of binary values, where `1` indicates an occupied location and `0` indications an unoccupied location.

`msg` — `'nav_msgs/OccupancyGrid'` ROS message
`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as a `OccupancyGrid` object handle.

Version History

Introduced in R2015a

See Also

Functions
`rosReadBinaryOccupancyGrid` | `rosReadOccupancyMap3D` | `rosReadOccupancyGrid` | `rosWriteOccupancyGrid`

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.

Note writeImage will be removed. Use `rosWriteImage` instead. For more information, see “ROS Message Structure Functions” on page 2-323

`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.

Version History

Introduced in R2019b

R2021a: ROS Message Structure Functions
Not recommended starting in R2021a

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To support message structures as inputs, new functions that operate on specialized ROS messages have been provided. These new functions are based on the existing object functions of message objects, but support ROS and ROS 2 message structures as inputs instead of message objects.

The object functions will be removed in a future release.

Message Types	Object Function Name	New Function Name
`Image` `CompressedImage`	`readImage` `writeImage`	`rosReadImage` `rosWriteImage`

Message Types	Object Function Name	New Function Name
LaserScan	readCartesian readScanAngles lidarScan plot	rosReadCartesian rosReadScanAngles rosReadLidarScan rosPlot
PointCloud2	apply readXYZ readRGB readAllFieldNames readField scatter3	rosApplyTransform rosReadXYZ rosReadRGB rosReadAllFieldNames rosReadField rosPlot
Quaternion	readQuaternion	rosReadQuaternion
OccupancyGrid	readBinaryOccupanyGrid readOccupancyGrid writeBinaryOccupanyGrid writeOccupanyGrid	rosReadOccupancyGrid rosReadBinaryOccupancyGrid rosReadOccupancyGrid rosWriteBinaryOccupancyGrid rosWriteOccupancyGrid
Octomap	readOccupancyMap3D	rosReadOccupancyMap3D
PointStamped PoseStamped QuaternionStamped Vector3Stamped TransformStamped	apply	rosApplyTransform
All messages	showdetails	rosShowDetails

See Also

rosReadImage | rosWriteImage | readRGB

writeOccupancyGrid

Write values from grid to ROS message

Syntax

`writeOccupancyGrid(msg,map)`

Description

`writeOccupancyGrid(msg,map)` writes occupancy values and other information to the ROS message, `msg`, from the occupancy grid, `map`.

Input Arguments

msg — `'nav_msgs/OccupancyGrid'` ROS message
`OccupancyGrid` object handle

`'nav_msgs/OccupancyGrid'` ROS message, specified as an `OccupancyGrid` ROS message object handle.

map — Occupancy map
`occupancyMap` object handle

Occupancy map, specified as an `occupancyMap` object handle.

Version History

Introduced in R2016b

See Also

Functions
`rosReadBinaryOccupancyGrid` | `rosReadOccupancyMap3D` | `rosReadOccupancyGrid` | `rosWriteOccupancyGrid`

3

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 anc
```

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

canTransform	Verify if transformation is available
getTransform	Retrieve transformation between two coordinate frames
readMessages	Read messages from rosbag
select	Select subset of messages in rosbag
timeseries	Create 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\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex32890909\ex_multiple_topics.b
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');
```

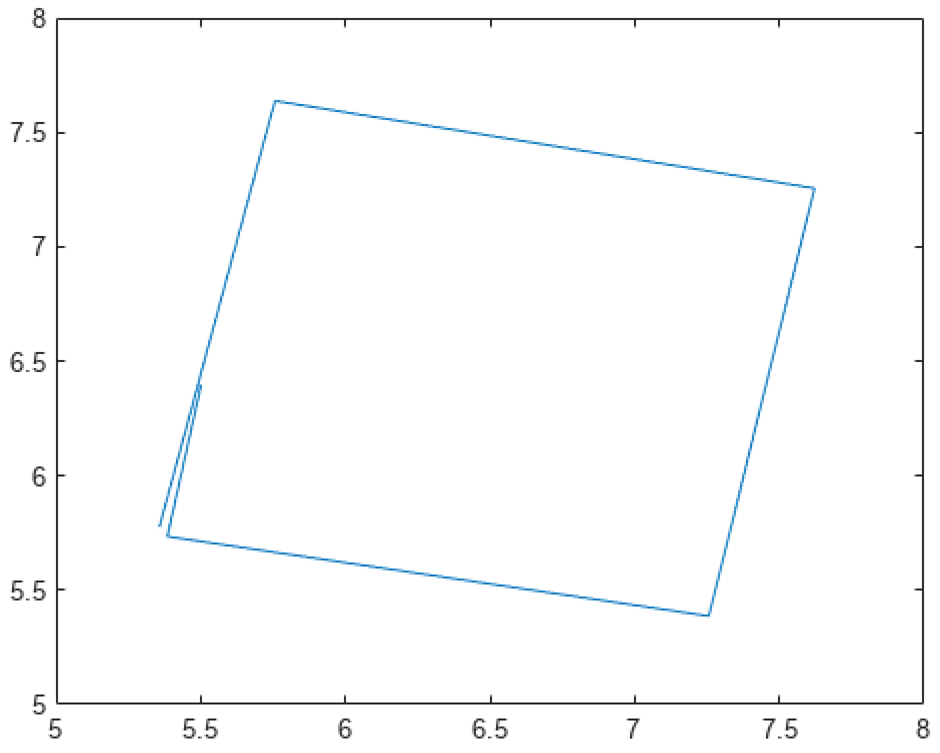
Read messages as a structure. Specify the `DataFormat` name-value pair when reading the messages. Inspect the first structure in the returned cell array of structures.

```
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)
```



Version History

Introduced in R2019b

See Also

`rosbag` | `readMessages` | `select` | `canTransform` | `getTransform` | `timeseries`

Topics

"Work with rosbag Logfiles"
"ROS Log Files (rosbags)"

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 On Specific Port

Create a ROS core on `localhost` and port `12000`.

```
core = ros.Core(12000)
```

```
Launching ROS Core...
..Done in 2.1671 seconds.

core = 
  Core with properties:

         Port: 12000
    MasterURI: 'http://172.30.131.134:12000'
```

Clear the ROS core to shut down the ROS network.

```
clear('core')
```

Version History

Introduced in R2019b

See Also

`rosinit` | `Node`

Topics

"Connect to a ROS Network"
"ROS Network Setup"

External Websites

ROS Core

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 = rosmessage('sensor_msgs/CompressedImage')`

Description

`msg = rosmessage('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 = rosmessage('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
```

Version History

Introduced in R2019b

See Also

`readImage` | `rosmessage` | `rossubscriber`

Topics

"Work with Specialized ROS Messages"

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 = rosmessage('sensor_msgs/Image')`

Description

`msg = rosmessage('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.

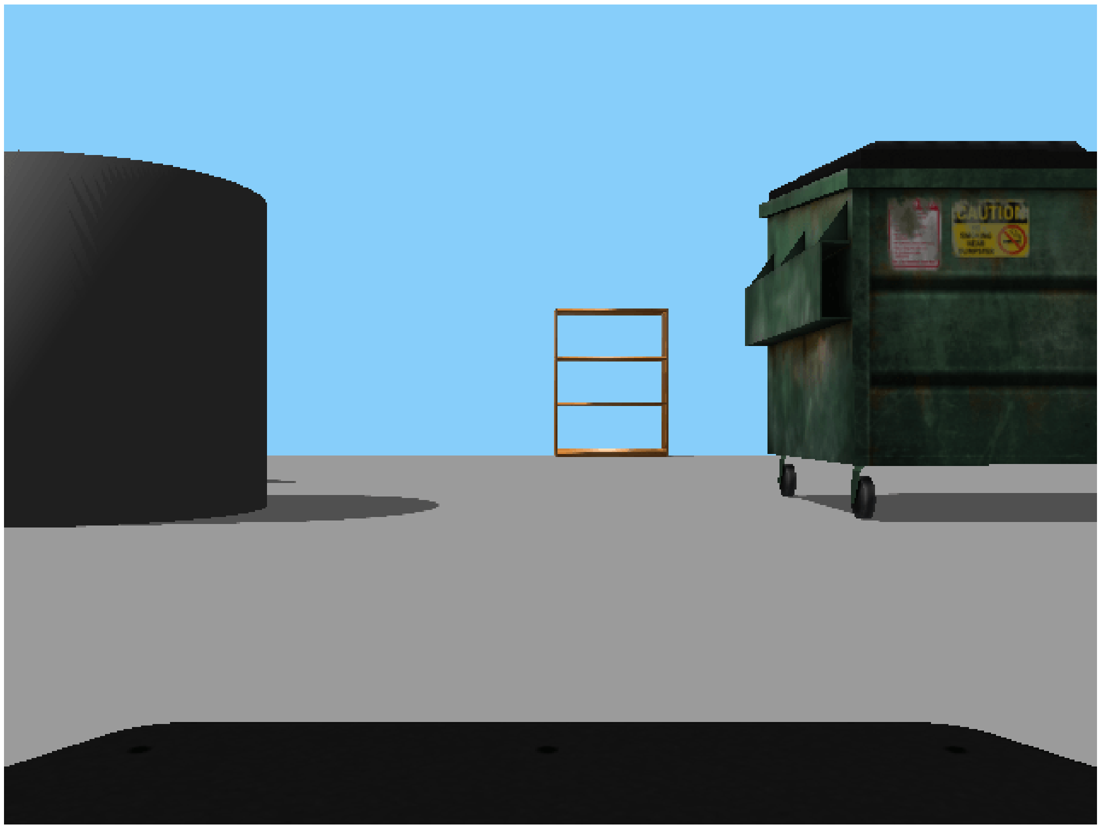
Load sample ROS messages and inspect the image message data. The `img` object is a sample ROS `Image` message object.

```
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 = rosmessage('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
```

Version History

Introduced in R2019b

See Also

`readImage` | `writeImage` | `rosmessage` | `rossubscriber`

Topics

"Work with Specialized ROS Messages"

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

lidarScan	Create object for storing 2-D lidar scan
plot	Display laser or lidar scan readings
readCartesian	Read laser scan ranges in Cartesian coordinates
readScanAngles	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.

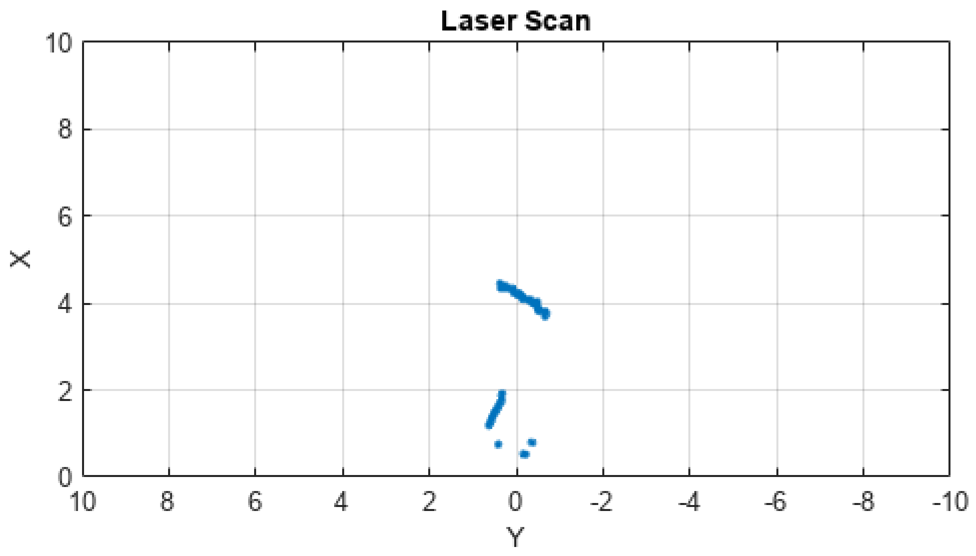
```
ranges = scan.Ranges;
angles = scan.readScanAngles;
size(ranges)
```

```
ans = 1×2

   640     1
```

```
size(angles)
```

```
ans = 1×2

   640     1
```

```
plot(scan)
```



Create Empty LaserScan Message

```
scan = rosmessage('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
```

Version History

Introduced in R2019b

See Also

`lidarScan` | `plot` | `readCartesian` | `readScanAngles` | `showdetails` | `rosmessage` | `rossubscriber`

Topics

"Work with Specialized ROS Messages"

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;
```

```
Launching ROS Core...
...Done in 3.2445 seconds.
```

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
```

```
Launching ROS Core...
..Done in 2.1948 seconds.
```

```
m2 = ros.Core(12000);
```

```
Launching ROS Core...
..Done in 2.6869 seconds.
```

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')
```

Version History

Introduced in R2019b

See Also

`rosinit` | `rosshutdown`

Topics

"ROS Network Setup"

External Websites

ROS Nodes

ros2node

Create a ROS 2 node on the specified network

Description

The `ros2node` object 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)
node = ros2node( ___ ,Parameters=params)
```

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.

By default the node uses the `'rmw_fastrtps_cpp'` ROS middleware (RMW) implementation unless otherwise specified by the `RMW_IMPLEMENTATION` environment variable. Set the `RMW_IMPLEMENTATION` environment variable before creating the `ros2node` object. For example, `setenv('RMW_IMPLEMENTATION','rmw_cyclonedds_cpp')` sets the RMW implementation to `'rmw_cyclonedds_cpp'`. For more information on RMW implementations see “Switching Between ROS Middleware Implementations”.

`node = ros2node(Name,ID)` will initialize the ROS 2 node with `Name` and connect to the network using domain `ID`.

`node = ros2node( ___ ,Parameters=params)` specifies parameters to be declared during the node startup using the name-value argument `Parameters`, using any of the arguments from the previous syntaxes. Specify `params` as a structure that contains the parameters as its fields. Each parameter in `params` can be a scalar or an array of `uint8`, `int64`, `logical`, `string`, `char` or `double` datatype.

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 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
getParameter	Get value of parameter declared in ROS 2 node
setParameter	Set value of parameter declared in ROS 2 node

Examples

Initialize a Node on Default ROS 2 Network

Initialize the node, `'/node_1_default'`, on the default ROS 2 network.

```
node1 = ros2node('node_1_default')
```

```
node1 = 
  ros2node with properties:

    Name: '/node_1_default'
```

```
      ID: 0
```

Initialize a Node on Specified ROS 2 Network

Initialize the node, `'/node_2_specified'`, on the ROS 2 network identified with domain `2`.

```
node2 = ros2node('node_2_specified', 2)
```

```
node2 = 
  ros2node with properties:

    Name: '/node_2_specified'
      ID: 2
```

Get and Set Parameters for ROS 2 Nodes

Create a structure that contains all the parameters for the ROS 2 node.

```
nodeParams.my_double = 2.0;
nodeParams.my_namespace.my_int = int64(1);
nodeParams.my_double_array = [1.1 2.2 3.3];
nodeParams.my_string = "Keyparams";
```

Create a ROS 2 node and specify `nodeParams` as the parameters.

```
node1 = ros2node("/node1",Parameters=nodeParams);
```

Set the parameter `my_double` to a new value.

```
setParameter(node1,"my_double",5.2);
```

Obtain the new value of the parameter `my_double`.

```
doubleValue = getParameter(node1,"my_double")
```

```
doubleValue = 5.2000
```

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only one `ros2node` object is allowed in a single MATLAB function. You can write individual MATLAB functions for each additional `ros2node`.

See Also

`ros2publisher` | `ros2subscriber` | `ros2param`

Topics

“Get Started with ROS 2”
“ROS Toolbox System Requirements”
“Switching Between ROS Middleware Implementations”

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 the `readBinaryOccupancyGrid` function.

To create an `occupancyMap` object , use the `readOccupancyGrid` function.

Creation

Syntax

```
msg = rosmessage('nav_msgs/OccupancyGrid');
```

Description

`msg = rosmessage('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

readOccupancyGrid	Read occupancy grid message
readBinaryOccupancyGrid	Read binary occupancy grid
writeBinaryOccupancyGrid	Write values from grid to ROS message
writeOccupancyGrid	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

  complexMap          41x52                 2132  logical
  emptyMap            26x27                  702  logical
  simpleMap           26x27                  702  logical
  ternaryMap         501x501             2008008  double
```

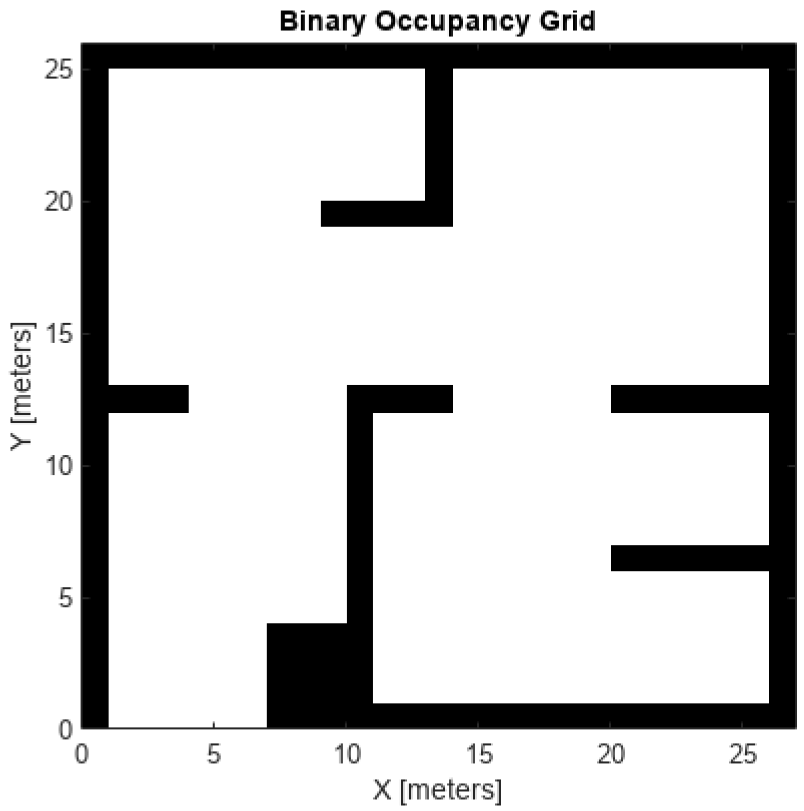
Create a ROS message from `simpleMap` using a `binaryOccupancyMap` object. Write the `OccupancyGrid` message using `writeBinaryOccupancyGrid`.

```
bogMap = binaryOccupancyMap(double(simpleMap));
mapMsg = rosmessage('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);
```



Version History

Introduced in R2019b

See Also

Objects

`occupancyMap` | `binaryOccupancyMap`

Functions

`readBinaryOccupancyGrid` | `readOccupancyGrid` | `writeBinaryOccupancyGrid` | `writeOccupancyGrid`

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 = rosmessage('sensor_msgs/PointCloud2')`

Description

`ptcloud = rosmessage('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 3-34.

Object Functions

readAllFieldNames	Get all available field names from ROS point cloud
readField	Read point cloud data based on field name
readRGB	Extract RGB values from point cloud data
readXYZ	Extract XYZ coordinates from point cloud data
scatter3	Display point cloud in scatter plot
showdetails	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
```

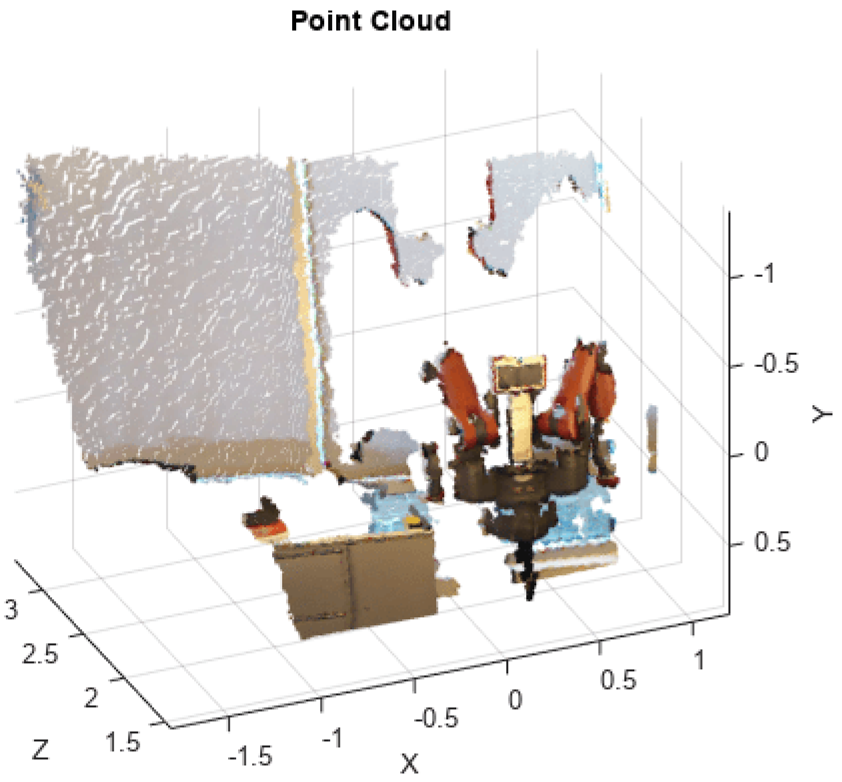
```
ptcloud = 
  ROS PointCloud2 message with properties:

    PreserveStructureOnRead: 0
                MessageType: 'sensor_msgs/PointCloud2'
                     Header: [1x1 Header]
                     Fields: [4x1 PointField]
                     Height: 480
                      Width: 640
                IsBigendian: 0
                  PointStep: 32
                    RowStep: 20480
                       Data: [9830400x1 uint8]
                    IsDense: 0

  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]
        Count: 307200
      XLimits: [-1.8147 1.1945]
      YLimits: [-1.3714 0.8812]
      ZLimits: [1.4190 3.3410]
        Color: [307200x3 uint8]
       Normal: []
    Intensity: []
```

Version History

Introduced in R2019b

See Also

`readAllFieldNames` | `readField` | `readRGB` | `readXYZ` | `scatter3` | `showdetails` | `rosmessage` | `rossubscriber`

Topics

"Work with Specialized ROS Messages"

rosdevice

Connect to remote ROS device

Description

The `rosdevice` object is used to create a connection with a ROS device. The ROS device can be the local device or a remote 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”.

You can also deploy ROS nodes generated from MATLAB code.

Note To connect to a remote ROS device, an SSH server must be installed on the device. To connect to the local host, an SSH server installation on the local device is not required if you specify the `deviceAddress` as `'localhost'`. Alternatively, if you specify the `deviceAddress` as `'127.0.0.1'` or as the host name referring to the local device, then an SSH server must be installed on the local device.

Creation

Syntax

```
device = rosdevice(deviceAddress,username,password)
device = rosdevice
device = rosdevice('localhost')
```

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`.

`device = rosdevice('localhost')` creates a `rosdevice` object connected to the local device.

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

runNode	Start ROS or ROS 2 node
stopNode	Stop ROS or ROS 2 node
isNodeRunning	Determine if ROS or ROS 2 node is running
runCore	Start ROS core
stopCore	Stop ROS core
isCoreRunning	Determine if ROS core is running
system	Execute system command on device
putFile	Copy file to device
getFile	Get file from device
deleteFile	Delete file from device
dir	List folder contents on device
openShell	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 = '192.168.203.131';
d = rosdevice(ipaddress,'user','password')
```

```
d = 
  rosdevice with properties:

      DeviceAddress: '192.168.203.131'
           Username: 'user'
          ROSFolder: '/opt/ros/indigo'
    CatkinWorkspace: '~/catkin_ws'
     AvailableNodes: {'voxel_grid_filter_sl'}
```

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 t
```

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

```
running = logical
   1
```

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

```
stopCore(d)
pause(2)
running = isCoreRunning(d)
```

```
running = logical
   0
```

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 = 1×2 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: {0×1 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/
```

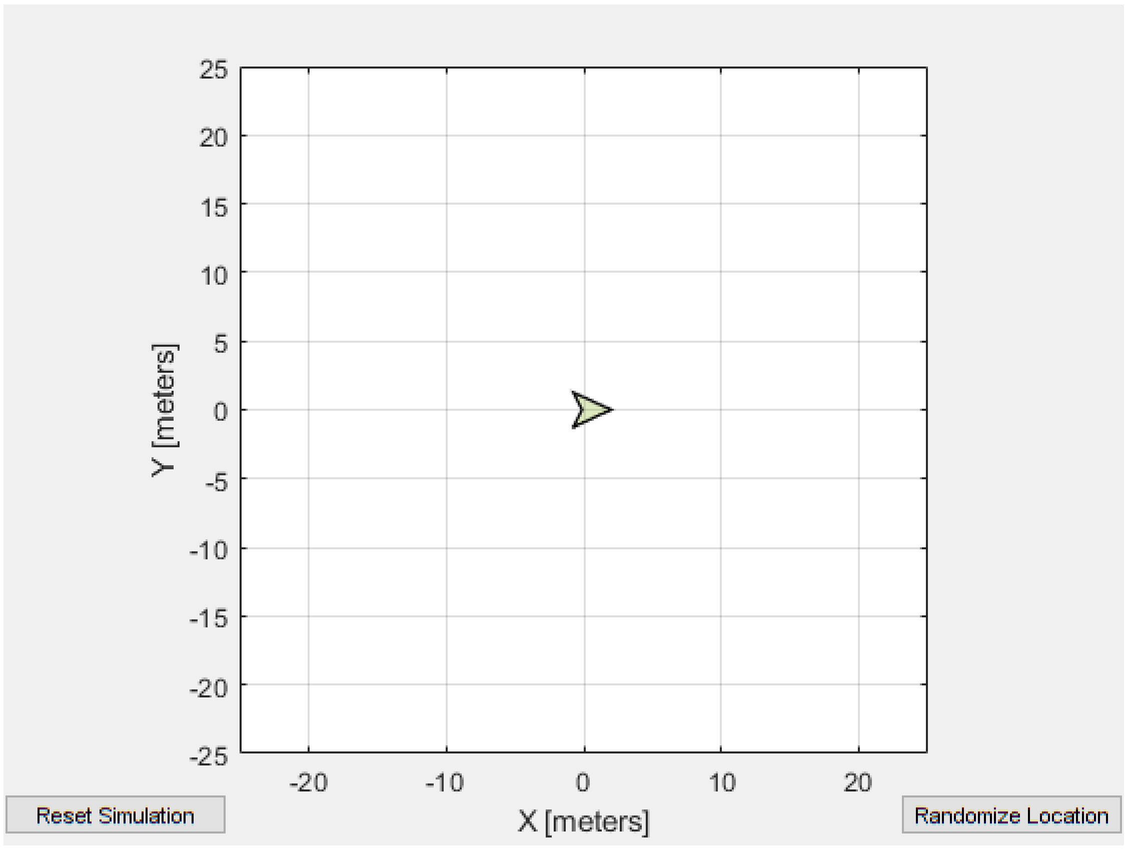
Check the available ROS nodes on the connected ROS device. The nodes listed in this example were generated from Simulink® models following the process in the “Generate a Standalone ROS Node from Simulink” example. Two separate nodes are generated, `'robotcontroller'` and `'robotcontroller2'`, which have the linear velocity set to 1 and 2 in the model respectively.

```
d.AvailableNodes
```

```
ans = 1×2 cell
    {'robotcontroller'}    {'robotcontroller2'}
```

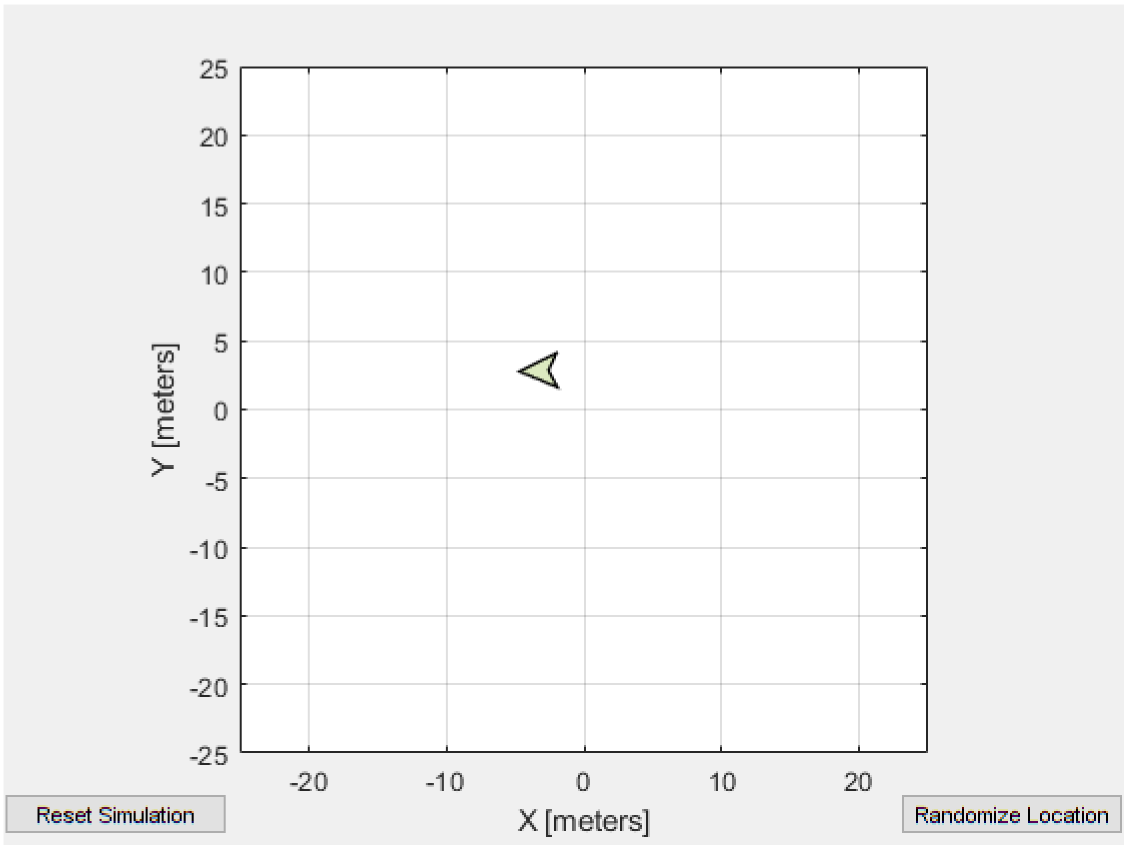
Start up the Robot Simulator using `ExampleHelperSimulinkRobotROS`. This simulator automatically connects to the ROS master on the ROS device. You will use this simulator to run a ROS node and control the robot.

```
sim = ExampleHelperSimulinkRobotROS;
```



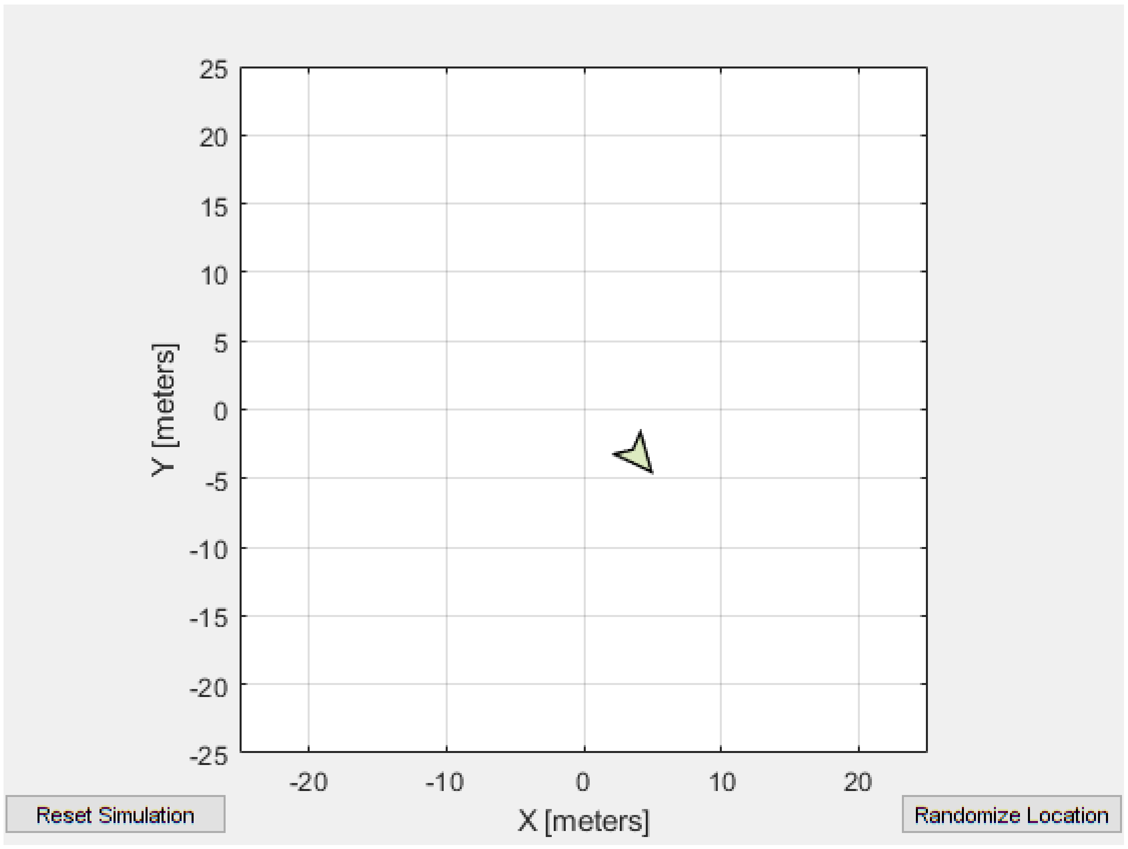
Run a ROS node, specifying the node name. The `'robotcontroller'` node commands the robot to a specific location (`[-10 10]`). Wait to see the robot drive.

```
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.

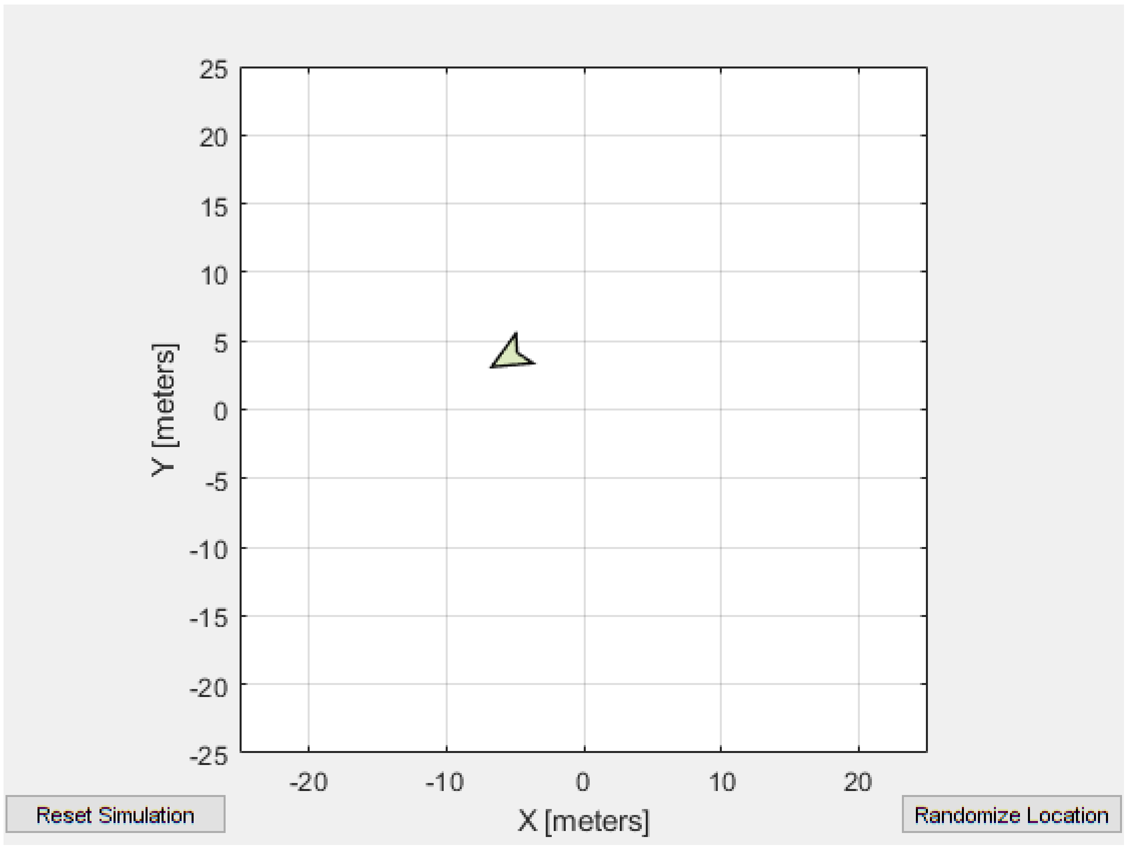
```
resetSimulation(sim.Simulator)
pause(5)
```



```
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')
rosshutdown
```

```
Shutting down global node /matlab_global_node_66434 with NodeURI http://192.168.203.1:59395/
```

```
stopCore(d)
```

Limitations

- You cannot change the `ROSFolder` property when connected to local host. For local host connections, it will always point to the ROS folder within MATLAB installation.

Version History

Introduced in R2019b

See Also

`runNode` | `stopNode` | `runCore` | `isNodeRunning`

Topics
"Generate a Standalone ROS Node from Simulink"

ros2device

Connect to remote ROS 2 device

Description

The `ros2device` object creates a connection with a ROS 2 device. The ROS 2 device can be the local device or a remote device. The object contains the necessary login information and other parameters of the ROS 2 distribution. Once you have made a connection using `ros2device`, you can run and stop ROS 2 nodes.

You can deploy ROS 2 nodes to a ROS 2 device using Simulink models. For an example, see “Generate a Standalone ROS 2 Node from Simulink”

You can also deploy ROS 2 nodes generated from MATLAB code.

.

Note To connect to a ROS 2 device, an SSH server must be installed on the device. To connect to the local host, an SSH server installation on the local device is not required if you specify the `deviceAddress` as `'localhost'`. Alternatively, if you specify the `deviceAddress` as `'127.0.0.1'` or as the host name referring to the local device, then an SSH server must be installed on the local device.

Creation

Syntax

```
device = ros2device(deviceAddress,username,password)
device = ros2device
device = ros2device('localhost')
```

Description

`device = ros2device(deviceAddress,username,password)` creates a `ros2device` object connected to the ROS 2 device at the specified address and with the specified user name and password.

`device = ros2device` creates a `ros2device` object connected to a ROS 2 device using the saved values for `deviceAddress`, `username`, and `password`.

`device = ros2device('localhost')` creates a `ros2device` object connected to the local device.

Properties

`DeviceAddress` — Host name or IP address of ROS 2 device
character vector

This property is read-only.

Host name or IP address of ROS 2 device, specified as a character vector.

Example: `'192.168.1.10'`

Example: `'samplehost.foo.com'`

`UserName` — User name used to connect to device
character vector

This property is read-only.

User name used to connect to ROS 2 device, specified as a character vector.

Example: `'user'`

`ROS2Folder` — Location of ROS 2 installation
character vector

Location of the ROS 2 installation, specified as a character vector. If you do not specify a folder, MATLAB tries to determine the correct folder for you. When you deploy a ROS 2 node, set this value from Simulink in the Configuration Parameters dialog box, under **Hardware Implementation**.

Example: `'/opt/ros/foxy'`

`ROS2Workspace` — ROS 2 project folder where models are deployed on device
character vector

ROS 2 project folder where models are deployed on device, specified as a character vector. When you deploy a ROS 2 node, set this value from Simulink in the Configuration Parameters dialog box, under **Hardware Implementation**.

Example: `'~/ros2_ws_test'`

`AvailableNodes` — Nodes available to run on ROS 2 device
cell array of character vectors

This property is read-only.

Nodes available to run on a ROS 2 device, returned as a cell array of character vectors. Nodes are only listed if they are part of the `ROS2Workspace` and have been deployed to the device using Simulink.

Example: `{'robotcontroller','publishernode'}`

Object Functions

runNode	Start ROS or ROS 2 node
stopNode	Stop ROS or ROS 2 node
isNodeRunning	Determine if ROS or ROS 2 node is running
system	Execute system command on device
putFile	Copy file to device
getFile	Get file from device
deleteFile	Delete file from device
dir	List folder contents on device
openShell	Open interactive command shell to device

Examples

Run a ROS 2 Node on Remote Device

Connect to a remote device and start a ROS 2 node using a `ros2device` object. Create a `ros2device` object by specifying the address, user name, and password of the remote device.

```
ipaddress = '192.168.203.131';
device = ros2device(ipaddress,'user','password');
device.ROS2Folder = '/opt/ros/foxy';
device.ROS2Workspace = '~/ros2_ws_test';
```

The `ros2device` object also contains information about the available ROS nodes. Check the available ROS 2 nodes on the connected device.

```
device.AvailableNodes
```

To execute the node on `'rmw_cyclonedds_cpp'` ROS middleware set the `RMW_IMPLEMENTATION` environment variable using `setenv`.

```
setenv("RMW_IMPLEMENTATION","rmw_cyclonedds_cpp")
```

Use the `runNode` object function to run a ROS 2 node on the remote device, and then check that if the node is running.

```
runNode(device,'ros2FeedbackControlExample')
isNodeRunning(device,'ros2FeedbackControlExample')
```

Stop the ROS 2 node.

```
stopNode(device,'ros2FeedbackControlExample')
```

Limitations

- You cannot change the `ROS2Folder` property when connected to local host. For local host connections, it will always point to the ROS 2 folder within MATLAB installation.

Version History

Introduced in R2021a

See Also

`runNode` | `stopNode` | `isNodeRunning`

Topics

“Generate a Standalone ROS 2 Node from Simulink”

ros2svcclient

Connect to ROS 2 service server

Description

Use `ros2svcclient` to create a ROS 2 service client object. This service client uses a connection to send requests to, and receive responses from, a ROS 2 service server. For more information, see "Call and Provide ROS 2 Services".

Creation

Syntax

```
client = ros2svcclient(node,servicename,servicetype)
client = ros2svcclient(___,Name=Value)

[client,reqmsg] = ros2svcclient(___)
```

Description

`client = ros2svcclient(node,servicename,servicetype)` creates a service client of the specified `servicetype` regardless of whether a service server offering `servicename` is available. It attaches the client to the ROS 2 node specified by the `ros2node` object, `node`.

`client = ros2svcclient(___,Name=Value)` sets the rest of the properties on page 3-51 based on the additional options specified by one or more `Name=Value` pair arguments, using the arguments from the previous syntax.

`[client,reqmsg] = ros2svcclient(___)` 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 input service type. The message is initialized with default values. You can also create the request message using `ros2message`.

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"`

`History` — Mode of storing requests in the queue
`"keeplast"` (default) | `"keepall"`

This property is read-only.

Mode of storing requests in the queue, specified as a string or character vector. If the queue fills with requests waiting to be processed, then old requests will be dropped to make room for new. When set to `"keeplast"`, the queue stores the number of requests set by the `Depth` property. Otherwise, when set to `"keepall"`, the queue stores all requests up to the MATLAB resource limits.

Example: `"keeplast"`

Data Types: `char` | `string`

`Depth` — Size of the request queue
10 (default) | `non-negative scalar integer`

This property is read-only.

Size of the request queue in number of requests stored in the queue, specified as a `non-negative scalar integer`. This only applies when `History` is set to `"keeplast"` or `Durability` is set to `"transientlocal"`.

Example: 42

Data Types: `double`

`Reliability` — Delivery guarantee of request
`"reliable"` (default) | `"besteffort"`

This property is read-only.

Requirement on delivery guarantee of request, specified as a string or character vector. If `"reliable"`, then delivery is guaranteed, but may retry calling multiple times. If `"besteffort"`, then attempt delivery and do not retry. `"reliable"` setting is recommended for services.

Note The `Reliability` and `Durability` quality of service settings must be compatible between service servers and clients for a connection to be made.

Example: `"reliable"`

Data Types: `char` | `string`

`Durability` — Persistence of the requests
`"volatile"` (default) | `"transientlocal"`

This property is read-only.

Requirement on persistence of the requests, specified as a string or character vector. If `"volatile"`, then requests do not persist. If `"transientlocal"`, then all recently sent requests persist, up to the number specified by `Depth`. `"volatile"` setting is recommended for services.

Note The `Reliability` and `Durability` quality of service settings must be compatible between service servers and clients for a connection to be made.

Example: `"volatile"`

Data Types: `char` | `string`

Object Functions

ros2message	Create ROS 2 message structures
call	Call ROS or ROS 2 service server and receive a response
isServerAvailable	Determine if ROS or ROS 2 service server is available
waitForServer	Wait for ROS or ROS 2 service server to start

Examples

Call ROS 2 Service Client With a Custom Callback Function

Create a sample ROS 2 network with two nodes.

```
node_1 = ros2node('node_1_service_client');
node_2 = ros2node('node_2_service_client');
```

Set up a service server and attach it to a ROS 2 node. Specify the callback function `flipstring`, which flips the input string. The callback function is defined at the end of this example.

```
server = ros2svcserver(node_1,'/test','test_msgs/BasicTypes',@flipString);
```

Set up a service client of the same service type and attach it to a different node.

```
client = ros2svcclient(node_2,'/test','test_msgs/BasicTypes');
```

Wait for the service client to connect to the server.

```
[connectionStatus,connectionStatustext] = waitForServer(client)
```

```
connectionStatus = logical
   1


connectionStatustext = 
'success'
```

Create a request message based on the client. Assign the string to the corresponding field in the message, `string_value`.

```
request = ros2message(client);
request.string_value = 'hello world';
```

Check whether the service server is available. If it is, send a service request and wait for a response. Specify that the service waits 3 seconds for a response.

```
if(isServerAvailable(client))
    response = call(client,request,'Timeout',3);
end
```

The response is a flipped string from the request message which you see in the `string_value` field.

```
response.string_value
```

```
ans = 
'dlrow olleh'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,request,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response.string_value)
end
```

```
dlrow olleh
```

The callback function used to flip the string is defined below.

```
function resp = flipString(req,resp)
% FLIPSTRING Reverses the order of a string in REQ and returns it in RESP.
resp.string_value = fliplr(req.string_value);
end
```

Tips

- ROS 2 service servers cannot communicate errors in callback execution directly to clients. In such situations, the servers only return the default response without any indication of failure. Hence, it is recommended to use try-catch blocks within the callback function, and set specific fields in the response message to communicate the success/failure of the callback execution on the server side.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `ServiceType` argument must be specified.
- Syntax with multiple output arguments is not supported.
- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`ros2svcserver` | `ros2`

Topics

"Call and Provide ROS 2 Services"

ros2svcserver

Create ROS 2 service server

Description

Use `ros2svcserver` to create a ROS 2 service server that can receive requests from, and send responses to, a ROS 2 service client.

When you create the service server, it registers itself with the ROS 2 network. When you create a service client, it establishes a connection to the server. The connection persists while both client and server exist and can reach each other. To get a list of services, or to get information about a particular service that is available on the current ROS 2 network, use the `ros2` function.

The service has an associated message type that 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 callback 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 2 network and issues a call. For more information, see “Call and Provide ROS 2 Services”.

Creation

Syntax

```
server = ros2svcserver(node,servicename,servicetype,callback)
server = ros2svcserver( ___ ,Name=Value)
```

Description

`server = ros2svcserver(node,servicename,servicetype,callback)` creates a service server of the specified `servicetype` available in the ROS 2 network under the name `servicename`. It attaches the server to the ROS 2 node specified by the `ros2node` object, `node`. It also specifies the `callback` function, which is set to the `NewRequestFcn` property. The input arguments `servicename` and `servicetype`, are set to the `ServiceType` and `ServiceName` properties, respectively.

`server = ros2svcserver( ___ ,Name=Value)` sets the rest of the properties on page 3-56 based on the additional options specified by one or more `Name=Value` pair arguments, using the arguments from the previous syntax.

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"

NewRequestFcn — 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 two input arguments with one output. The first argument, `reqMsg`, is the request message object sent by the service client. The second 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(reqMsg,defaultRespMsg)
```

Specify the `NewRequestFcn` property as:

```
server.NewRequestFcn = @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(reqMsg,defaultRespMsg,userData)
```

Specify the `NewRequestFcn` property as:

```
server.NewRequestFcn = {@serviceCallback,userData};
```

History — Mode of storing requests in the queue

"keeplast" (default) | "keepall"

This property is read-only.

Mode of storing requests in the queue, specified as a string or character vector. If the queue fills with requests waiting to be processed, then old requests will be dropped to make room for new. When set to `"keeplast"`, the queue stores the number of requests set by the `Depth` property. Otherwise, when set to `"keepall"`, the queue stores all requests up to the MATLAB resource limits.

Example: "keeplast"

Data Types: char | string

Depth — Size of the request queue

10 (default) | non-negative scalar integer

This property is read-only.

Size of the request queue in number of requests stored in the queue, specified as a `non-negative scalar integer`. This only applies when `History` is set to `"keeplast"`.

Example: `42`

Data Types: `double`

`Reliability` — Delivery guarantee of request and response
`"reliable"` (default) | `"besteffort"`

This property is read-only.

Requirement on delivery guarantee of request and response, specified as a string or character vector. If `"reliable"`, then delivery is guaranteed, but may retry multiple times. If `"besteffort"`, then attempt delivery and do not retry. `"reliable"` setting is recommended for services.

Note The `Reliability` and `Durability` quality of service settings must be compatible between service servers and clients for a connection to be made.

Example: `"reliable"`

Data Types: `char` | `string`

`Durability` — Persistence of the client
`"volatile"` (default) | `"transientlocal"`

This property is read-only.

Requirement on persistence of the client, specified as a string or character vector. If `"volatile"`, then requests are not required to persist. If `"transientlocal"`, then the server will require clients to persist and receive responses for the number of previous requests specified by `Depth`. `"volatile"` setting is recommended to prevent servers from receiving out of date requests in the event of a server restart.

Note The `Reliability` and `Durability` quality of service settings must be compatible between service servers and clients for a connection to be made.

Example: `"volatile"`

Data Types: `char` | `string`

Object Functions

ros2message	Create ROS 2 message structures

Examples

Call ROS 2 Service Client With a Custom Callback Function

Create a sample ROS 2 network with two nodes.

```
node_1 = ros2node('node_1_service_client');
node_2 = ros2node('node_2_service_client');
```

Set up a service server and attach it to a ROS 2 node. Specify the callback function `flipstring`, which flips the input string. The callback function is defined at the end of this example.

```
server = ros2svcserver(node_1,'/test','test_msgs/BasicTypes',@flipString);
```

Set up a service client of the same service type and attach it to a different node.

```
client = ros2svcclient(node_2,'/test','test_msgs/BasicTypes');
```

Wait for the service client to connect to the server.

```
[connectionStatus,connectionStatustext] = waitForServer(client)
```

```
connectionStatus = logical
   1


connectionStatustext =
'success'
```

Create a request message based on the client. Assign the string to the corresponding field in the message, `string_value`.

```
request = ros2message(client);
request.string_value = 'hello world';
```

Check whether the service server is available. If it is, send a service request and wait for a response. Specify that the service waits 3 seconds for a response.

```
if(isServerAvailable(client))
    response = call(client,request,'Timeout',3);
end
```

The response is a flipped string from the request message which you see in the `string_value` field.

```
response.string_value
```

```
ans =
'dlrow olleh'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,request,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response.string_value)
end
```

```
dlrow olleh
```

The callback function used to flip the string is defined below.

```
function resp = flipString(req,resp)
% FLIPSTRING Reverses the order of a string in REQ and returns it in RESP.
resp.string_value = fliplr(req.string_value);
end
```

Tips

- ROS 2 service servers cannot communicate errors in callback execution directly to clients. In such situations, the servers only return the default response without any indication of failure. Hence, it is recommended to use try-catch blocks within the callback function, and set specific fields in the response message to communicate the success/failure of the callback execution on the server side.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `ServiceType` argument must be specified.
- Callback functions must be assigned at the time of `ros2svcserver` object creation, and cannot be changed during run-time.
- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`ros2svcclient` | `ros2`

Topics
"Call and Provide ROS 2 Services"

rosactionserver

Create ROS Action Server

Description

Use `rosactionserver` to create an action server as a `SimpleActionServer` object. Then, use the `rosactionclient` object to create an action client and connect to the action server to request the execution of action goals. When a connected client sends a goal execution request, the server executes the specified callback function. You can use the `rosActionServerExecuteGoalFcn` function to customize the callback function based on a predefined framework. The server can provide periodic feedback on execution progress to the clients, and stop goal execution if specified or if a new goal is received.

When you create the action server, it registers itself with the ROS master. To get a list of actions, or to get information about a particular action that is available on the current ROS network, use the `rosaction` function.

An action is defined by a type and three messages: one for the goal, one for the feedback, and one for the result. On receiving a goal, the server goal execution callback must periodically send feedback to the client during goal execution, and return an appropriate result when goal execution completes. The behavior of the action server is inherently asynchronous because it becomes active only when an action client connects to the ROS network and issues a goal execution request.

Creation

Syntax

```
server = rosactionserver(actionname,actiontype,ExecuteGoalFcn=cb)
[server] = rosactionserver(___,"DataFormat","struct")

server = ros.SimpleActionServer(node, actionname,actiontype,
ExecuteGoalFcn=cb)
[server] = ros.SimpleActionServer(___,DataFormat="struct")
```

Description

`server = rosactionserver(actionname,actiontype,ExecuteGoalFcn=cb)` creates an action server object, `server`, that corresponds to the ROS action of the specified name, `actionname` and type, `actiontype`. You must also specify the `ExecuteGoalFcn` property as a function handle callback, `cb`, which handles the goal execution when the client sends a request.

`[server] = rosactionserver(___,"DataFormat","struct")` specifies to use message structures instead of objects, in addition to all input arguments from the previous syntax. For more information, see “Improve Performance of ROS Using Message Structures”.

`server = ros.SimpleActionServer(node, actionname,actiontype, ExecuteGoalFcn=cb)` attaches the created action server to the specified ROS node `node`.

[server] = ros.SimpleActionServer(___ ,DataFormat="struct") uses message structures instead of objects. For more information, see “Improve Performance of ROS Using Message Structures”.

Properties

ActionName — Name of the action

string scalar | character vector

This property is read-only.

Name of the action, specified as a string scalar or character vector.

Example: "/fibonacci"

Data Types: char | string

ActionType — Type of action

string scalar | character vector

This property is read-only.

Type of action, specified as a string scalar or character vector.

Example: "actionlib_tutorials/Fibonacci"

Data Types: char | string

ExecuteGoalFcn — Action callback function

function handle | cell array

Action callback function, 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. To get a predefined framework to customize the callback function, use rosActionServerExecuteGoalFcn.

The action callback function requires at least four input arguments with one output. The first argument, src, is the associated action server object. The second argument, goal, is the goal message sent by the action client. The third argument is the default response message, defaultFeedback. The fourth argument is the default result message, defaultResultMsg. The callback returns a result message, result, based on the input goal message and sends it back to the action client. Use the default response message as a starting point for constructing the request message. The callback also returns success as true if the goal was successfully reached, or as false if the goal was aborted or preempted by another goal. The function header for the callback is:

```
function [result,success] = actionCallback(src,goalMsg,defaultFeedbackMsg,defaultResultMsg)
```

Specify the ExecuteGoalFcn property while creating the action server using the name-value pair as:

```
server = rosactionserver(actionname,actiontype,ExecuteGoalFcn=@actionCallback)
```

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 such a callback is:

```
function [result,success] = actionCallback(src,goalMsg,defaultFeedbackMsg,defaultResultMsg,userDa
```

Specify the `ExecuteGoalFcn` property while creating the action server using the name-value pair as:

```
server = rosactionserver(actionname,actiontype,ExecuteGoalFcn={@actionCallback,userData})
```

`DataFormat` — Message format
`"object"` (default) | `"struct"`

Message format, specified as `"object"` or `"struct"`. You must set this property on creation using the name-value input. For more information, see “Improve Performance of ROS Using Message Structures”.

Object Functions

getFeedbackMessage	Create new action feedback message
isPreemptRequested	Check if a goal has been preempted
sendFeedback	Send feedback to action client during goal execution

Examples

Create a ROS Action Server and Execute a Goal

This example shows how to create a ROS action server, connect an action client to it, receive goal, and execute it.

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6875 seconds.
Initializing ROS master on http://172.30.131.134:51566.
Initializing global node /matlab_global_node_50700 with NodeURI http://bat6234win64:63032/ and Ma
```

Set up an action server for calculating Fibonacci sequence. Use structures for the ROS message data format. Use `fibbonacciExecution` on page 3-64 function as the callback.

```
cb = @fibonacciExecution;
server = rosactionserver("/fibonacci","actionlib_tutorials/Fibonacci",ExecuteGoalFcn=cb,DataForma
```

```
server = 
  SimpleActionServer with properties:

        ActionName: '/fibonacci'
        ActionType: 'actionlib_tutorials/Fibonacci'
    ExecuteGoalFcn: @fibonacciExecution
        DataFormat: 'struct'
```

Create action client and send a goal to the server to calculate the Fibonacci sequence up to 10 terms past the first two terms, `0` and `1`. Display the result sequence.

```
client = rosactionclient("/fibonacci","actionlib_tutorials/Fibonacci",DataFormat="struct");
goal = rosmessage(client);
goal.Order = int32(10);
result = sendGoalAndWait(client,goal);
result.Sequence
```

```
ans = 12x1 int32 column vector

    0
    1
    1
    2
    3
    5
    8
   13
   21
   34
      ⋮
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_50700 with NodeURI http://bat6234win64:63032/ and M
Shutting down ROS master on http://172.30.131.134:51566.
```

Supporting Functions

The callback function `fibbonacciExecution` is executed every time the server receives a goal execution request from the client. This function checks if the goal has been preempted, executes the goal and sends feedback to the client during goal execution.

```
function [result,success] = fibonacciExecution(src,goal,defaultFeedback,defaultResult)

    % Initialize variables
    success = true;
    result = defaultResult;
    feedback = defaultFeedback;
    feedback.Sequence = int32([0 1]);

    for k = 1:goal.Order
        % Check that the client has not canceled or sent a new goal
        if isPreemptRequested(src)
            success = false;
            break
        end

        % Send feedback to the client periodically
        feedback.Sequence(end+1) = feedback.Sequence(end-1) + feedback.Sequence(end);
        sendFeedback(src,feedback)

        % Pause to allow time to complete other callbacks (like client feedback)
        pause(0.2)
    end

    if success
        result.Sequence = feedback.Sequence;
    end

end
```

Create Custom Callback for a ROS Action Server Using the Predefined Callback Framework

This example shows how to create a custom callback for a ROS action server using `rosActionServerExecuteGoalFcn`, which provides a customizable predefined callback framework.

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6975 seconds.
Initializing ROS master on http://172.30.131.134:52921.
Initializing global node /matlab_global_node_29865 with NodeURI http://bat6234win64:57770/ and Ma
```

Set up an action server callback for calculating the Fibonacci sequence using `rosActionServerExecuteGoalFcn`. Specify the custom callback functions for the tasks in the callback framework. All the callback functions use a shared object to store data. For definition of these custom functions, see Supporting Functions on page 3-66.

```
% Store the first two terms 0 and 1 in shared object
fibSequence = int32([0 1]);
% Create the callback
cb = rosActionServerExecuteGoalFcn(IsGoalReachedFcn=@isGoalReached,...
        StepExecutionFcn=@nextFibNumber,...
        CreateFeedbackFcn=@assignUserDataToMessage,...
        CreateSuccessfulResultFcn=@assignUserDataToMessage,...
        StepDelay=0.2,...
        UserData=fibSequence);
```

Use the created custom callback, `cb` and set up an action server for calculating Fibonacci sequence. Use structures for the ROS message data format.

```
server = rosactionserver("/fibonacci","actionlib_tutorials/Fibonacci",ExecuteGoalFcn=cb,DataForma
```

Create action client and send a goal to the server, which calculates the first 10 terms in the Fibonacci sequence. Display the result sequence.

```
client = rosactionclient("/fibonacci","actionlib_tutorials/Fibonacci",DataFormat="struct");
goal = rosmessage(client);
goal.Order = int32(10);
result = sendGoalAndWait(client,goal);
result.Sequence
```

```
ans = 10x1 int32 column vector

    0
    1
    1
    2
    3
    5
    8
   13
   21
   34
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_29865 with NodeURI http://bat6234win64:57770/ and M
Shutting down ROS master on http://172.30.131.134:52921.
```

Supporting Functions

The function `isGoalReached` checks whether the goal is reached. In this case, it checks whether the number of terms in the calculated Fibonacci sequence exceeds the goal from the client.

```
function status = isGoalReached(sharedObj,goal)
    status = numel(sharedObj.UserData) >= goal.Order;
end
```

The function `nextFibNumber` is the step execution function that calculates the next term in the sequence in every iteration towards goal execution.

```
function nextFibNumber(sharedObj,~)
    sharedObj.UserData(end+1) = sharedObj.UserData(end-1) + sharedObj.UserData(end);
end
```

The function `assignUserDataToMessage` assigns the current sequence to the appropriate field in the result message. In this specific case of Fibonacci action, the feedback message also uses the same field, `Sequence` as the result message. Hence, this function can be used for both creating a feedback message and result message to the client.

```
function msg = assignUserDataToMessage(sharedObj,msg)
    msg.Sequence = sharedObj.UserData;
end
```

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Only the syntax that uses message structures with `DataFormat="struct"` option is supported.
- Only one action server per ROS node is supported.
- The `UserData` specified for the callback created using `rosActionServerExecuteGoalFcn` function must be a 1-D array.

See Also

`rosActionServerExecuteGoalFcn` | `rosactionclient` | `getFeedbackMessage` | `isPreemptRequested` | `sendFeedback` | `rosaction`

Topics
"ROS Actions Overview"

rosactionclient

Create ROS action client

Description

Use the `rosactionclient` 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 = rosactionclient(actionname)
client = rosactionclient(actionname,actiontype)
[client,goalMsg] = rosactionclient(___)
[___] = rosactionclient(___,"DataFormat","struct")

client = ros.SimpleActionClient(node,actionname)
client = ros.SimpleActionClient(node,actionname,actiontype)
client = ros.SimpleActionClient(___,"DataFormat","struct")
```

Description

`client = rosactionclient(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 `rosactionclient` 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 = rosactionclient(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] = rosactionclient(___)` 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.

`[___] = rosactionclient(___,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-73.

`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`.

`client = ros.SimpleActionClient( ___ ,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-73.

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.

`DataFormat` — Message format
`"object"` (default) | `"struct"`

Message format, specified as `"object"` or `"struct"`. You must set this property on creation using the name-value input. For more information, see “ROS Message Structures” on page 3-73.

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:

```
rosrun 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 = '192.168.203.133';
rosinit(ipaddress,11311)
```

```
Initializing global node /matlab_global_node_81947 with NodeURI http://192.168.203.1:54283/
```

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 by specifying the action name. Use structures for ROS messages.

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

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. Set the order to an `int32` value of 8.

```
goalMsg.Order = int32(8);
```

Send the goal and wait for its completion. Specify a timeout of 10 seconds to complete the action.

```
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg,10);
```

```
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  actionlib_tutorials/FibonacciResult
       Sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_81947 with NodeURI http://192.168.203.1:54283/
```

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:

```
rosrun 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('192.168.203.133',11311)
```

```
Initializing global node /matlab_global_node_18287 with NodeURI http://192.168.203.1:55284/
```

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

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

```
goalMsg.Order = int32(4);
[resultMsg,resultState] = sendGoalAndWait(actClient,goalMsg)
```

```
resultMsg = struct with fields:
    MessageType: 'actionlib_tutorials/FibonacciResult'
       Sequence: [0 1 1 2 3]


resultState = 
'succeeded'
```

```
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  actionlib_tutorials/FibonacciResult
       Sequence    :  [0, 1, 1, 2, 3]'
```

Send a new goal message without waiting.

```
goalMsg.Order = int32(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_18287 with NodeURI http://192.168.203.1:55284/
```

Version History

Introduced in R2019b

R2021a: ROS Message Structures
Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for `struct` messages.
- `ActionType` argument must be specified.
- Callback functions must be assigned at the time of `rosactionclient` or `ros.SimpleActionClient` object creation, and cannot be changed during run-time.
- For `ros.SimpleActionClient`, `node` input argument must be empty.
- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also

`rosactionserver` | `sendGoal` | `cancelGoal` | `waitForServer` | `rosmessage` | `rosaction`

Topics
“ROS Actions Overview”
“Move a Turtlebot Robot Using ROS Actions”

External Websites
ROS Actions

ros2actionclient

Create ROS 2 action client

Description

Use the `ros2actionclient` object to connect to an action server and request the execution of action goals. You can send multiple goals from an action client, get feedback on the execution progress, and cancel goals at any time.

Creation

Syntax

```
client = ros2actionclient(node,actionname,actiontype)
client = ros2actionclient(node,actionname,actiontype,Name=Value)
[client,goalMsg] = ros2actionclient( ___ )
```

Description

`client = ros2actionclient(node,actionname,actiontype)` creates a client for the ROS 2 action with name `actionname` and type `actiontype`. It attaches the action client to the ROS 2 node specified by the `ros2node` object, `node`. If the action server offering `actionname` is not available, the function does not display an error.

`client = ros2actionclient(node,actionname,actiontype,Name=Value)` sets the rest of the properties based on the additional options specified by one or more `Name=Value` pair arguments.

`[client,goalMsg] = ros2actionclient( ___ )` returns a goal message initialized with default values for the action type of the client, using any of the arguments from the previous syntaxes. You can use this message to modify the goal and send it to the action server.

Properties

`ActionName` — ROS 2 action name
string scalar | character vector

ROS 2 action name, returned as a character vector. The action name must match one of the topics that `ros2 action list` outputs. For more information, see `ros2`.

`ActionType` — Action type for ROS 2 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 `ros2 action type <action_name>`. For more information, see `ros2`.

`IsServerConnected` — Indicates if client is connected to ROS 2 action server
`false` (default) | `true`

This property is read-only.

Indicator of whether the client is connected to a ROS 2 action server, returned as `false` or `true`. Use `waitForserver` to wait until the server is connected when setting up an action client.

`GoalServiceQoS` — Quality of Service (QoS) settings for sending goal
`'History: keeplast, Depth: 10, Reliability: reliable, Durability: volatile'` (default) | structure

Quality of Service (QoS) settings for sending goal to the action server, returned as a character vector. When you set this QoS setting as a name-value argument, you must specify it as a structure with the one or more of these fields:

- `History`
- `Depth`
- `Reliability`
- `Durability`

`ResultServiceQoS` — Quality of Service (QoS) settings for getting result
`'History: keeplast, Depth: 10, Reliability: reliable, Durability: volatile'` (default) | structure

Quality of Service (QoS) settings for getting result from the action server, returned as a character vector. When you set this QoS setting as a name-value argument, you must specify it as a structure with the one or more of these fields:

- `History`
- `Depth`
- `Reliability`
- `Durability`

`CancelServiceQoS` — Quality of Service (QoS) settings for canceling goal
`'History: keeplast, Depth: 10, Reliability: reliable, Durability: volatile'` (default) | structure

Quality of Service (QoS) settings for canceling goals sent to the action server, returned as a character vector. When you set this QoS setting as a name-value argument, you must specify it as a structure with the one or more of these fields:

- `History`
- `Depth`
- `Reliability`
- `Durability`

`FeedbackTopicQoS` — Quality of Service (QoS) settings for receiving feedback
`'History: keeplast, Depth: 10, Reliability: reliable, Durability: volatile'` (default) | structure

Quality of Service (QoS) settings for receiving feedback from the action server, returned as a character vector. When you set this QoS setting as a name-value argument, you must specify it as a structure with the one or more of these fields:

- `History`
- `Depth`
- `Reliability`
- `Durability`

`StatusTopicQoS` — Quality of Service (QoS) settings for receiving goal status
`'History: keeplast, Depth: 10, Reliability: reliable, Durability: volatile'` (default) | structure

Quality of Service (QoS) settings for receiving goal execution status from the action server, returned as a character vector. When you set this QoS setting as a name-value argument, you must specify it as a structure with the one or more of these fields:

- `History`
- `Depth`
- `Reliability`
- `Durability`

Object Functions

ros2message	Create ROS 2 message structures
waitForServer	Wait for ROS 2 action server to be ready for sending goals
sendGoal	Send goal message to ROS 2 action server
getStatus	Get execution status of specific goal sent by ROS 2 action client
getResult	Get result of specific goal associated with goal handle
cancelGoal	Cancel specific goal sent by ROS 2 action client
cancelGoalAndWait	Cancel specific goal sent by ROS 2 action client and wait for cancel response
cancelGoalsBefore	Cancel goals sent by ROS 2 action client before timestamp
cancelGoalsBeforeAndWait	Cancel goals sent by ROS 2 action client before timestamp and wait for cancel response
cancelAllGoals	Cancel all active goals sent by ROS 2 action client
cancelAllGoalsAndWait	Cancel all active goals sent by ROS 2 action client and wait for cancel response

Examples

Set Up ROS 2 Action Client and Execute an Action

This example shows how to create a ROS 2 action client and execute the action. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

List the actions available on the network. The `/fibonacci` action must be on the list.

```
ros2 action list
```

```
/fibonacci
```

Get the action type for the `/fibonacci` action.

```
ros2 action type /fibonacci
```

```
action_tutorials_interfaces/Fibonacci
```

Create a ROS 2 node.

```
node = ros2node("/node_1");
```

Create an action client by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
```

Wait for the action client to connect to the server.

```
status = waitForServer(client)
```

```
status = logical
   1
```

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. Set the order to an `int32` value of 8. If the input requires a 1-D array, set it as a column vector.

```
goalMsg.order = int32(8);
```

Send Goal and Execute Action

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response and the final result using the name-value arguments.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback,ResultFcn=@helperRes
```

Send the goal to the action server using the `sendGoal` function. Specify the callback options. During goal execution, you see outputs from the feedback and result callbacks displayed on the MATLAB® command window.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
```

```
Goal with GoalUUID 3d10ab880f960666fde5666f45f621a accepted by server, waiting for result!
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5  8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Full sequence result for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8  13
```

Get the status of goal execution.

```
exStatus = getStatus(client,goalHandle)
```

```
exStatus = int8
    2
```

Get the result using the action client and goal handle inputs. Display the result. The `getResult` function returns the sequence as a column vector.

```
resultMsg = getResult(client,goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Alternatively, you can only use the goal handle as input to get the result.

```
resultMsg = getResult(goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
    seq = feedbackMsg.partial_sequence;
    disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperResultCallback` defines the callback function to execute when the client receives the result message from the action server.

```
function helperResultCallback(goalHandle,wrappedResultMsg)
    seq = wrappedResultMsg.result.sequence;
    disp(['Full sequence result for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order 8. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);
```

```
Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
```

```
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The callback functions used to send and cancel goals must be specified when you create the `ros2actionclient` object using these name-value arguments.

  - `'SendGoalOptions'` — Specify a cell array of callback options structures for sending goals that you create using the `ros2ActionSendGoalOptions` function.
  - `'CancelFcn'` — Specify the cancel response callback that you use with the `cancelGoal` function.
  - `'CancelAllFcn'` — Specify the cancel response callback that you use with the `cancelAllGoals` function.
  - `'CancelBeforeFcn'` — Specify the cancel response callback that you use with the `cancelGoalsBefore` function.

  Note that these arguments are used for definition of `ros2actionclient` only and must be specified again at the time of usage in the respective function.

  ```
  callbackOpts1 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback,ResultFcn=@glResult);
  callbackOpts2 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback2,ResultFcn=@glResult2);
  [client] = ros2actionClient(node,"my_action","my_action_type",SendGoalOptions={callbackOpts1,c
  goalMsg = ros2message(client);
  goalHandle = sendGoal(client,goalMsg,callbackOpts1);
  cancelGoal(client,goalHandle,CancelFcn=@cancelGoalCB);
  ```
- The syntax with `goalMsg` output argument is not supported.

  ```
  [client,goalMsg] = ros2actionClient(node,"my_action","my_action_type")
  ```
- When you create callback options to send goals using the `ros2ActionSendGoalOptions` function, any additional input arguments to `GoalRespFcn`, `FeedbackFcn` and, `ResultFcn` callbacks are not supported. Only the function signatures with the goal handle and received message as input arguments is supported.

See Also

`sendGoal` | `getResult` | `getStatus` | `cancelGoal` | `waitForServer`

ActionClientGoalHandle

Goal handle object for ROS 2 action client goals

Description

Use `ActionClientGoalHandle` object to inspect and interact with goals sent by ROS 2 action clients. Each goal has its unique goal handle associated with the action client that sent out the goal. You can use the properties of the `ActionClientGoalHandle` object to track the goal execution asynchronously. To get the goal execution result synchronously, use the `getResult` object function.

Creation

To create a `ActionClientGoalHandle` object associated with a goal, use the `sendGoal` function on a `ros2actionclient` object and send a goal from the action client to the action server.

```
goalHandle = sendGoal(client,goalMsg,callbackOptions)
```

Properties

`GoalUUID` — Unique ID for associated action client goal
nonnegative integer

This property is read-only.

Unique ID for the associated action client goal, returned as a nonnegative integer.

Data Types: `uint8`

`Status` — Execution status of associated goal
nonnegative integer

This property is read-only.

Execution status of the associated goal, returned as a nonnegative integer. Each integer denotes a specific status as defined in the `action_msgs/msg/GoalStatus` ROS 2 message definition:

- `0` — Unknown
- `1` — Accepted
- `2` — Executing
- `3` — Canceling
- `4` — Succeeded
- `5` — Canceled
- `6` — Aborted

Data Types: `int8`

`Timestamp` — Timestamp when goal was accepted
`builtin_interfaces/Time` message structure

This property is read-only.

Timestamp when the goal was accepted, returned as a `builtin_interfaces/Time` message structure.

Data Types: `struct`

`FeedbackFcn` — Callback function to execute when feedback response is received
function handle

This property is read-only.

Callback function to execute when a feedback response is received by the action client, returned as a function handle. You can customize this callback using the `ros2ActionSendGoalOptions` function and then specify the custom callback options when you send a goal using the `sendGoal` function.

Data Types: `function_handle`

`ResultFcn` — Callback function to execute when result response is received
function handle

This property is read-only.

Callback function to execute when result response is received, returned as a function handle. You can customize this callback using the `ros2ActionSendGoalOptions` function and then specify the custom callback options when you send a goal using the `sendGoal` function.

Data Types: `function_handle`

Object Functions

getResult	Get result of specific goal associated with goal handle

Examples

Set Up ROS 2 Action Client and Execute an Action

This example shows how to create a ROS 2 action client and execute the action. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

List the actions available on the network. The `/fibonacci` action must be on the list.

```
ros2 action list
```

```
/fibonacci
```

Get the action type for the `/fibonacci` action.

```
ros2 action type /fibonacci
```

```
action_tutorials_interfaces/Fibonacci
```

Create a ROS 2 node.

```
node = ros2node("/node_1");
```

Create an action client by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
```

Wait for the action client to connect to the server.

```
status = waitForServer(client)
```

```
status = logical
   1
```

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. Set the order to an `int32` value of 8. If the input requires a 1-D array, set it as a column vector.

```
goalMsg.order = int32(8);
```

Send Goal and Execute Action

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response and the final result using the name-value arguments.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback,ResultFcn=@helperRes
```

Send the goal to the action server using the `sendGoal` function. Specify the callback options. During goal execution, you see outputs from the feedback and result callbacks displayed on the MATLAB® command window.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
```

```
Goal with GoalUUID 3d10ab880f960666fde5666f45f621a accepted by server, waiting for result!
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5  8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Full sequence result for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8  13
```

Get the status of goal execution.

```
exStatus = getStatus(client,goalHandle)
```

```
exStatus = int8
    2
```

Get the result using the action client and goal handle inputs. Display the result. The `getResult` function returns the sequence as a column vector.

```
resultMsg = getResult(client,goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Alternatively, you can only use the goal handle as input to get the result.

```
resultMsg = getResult(goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
    seq = feedbackMsg.partial_sequence;
    disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperResultCallback` defines the callback function to execute when the client receives the result message from the action server.

```
function helperResultCallback(goalHandle,wrappedResultMsg)
    seq = wrappedResultMsg.result.sequence;
    disp(['Full sequence result for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order 8. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);
```

```
Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
```

```
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`ros2actionclient` | `sendGoal` | `getResult` | `getStatus` | `ros2ActionSendGoalOptions` | `waitForServer` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait` | `cancelAllGoals` | `cancelAllGoalsAndWait`

rosbagreader

Access rosbag log file information

Description

The `rosbagreader` 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 time series of the message properties.

Creation

Syntax

`bagreader = rosbagreader(filepath)`

Description

`bagreader = rosbagreader(filepath)` creates an indexable `rosbagreader` object, `bagreader`, that contains all the messages from the rosbag log file at the input path `filepath`. The `filepath` input argument sets the FilePath property. To access the data, you can call `readMessages` or `timeseries` to extract relevant data.

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.

Data Types: `table`

`AvailableFrames` — List of available coordinate frames
cell array of character vectors

This property is read-only.

List of available coordinate frames, specified 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`

`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

select	Select subset of messages in rosbag
readMessages	Read messages from rosbag
timeseries	Create time series object for selected message properties
canTransform	Verify if transformation is available
getTransform	Retrieve transformation between two coordinate frames

Examples

Create rosbag Selection Using `rosbagreader` Object

Load a rosbag log file and parse out specific messages based on the selected criteria.

Create a `rosbagreader` object of all the messages in the rosbag log file.

```
bagMsgs = rosbagreader("ros_multi_topics.bag")
```

```
bagMsgs = 
  rosbagreader with properties:

           FilePath: 'B:\matlab\toolbox\robotics\robotexamples\ros\data\bags\ros_multi_topics.bag
          StartTime: 201.3400
            EndTime: 321.3400
        NumMessages: 36963
    AvailableTopics: [4x3 table]
    AvailableFrames: {0x1 cell}
        MessageList: [36963x4 table]
```

Select a subset of the messages based on their timestamp and topic.

```
bagMsgs2 = select(bagMsgs,...
    Time=[bagMsgs.StartTime bagMsgs.StartTime + 1],...
    Topic='/odom')
```

```
bagMsgs2 = 
  rosbagreader with properties:

           FilePath: 'B:\matlab\toolbox\robotics\robotexamples\ros\data\bags\ros_multi_topics.bag
          StartTime: 201.3400
            EndTime: 202.3200
        NumMessages: 99
    AvailableTopics: [1x3 table]
    AvailableFrames: {0x1 cell}
        MessageList: [99x4 table]
```

Retrieve the messages in the selection as a cell array.

```
msgs = readMessages(bagMsgs2)
```

```
msgs=99×1 cell array
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
      ⋮
```

Return certain message properties as a time series.

```
ts = timeseries(bagMsgs2,...
    'Pose.Pose.Position.X', ...
    'Twist.Twist.Angular.Y')
```

```
  timeseries

  Timeseries contains duplicate times.

  Common Properties:
            Name: '/odom Properties'
            Time: [99x1 double]
        TimeInfo: tsdata.timemetadata
            Data: [99x2 double]
        DataInfo: tsdata.datametadata
```

Get Transformations from rosbag File Using `rosbagreader` Object

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.

```
bagMsgs = rosbagreader("ros_turtlesim.bag")
```

```
bagMsgs = 
  rosbagreader with properties:

           FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\26\tp4cf343c3\ros-ex81142742\ros_tur
          StartTime: 1.5040e+09
            EndTime: 1.5040e+09
        NumMessages: 6089
    AvailableTopics: [6x3 table]
    AvailableFrames: {2x1 cell}
        MessageList: [6089x4 table]
```

Get a list of available frames.

```
frames = bagMsgs.AvailableFrames
```

```
frames = 2x1 cell
    {'turtle1'}
    {'world'  }
```

Get the latest transformation between two coordinate frames.

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

```
tf = 
  ROS TransformStamped message with properties:

     MessageType: 'geometry_msgs/TransformStamped'
          Header: [1x1 Header]
       Transform: [1x1 Transform]
```

```
    ChildFrameId: 'turtle1'

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

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

Version History

Introduced in R2021b

See Also

`select` | `readMessages` | `timeseries` | `canTransform` | `getTransform`

rosbagwriter

Create and write logs to rosbag log file

Description

Use the `rosbagwriter` object to create a rosbag log file and write logs to the bag file. Each log contains a topic, its corresponding timestamp, and a ROS message.

Note The `rosbagwriter` object locks the created bag file for use, it is necessary to delete and clear the `rosbagwriter` object in order to use the bag file with a reader or perform other operations.

Creation

Syntax

```
bagwriter = rosbagwriter(filepath)
bagwriter = rosbagwriter( ___ ,Name,Value)
```

Description

`bagwriter = rosbagwriter(filepath)` creates a rosbag log file in the location specified by `path` and returns the corresponding `rosbagwriter` object.

If you do not specify the name of the bag file in the `filepath`, the object assigns the current timestamp as the file name. If the folders you specify in `filepath` are not present in the directory, the object creates them and places the bag file accordingly. The `filepath` input argument sets the FilePath property.

`bagwriter = rosbagwriter( ___ ,Name,Value)` sets the Compression and ChunkSize properties using name-value arguments. Use this syntax with the input argument in the previous syntax.

Properties

`FilePath` — Path to rosbag file
character vector

This property is read-only.

Path to the rosbag file, specified as a character vector.

Data Types: `char`

`StartTime` — Timestamp of first message written to bag file
scalar

This property is read-only.

Timestamp of the first message written to the bag file, specified as a scalar in seconds.

Data Types: `double`

`EndTime` — Timestamp of last message written to bag file
scalar

This property is read-only.

Timestamp of the last message written to the bag file, specified as a scalar in seconds.

Data Types: `double`

`NumMessages` — Number of messages written to bag file
scalar

This property is read-only.

Number of messages written to the bag file, specified as a scalar.

Data Types: `double`

`Compression` — Compression format of message chunks
`"uncompressed"` (default) | `"bz2"` | `"lz4"`

Compression format of the message chunks, specified as `"bz2"`, `"lz4"`, or `"uncompressed"`.

Example: `"Compression","lz4"`

Data Types: `char` | `string`

`ChunkSize` — Size of each message chunk
`786432` (default) | nonzero positive integer

Size of each message chunk, specified as a nonzero positive integer in bytes. The value specify the buffer within the bag file object. Reducing this value results in more writes to disk.

Example: `"ChunkSize",819200`

Data Types: `double`

`FileSize` — Current bag file size
nonnegative integer

This property is read-only.

Current bag file size, specified as a nonnegative integer in bytes.

Data Types: `double`

Object Functions

write	Write logs to rosbag log file
delete	Remove rosbag writer object from memory

Examples

Write Log to rosbag File Using `rosbagwriter` Object

Retrieve all the information from the rosbag log file.

```
rosbag('info','path_record.bag')
```

```
Path:     C:\TEMP\Bdoc23a_2213998_3568\ib570499\10\tpf8d9c23d\ros-ex73035957\path_record.bag
Version:  2.0
Duration: 10.5s
Start:    Jul 05 2021 08:09:52.86 (1625486992.86)
End:      Jul 05 2021 08:10:03.40 (1625487003.40)
Size:     13.3 KB
Messages: 102
Types:    geometry_msgs/Point [4a842b65f413084dc2b10fb484ea7f17]
Topics:   /circle  51 msgs  : geometry_msgs/Point
          /line    51 msgs  : geometry_msgs/Point
```

Create a `rosbagreader` object of all the messages in the rosbag log file.

```
reader = rosbagreader('path_record.bag');
```

Select all the messages related to the topic `'/circle'`.

```
bagSelCircle = select(reader,'Topic','/circle');
```

Retrieve the list of timestamps from the topic.

```
timeStamps = bagSelCircle.MessageList.Time;
```

Retrieve the messages in the selection as a cell array.

```
messages = readMessages(bagSelCircle);
```

Create a `rosbagwriter` object to write the messages to a new rosbag file.

```
circleWriter = rosbagwriter('circular_path_record.bag');
```

Write all the messages related to the topic `'/circle'` to the new rosbag file.

```
write(circleWriter,'/circle',timeStamps,messages);
```

Remove the `rosbagwriter` object from memory and clear the associated object.

```
delete(circleWriter)
clear circleWriter
```

Retrieve all the information from the new rosbag log file.

```
rosbag('info','circular_path_record.bag')
```

```
Path:     C:\TEMP\Bdoc23a_2213998_3568\ib570499\10\tpf8d9c23d\ros-ex73035957\circular_path_recor
Version:  2.0
Duration: 10.4s
Start:    Jul 05 2021 08:09:52.86 (1625486992.86)
End:      Jul 05 2021 08:10:03.29 (1625487003.29)
Size:     8.8 KB
Messages: 51
Types:    geometry_msgs/Point [4a842b65f413084dc2b10fb484ea7f17]
Topics:   /circle  51 msgs  : geometry_msgs/Point
```

Load the new rosbag log file.

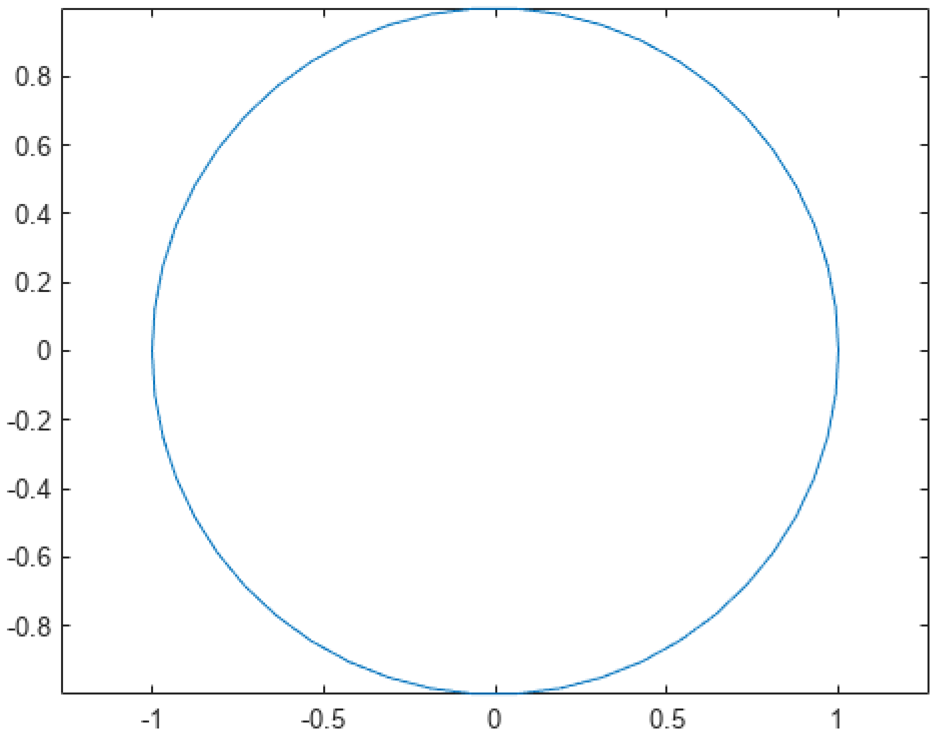
```
readerCircle = rosbagreader('circular_path_record.bag');
```

Create a time series for the coordinates.

```
tsCircle = timeseries(readerCircle,'X','Y');
```

Plot the coordinates.

```
plot(tsCircle.Data(:,1),tsCircle.Data(:,2))
axis equal
```



Create rosbag File Using `rosbagwriter` Object

Create a `rosbagwriter` object and a rosbag file in the current working directory. Specify the compression format of the message chunks and the size of each message chunk.

```
bagwriter = rosbagwriter("bagfile.bag", ...
    "Compression","lz4",...
    "ChunkSize",1500)
```

```
bagwriter = 
  rosbagwriter with properties:

       FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex26181333\bagfile.bag
      StartTime: 0
        EndTime: 0
    NumMessages: 0
    Compression: 'lz4'
      ChunkSize: 1500
       FileSize: 4117
```

Start node and connect to ROS master.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.1985 seconds.
Initializing ROS master on http://172.30.131.134:54539.
Initializing global node /matlab_global_node_21996 with NodeURI http://bat6234win64:64034/ and Ma
```

Write a single log to the rosbag file.

```
timeStamp = rostime("now");
rosMessage = rosmessage("nav_msgs/Odometry");
write(bagwriter,"/odom",timeStamp,rosMessage);
bagwriter
```

```
bagwriter = 
  rosbagwriter with properties:

       FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex26181333\bagfile.bag
      StartTime: 1.6779e+09
        EndTime: 1.6779e+09
    NumMessages: 1
    Compression: 'lz4'
      ChunkSize: 1500
       FileSize: 4172
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_21996 with NodeURI http://bat6234win64:64034/ and M
Shutting down ROS master on http://172.30.131.134:54539.
```

Remove rosbag writer object from memory and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Create a `rosbagreader` object and load all the messages in the rosbag log file. Verify the recently written log.

```
bagreader = rosbagreader('bagfile.bag')
```

```
bagreader = 
  rosbagreader with properties:

           FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex26181333\bagfile
          StartTime: 1.6779e+09
            EndTime: 1.6779e+09
        NumMessages: 1
    AvailableTopics: [1x3 table]
    AvailableFrames: {0x1 cell}
```

```
        MessageList: [1x4 table]
```

```
bagreader.AvailableTopics
```

```
ans=1×3 table
             NumMessages      MessageType            MessageDefinition
             ___________    __________________    ______________________________

    /odom         1         nav_msgs/Odometry    {'std_msgs/Header Header...'}
```

Version History

Introduced in R2021b

See Also

Objects
`rosbagreader`

Functions
`write` | `delete`

ros2bagwriter

Create and write logs to ROS 2 bag log file

Description

Use the `ros2bagwriter` object to create a ROS 2 bag log file (`.db3`) in a folder that you specify. Use the `write` function to write logs to the ROS 2 bag file. Each log contains a topic, its corresponding timestamp, and a ROS 2 message. After writing the logs to the ROS 2 bag file, call the `delete` function to close the opened ROS 2 bag file, create the `metadata.yaml` file, and remove the object from memory.

Note The `ros2bagwriter` object locks the created ROS 2 bag file. Delete and clear the `ros2bagwriter` object to use the ROS 2 bag file.

Creation

Syntax

```
bagwriter = ros2bagwriter(path)
bagwriter = ros2bagwriter(path,Name=Value)
```

Description

`bagwriter = ros2bagwriter(path)` creates a ROS 2 bag file in the location specified by `path` and returns its corresponding `ros2bagwriter` object. Use the object to write records into the ROS 2 bag file. Use the `path` input argument to set the Path property.

The name of the ROS 2 bag file is the name of the folder containing it. If the folders specified in `path` are not in the directory, the object creates them and places the ROS 2 bag file accordingly.

`bagwriter = ros2bagwriter(path,Name=Value)` sets the CacheSize property using a name-value argument.

Properties

`Path` — Path to ROS 2 bag folder
string scalar | character vector

Note This property becomes a read-only after creation of the object.

Path to the ROS 2 bag folder, specified as a string scalar or character vector.

Data Types: `char` | `string`

`StartTime` — Earliest timestamp of messages written to ROS 2 bag file
nonnegative numeric scalar

This property is read-only.

Earliest timestamp of the messages written to the ROS 2 bag file, specified as a nonnegative numeric scalar in seconds.

Data Types: `single` | `double`

`EndTime` — Latest timestamp of messages written to ROS 2 bag file
nonnegative numeric scalar

This property is read-only.

Latest timestamp of the messages written to the ROS 2 bag file, specified as a nonnegative numeric scalar in seconds.

Data Types: `single` | `double`

`NumMessages` — Number of messages written to ROS 2 bag file
nonnegative numeric scalar

This property is read-only.

Number of messages written to the ROS 2 bag file, specified as a nonnegative numeric scalar.

Data Types: `single` | `double`

`CacheSize` — Size of cache for writing messages to ROS 2 bag file
`104857600` (default) | nonnegative integer

This property is read-only.

Size of the cache for writing messages to the ROS 2 bag file, specified as a positive integer in bytes. This value specifies the total size of the messages, that the buffer of the cache holds in the object. If you reduce this value, the object writes more messages to the disk, which consumes more time and decreases performance of the drive.

Data Types: `uint64`

`StorageFormat` — Storage format of ROS 2 bag file
`'sqlite3'`

This property is read-only.

Storage format of the ROS 2 bag file, specified as `'sqlite3'`.

Data Types: `char` | `string`

`SerializationFormat` — Serialization format of messages in ROS 2 bag file
`'cdr'`

This property is read-only.

Serialization format of messages in the ROS 2 bag file, specified as `'cdr'`. This value is the default binary serialization format used by Data Distribution Service (DDS), which is the default middleware of ROS 2.

Data Types: `char` | `string`

Object Functions

write	Write logs to ROS 2 bag log file
delete	Remove ros2bagwriter object from memory

Examples

Write Log Using `ros2bagwriter` Object by Reading Messages from ROS 2 Bag File

Extract the zip file that contains the ROS 2 bag log file and specify the full path to the log folder.

```
unzip('ros2_netwrk_bag.zip');
folderPath = fullfile(pwd,'ros2_netwrk_bag');
```

Get all the information from the ROS 2 bag log file.

```
bag2info = ros2("bag","info",folderPath);
```

Create a `ros2bagreader` object that contains all messages in the log file.

```
bag = ros2bagreader(folderPath);
bag.AvailableTopics
```

```
ans=4×3 table
                NumMessages          MessageType
                ___________    _____________________    ________________________________________

    /clock       1.607e+05     rosgraph_msgs/Clock      {'%...'
    /cmd_vel             3     geometry_msgs/Twist      {'...'
    /odom             5275     nav_msgs/Odometry        {'% The pose in this message should be sp
    /scan              892     sensor_msgs/LaserScan    {'%...'
```

Select a subset of the messages, by applying filters to the topic and timestamp.

```
start = bag.StartTime;
odomBagSel = select(bag,"Time",[start start + 30],"Topic","/odom")
```

```
odomBagSel = 
  ros2bagreader with properties:

           FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\10\tpf8d9c23d\ros-ex95368813\ros2_ne
          StartTime: 1.6020e+09
            EndTime: 1.6020e+09
    AvailableTopics: [1x3 table]
        MessageList: [801x3 table]
        NumMessages: 801
```

Get the messages in the selection.

```
odomMsgs = readMessages(odomBagSel);
```

Retrieve the list of timestamps from the topic.

```
timestamps = odomBagSel.MessageList.Time;
```

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder.

```
bagwriter = ros2bagwriter("myRos2bag");
```

Write the messages related to the topic '`/odom`' to the ROS 2 bag file.

```
write(bagwriter,"/odom",timestamps,odomMsgs)
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Load the new ROS 2 bag log file.

```
bagOdom = ros2bagreader("myRos2bag");
```

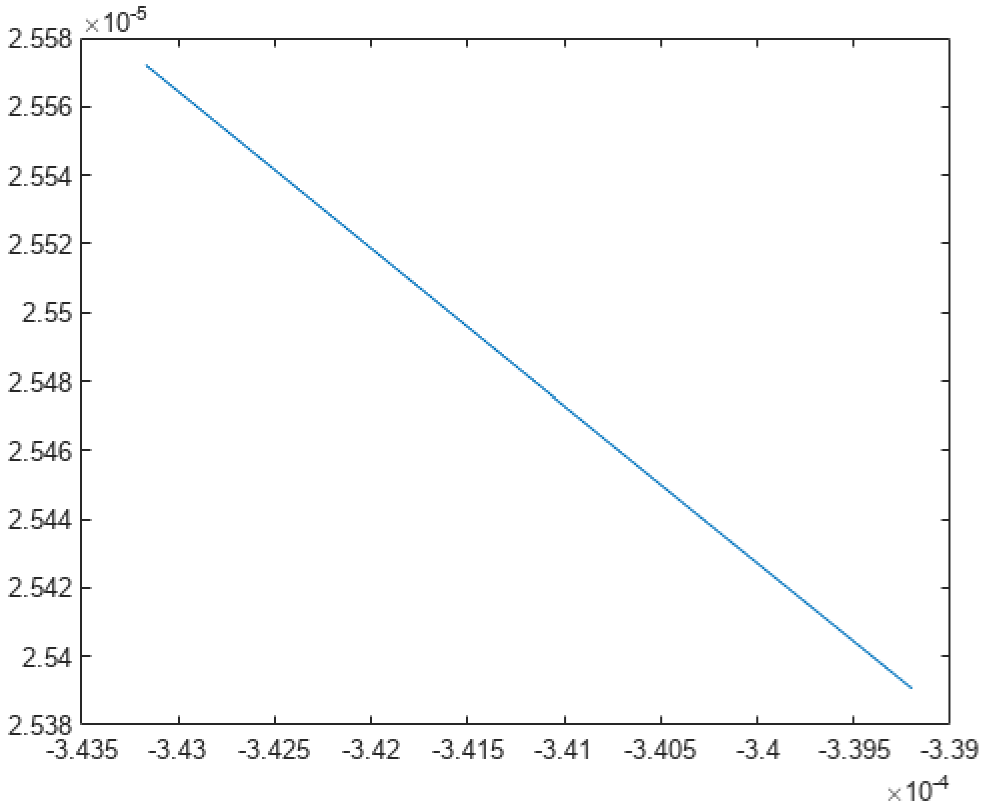
Retrieve messages from the ROS 2 bag log file.

```
msgs = readMessages(bagOdom);
```

Plot the coordinates for the messages in the ROS 2 bag log file.

Remove the `myRos2bag` file and the `ros2_netwrk_bag` file from memory to run the example again.

```
plot(cellfun(@(msg) msg.pose.pose.position.x,msgs),cellfun(@(msg) msg.twist.twist.angular.z,msgs
```



Create Single Log and Write to ROS 2 Bag File

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder.

```
bagwriter = ros2bagwriter("myRos2bag");
```

Write a single log to the ROS 2 bag file.

```
topic = "/odom";
message = ros2message("nav_msgs/Odometry");
timestamp = ros2time(1.6170e+09);
write(bagwriter,topic,timestamp,message)
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Create Multiple Logs and Write to ROS 2 Bag File

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder. Specify the cache size for each message.

```
bagwriter = ros2bagwriter("bag_files/my_bag_file",CacheSize=1500);
```

Write multiple logs to the ROS 2 bag file.

```
message1 = ros2message("nav_msgs/Odometry");
message2 = ros2message("geometry_msgs/Twist");
message3 = ros2message("sensor_msgs/Image");
write(bagwriter, ...
      ["/odom","cmd_vel","/camera/rgb/image_raw"], ...
      {ros2time(1.6160e+09),ros2time(1.6170e+09),ros2time(1.6180e+09)}, ...
      {message1,message2,message3})
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Create Multiple Logs for Same Topic and Write to ROS 2 Bag File

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder.

```
bagwriter = ros2bagwriter("myBag");
```

Write multiple logs for the same topic to the ROS 2 bag file.

```
pointMsg1 = ros2message("geometry_msgs/Point");
pointMsg2 = pointMsg1;
pointMsg3 = pointMsg1;
pointMsg1.x = 1;
pointMsg2.x = 2;
```

```
pointMsg3.x = 3;
write(bagwriter, ...
      "/point", ...
      {1.6190e+09, 1.6200e+09,1.6210e+09}, ...
      {pointMsg1,pointMsg2,pointMsg3})
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Version History

Introduced in R2022b

See Also

Functions

`write` | `delete`

Topics

"Write Log to rosbag File Using rosbagwriter Object" on page 3-97

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	`int32`
boolean	`logical`
double	`double`
string	character vector (`char`)
list	cell array (`cell`)
dictionary	structure (`struct`)

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;
```

```
Launching ROS Core...
..Done in 2.1617 seconds.
```

```
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=1×2 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
```

```
Launching ROS Core...
..Done in 2.7008 seconds.
Initializing ROS master on http://172.30.131.134:54064.
Initializing global node /matlab_global_node_04167 with NodeURI http://bat6234win64:65339/ and Ma
```

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 parameter 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_04167 with NodeURI http://bat6234win64:65339/ and N
Shutting down ROS master on http://172.30.131.134:54064.
```

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Code generation is supported for global node but not for `Node` since a node cannot create another node in code generation.

See Also

`rosparam` | `get` | `has` | `search` | `set` | `del`

Topics

"Access the ROS Parameter Server"

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.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see “ROS Message Structures” on page 3-116.

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,msg] = rospublisher(___,"DataFormat","struct")

pub = ros.Publisher(node,topicname)
pub = ros.Publisher(node,topicname,type)
pub = ros.Publisher( ___ ,"IsLatching",value)
[pub,msg] = ros.Publisher( ___ ,"DataFormat","struct")
```

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,msg] = rospublisher( ___ ,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-116

`pub = ros.Publisher(node,topicname)` creates a publisher for a topic with name, `topicname`. `node` is the `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.

`[pub,msg] = ros.Publisher( ___ ,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-116

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`

`DataFormat` — Message format
`"object"` (default) | `"struct"`

Message format, specified as `"object"` or `"struct"`. You must set this property on creation using the name-value input. For more information, see “ROS Message Structures” on page 3-116.

Object Functions

send	Publish ROS message to topic
rosmessage	Create ROS messages

Examples

Create ROS Publisher and Send Data

Start ROS master.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.705 seconds.
```

```
Initializing ROS master on http://172.30.131.134:53000.
Initializing global node /matlab_global_node_37957 with NodeURI http://bat6234win64:63529/ and Ma
```

Create publisher for the `/chatter` topic with the `std_msgs/String` message type. Set the `"DataFormat"` name-value argument to structure ROS messages.

```
chatpub = rospublisher("/chatter","std_msgs/String","DataFormat","struct");
```

Create a message to send. Specify the `Data` property with a character vector.

```
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_37957 with NodeURI http://bat6234win64:63529/ and M
Shutting down ROS master on http://172.30.131.134:53000.
```

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
```

```
Launching ROS Core...
..Done in 2.189 seconds.
Initializing ROS master on http://172.30.131.134:55162.
Initializing global node /matlab_global_node_09723 with NodeURI http://bat6234win64:63941/ and Ma
```

Create a publisher and view its properties.

```
pub = rospublisher('/chatter','std_msgs/String','DataFormat','struct');
```

```
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','DataFormat','struct');
pause(1)
```

```
subCount = pub.NumSubscribers
subCount = 1
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_09723 with NodeURI http://bat6234win64:63941/ and M
Shutting down ROS master on http://172.30.131.134:55162.
```

Use ROS Publisher Object

Create a `Publisher` object using the class constructor.

Start the ROS core.

```
core = ros.Core;
```

```
Launching ROS Core...
..Done in 2.3196 seconds.
```

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','DataFormat','struct');
msg = rosmessage(pub);
msg.Data = 'robot1';
send(pub,msg);
```

Clear the publisher and ROS node. Shut down the ROS master.

```
clear('pub','node')
clear('master')
```

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as "struct" for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for `struct` messages.
- `MessageType` argument must be specified.
- Introspection syntax `rospublisher(topicname,message)` is not supported.

See Also

Functions

`send` | `rosmessage`

Topics

“Exchange Data with ROS Publishers and Subscribers”

ros2bagreader

Access ROS 2 bag log file information

Description

The `ros2bagreader` object is an index of the messages within a ROS 2 bag file. You can use it to extract message data from a ROS 2 bag file or select messages based on specific criteria.

Creation

Syntax

`bagreader = ros2bagreader(folderpath)`

Description

`bagreader = ros2bagreader(folderpath)` creates an indexable `ros2bagreader` object, `bagreader`, that contains all the messages from the ROS 2 bag file at the input path `filepath`. The `folderpath` input sets the value of the FilePathproperty.

ROS 2 bag files are used for storing message data. Their primary use is in the logging of messages transmitted over a ROS 2 network. The resulting bag file can be used for offline analysis, visualization, and storage. MATLAB provides functionality for reading existing bag files.

Note If the ROS 2 bag log file contains custom messages, create custom messages for MATLAB using `ros2genmsg` function before creating the `ros2bagreader` object.

Properties

`FilePath` — Absolute path to ROS 2 bag file
character vector

This property is read-only.

Absolute path to the ROS 2 bag files, specified as a character vector.

Data Types: `char`

`StartTime` — Timestamp of first message
scalar

This property is read-only.

Timestamp of the first message, specified as a scalar in seconds.

Data Types: `double`

`EndTime` — Timestamp of last message
scalar

This property is read-only.

Timestamp of the last message, specified as a scalar in seconds.

Data Types: `double`

`NumMessages` — Number of messages
scalar

This property is read-only.

Number of messages, specified as a scalar.

Data Types: `double`

`AvailableTopics` — Table of available topics
table

This property is read-only.

Table of available topics, specified as a table. Each row in the table lists one topic, the number of messages for this topic, the message type, and the message definition.

Data Types: `table`

`MessageList` — List of messages
table

This property is read-only.

List of messages, specified as a table. Each row in the table lists one message.

Data Types: `table`

Object Functions

readMessages	Read messages from ros2bagreader object
select	Select subset of messages in ros2bagreader

Examples

Read Messages from ROS 2 Bag Log File

Extract the zip file that contains the ROS 2 bag log file and specify the full path to the log folder.

```
unzip('ros2_netwrk_bag.zip');
folderPath = fullfile(pwd,'ros2_netwrk_bag');
```

Create a `ros2bagreader` object that contains all messages in the log file.

```
bag = ros2bagreader(folderPath);
```

Get information on the contents of the `ros2bagreader` object.

```
baginfo = ros2("bag","info",folderPath)
```

```
baginfo = struct with fields:
         Path: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\26\tp4cf343c3\ros-ex96596996\ros2_netwrk_ba
      Version: '1'
    StorageId: 'sqlite3'
     Duration: 207.9020
        Start: [1x1 struct]
          End: [1x1 struct]
         Size: 16839538
     Messages: 166867
        Types: [4x1 struct]
       Topics: [4x1 struct]
```

Get all the messages in the `ros2bagreader` object.

```
msgs = readMessages(bag);
```

Select a subset of the messages, filtered by topic.

```
bagSel = select(bag,"Topic","/odom");
```

Get the messages in the selection.

```
msgsFiltered = readMessages(bagSel);
```

Version History

Introduced in R2021a

R2022b: `ros2bagreader` was renamed

Behavior change in future release

The `ros2bagreader` object was renamed from `ros2bag`. Use `ros2bagreader` when creating the object.

R2022a: `folderpath` Input

Behavior changed in R2022a

- The `ros2bagreader` object in Foxy can accept file name of the ROS 2 bag log file (`.db3`) as the `folderpath` input, when there is no `metadata.yaml` file.
- If there is `metadata.yaml` file along with the `.db3` file in a folder, it accepts the folder name as the `folderpath` input.

R2022a: ROS 2 Bag Log File Version

Behavior changed in R2022a

The `ros2bagreader` object in Foxy can accept different versions of bag file from version 1 to 4.

R2022a: Empty Messages

Behavior changed in R2022a

The `ros2bagreader` object in Foxy discards empty messages recorded on a topic. Whereas in Dashing the `ros2bagreader` object accepts `/rosout` and `/param_events` topics.

See Also

Functions
`readMessages` | `select`

ros2param

Create object to access parameters from ROS 2 nodes

Description

Create a `ros2param` object and use its object functions to interact with the parameters associated with any node on the ROS 2 network. You can get, set, list and search for parameters of the specified ROS 2 node.

Creation

Syntax

```
paramObj = ros2param(nodeName)
paramObj = ros2param(nodeName,DomainID=ID)
```

Description

`paramObj = ros2param(nodeName)` returns a `ros2param` object `paramObj` which you can use to interact with the parameters associated with the specified ROS 2 node, `nodeName`.

`paramObj = ros2param(nodeName,DomainID=ID)` specifies a Domain identification of the ROS 2 network to connect to using name-value argument `DomainID`.

Input Arguments

nodeName — 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 enforced. When creating a new node, use `ros2` function to list existing nodes.

Object Functions

get	Get value of parameter in associated ROS 2 node
set	Set value of parameter in associated ROS 2 node
list	List all parameters in associated ROS 2 node
has	Check if parameter exists in ROS 2 node
search	Search for parameter names in ROS 2 node

Examples

Interact with Parameters of ROS 2 Node

Create a ROS 2 node with parameters.

```
nodeParams.my_double = 2.0;
nodeParams.my_namespace.my_int = int64(1);
nodeParams.my_double_array = [1.1 2.2 3.3];
nodeParams.my_string = "Keyparams";
node1 = ros2node("/node1",Parameters=nodeParams);
```

Create a `ros2param` object to interact with the parameters of the ROS 2 node, `/node1`.

```
paramObj = ros2param("/node1");
```

Use the `set` function to change the value of the parameter `my_string`.

```
set(paramObj,"my_string","Newparams");
```

Use the `get` function to obtain the new value of `my_string`.

```
stringVal = get(paramObj,"my_string")
```

```
stringVal =
'Newparams'
```

Use the `has` function to check if the parameter `my_char` exists in the ROS 2 node, `/node1`.

```
flag = has(paramObj,"my_char")
```

```
flag = logical
   0
```

Use the `search` function to search for names of all the parameters that contain the string "`my_d`". Obtain the values of the matching parameters.

```
[pNames,pVals] = search(paramObj,"my_d")
```

```
pNames = 2x1 cell
    {'my_double'      }
    {'my_double_array'}


pVals=2×1 cell array
    {[                  2]}
    {[1.1000 2.2000 3.3000]}
```

Use the `list` function to list the names of all parameters in the ROS 2 node.

```
pList = list(paramObj)
```

```
pList = 5x1 cell
    {'my_double'          }
    {'my_double_array'    }
    {'my_namespace.my_int'}
    {'my_string'          }
    {'use_sim_time'       }
```

Version History

Introduced in R2022b

See Also

`get` | `set` | `ros2node`

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] = ros2publisher(___)
```

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] = ros2publisher(___)` 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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`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('node1_empty_ros2_msg');
```

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:
    MessageType: 'std_msgs/String'
           data: ''
```

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `MessageType` argument must be specified.
- ros2publisher(___) does not return a message, `msg`, that you can send with the publisher, `pub`. You can get the ROS message using the `ros2message` function.

See Also

`ros2subscriber` | `ros2message` | `send`

Topics
"Manage Quality of Service Policies in ROS 2"

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(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 pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

`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

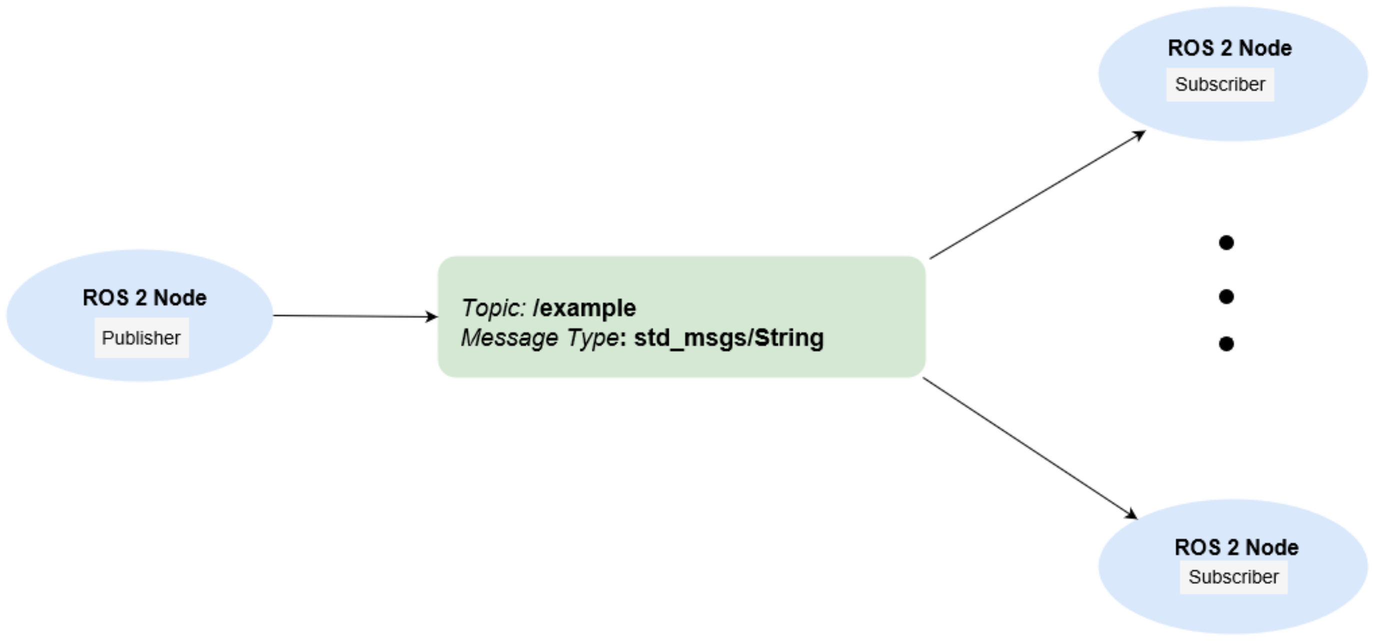
Exchange Data with ROS 2 Publishers and Subscribers

This example shows how to publish and subscribe to topics in a ROS 2 network.

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.



Besides how to publish and subscribe to topics in a ROS 2 network, this example 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
/rosout
/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(5)
laserSub = ros2subscriber(detectNode,"/scan");
pause(5)
```

Use `receive` to wait for a new message. Specify a timeout of 10 seconds. The output `scanData` contains the received message data. `status` indicates whether a message was received successfully and `statustext` provides additional information about the `status`.

```
[scanData,status,statustext] = 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");
pause(5)
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)
```

```
       0.00235920447111606       -0.0201184589892978        0.0203969078651195
```

```
disp(orient)
```

```
       -0.0118389124011118       0.00676849978014866        0.0387860955311228
```

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 “Create Callbacks for Graphics Objects” 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
/rosout
/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"
- "Generate ROS 2 Custom Messages in MATLAB"

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `MessageType` argument must be specified.
- Callback functions must be assigned at the time of `ros2subscriber` object creation, and cannot be changed during run-time.

See Also

`ros2publisher` | `ros2node`

Topics

"Manage Quality of Service Policies in ROS 2"
"ROS Custom Message Support"

rosrate

Execute loop at fixed frequency

Description

The `rosrate` 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 `rosrate` 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 = rosrate(desiredRate)
rate = ros.Rate(node,desiredRate)
```

Description

`rate = rosrate(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.

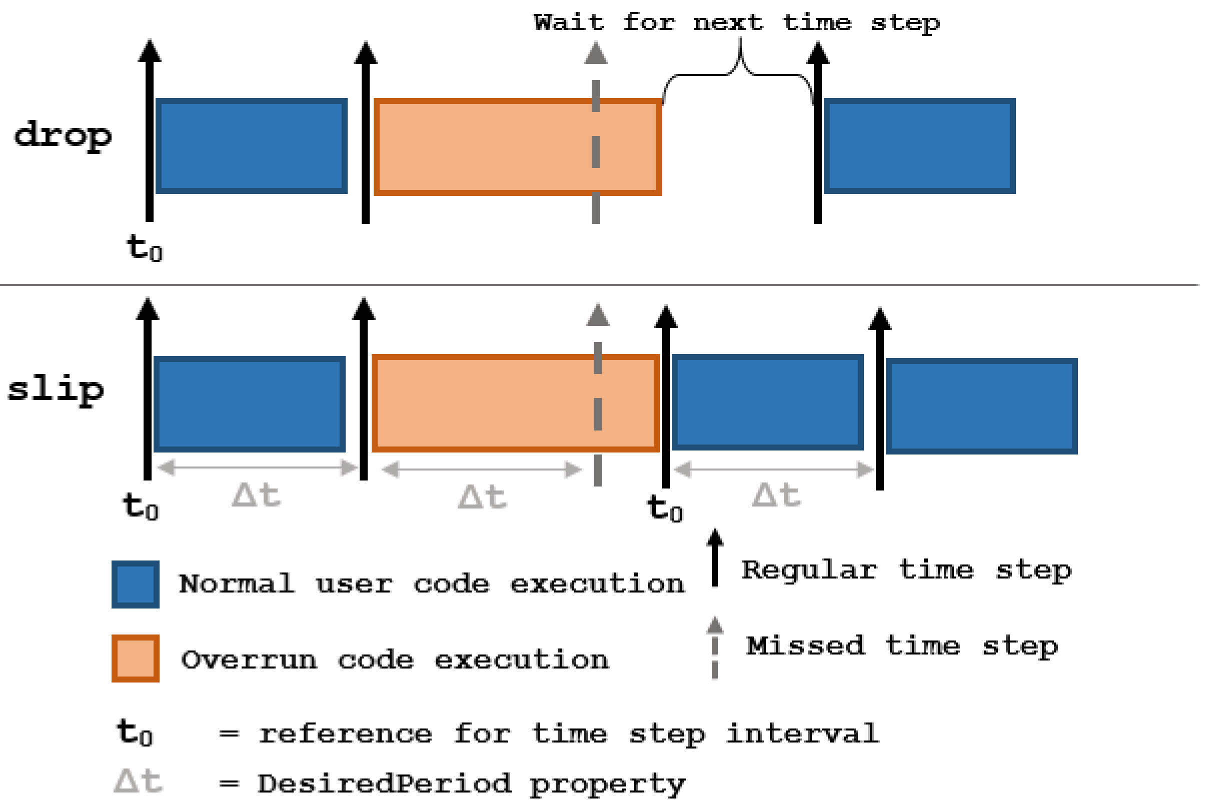
`LastPeriod` — Elapsed time between last two calls to `waitfor`
`NaN` (default) | scalar

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 the global node.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.7212 seconds.
Initializing ROS master on http://172.30.131.134:53864.
Initializing global node /matlab_global_node_84260 with NodeURI http://bat6234win64:63557/ and Ma
```

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: 0.006325
Iteration: 2 - Time Elapsed: 1.011267
Iteration: 3 - Time Elapsed: 2.009054
Iteration: 4 - Time Elapsed: 3.000561
Iteration: 5 - Time Elapsed: 4.000956
Iteration: 6 - Time Elapsed: 5.010741
Iteration: 7 - Time Elapsed: 6.000135
Iteration: 8 - Time Elapsed: 7.000384
Iteration: 9 - Time Elapsed: 8.000150
Iteration: 10 - Time Elapsed: 9.003515
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_84260 with NodeURI http://bat6234win64:63557/ and M
Shutting down ROS master on http://172.30.131.134:53864.
```

Run Loop At Fixed Rate Using ROS Time

Initialize the ROS master and node.

```
rosinit
```

```
Launching ROS Core...
....Done in 4.1733 seconds.
Initializing ROS master on http://192.168.88.1:51279.
Initializing global node /matlab_global_node_86106 with NodeURI http://ah-avijayar:50550/
```

```
node = ros.Node('/testTime');
```

```
Using Master URI http://localhost:51279 from the global node to connect to the ROS master.
```

Create a `ros.Rate` object running at 20 Hz.

```
r = 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_86106 with NodeURI http://ah-avijayar:50550/
Shutting down ROS master on http://192.168.88.1:51279.
```

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- `statistics` object function is not supported.

See Also

`rateControl` | `waitfor`

Topics

“Execute Code Based on ROS Time”

ros2rate

Execute loop at fixed frequency

Description

The `ros2rate` object allows you to execute a loop at a fixed frequency. It uses the ROS 2 node as a source for time information. Therefore, it can use the ROS 2 simulation time or wall clock time (see the `IsSimulationTime` property).

The performance of the `ros2rate` object, and the ability to maintain the `DesiredRate` value depend on the publishing of the clock information in ROS 2 network. Because the ros2rate object relies on the `pause` function, disabling `pause` will result in inaccurate execution.

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 when executing MATLAB code. To improve performance and execution speeds, use code generation.

Creation

`rate = ros2rate(node,desiredRate)`

Description

`rate = ros2rate(node,desiredRate)` creates a `ros2rate` object, `rate`, that enables you to execute a loop at a fixed frequency, `desiredRate`. The object uses the time source of the specified ROS 2 node object, `node`.

Properties

`IsSimulationTime` — Loop execution uses simulation or wall clock time
`true` or `1` | `false` or `0`

Loop execution uses simulation or wall clock time, specified as `1 (true)` or `0 (false)`. If `true`, the `ros2rate` object uses the ROS simulation time to regulate the rate of loop execution.

`DesiredRate` — Desired execution rate
positive scalar

Desired execution rate of the loop, specified as a scalar in Hz. When using `waitfor`, the loop operates every `DesiredRate` seconds, unless the loop takes longer than that to execute. It then begins the next loop based on the specified `OverrunAction`.

`DesiredPeriod` — Desired time period between executions
positive scalar

Desired time period between executions, specified as a positive scalar in seconds. This property is equal to the inverse of `DesiredRate`.

`TotalElapsedTime` — Elapsed time since construction or reset
positive scalar

Elapsed time since construction or reset, specified as a positive scalar in seconds.

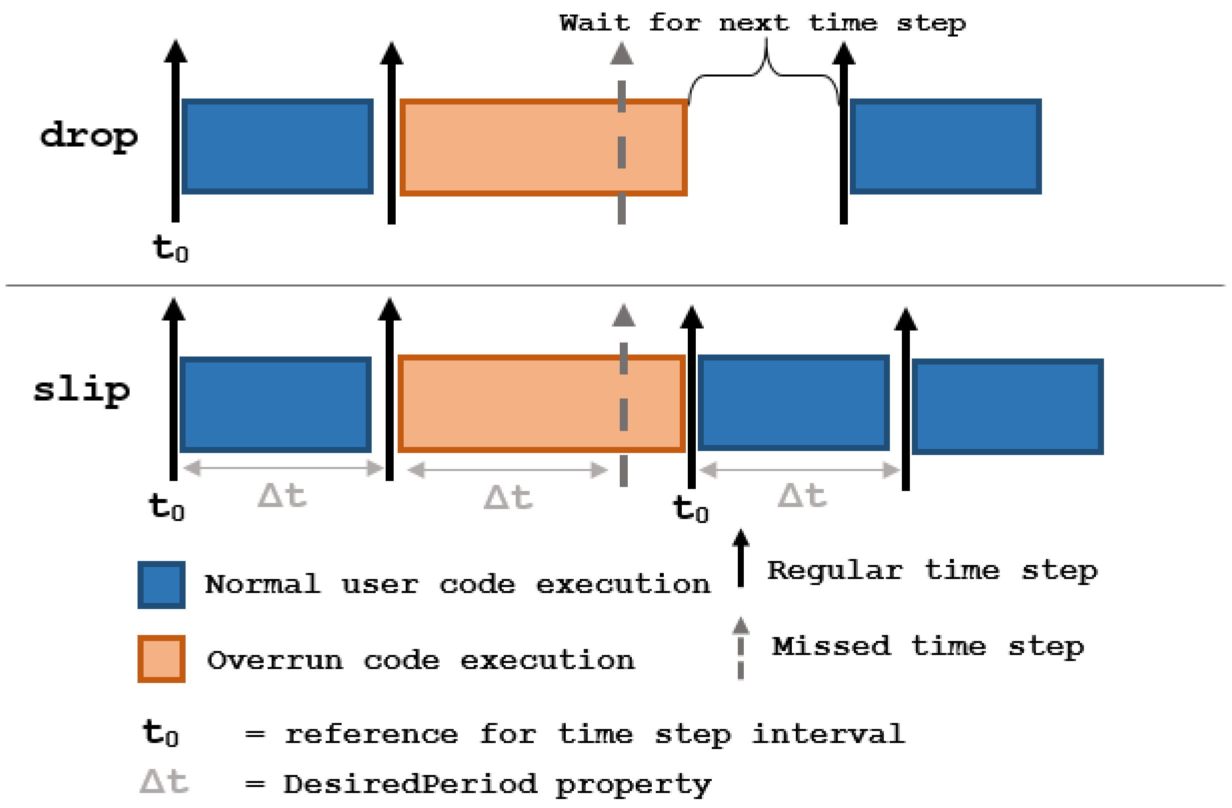
`LastPeriod` — Elapsed time between last two calls to `waitfor`
`NaN` (default) | scalar

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.

`OverrunAction` — Method for handling overruns
`'slip'` (default) | `'drop'`

Method for handling overruns, specified as one of these character vectors:

- `'drop'` — Executes the next iteration of the loop at the next time step equal to a multiple of `DesiredPeriod`.
- `'slip'` — Immediately executes the next iteration of the loop.



Each code section calls `waitfor` at the end of execution.

Object Functions

waitfor	Pause code execution to achieve desired execution rate
statistics	Statistics of past execution periods

reset	Reset ros2rate object

Examples

Run Loop at Fixed Rate Using `ros2rate`

Create a ROS 2 node.

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

Create a publisher to publish a standard integer message.

```
pub = ros2publisher(node,"/my_int","std_msgs/Int64");
```

Create a `ros2rate` object that runs at 2 Hz.

```
r = ros2rate(node,2);
```

Start loop that prints the current 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: 0.008841
Iteration: 2 - Time Elapsed: 0.519071
Iteration: 3 - Time Elapsed: 1.015046
Iteration: 4 - Time Elapsed: 1.512195
Iteration: 5 - Time Elapsed: 2.012841
Iteration: 6 - Time Elapsed: 2.510505
Iteration: 7 - Time Elapsed: 3.002018
Iteration: 8 - Time Elapsed: 3.500703
Iteration: 9 - Time Elapsed: 4.014428
Iteration: 10 - Time Elapsed: 4.500222
```

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `statistics` object function is not supported.

- The `OverrunAction` setting `'drop'` is not supported.

See Also

`waitfor` | `reset`

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.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see “ROS Message Structures” on page 3-152.

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 = rossubscriber(___,"DataFormat","struct")
```

```
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)
sub = ros.Subscriber(___,"DataFormat","struct")
```

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 = rossubscriber(___,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-152

`sub = ros.Subscriber(node,topicname)` subscribes to a topic with name, `TopicName`. The `node` is the `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.

`sub = ros.Subscriber(___,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-152

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};
```

DataFormat — Message format
"object" (default) | "struct"

Message format, specified as "object" or "struct". You must set this property on creation using the name-value input. For more information, see "ROS Message Structures" on page 3-152.

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
```

```
Launching ROS Core...
..Done in 2.6893 seconds.
Initializing ROS master on http://172.30.131.134:57346.
Initializing global node /matlab_global_node_49538 with NodeURI http://bat6234win64:59473/ and Ma
```

```
exampleHelperROSCreateSampleNetwork
```

Create a subscriber for the '/scan' topic using message structures. Wait for the subscriber to register with the master.

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

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

```
[msg2,status,statustext] = receive(sub,10)
```

```
msg2 = struct with fields:
       MessageType: 'sensor_msgs/LaserScan'
            Header: [1x1 struct]
          AngleMin: -0.5467
          AngleMax: 0.5467
    AngleIncrement: 0.0017
     TimeIncrement: 0
          ScanTime: 0.0330
          RangeMin: 0.4500
          RangeMax: 10
            Ranges: [640x1 single]
       Intensities: []


status = logical
   1


statustext =
'success'
```

Shutdown the timers used by sample network.

```
exampleHelperROSShutDownSampleNetwork
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_49538 with NodeURI http://bat6234win64:59473/ and M
Shutting down ROS master on http://172.30.131.134:57346.
```

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
```

```
Launching ROS Core...
..Done in 2.1585 seconds.
Initializing ROS master on http://172.30.131.134:51178.
Initializing global node /matlab_global_node_46762 with NodeURI http://bat6234win64:59585/ and Ma
```

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','DataFormat','struct');
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,'DataFormat','struct');
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_46762 with NodeURI http://bat6234win64:59585/ and M
Shutting down ROS master on http://172.30.131.134:51178.
```

Use ROS Subscriber Object

Use a ROS `Subscriber` object to receive messages over the ROS network.

Start the ROS core and node.

```
master = ros.Core;
```

```
Launching ROS Core...
..Done in 2.0694 seconds.
```

```
node = ros.Node('/test');
```

Create a publisher and subscriber to send and receive a message over the ROS network. Use ROS messages as structures.

```
pub = ros.Publisher(node,'/chatter','std_msgs/String','DataFormat','struct');
sub = ros.Subscriber(node,'/chatter','std_msgs/String','DataFormat','struct');
```

Send a message over the network.

```
msg = rosmessage(pub);
msg.Data = 'hello world';
send(pub,msg)
```

View the message data using the `LatestMessage` property of the `Subscriber` object.

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

```
ans = struct with fields:
    MessageType: 'std_msgs/String'
           Data: 'hello world'
```

Clear the publisher, subscriber, and ROS node. Shut down the ROS master.

```
clear('pub','sub','node')
clear('master')
```

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for `struct` messages.
- `MessageType` argument must be specified.
- Callback functions must be assigned at the time of `Subscriber` object creation, and cannot be changed during run-time.

See Also

`receive` | `rospublisher` | `rosmessage`

Topics

“Exchange Data with ROS Publishers and Subscribers”

rossvcclient

Connect to ROS service server

Description

Use `rossvcclient` or `ros.ServiceClient` to create a service client object over ROS network. 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.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see "ROS Message Structures" on page 3-158.

Creation

Syntax

```
client = rossvcclient(servicename,servicetype)
client = rossvcclient(servicename,servicetype,Name,Value)

[client,reqmsg] = rossvcclient( ___ )
[ ___ ] = rossvcclient( ___ ,"DataFormat","struct")
[ ___ ] = rossvcclient( ___ ,"IsPersistent","true")

client = ros.ServiceClient(node, name)
client = ros.ServiceClient(node, name,"Timeout",timeout)
[ ___ ] = ros.ServiceClient( ___ ,"DataFormat","struct")
```

Description

`client = rossvcclient(servicename,servicetype)` creates a service client with the given `ServiceName` that connects to a service serve of type `ServiceType`. This command syntax prevents the current MATLAB program from running until it can connect to the service server.

`client = rossvcclient(servicename,servicetype,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 `rosmessage`.

`[___] = rossvcclient(___,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-158.

`[___] = rossvcclient(___,"IsPersistent","true")` determines that the service client is persistent.

`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.

`[___] = ros.ServiceClient(___,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-158.

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"`

`DataFormat` — Message format
`"object"` (default) | `"struct"`

Message format, specified as `"object"` or `"struct"`. You must set this property on creation using the name-value input. For more information, see “ROS Message Structures” on page 3-158.

`IsPersistent` — Determines if the service connection is persistent
`true` or `1` (default) | `false` or `0`

Option to determine if the service connection is persistent, specified as a numeric or logical `1`(`true`) or `0`(`false`).

With a persistent connection, a client always stays connected to a service. The connection persists until the service client gets deleted or the service server becomes unavailable. Persistent connections significantly improve performance for repeated requests, however, they also make the client more fragile to service failures when the connection is lost.

In case of non-persistent connection, a client normally does a lookup and reconnects to a service each time. This connection is slower but potentially allows a client to connect to a different node each time it does a service call, assuming that the lookup return a different node.

Data Types: `logical`

Object Functions

rosmessage	Create ROS messages
call	Call ROS or ROS 2 service server and receive a response
isServerAvailable	Determine if ROS or ROS 2 service server is available
waitForServer	Wait for ROS or ROS 2 service server to start

Examples

Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6739 seconds.
Initializing ROS master on http://172.30.131.134:59927.
Initializing global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and Ma
```

Set up a service server. Use structures for the ROS message data format.

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

Check whether the service server is available. If it is, wait for the service client to connect to the server.

```
if(isServerAvailable(client))
    [connectionStatus,connectionStatustext] = waitForServer(client)
end
```

```
connectionStatus = logical
   1


connectionStatustext = 
'success'
```

Call service server with default message.

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

```
response = struct with fields:
    MessageType: 'std_srvs/EmptyResponse'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,"Timeout",3);
```

```
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response)
end

    MessageType: 'std_srvs/EmptyResponse'
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and N
Shutting down ROS master on http://172.30.131.134:59927.
```

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;
```

```
Launching ROS Core...
.Done in 1.5783 seconds.
```

```
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. Use structures for the ROS message data format.

```
server = ros.ServiceServer(node,'/data/string',...
                           'roseus/StringString','DataFormat','struct');
```

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. Use structures for the ROS message data format.

Create a request message based on the client.

```
client = ros.ServiceClient(node,'/data/string','DataFormat','struct');
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 = struct with fields:
    MessageType: 'roseus/StringStringResponse'
            Str: 'dlrow olleh'
```

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
```

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

R2021b: Specify `ServiceType` when you create the client
Behavior change in future release

In a future release, you must specify the `ServiceType` property during the client creation.

R2021b: `Timeout` name-value pair argument will be removed in a future release
Not recommended starting in R2021b

In a future release, `Timeout` name-value pair argument will be removed for `rossvcclient`. Use `waitForServer` instead to specify a timeout and obtain the connection status of the client to the service server.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for `struct` messages.
- `ServiceType` property must be specified.
- Callback functions must be assigned at the time of `rossvcclient` object creation, and cannot be changed during run-time.
- For `ros.ServiceClient`, `node` input argument must be empty.
- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also
`rosservice` | `rossvcserver` | `call` | `isServerAvailable` | `waitForServer` | `rosmessage`

Topics
“Call and Provide ROS Services”

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 `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.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see “ROS Message Structures” on page 3-163.

Creation

Syntax

```
server = rossvcserver(servicename,svctype)
server = rossvcserver(servicename,svctype,callback)
[ ___ ] = rossvcclient( ___ ,"DataFormat","struct")

server = ros.ServiceServer(node, name,type)
server = ros.ServiceServer(node, name,type,callback)
[ ___ ] = ros.ServiceServer( ___ ,"DataFormat","struct")
```

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.

`[ ___ ] = rossvcclient( ___ ,"DataFormat","struct")` uses message structures instead of objects with any of the arguments in previous syntaxes. For more information, see “ROS Message Structures” on page 3-163.

`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.

`[ ___ ] = ros.ServiceServer( ___ ,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-163.

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};
```

`DataFormat` — Message format
`"object"` (default) | `"struct"`

Message format, specified as `"object"` or `"struct"`. You must set this property on creation using the name-value input. For more information, see "ROS Message Structures" on page 3-163.

Object Functions

rosmessage   Create ROS messages

Examples

Call Service Client with Default Message

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6739 seconds.
Initializing ROS master on http://172.30.131.134:59927.
Initializing global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and Ma
```

Set up a service server. Use structures for the ROS message data format.

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

Check whether the service server is available. If it is, wait for the service client to connect to the server.

```
if(isServerAvailable(client))
    [connectionStatus,connectionStatustext] = waitForServer(client)
end
```

```
connectionStatus = logical
   1
```

```
connectionStatustext = 
'success'
```

Call service server with default message.

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

```
response = struct with fields:
    MessageType: 'std_srvs/EmptyResponse'
```

If the `call` function above fails, it results in an error. Instead of an error, if you would prefer to react to a call failure using conditionals, return the `status` and `statustext` outputs from the call function. The `status` output indicates if the call succeeded, while `statustext` provides additional information.

```
numCallFailures = 0;
[response,status,statustext] = call(client,"Timeout",3);
if ~status
    numCallFailures = numCallFailues + 1;
    fprintf("Call failure number %d. Error cause: %s\n",numCallFailures,statustext)
else
    disp(response)
end
```

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

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_12960 with NodeURI http://bat6234win64:51978/ and 
Shutting down ROS master on http://172.30.131.134:59927.
```

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see "Improve Performance of ROS Using Message Structures".

R2021b: Specify `NewMessageFcn` call back property when you create the server
Behavior change in future release

In a future release, you must specify the `NewMessageFcn` callback property during the server creation.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for `struct` messages.
- `ServiceType` argument must be specified.
- Callback functions must be assigned at the time of `rossvcserver` or `ros.ServiceServer` object creation, and cannot be changed during run-time.
- For `ros.ServiceServer`, `node` input argument must be empty.
- Supported only for the Build Type, `Executable`.
- Usage in MATLAB Function block is not supported.

See Also
`rossvcclient` | `call` | `rosmessage`

Topics
"Call and Provide ROS Services"

rostf

Receive, send, and apply ROS transformations

Description

Calling the `rostf` 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 `rostf` to create the transformation tree.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see “ROS Message Structures” on page 3-169.

Creation

Syntax

```
tfTree = rostf
tfTree = rostf("DataFormat","struct")

trtree = ros.TransformationTree(node)
tfTree = ros.TransformationTree(node,"DataFormat","struct")
```

Description

`tfTree = rostf` creates a ROS `TransformationTree` object.

`tfTree = rostf("DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-169.

`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.

`tfTree = ros.TransformationTree(node,"DataFormat","struct")` uses message structures instead of objects. For more information, see “ROS Message Structures” on page 3-169.

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

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.

`DataFormat` — Message format
`"object"` (default) | `"struct"`

Message format, specified as `"object"` or `"struct"`. You must set this property on creation using the name-value input. For more information, see “ROS Message Structures” on page 3-169.

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. Create a node. Use the example helper function to publish transformation data.

```
rosinit
```

```
Launching ROS Core...
...Done in 3.7343 seconds.
Initializing ROS master on http://172.30.131.134:59915.
Initializing global node /matlab_global_node_24535 with NodeURI http://bat6234win64:61514/ and Ma
```

```
node = ros.Node('/testTf');
```

```
Using Master URI http://localhost:59915 from the global node to connect to the ROS master.
```

```
exampleHelperROSStartTfPublisher
```

Create a transformation tree. Use structures as the ROS message data format. Use the `AvailableFrames` property to see the transformation frames available. These transformations were specified separately prior to connecting to the network.

```
tree = rostf('DataFormat','struct');
pause(1);
tree.AvailableFrames
```

```
ans = 3x1 cell
    {'camera_center' }
    {'mounting_point'}
    {'robot_base'    }
```

Disconnect from the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_24535 with NodeURI http://bat6234win64:61514/ and M
Shutting down ROS master on http://172.30.131.134:59915.
```

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
```

```
Launching ROS Core...
..Done in 2.0852 seconds.
Initializing ROS master on http://172.30.131.134:59798.
Initializing global node /matlab_global_node_24302 with NodeURI http://bat6234win64:57081/ and Ma
```

```
node = ros.Node('/testTf');
```

```
Using Master URI http://localhost:59798 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,'DataFormat','struct');
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 = struct with fields:
     Sec: 1677880294
    Nsec: 575883700
```

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 = rosmessage('geometry_msgs/PointStamped','DataFormat','struct');
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 = struct with fields:
    MessageType: 'geometry_msgs/PointStamped'
         Header: [1x1 struct]
          Point: [1x1 struct]
```

Display the transformed point coordinates.

```
tfpt.Point
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Point'
              X: 1.2000
              Y: 1.5000
              Z: -2.5000
```

Clear ROS node. Shut down ROS master.

```
clear('node')
rosshutdown
```

```
Shutting down global node /matlab_global_node_24302 with NodeURI http://bat6234win64:57081/ and M
Shutting down ROS master on http://172.30.131.134:59798.
```

Limitations

- In ROS Noetic, multiple coordinate frames with redundant timestamp cannot be published.

Version History

Introduced in R2019b

R2021a: ROS Message Structures
Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Accessing the last update time from the `TransformationTree` object using its `LastUpdateTime` property is not supported.
- Usage in MATLAB Function block is not supported.

See Also

`waitForTransform` | `getTransform` | `transform` | `sendTransform`

Topics
“Access the tf Transformation Tree in ROS”

ros2tf

Receive, send, verify and apply ROS 2 transformations

Description

Use `ros2tf` to create a ROS 2 transformation tree, which allows you to access the dynamic transformations `tf` and static transformations `tf_static`, that are shared over the ROS 2 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 2 network.

1. Dynamic transform broadcasts the transformation message over coordinate frames that change relative to each other through time. These apply to the moving parts of the robot.
2. Static transform broadcasts the transformation message over fixed coordinate frames. These apply to the non moving parts of the robot.

ROS 2 uses the `tf2_ros` transformation library to keep track of the relationship between multiple coordinate frames. The relative transformations between these coordinate frames are maintained in a tree structure. You can transform entities like poses and points between any two coordinate frames by querying this tree.

The transformation tree changes over time and by default, the transformations are buffered for up to 10 seconds. You can access transformations at any time in this buffer window and the result will be interpolated to the exact time that you specify.

Creation

Syntax

```
tftree = ros2tf(node)
tftree = ros2tf(
node,'DynamicBroadcasterQoS',struct('Depth',200,'Reliability','besteffort'))
tftree = ros2tf(node,'StaticBroadcasterQoS',struct('Depth',50))
tftree = ros2tf(
node',DynamicListenerQoS',struct('Depth',200,'Reliability','besteffort'))
tftree = ros2tf(node,'StaticListenerQoS',struct('Depth',50))
```

Description

`tftree = ros2tf(node)` creates a ROS transformation tree object handle that the transformation tree is attached to. The `node` is the node connected to the ROS 2 network that publishes transformations.

`tftree = ros2tf(`
`node,'DynamicBroadcasterQoS',struct('Depth',200,'Reliability','besteffort'))` declares quality of service settings for the dynamic transform broadcaster while creating transformation tree.

`tftree = ros2tf(node,'StaticBroadcasterQoS',struct('Depth',50))` declares quality of service settings for the static transform broadcaster while creating transformation tree.

`tftree = ros2tf( node',DynamicListenerQoS',struct('Depth',200,'Reliability','besteffort'))` declares quality of service settings for the dynamic transform listener while creating transformation tree.

`tftree = ros2tf(node,'StaticListenerQoS',struct('Depth',50))` declares quality of service settings for the static transform listener while creating transformation tree.

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. It is empty if no frames are in the tree.

Example: `{'camera_center';'mounting_point';'robot_base'}`

Data Types: `cell`

`LastUpdateTime` — Time when the last transform was received

`"struct"`

This property is read-only.

Time when the last transformation was received, specified as a structure that resembles `ros2time`. It is empty if no transformation is received.

Data Types: `struct`

`BufferTime` — Length of time for which transformations are buffered

10 (default) | scalar

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.

`DynamicBroadcasterQoS` — Broadcaster of QoS settings for dynamic transforms

`"struct"`

This property is read-only.

Quality of service settings to be declared for the dynamic transform broadcaster while creating transformation tree, specified as `"struct"`. Specify a structure containing QoS settings such as `History`, `Depth`, `Reliability` and `Durability`. These QoS settings are used by the `TransformBroadCaster` of `tf2_ros`, while broadcasting `tf2_msgs/TFMessage` onto `/tf` topic.

Example: `{'History: keeplast, Depth: 100, Reliability: reliable, Durability: volatile'}`

Data Types: `struct`

`StaticBroadcasterQoS` — Broadcaster of QoS settings for static transforms
`"struct"`

This property is read-only.

Quality of service settings to be declared for the static transform broadcaster while creating transformation tree, specified as `"struct"`. Specify a structure containing QoS settings such as `History`, `Depth`, `Reliability` and `Durability`. These QoS settings are used by the `StaticTransformBroadCaster` of `tf2_ros`, while broadcasting `tf2_msgs/TFMessage` onto `/tf_static` topic.

Example: `{'History: keeplast, Depth: 1, Reliability: reliable, Durability: transientlocal'}`

Data Types: `struct`

`DynamicListenerQoS` — Listener QoS settings for dynamic transforms
`"struct"`

This property is read-only.

Quality of service settings to be declared for the dynamic transform listener while creating transformation tree, specified as `"struct"`. Specify a structure containing QoS settings such as `History`, `Depth`, `Reliability` and `Durability`. These QoS settings are used by the `TransformListener` of `tf2_ros`, while listening to `tf2_msgs/TFMessage` onto `/tf` topic.

Example: `{'History: keeplast, Depth: 100, Reliability: reliable, Durability: volatile'}`

Data Types: `struct`

`StaticListenerQoS` — Listener QoS settings for static transforms
`"struct"`

This property is read-only.

Quality of service settings to be declared for the Static transform listener while creating transformation tree, specified as `"struct"`. Specify a structure containing QoS settings such as `History`, `Depth`, `Reliability` and `Durability`. These QoS settings are used by the `StaticTransformListener` of `tf2_ros`, while listening to `tf2_msgs/TFMessage` onto `/tf_static` topic.

Example: `{'History: keeplast, Depth: 100, Reliability: reliable, Durability: transientlocal'}`

Data Types: `struct`

Object Functions

getTransform	Return the transformation between two coordinate frames
transform	Transform message entities into target coordinate frame
sendTransform	Send a transformation to the ROS 2 network
canTransform	Verify if transformation is available

Examples

Create a ROS 2 Transformation Tree

This example assumes that a ROS 2 node publishes transformations between `robot_base` and `camera_center`. For example, a real or simulated TurtleBot would do that.

Create a ROS 2 node on domain ID 25. Use the example helper function to publish transformation data.

```
node = ros2node("/matlabNode",25);
exampleHelperROS2StartTfPublisher
```

Retrieve the transformation tree object.

```
tftree = ros2tf(node);
pause(1)
```

Use the `AvailableFrames` property to see the transformation frames available. These transformations were specified separately prior to connecting to the network.

```
frames = tftree.AvailableFrames
```

```
frames = 3×1 cell
    {'camera_center' }
    {'mounting_point'}
    {'robot_base'    }
```

Use the `LastUpdateTime` property to see the time when the last transformation was received.

```
updateTime = tftree.LastUpdateTime
```

```
updateTime = struct with fields:
    MessageType: 'builtin_interfaces/Time'
            sec: 1670925404
        nanosec: 440832800
```

Wait for the transformation that takes data from `camera_center` to `robot_base`. It waits for the transformation to be valid within 5 seconds.

```
getTransform(tftree,'robot_base','camera_center', Timeout=5)
```

```
ans = struct with fields:
       MessageType: 'geometry_msgs/TransformStamped'
            header: [1×1 struct]
    child_frame_id: 'camera_center'
         transform: [1×1 struct]
```

Define a point [3 1.5 0.2] in the camera's coordinate frame.

```
pt = ros2message('geometry_msgs/PointStamped');
pt.header.frame_id = 'camera_center';
pt.point.x = 3;
pt.point.y = 1.5;
pt.point.z = 0.2;
```

The transformation is now available, so transform the point into the `robot_base` frame.

```
tfpt = transform(tftree,'robot_base',pt)
```

```
tfpt = struct with fields:
    MessageType: 'geometry_msgs/PointStamped'
         header: [1×1 struct]
          point: [1×1 struct]
```

Display the transformed point coordinates.

```
tfpt.point
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Point'
              x: 1.2000
              y: 1.5000
              z: -2.5000
```

Stop the example transformation publisher.

```
exampleHelperROS2StopTfPublisher
```

Clear the node.

```
clear('node')
```

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Accessing the last update time from the `ros2tf` object using its `LastUpdateTime` property is not supported when you are generating code using MATLAB Coder™.
- Usage of the function inside MATLAB Function block in Simulink is not supported.

See Also

`getTransform` | `transform` | `sendTransform` | `canTransform`

Topics

"Access the tf Transformation Tree in ROS 2"

rostime

Access ROS time functionality

Description

A ROS `Time` message represents 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")

time = rostime( ___ ,"DataFormat","struct")
```

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.

Note In a future release, ROS Toolbox will use message structures instead of objects for ROS messages.

To use message structures now, set the `"DataFormat"` name-value argument to `"struct"`. For more information, see "ROS Message Structures" on page 3-179.

`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.

`time = rostime( ___ ,"DataFormat","struct")` uses message structures instead of objects with any of the arguments in previous syntaxes. For more information, see “ROS Message Structures” on page 3-179.

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. It 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`.

`DataFormat` — Message format
`"object"` (default) | `"struct"`

Message format, specified as `"object"` or `"struct"`. You must set this property on creation using the name-value input. For more information, see “ROS Message Structures” on page 3-179.

Examples

Get Current ROS Time

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.1866 seconds.
Initializing ROS master on http://172.30.131.134:56360.
Initializing global node /matlab_global_node_75227 with NodeURI http://bat6234win64:59816/ and Ma
```

Get the current ROS Time as a ROS message structure. You can also check whether is it system time by getting the `issim` output.

```
[t,issim] = rostime('now','DataFormat','struct')
```

```
t = struct with fields:
     Sec: 1677880399
    Nsec: 78270100


issim = logical
   0
```

Shutdown the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_75227 with NodeURI http://bat6234win64:59816/ and M
Shutting down ROS master on http://172.30.131.134:56360.
```

Timestamp ROS Message Data

Initialize a ROS network.

```
rosinit
```

```
Launching ROS Core...
...Done in 3.3109 seconds.
Initializing ROS master on http://172.30.131.134:53734.
Initializing global node /matlab_global_node_01616 with NodeURI http://bat6234win64:60323/ and Ma
```

Create a stamped ROS message using structures. Specify the `Header.Stamp` property with the current system time.

```
point = rosmessage('geometry_msgs/PointStamped','DataFormat','struct');
point.Header.Stamp = rostime('now','system','DataFormat','struct');
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_01616 with NodeURI http://bat6234win64:60323/ and M
Shutting down ROS master on http://172.30.131.134:53734.
```

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 the ROS time as a structure message.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.7011 seconds.
```

```
Initializing ROS master on http://172.30.131.134:53574.
Initializing global node /matlab_global_node_81249 with NodeURI http://bat6234win64:64571/ and Ma
```

```
t = rostime('now','DataFormat','struct');
```

Convert ROS Time to double using the `seconds` function and set time to a specified MATLAB format, `datetime`.

```
time = datetime(t.Sec + 10^-9*t.Nsec,'ConvertFrom','posixtime')
```

```
time = datetime
   03-Mar-2023 21:52:48
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_81249 with NodeURI http://bat6234win64:64571/ and M
Shutting down ROS master on http://172.30.131.134:53574.
```

Version History

Introduced in R2019b

R2021a: ROS Message Structures

Behavior change in future release

You can now create messages as structures with fields matching the message object properties. Using structures typically improves performance of creating, updating, and using ROS messages, but message fields are no longer validated when set. Message types and corresponding field values from the structures are validated when sent across the network.

To use ROS messages as structures, use the `"DataFormat"` name-value argument when creating your publishers, subscribers, or other ROS objects. Any messages generated from these objects will utilize structures.

```
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
msg = rosmessage(pub)
```

You can also create messages as structures directly, but make sure to specify the data format as `"struct"` for the publisher, subscriber, or other ROS objects as well. ROS objects still use message objects by default.

```
msg = rosmessage("/scan","sensor_msgs/LaserScan","DataFormat","struct")
...
pub = rospublisher("/scan","sensor_msgs/LaserScan","DataFormat","struct")
```

In a future release, ROS messages will use structures by default and ROS message objects will be removed.

For more information, see “Improve Performance of ROS Using Message Structures”.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

See Also

`rosduration` | `rosmessage`

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
```

```
Launching ROS Core...
..Done in 2.6922 seconds.
Initializing ROS master on http://172.30.131.134:51325.
Initializing global node /matlab_global_node_56947 with NodeURI http://bat6234win64:63647/ and Ma
```

```
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)
```

```
  Header
    Stamp
      Sec  :  1677880106
      Nsec :  462756100
    Seq     :  0
    FrameId :  camera_center
  Transform
    Translation
      X :  0.5
      Y :  0
      Z :  -1
    Rotation
      X :  0
      Y :  -0.7071067811865476
      Z :  0
      W :  0.7071067811865476
  ChildFrameId :  robot_base
```

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: -1.0000

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

Stop the example transformation publisher.

```
exampleHelperROSStopTfPublisher
```

Shutdown ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_56947 with NodeURI http://bat6234win64:63647/ and M
Shutting down ROS master on http://172.30.131.134:51325.
```

Apply Transformation using `TransformStamped` Object

Apply a transformation from a `TransformStamped` object to a `PointStamped` message.

Start ROS network and setup transformations.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6105 seconds.
Initializing ROS master on http://172.30.131.134:52598.
Initializing global node /matlab_global_node_71525 with NodeURI http://bat6234win64:49546/ and Ma
```

```
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)
```

```
  Header 
    Stamp 
      Sec  :  1677879474
      Nsec :  912193000
    Seq     :  0
    FrameId :  camera_center
  Transform
```

```
    Translation
      X :  0.5
      Y :  0
      Z :  -1
    Rotation
      X :  0
      Y :  -0.7071067811865476
      Z :  0
      W :  0.7071067811865476
  ChildFrameId :  robot_base
```

Create point to transform. You could also get this point message off the ROS network.

```
pt = rosmessage('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.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_71525 with NodeURI http://bat6234win64:49546/ and M
Shutting down ROS master on http://172.30.131.134:52598.
```

Version History

Introduced in R2019b

See Also

Functions
`rostf` | `apply` | `getTransform` | `transform` | `waitForTransform`

Topics
"Access the tf Transformation Tree in ROS"

velodyneROSMessageReader

Read Velodyne ROS messages

Description

The `velodyneROSMessageReader` object reads point cloud data from VelodyneScan ROS messages, collected from a Velodyne® lidar sensor. To read this point cloud data into the workspace as point cloud object, use the `readFrame` object function. To check for additional point clouds in the message set, use the `hasFrame` object function.

Creation

Syntax

```
veloReader = velodyneROSMessageReader(msgs,devicemodel)
veloReader = velodyneFileReader(fileName,deviceModel,Name=Value)
```

Description

`veloReader = velodyneROSMessageReader(msgs,devicemodel)` creates a Velodyne ROS message reader object for a set of `VelodyneScan` ROS messages `msgs` from a specified device model `devicemodel`.

`veloReader = velodyneFileReader(fileName,deviceModel,Name=Value)` specifies options using one or more name-value arguments in addition to any combination of arguments from previous syntaxes. For example, (`OrganizePoints=true`) returns an organized point cloud.

Input Arguments

`msgs` — ROS Velodyne scan messages
cell array of VelodyneScan message objects | structure array

Velodyne scan ROS messages, specified as a cell array of VelodyneScan message objects or a structure array. The message type is `velodyne_msgs/VelodyneScan`. This argument sets the `VelodyneMessages` property.

`devicemodel` — Name of device model
`'VLP16'` | `'VLP32C'` | `'HDL32E'` | `'HDL64E'`

Name of the device model, specified as a character vector:

- `'VLP16'`
- `'VLP32C'`
- `'HDL32E'`
- `'HDL64E'`

Note Specifying a device model other than the one that captured the scans may result in improperly calibrated point clouds.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: (`OrganizePoints=true`) returns an organized point cloud.

`CalibrationFile` — Calibration XML file
`''` (default) | string

Calibration XML file, specified as a string. If you do not specify a calibration file, the reader selects a default calibration file containing data from the Velodyne device manual.

`OrganizePoints` — Logical to set the structure for the output point cloud
`true` (default) | `false`

Logical to set the structure for the output point cloud, specified as a numeric or logical `1` (`true`) or `0` (`false`).

To return an organized point cloud structure, set `OrganizePoints` to `true`. For an organized point cloud, every row represents a separate laser scan, and the number of columns is based on the horizontal angle resolution of the sensor.

To return an organized point cloud structure, set `OrganizePoints` to `false`.

Properties

`VelodyneMessages` — Raw Velodyne ROS messages
cell array of VelodyneScan message objects | structure array

This property is read-only.

Raw Velodyne ROS messages, specified as a cell array of VelodyneScan message objects or structure array. The ROS messages are of type `velodyne_msgs/VelodyneScan`.

`DeviceModel` — Velodyne device model name
`'VLP16'` | `'VLP32C'` | `'HDL32E'` | `'HDL64E'` | `'VLS128'`

This property is read-only.

Velodyne device model name, specified as `'VLP16'`, `'VLP32C'`, `'HDL32E'`, `'HDL64E'`, or `'VLS128'`.

Note Specifying a device model other than the one that captured the scans may result in improperly calibrated point clouds.

`CalibrationFile` — Name of Velodyne calibration XML file
character vector | string scalar

This property is read-only.

Name of the Velodyne calibration XML file, specified as a character vector or string scalar. Each device model includes a default calibration file.

`NumberOfFrames` — Total number of point clouds
positive integer

This property is read-only.

Total number of point clouds in the file, specified as a positive integer.

`Duration` — Total duration of file in seconds
`duration` scalar

This property is read-only.

Total duration of the file, specified as a `duration` scalar, in seconds.

`StartTime` — Timestamp of first point cloud reading
`duration` scalar

This property is read-only.

Timestamp of the first point cloud, specified as a `duration` scalar in seconds.

Start and end times are specified relative to the previous whole hour. For instance, if the file is recorded for 7 minutes from 13:58 to 14:05, then:

- `StartTime` = 58 min = 3480 s
- `EndTime` = `StartTime` + 7 min = 65 min = 3900 s

`EndTime` — Timestamp of last point cloud reading
`duration` scalar

This property is read-only.

Timestamp of the last point cloud reading, specified as a `duration` scalar, in seconds.

Start and end times are relative to the whole hour. For example, if the data is recorded over the 7 minutes from 1:58 PM to 2:05 PM, then:

- `StartTime` = 58 min = 3840 s
- `EndTime` = `StartTime` + 7 min = 65 min = 3900 s

`Timestamps` — Timestamp of point cloud frames
`duration` vector

This property is read-only.

Timestamps of the point cloud frames in seconds, specified as a `duration` vector. The length of the vector is equal to the value of the `NumberOfFrames` property. The value of the first element in the vector is the same as that of the `StartTime` property. You can use this property to read point cloud frames captured at different times.

For example, read the timestamp of a point cloud frame from the `Timestamps` property. Use the start time as an input for the `readFrame` object function to read the corresponding point cloud frame.

```
veloReader = velodyneROSMessageReader(msgs,'HDL32E')
frameTime = veloReader.Timestamps(10);
ptCloud = readFrame(veloReader,frameTime);
```

`CurrentTime` — Timestamp of current point cloud
`duration` scalar

Timestamp of the current point cloud reading, specified as a `duration` scalar, in seconds. As you read point clouds using the `readFrame` object function, the object updates this property with the most recently read point cloud. You can use the `reset` object function to reset this property to the default value. The default value matches the `StartTime` property.

Object Functions

hasFrame	Determine if another Velodyne point cloud is available in the ROS messages
readFrame	Read point cloud frame from ROS message
reset	Reset CurrentTime property of velodyneROSMessageReader object to default value

Examples

Work with Velodyne ROS Messages

This example shows how to handle `VelodyneScan` messages from a Velodyne LiDAR.

Velodyne ROS messages store data in a format that requires some interpretation before it can be used for further processing. MATLAB® can help you by formatting Velodyne ROS messages for easy use.

Prerequisites: "Work with Basic ROS Messages"

Load Sample Messages

Load sample Velodyne messages. These messages are populated with data gathered from Velodyne LiDAR sensor.

```
load("lidarData_ConstructionRoad.mat")
```

VelodyneScan Messages

`VelodyneScan` messages are ROS messages that contain Velodyne LIDAR scan packets. You can see the standard ROS format for a `VelodyneScan` message by creating an empty message of the appropriate type. Use messages in structure format for better performance.

```
emptyveloScan = rosmessage("velodyne_msgs/VelodyneScan","DataFormat","struct")
```

```
emptyveloScan = struct with fields:
    MessageType: 'velodyne_msgs/VelodyneScan'
         Header: [1×1 struct]
        Packets: [0×1 struct]
```

Since you created an empty message, `emptyveloScan` does not contain any meaningful data. For convenience, the loaded `lidarData_ConstructionRoad.mat` file contains a set of `VelodyneScan` messages that are fully populated and stored in the msgs variable. Each element in the msgs cell array is a VelodyneScan ROS message struct. The primary data in each VelodyneScan message is in the Packets property, it contains multiple VelodynePacket messages. You can see the standard ROS format for a VelodynePacket message by creating an empty message of the appropriate type.

```
emptyveloPkt = rosmessage("velodyne_msgs/VelodynePacket","DataFormat","struct")
```

```
emptyveloPkt = struct with fields:
    MessageType: 'velodyne_msgs/VelodynePacket'
          Stamp: [1×1 struct]
           Data: [1206×1 uint8]
```

Create Velodyne ROS Message Reader

The velodyneROSMessageReader object reads point cloud data from VelodyneScan ROS messages based on their specified model type. Note that providing an incorrect device model may result in improperly calibrated point clouds. This example uses messages from the "HDL32E" model.

```
veloReader = velodyneROSMessageReader(msgs,"HDL32E")
```

```
veloReader = 
  velodyneROSMessageReader with properties:

    VelodyneMessages: {28×1 cell}
         DeviceModel: 'HDL32E'
     CalibrationFile: 'M:\jobarchive\Bdoc21b\2021_06_16_h16m50s15_job1697727_pass\matlab\toolbox\
      NumberOfFrames: 55
            Duration: 2.7477 sec
           StartTime: 1145.2 sec
             EndTime: 1147.9 sec
          Timestamps: [1145.2 sec    1145.2 sec    1145.3 sec    1145.3 sec    1145.4 sec    114
         CurrentTime: 1145.2 sec
```

Extract Point Clouds

You can extract point clouds from the raw packets message with the help of this velodyneROSMessageReader object. By providing a specific frame number or timestamp, one point cloud can be extracted from velodyneROSMessageReader object using the readFrame object function. If you call readFrame without a frame number or timestamp, it extracts the next point cloud in the sequence based on the CurrentTime property.

Create a duration scalar that represents one second after the first point cloud reading.

```
timeDuration = veloReader.StartTime + seconds(1);
```

Read the first point cloud recorded at or after the given time duration.

```
ptCloudObj = readFrame(veloReader,timeDuration);
```

Access Location data in the point cloud.

```
ptCloudLoc = ptCloudObj.Location;
```

Reset the CurrentTime property of veloReader to the default value

```
reset(veloReader)
```

Display All Point Clouds

You can also loop through all point clouds in the input Velodyne ROS messages.

Define *x*-, *y*-, and *z*-axes limits for `pcplayer` in meters. Label the axes.

```
xlimits = [-60 60];
ylimits = [-60 60];
zlimits = [-20 20];
```

Create the point cloud player.

```
player = pcplayer(xlimits,ylimits,zlimits);
```

Label the axes.

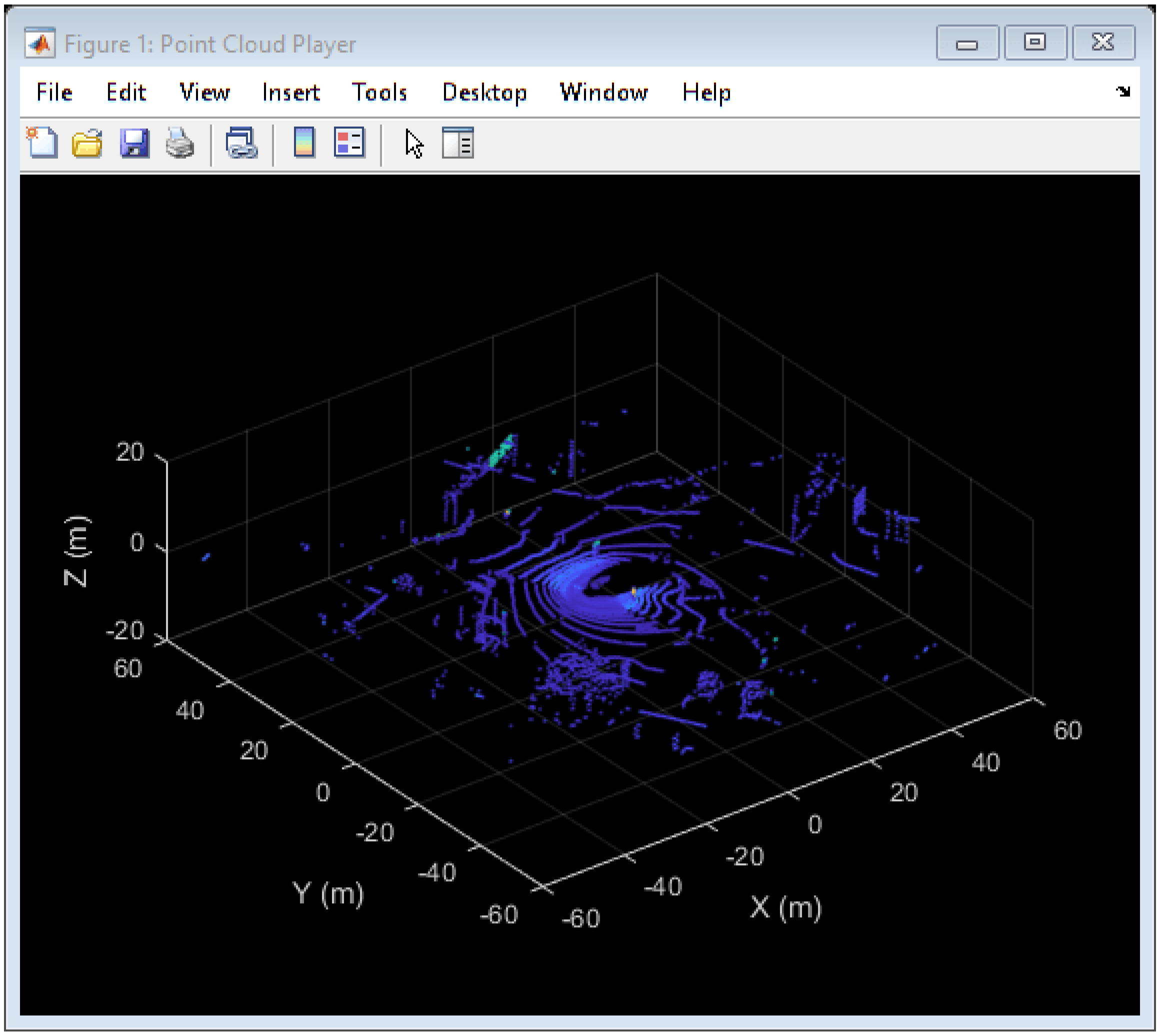
```
xlabel(player.Axes,"X (m)");
ylabel(player.Axes,"Y (m)");
zlabel(player.Axes,"Z (m)");
```

The first point cloud of interest is captured at 0.3 second into the input messages. Set the `CurrentTime` property to that time to begin reading point clouds from there.

```
veloReader.CurrentTime = veloReader.StartTime + seconds(0.3);
```

Display the point cloud stream for 2 seconds. To check if a new frame is available and continue past 2 seconds, remove the last `while` condition. Iterate through the file by calling `readFrame` to read in point clouds. Display them using the point cloud player.

```
while(hasFrame(veloReader) && isOpen(player) && (veloReader.CurrentTime < veloReader.StartTime +
    ptCloudObj = readFrame(veloReader);
    view(player,ptCloudObj.Location,ptCloudObj.Intensity);
    pause(0.1);
end
```



Tips

- Providing an incorrect device model will result in improperly calibrated point clouds.
- Not providing a calibration file can lead to inaccurate results.

Version History

Introduced in R2020b

See Also

`pointCloud` | `hasFrame` | `readFrame` | `reset`

pcplayer

Visualize streaming 3-D point cloud data

Description

Visualize 3-D point cloud data streams from devices such as Microsoft®Kinect®.

To improve performance, `pcplayer` automatically downsamples the rendered point cloud during interaction with the figure. The downsampling occurs only for rendering the point cloud and does not affect the saved points.

Creation

Syntax

```
player = pcplayer(xlimits,ylimits,zlimits)
player = pcplayer(xlimits,ylimits,zlimits,Name,Value)
```

Description

`player = pcplayer(xlimits,ylimits,zlimits)` returns a player with `xlimits`,`ylimits`, and `zlimits` set for the axes limits.

`player = pcplayer(xlimits,ylimits,zlimits,Name,Value)` returns a player with additional properties specified by one or more `Name,Value` pair arguments.

Input Arguments

`xlimits` — Range of *x*-axis coordinates
1-by-2 vector

Range of *x*-axis coordinates, specified as a 1-by-2 vector in the format [*min max*]. `pcplayer` does not display data outside these limits.

`ylimits` — Range of *y*-axis coordinates
1-by-2 vector

Range of *y*-axis coordinates, specified as a 1-by-2 vector in the format [*min max*]. `pcplayer` does not display data outside these limits.

`zlimits` — Range of *z*-axis coordinates
1-by-2 vector

Range of *z*-axis coordinates, specified as a 1-by-2 vector in the format [*min max*].`pcplayer` does not display data outside these limits.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `'VerticalAxisDir'`, `'Up'`.

`MarkerSize` — Diameter of marker

`6` (default) | positive scalar

Diameter of marker, specified as the comma-separated pair consisting of '`MarkerSize`' and a positive scalar. The value specifies the approximate diameter of the point marker. MATLAB graphics defines the unit as points. A marker size larger than six can reduce the rendering performance.

`VerticalAxis` — Vertical axis

`'Z'` (default) | `'X'` | `'Y'`

Vertical axis, specified as the comma-separated pair consisting of '`VerticalAxis`' and `'X'`, `'Y'`, or `'Z'`. When you reload a saved figure, any action on the figure resets the vertical axis to the *z*-axis.

`VerticalAxisDir` — Vertical axis direction

`'Up'` (default) | `'Down'`

Vertical axis direction, specified as the comma-separated pair consisting of '`VerticalAxisDir`' and `'Up'` or `'Down'`. When you reload a saved figure, any action on the figure resets the direction to the up direction.

Properties

`Axes` — Player axes handle

`axes` graphics object

Player axes handle, specified as an `axes` graphics object.

Usage

Color and Data Point Values in Figure

To view point data or modify color display values, hover over the axes toolbar and select one of the following options.

Feature	Description
Datatip	Click **Data Tips** to view the data point values for any point in the point cloud figure. For a normal point cloud, the **Data Tips** displays the *x*,*y*,*z* values. Additional data properties for the depth image and lidar are:

Point Cloud Data	Data Value Properties
Depth image (RGB-D sensor)	Color, row, column
Lidar	Intensity, range, azimuth angle, elevation angle, row, column

Feature	Description
Background color	Click **Rotate** and then right-click in the figure for background options.
Colormap value	Click **Rotate** and then right-click in the figure for colormap options. You can modify colormap values for the coordinate and range values available, depending on the type of point cloud displayed.
View	Click **Rotate** to change the viewing angle of the point cloud figure to the *XZ*, *ZX*,*YZ*, *ZY*, *XY*, or the *YX* plane. Click **Restore View** to reset the viewing angle.

OpenGL Option

`pcplayer` supports the `'opengl'` option for the `Renderer` `figure` property only.

Object Functions

hide	Hide player figure
isOpen	Visible or hidden status for player
show	Show player
view	Display point cloud

Examples

Terminate a Point Cloud Processing Loop

Create the player and add data.

```
player = pcplayer([0 1],[0 1],[0 1]);
```

Display continuous player figure. Use the `isOpen` function to check if player figure window is open.

```
while isOpen(player)
     ptCloud = pointCloud(rand(1000,3,'single'));
     view(player,ptCloud);
end
```

Terminate while-loop by closing pcplayer figure window.

Version History

Introduced in R2020b

See Also

`pointCloud`

hide

Hide player figure

Syntax

`hide(player)`

Description

`hide(player)` hides the figure. To redisplay the player, use `show(player)`.

Input Arguments

`player` — Player
object

Video player, specified as a `pcplayer` object.

Version History

Introduced in R2020b

isOpen

Visible or hidden status for player

Syntax

`isOpen(player)`

Description

`isOpen(player)` returns `true` or `false` to indicate whether the player is visible.

Input Arguments

`player` — Player
object

Video player, specified as a `pcplayer` object.

Version History

Introduced in R2020b

show

Show player

Syntax

`show(player)`

Description

`show(player)` makes the player figure visible again after closing or hiding it.

Input Arguments

`player` — Player
object

Player for visualizing data streams, specified as a `pcplayer` object. Use this method to view the figure after you have removed it from display. For example, after you x-out of a figure and you want to view it again. This is particularly useful to use after a while loop that contains display code ends.

Version History

Introduced in R2020b

view

Display point cloud

Syntax

```
view(player,ptCloud)
view(player,xyzPoints)
view(player,xyzPoints,color)
view(player,xyzPoints,colorMap)
```

Description

`view(player,ptCloud)` displays a point cloud in the `pcplayer` figure window, `player`. The points, locations, and colors are stored in the `ptCloud` object.

`view(player,xyzPoints)` displays the points of a point cloud at the locations specified by the `xyzPoints` matrix. The color of each point is determined by the *z* value.

`view(player,xyzPoints,color)` displays a point cloud with colors specified by `color`.

`view(player,xyzPoints,colorMap)` displays a point cloud with colors specified by `colorMap`.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object. The object contains the locations, intensities, and RGB colors to render the point cloud.

Point Cloud Property	Color Rendering Result
`Location` only	Maps the z-value to a color value in the current color map.
`Location` and `Intensity`	Maps the intensity to a color value in the current color map.
`Location` and `Color`	Use provided color.
`Location`, `Intensity`, and `Color`	Use provided color.

`player` — Player
`pcplayer` object

Player for visualizing 3-D point cloud data streams, specified as a `pcplayer` object.

`xyzPoints` — Point cloud *x*, *y*, and *z* locations
M-by-3 numeric matrix | *M*-by-*N*-by-3 numeric matrix

Point cloud *x*, *y*, and *z* locations, specified as either an *M*-by-3 or an *M*-by-*N*-by-3 numeric matrix. The *M*-by-*N*-by-3 numeric matrix is commonly referred to as an organized point cloud. The `xyzPoints` numeric matrix contains *M* or *M*-by-*N* [*x*,*y*,*z*] points. The *z* values in the numeric matrix, which generally correspond to depth or elevation, determine the color of each point.

`color` — Point cloud color
1-by-3 RGB vector | short name of color | long name of color | *M*-by-3 matrix | *M*-by-*N*-by-3 matrix

Point cloud color of points, specified as one of:

- RGB triplet
- A color name or a short name
- *M*-by-3 matrix
- *M*-by-*N*-by-3 matrix

Color Name	Short Name	RGB Triplet	Appearance
`"red"`	`"r"`	`[1 0 0]`	🟥
`"green"`	`"g"`	`[0 1 0]`	🟩
`"blue"`	`"b"`	`[0 0 1]`	🟦
`"cyan"`	`"c"`	`[0 1 1]`	🟦
`"magenta"`	`"m"`	`[1 0 1]`	🟪
`"yellow"`	`"y"`	`[1 1 0]`	🟨
`"black"`	`"k"`	`[0 0 0]`	⬛
`"white"`	`"w"`	`[1 1 1]`	⬜

You can specify the same color for all points or a different color for each point. When you set `color` to `single` or `double`, the RGB values range between [0, 1]. When you set `color` to `uint8`, the values range between [0, 255].

Points Input	Color Selection	Valid Values of `C`
`xyzPoints`	Same color for all points	1-by-3 RGB vector, or the short or long name of a color
	Different color for each point	*M*-by-3 matrix or *M*-by-*N*-by-3 matrix containing RGB values for each point.

`colorMap` — Point cloud color map
M-by-1 vector | *M*-by-*N* matrix

Point cloud color of points, specified as one of:

- *M*-by-1 vector
- *M*-by-*N* matrix

Points Input	Color Selection	Valid Values of `C`
`xyzPoints`	Different color for each point	Vector or *M*-by-*N* matrix. The matrix must contain values that are linearly mapped to a color in the current `colormap`.

Version History

Introduced in R2020b

pointCloud

Object for storing 3-D point cloud

Description

The `pointCloud` object creates point cloud data from a set of points in 3-D coordinate system. The point cloud data is stored as an object with the properties listed in “Properties” on page 3-203. Use “Object Functions” on page 3-204 to retrieve, select, and remove desired points from the point cloud data.

Creation

Syntax

```
ptCloud = pointCloud(xyzPoints)
ptCloud = pointCloud(xyzPoints,Name,Value)
```

Description

`ptCloud = pointCloud(xyzPoints)` returns a point cloud object with coordinates specified by `xyzPoints`.

`ptCloud = pointCloud(xyzPoints,Name,Value)` creates a `pointCloud` object with properties specified as one or more `Name,Value` pair arguments. For example, `pointCloud(xyzPoints,'Color',[0 0 0])` sets the `Color` property of the point `xyzPoints` as `[0 0 0]`. Enclose each property name in quotes. Any unspecified properties have default values.

Input Arguments

`xyzPoints` — 3-D coordinate points
M-by-3 list of points | *M*-by-*N*-by-3 array for organized point cloud

3-D coordinate points, specified as an *M*-by-3 list of points or an *M*-by-*N*-by-3 array for an organized point cloud. The 3-D coordinate points specify the *x*, *y*, and *z* positions of a point in the 3-D coordinate space. The first two dimensions of an organized point cloud correspond to the scanning order from sensors such as RGBD or lidar. This argument sets the `Location` property.

Data Types: `single` | `double`

Output Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, returned as a `pointCloud` object with the properties listed in “Properties” on page 3-203.

Properties

`Location` — Position of the points in 3-D coordinate space

M-by-3 array | *M*-by-*N*-by-3 array

This property is read-only.

Position of the points in 3-D coordinate space, specified as an *M*-by-3 or *M*-by-*N*-by-3 array. Each entry specifies the *x*, *y*, and *z* coordinates of a point in the 3-D coordinate space. You cannot set this property as a name-value pair. Use the `xyzPoints` input argument.

Data Types: `single` | `double`

`Color` — Point cloud color

`[]` (default) | *M*-by-3 array | *M*-by-*N*-by-3 array

Point cloud color, specified as an *M*-by-3 or *M*-by-*N*-by-3 array. Use this property to set the color of points in point cloud. Each entry specifies the RGB color of a point in the point cloud data. Therefore, you can specify the same color for all points or a different color for each point.

- The specified RGB values must lie within the range [0, 1], when you specify the data type for `Color` as `single` or `double`.
- The specified RGB values must lie within the range [0, 255], when you specify the data type for `Color` as `uint8`.

Coordinates	Valid assignment of `Color`
M-by-3 array	*M*-by-3 array containing RGB values for each point
M-by-*N*-by-3 array	*M*-by-*N*-by-3 array containing RGB values for each point

Data Types: `uint8`

`Normal` — Surface normals

`[]` (default) | *M*-by-3 array | *M*-by-*N*-by-3 array

Surface normals, specified as a *M*-by-3 or *M*-by-*N*-by-3 array. Use this property to specify the normal vector with respect to each point in the point cloud. Each entry in the surface normals specifies the *x*, *y*, and *z* component of a normal vector.

Coordinates	Surface Normals
M-by-3 array	*M*-by-3 array, where each row contains a corresponding normal vector.
M-by-*N*-by-3 array	*M*-by-*N*-by-3 array containing a 1-by-1-by-3 normal vector for each point.

Data Types: `single` | `double`

`Intensity` — Grayscale intensities

`[]` (default) | *M*-by-1 vector | *M*-by-*N* matrix

Grayscale intensities at each point, specified as a *M*-by-1 vector or *M*-by-*N* matrix. The function maps each intensity value to a color value in the current colormap.

Coordinates	Intensity
M-by-3 array	*M*-by-1 vector, where each row contains a corresponding intensity value.

Coordinates	Intensity
M-by-*N*-by-3 array	*M*-by-*N* matrix containing intensity value for each point.

Data Types: `single` | `double` | `uint8`

`Count` — Number of points
positive integer

This property is read-only.

Number of points in the point cloud, stored as a positive integer.

`XLimits` — Range of *x* coordinates
1-by-2 vector

This property is read-only.

Range of coordinates along *x*-axis, stored as a 1-by-2 vector.

`YLimits` — Range of *y* coordinates
1-by-2 vector

This property is read-only.

Range of coordinates along *y*-axis, stored as a 1-by-2 vector.

`ZLimits` — Range of *z* coordinates
1-by-2 vector

This property is read-only.

Range of coordinates along *z*-axis, stored as a 1-by-2 vector.

Object Functions

findNearestNeighbors	Find nearest neighbors of a point in point cloud
findNeighborsInRadius	Find neighbors within a radius of a point in the point cloud
findPointsInROI	Find points within a region of interest in the point cloud
removeInvalidPoints	Remove invalid points from point cloud
select	Select points in point cloud
copy	Copy array of handle objects

Tips

The `pointCloud` object is a `handle` object. If you want to create a separate copy of a point cloud, you can use the MATLAB `copy` method.
`ptCloudB = copy(ptCloudA)`

If you want to preserve a single copy of a point cloud, which can be modified by point cloud functions, use the same point cloud variable name for the input and output.
`ptCloud =` *pcFunction*`(ptCloud)`

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- GPU code generation for variable input sizes is not optimized. Consider using constant size inputs for an optimized code generation.
- GPU code generation supports the `'Color'`, `'Normal'`, and `'Intensity'` name-value pairs.
- GPU code generation supports the `findNearestNeighbors`, `findNeighborsInRadius`, `findPointsInROI`, `removeInvalidPoints`, and `select` methods.
- For very large inputs, the memory requirements of the algorithm may exceed the GPU device limits. In such cases, consider reducing the input size to proceed with code generation.

See Also

Objects
`pcplayer`

Functions
`findNearestNeighbors` | `findNeighborsInRadius` | `findPointsInROI` | `removeInvalidPoints` | `select`

findNearestNeighbors

Find nearest neighbors of a point in point cloud

Syntax

```
[indices,dists] = findNearestNeighbors(ptCloud,point,K)
[indices,dists] = findNearestNeighbors( ___ ,Name,Value)
```

Description

`[indices,dists] = findNearestNeighbors(ptCloud,point,K)` returns the `indices` for the K-nearest neighbors of a query point in the input point cloud. `ptCloud` can be an unorganized or organized point cloud. The K-nearest neighbors of the query point are computed by using the Kd-tree based search algorithm. This function requires a Computer Vision Toolbox license.

`[indices,dists] = findNearestNeighbors( ___ ,Name,Value)` specifies options using one or more name-value arguments in addition to the input arguments in the preceding syntaxes.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`point` — Query point
three-element vector of form [*x y z*]

Query point, specified as a three-element vector of form [*x y z*].

`K` — Number of nearest neighbors
positive integer

Number of nearest neighbors, specified as a positive integer.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `findNearestNeighbors(ptCloud,point,k,'Sort',true)`

`Sort` — Sort indices
`false` (default) | `true`

Sort indices, specified as a comma-separated pair of `'Sort'` and a logical scalar. When you set `Sort` to `true`, the returned indices are sorted in the ascending order based on the distance from a query point. To turn off sorting, set `Sort` to `false`.

`MaxLeafChecks` — Number of leaf nodes to check
`Inf` (default) | integer

Number of leaf nodes to check, specified as a comma-separated pair consisting of `'MaxLeafChecks'` and an integer. When you set this value to `Inf`, the entire tree is searched. When the entire tree is searched, it produces exact search results. Increasing the number of leaf nodes to check increases accuracy, but reduces efficiency.

Note The name-value argument `'MaxLeafChecks'` is valid only with Kd-tree based search method.

Output Arguments

`indices` — Indices of stored points
column vector

Indices of stored points, returned as a column vector. The vector contains K linear indices of the nearest neighbors stored in the point cloud.

`dists` — Distances to query point
column vector

Distances to query point, returned as a column vector. The vector contains the Euclidean distances between the query point and its nearest neighbors.

Version History

Introduced in R2020b

References

[1] Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For code generation in non-host platforms, the value for `'MaxLeafChecks'` must be set to the default value `Inf`. If you specify values other than `Inf`, the function generates a warning and automatically assigns the default value for `'MaxLeafChecks'`.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For GPU code generation, the `'MaxLeafChecks'` name-value pair option is ignored.

See Also

Objects
`pointCloud`

Functions
`findNeighborsInRadius` | `findPointsInROI` | `removeInvalidPoints` | `select`

findNeighborsInRadius

Find neighbors within a radius of a point in the point cloud

Syntax

```
[indices,dists] = findNeighborsInRadius(ptCloud,point,radius)
[indices,dists] = findNeighborsInRadius( ___ ,Name,Value)
```

Description

`[indices,dists] = findNeighborsInRadius(ptCloud,point,radius)` returns the `indices` of neighbors within a radius of a query point in the input point cloud. `ptCloud` can be an unorganized or organized point cloud. The neighbors within a radius of the query point are computed by using the Kd-tree based search algorithm. This function requires a Computer Vision Toolbox license.

`[indices,dists] = findNeighborsInRadius( ___ ,Name,Value)` specifies options using one or more name-value pair arguments in addition to the input arguments in the preceding syntaxes.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`point` — Query point
three-element vector of form `[`*`x y z`*`]`

Query point, specified as a three-element vector of form `[`*`x y z`*`]`.

`radius` — Search radius
scalar

Search radius, specified as a scalar. The function finds the neighbors within the specified `radius` around a query point in the input point cloud.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `findNeighborsInRadius(ptCloud,point,radius,'Sort',true)`

`Sort` — Sort indices
`false` (default) | `true`

Sort indices, specified as a comma-separated pair of `'Sort'` and a logical scalar. When you set `Sort` to `true`, the returned indices are sorted in the ascending order based on the distance from a query point. To turn off sorting, set `Sort` to `false`.

`MaxLeafChecks` — Number of leaf nodes
`Inf` (default) | integer

Number of leaf nodes, specified as a comma-separated pair consisting of `'MaxLeafChecks'` and an integer. When you set this value to `Inf`, the entire tree is searched. When the entire tree is searched, it produces exact search results. Increasing the number of leaf nodes to check increases accuracy, but reduces efficiency.

Output Arguments

`indices` — Indices of stored points
column vector

Indices of stored points, returned as a column vector. The vector contains the linear indices of the radial neighbors stored in the point cloud.

`dists` — Distances to query point
column vector

Distances to query point, returned as a column vector. The vector contains the Euclidean distances between the query point and its radial neighbors.

Version History

Introduced in R2020b

References

[1] Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- For code generation in non-host platforms, the value for `'MaxLeafChecks'` must be set to the default value `Inf`. If you specify values other than `Inf`, the function generates a warning and automatically assigns the default value for `'MaxLeafChecks'`.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

Usage notes and limitations:

- For GPU code generation, the `'MaxLeafChecks'` name-value pair option is ignored.

See Also

Objects
`pointCloud`

Functions
`findNearestNeighbors` | `findPointsInROI` | `removeInvalidPoints` | `select`

findPointsInROI

Find points within a region of interest in the point cloud

Syntax

`indices = findPointsInROI(ptCloud,roi)`

Description

`indices = findPointsInROI(ptCloud,roi)` returns the points within a region of interest (ROI) in the input point cloud. The points within the specified ROI are obtained using a Kd-tree based search algorithm. This function requires a Computer Vision Toolbox license.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`roi` — Region of interest
six-element vector of form [*xmin xmax ymin ymax zmin zmax*]

Region of interest, specified as a six-element vector of form [*xmin xmax ymin ymax zmin zmax*], where:

- *xmin* and *xmax* are the minimum and the maximum limits along the *x*-axis respectively.
- *ymin* and *ymax* are the minimum and the maximum limits along the *y*-axis respectively.
- *zmin* and *zmax* are the minimum and the maximum limits along the *z*-axis respectively.

Output Arguments

`indices` — Indices of stored points
column vector

Indices of stored points, returned as a column vector. The vector contains the linear indices of the ROI points stored in the point cloud.

Version History

Introduced in R2020b

References

[1] Muja, M. and David G. Lowe. "Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration". *In VISAPP International Conference on Computer Vision Theory and Applications*. 2009. pp. 331–340.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

-

See Also

Objects
`pointCloud`

Functions
`findNearestNeighbors` | `findNeighborsInRadius` | `removeInvalidPoints` | `select`

removeInvalidPoints

Remove invalid points from point cloud

Syntax

```
[ptCloudOut,indices] = removeInvalidPoints(ptCloud)
```

Description

`[ptCloudOut,indices] = removeInvalidPoints(ptCloud)` removes points with `Inf` or `NaN` coordinate values from point cloud and returns the indices of valid points. This function requires a Computer Vision Toolbox license.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

Output Arguments

`ptCloudOut` — Point cloud with points removed
`pointCloud` object

Point cloud, returned as a `pointCloud` object with `Inf` or `NaN` coordinates removed.

Note The output is always an unorganized (*X*-by-3) point cloud. If the input `ptCloud` is an organized point cloud (*M*-by-*N*-by-3), the function returns the output as an unorganized point cloud.

`indices` — Indices of valid points
vector

Indices of valid points in the point cloud, specified as a vector.

Version History

Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Objects
`pointCloud`

Functions
`findNearestNeighbors` | `findNeighborsInRadius` | `findPointsInROI` | `select`

select

Select points in point cloud

Syntax

```
ptCloudOut = select(ptCloud,indices)
ptCloudOut = select(ptCloud,row,column)
ptCloudOut = select( ___ ,'OutputSize',outputSize)
```

Description

`ptCloudOut = select(ptCloud,indices)` returns a `pointCloud` object containing only the points that are selected using linear indices. This function requires a Computer Vision Toolbox license.

`ptCloudOut = select(ptCloud,row,column)` returns a `pointCloud` object containing only the points that are selected using row and column subscripts. This syntax applies only if the input is an organized point cloud data of size *M*-by-*N*-by-3.

`ptCloudOut = select( ___ ,'OutputSize',outputSize)` returns the selected points as a `pointCloud` object of size specified by `outputSize`.

Input Arguments

`ptCloud` — Point cloud
`pointCloud` object

Point cloud, specified as a `pointCloud` object.

`indices` — Indices of selected points
vector

Indices of selected points, specified as a vector.

`row` — Row indices
vector

Row indices, specified as a vector. This argument applies only if the input is an organized point cloud data of size *M*-by-*N*-by-3.

`column` — Column indices
vector

Column indices, specified as a vector. This argument applies only if the input is an organized point cloud data of size *M*-by-*N*-by-3.

`outputSize` — Size of output point cloud
`'selected'` (default) | `'full'`

Size of the output point cloud, `ptCloudOut`, specified as `'selected'` or `'full'`.

- If the size is `'selected'`, then the output contains only the selected points from the input point cloud, `ptCloud`.
- If the size is `'full'`, then the output is same size as the input point cloud `ptCloud`. Cleared points are filled with `NaN` and the color is set to `[0 0 0]`.

Output Arguments

`ptCloudOut` — Selected point cloud
`pointCloud` object

Point cloud, returned as a `pointCloud` object.

Version History
Introduced in R2020b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

GPU Code Generation
Generate CUDA® code for NVIDIA® GPUs using GPU Coder™.

See Also

Objects
`pointCloud`

Functions
`findNearestNeighbors` | `findNeighborsInRadius` | `findPointsInROI` | `removeInvalidPoints`

Methods

getFeedbackMessage

Create new action feedback message

Syntax

```
msg = getFeedbackMessage(server)
```

Description

`msg = getFeedbackMessage(server)` creates and returns an empty message, `msg`, whose message type is determined by the action type of `server`. The format of `msg` is determined by the `DataFormat` property of the action server. This message is the default feedback message that this server sends to the client while executing a goal.

Examples

Create a ROS Action Server and Execute a Goal

This example shows how to create a ROS action server, connect an action client to it, receive goal, and execute it.

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6875 seconds.
Initializing ROS master on http://172.30.131.134:51566.
Initializing global node /matlab_global_node_50700 with NodeURI http://bat6234win64:63032/ and Ma
```

Set up an action server for calculating Fibonacci sequence. Use structures for the ROS message data format. Use `fibbonacciExecution` on page 4-3 function as the callback.

```
cb = @fibonacciExecution;
server = rosactionserver("/fibonacci","actionlib_tutorials/Fibonacci",ExecuteGoalFcn=cb,DataForma
```

```
server = 
  SimpleActionServer with properties:

        ActionName: '/fibonacci'
        ActionType: 'actionlib_tutorials/Fibonacci'
    ExecuteGoalFcn: @fibonacciExecution
        DataFormat: 'struct'
```

Create action client and send a goal to the server to calculate the Fibonacci sequence up to 10 terms past the first two terms, `0` and `1`. Display the result sequence.

```
client = rosactionclient("/fibonacci","actionlib_tutorials/Fibonacci",DataFormat="struct");
goal = rosmessage(client);
goal.Order = int32(10);
```

```
result = sendGoalAndWait(client,goal);
result.Sequence
```

```
ans = 12x1 int32 column vector

     0
     1
     1
     2
     3
     5
     8
    13
    21
    34
      ⋮
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_50700 with NodeURI http://bat6234win64:63032/ and M
Shutting down ROS master on http://172.30.131.134:51566.
```

Supporting Functions

The callback function `fibbonacciExecution` is executed every time the server receives a goal execution request from the client. This function checks if the goal has been preempted, executes the goal and sends feedback to the client during goal execution.

```
function [result,success] = fibonacciExecution(src,goal,defaultFeedback,defaultResult)

    % Initialize variables
    success = true;
    result = defaultResult;
    feedback = defaultFeedback;
    feedback.Sequence = int32([0 1]);

    for k = 1:goal.Order
        % Check that the client has not canceled or sent a new goal
        if isPreemptRequested(src)
            success = false;
            break
        end

        % Send feedback to the client periodically
        feedback.Sequence(end+1) = feedback.Sequence(end-1) + feedback.Sequence(end);
        sendFeedback(src,feedback)

        % Pause to allow time to complete other callbacks (like client feedback)
        pause(0.2)
    end

    if success
        result.Sequence = feedback.Sequence;
    end
```

```
end
```

Input Arguments

`server` — ROS action server
`SimpleActionServer` object handle

ROS action server, specified as a `SimpleActionServer` object handle.

Output Arguments

`msg` — Default feedback message
ROS message

Default feedback message, returned as a ROS message whose type is determined by the action type of `server`.

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rosactionserver` | `sendFeedback` | `rosActionServerExecuteGoalFcn` | `rosmessage`

isPreemptRequested

Check if a goal has been preempted

Syntax

```
status = isPreemptRequested(server)
```

Description

`status = isPreemptRequested(server)` checks whether the goal currently being executed by the action server, `server`, has been preempted and returns the `status` accordingly. The action client connected to `server` initiates the goal preemption either by cancelling the current goal or sending a new goal to execute.

Examples

Create a ROS Action Server and Execute a Goal

This example shows how to create a ROS action server, connect an action client to it, receive goal, and execute it.

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6875 seconds.
Initializing ROS master on http://172.30.131.134:51566.
Initializing global node /matlab_global_node_50700 with NodeURI http://bat6234win64:63032/ and Ma
```

Set up an action server for calculating Fibonacci sequence. Use structures for the ROS message data format. Use `fibbonacciExecution` on page 4-6 function as the callback.

```
cb = @fibonacciExecution;
server = rosactionserver("/fibonacci","actionlib_tutorials/Fibonacci",ExecuteGoalFcn=cb,DataForma
```

```
server = 
  SimpleActionServer with properties:

        ActionName: '/fibonacci'
        ActionType: 'actionlib_tutorials/Fibonacci'
    ExecuteGoalFcn: @fibonacciExecution
        DataFormat: 'struct'
```

Create action client and send a goal to the server to calculate the Fibonacci sequence up to 10 terms past the first two terms, `0` and `1`. Display the result sequence.

```
client = rosactionclient("/fibonacci","actionlib_tutorials/Fibonacci",DataFormat="struct");
goal = rosmessage(client);
goal.Order = int32(10);
```

```
result = sendGoalAndWait(client,goal);
result.Sequence
```

```
ans = 12x1 int32 column vector

    0
    1
    1
    2
    3
    5
    8
   13
   21
   34
      ⋮
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_50700 with NodeURI http://bat6234win64:63032/ and M
Shutting down ROS master on http://172.30.131.134:51566.
```

Supporting Functions

The callback function `fibbonacciExecution` is executed every time the server receives a goal execution request from the client. This function checks if the goal has been preempted, executes the goal and sends feedback to the client during goal execution.

```
function [result,success] = fibonacciExecution(src,goal,defaultFeedback,defaultResult)

    % Initialize variables
    success = true;
    result = defaultResult;
    feedback = defaultFeedback;
    feedback.Sequence = int32([0 1]);

    for k = 1:goal.Order
        % Check that the client has not canceled or sent a new goal
        if isPreemptRequested(src)
            success = false;
            break
        end

        % Send feedback to the client periodically
        feedback.Sequence(end+1) = feedback.Sequence(end-1) + feedback.Sequence(end);
        sendFeedback(src,feedback)

        % Pause to allow time to complete other callbacks (like client feedback)
        pause(0.2)
    end

    if success
        result.Sequence = feedback.Sequence;
    end
```

```
end
```

Input Arguments

`server` — ROS action server
`SimpleActionServer` object handle

ROS action server, specified as a `SimpleActionServer` object handle.

Output Arguments

`status` — Status of goal preemption
`logical` scalar

Status of goal preemption, retuned as a `logical` scalar. If the goal has been preempted, the function returns the status as `true`.

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rosactionserver` | `rosActionServerExecuteGoalFcn`

sendFeedback

Send feedback to action client during goal execution

Syntax

```
sendFeedback(server,feedbackMsg)
```

Description

`sendFeedback(server,feedbackMsg)` sends the feedback message, `feedbackMsg`, to the action client that sent the goal currently being executed by the action server, `server`.

Examples

Create a ROS Action Server and Execute a Goal

This example shows how to create a ROS action server, connect an action client to it, receive goal, and execute it.

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6875 seconds.
Initializing ROS master on http://172.30.131.134:51566.
Initializing global node /matlab_global_node_50700 with NodeURI http://bat6234win64:63032/ and Ma
```

Set up an action server for calculating Fibonacci sequence. Use structures for the ROS message data format. Use `fibbonacciExecution` on page 4-9 function as the callback.

```
cb = @fibonacciExecution;
server = rosactionserver("/fibonacci","actionlib_tutorials/Fibonacci",ExecuteGoalFcn=cb,DataForma
```

```
server = 
  SimpleActionServer with properties:

        ActionName: '/fibonacci'
        ActionType: 'actionlib_tutorials/Fibonacci'
    ExecuteGoalFcn: @fibonacciExecution
        DataFormat: 'struct'
```

Create action client and send a goal to the server to calculate the Fibonacci sequence up to 10 terms past the first two terms, `0` and `1`. Display the result sequence.

```
client = rosactionclient("/fibonacci","actionlib_tutorials/Fibonacci",DataFormat="struct");
goal = rosmessage(client);
goal.Order = int32(10);
result = sendGoalAndWait(client,goal);
result.Sequence
```

```
ans = 12x1 int32 column vector

    0
    1
    1
    2
    3
    5
    8
   13
   21
   34
      ⋮
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_50700 with NodeURI http://bat6234win64:63032/ and M
Shutting down ROS master on http://172.30.131.134:51566.
```

Supporting Functions

The callback function `fibbonacciExecution` is executed every time the server receives a goal execution request from the client. This function checks if the goal has been preempted, executes the goal and sends feedback to the client during goal execution.

```
function [result,success] = fibonacciExecution(src,goal,defaultFeedback,defaultResult)

    % Initialize variables
    success = true;
    result = defaultResult;
    feedback = defaultFeedback;
    feedback.Sequence = int32([0 1]);

    for k = 1:goal.Order
        % Check that the client has not canceled or sent a new goal
        if isPreemptRequested(src)
            success = false;
            break
        end

        % Send feedback to the client periodically
        feedback.Sequence(end+1) = feedback.Sequence(end-1) + feedback.Sequence(end);
        sendFeedback(src,feedback)

        % Pause to allow time to complete other callbacks (like client feedback)
        pause(0.2)
    end

    if success
        result.Sequence = feedback.Sequence;
    end
```

```
end
```

Input Arguments

`server` — ROS action server
`SimpleActionServer` object handle

ROS action server, specified as a `SimpleActionServer` object handle.

`feedbackMsg` — Feedback message for action client
ROS message

Feedback message for action client, specified as a ROS message. The type and format of the `feedbackMsg` must match the `ActionType` and `DataFormat` properties of `server`, respectively.

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`rosactionserver` | `getFeedbackMessage` | `rosActionServerExecuteGoalFcn`

rosActionServerExecuteGoalFcn

Return function handle for ROS action server callback

Syntax

```
cb = rosActionServerExecuteGoalFcn
cb = rosActionServerExecuteGoalFcn(Name=Value)
```

Description

`rosActionServerExecuteGoalFcn` provides a predefined callback framework for use as the goal execution callback in a ROS action server. The callback framework is a set of callback functions, one for each of these tasks the server must carry out during goal execution:

- Check if the goal is reached
- Execute items in every iteration towards the goal
- Construct feedback message for the action client
- Construct result message if the goal is preempted
- Construct result message if the goal is reached successfully

You can specify custom functions for these tasks by using the respective name-value arguments of `rosActionServerExecuteGoalFcn`.

`cb = rosActionServerExecuteGoalFcn` returns a function handle, `cb`, with a predefined callback framework for action server goal execution. You can specify `cb` as value for the `ExecuteGoalFcn` name-value argument when you create the `rosactionserver` object. When you use the function handle from this syntax in the action server, the callback immediately indicates that the goal has been reached and returns the default result message.

`cb = rosActionServerExecuteGoalFcn(Name=Value)` specifies additional options using one or more name-value arguments. To customize the behavior of the predefined callback framework, specify handles of custom functions using the corresponding name-value arguments. The custom functions must have two input arguments: a shared object containing `UserData` as the first, and an appropriate ROS message as the second. Most functions must also provide appropriate output. For more information about each function signature, see Name-Value Arguments on page 4-13.

Examples

Create Custom Callback for a ROS Action Server Using the Predefined Callback Framework

This example shows how to create a custom callback for a ROS action server using `rosActionServerExecuteGoalFcn`, which provides a customizable predefined callback framework.

Connect to a ROS network.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.6975 seconds.
Initializing ROS master on http://172.30.131.134:52921.
Initializing global node /matlab_global_node_29865 with NodeURI http://bat6234win64:57770/ and Ma
```

Set up an action server callback for calculating the Fibonacci sequence using `rosActionServerExecuteGoalFcn`. Specify the custom callback functions for the tasks in the callback framework. All the callback functions use a shared object to store data. For definition of these custom functions, see Supporting Functions on page 4-12.

```
% Store the first two terms 0 and 1 in shared object
fibSequence = int32([0 1]);
% Create the callback
cb = rosActionServerExecuteGoalFcn(IsGoalReachedFcn=@isGoalReached,...
        StepExecutionFcn=@nextFibNumber,...
        CreateFeedbackFcn=@assignUserDataToMessage,...
        CreateSuccessfulResultFcn=@assignUserDataToMessage,...
        StepDelay=0.2,...
        UserData=fibSequence);
```

Use the created custom callback, `cb` and set up an action server for calculating Fibonacci sequence. Use structures for the ROS message data format.

```
server = rosactionserver("/fibonacci","actionlib_tutorials/Fibonacci",ExecuteGoalFcn=cb,DataForma
```

Create action client and send a goal to the server, which calculates the first 10 terms in the Fibonacci sequence. Display the result sequence.

```
client = rosactionclient("/fibonacci","actionlib_tutorials/Fibonacci",DataFormat="struct");
goal = rosmessage(client);
goal.Order = int32(10);
result = sendGoalAndWait(client,goal);
result.Sequence
```

```
ans = 10x1 int32 column vector

    0
    1
    1
    2
    3
    5
    8
   13
   21
   34
```

Shut down ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_29865 with NodeURI http://bat6234win64:57770/ and M
Shutting down ROS master on http://172.30.131.134:52921.
```

Supporting Functions

The function `isGoalReached` checks whether the goal is reached. In this case, it checks whether the number of terms in the calculated Fibonacci sequence exceeds the goal from the client.

```
function status = isGoalReached(sharedObj,goal)
    status = numel(sharedObj.UserData) >= goal.Order;
end
```

The function `nextFibNumber` is the step execution function that calculates the next term in the sequence in every iteration towards goal execution.

```
function nextFibNumber(sharedObj,~)
    sharedObj.UserData(end+1) = sharedObj.UserData(end-1) + sharedObj.UserData(end);
end
```

The function `assignUserDataToMessage` assigns the current sequence to the appropriate field in the result message. In this specific case of Fibonacci action, the feedback message also uses the same field, `Sequence` as the result message. Hence, this function can be used for both creating a feedback message and result message to the client.

```
function msg = assignUserDataToMessage(sharedObj,msg)
    msg.Sequence = sharedObj.UserData;
end
```

Input Arguments

Name-Value Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `StepDelay=0.01`

`IsGoalReachedFcn` — Callback function to determine if goal is reached
function handle

Callback function to determine if the goal is reached, specified as a function handle. In the default framework, this function always returns `true`. When specifying the handle for a custom function, the function must have two input arguments: a shared object containing `UserData` as the first, and the goal message as the second. This is an example function header signature:

```
function atGoal = isGoalReached(sharedObj,goalMsg)
```

You can use the data or resources in `sharedObj.UserData` to check the current state, determine whether the goal is reached, and return `true` or `false` appropriately.

Example: `@isGoalReached`

Data Types: `function_handle`

`StepExecutionFcn` — Callback function to progress toward goal each iteration
empty (default) | function handle

Callback function to progress toward the goal each iteration, specified as a function handle. In the default framework, `StepExecutionFcn` is empty, and no step function will be executed. When specifying the handle for a custom function, the function must have two input arguments: a shared object containing `UserData` as the first, and the goal message as the second. This is an example function header signature:

```
function stepExecution(sharedObj,goalMsg)
```

You can use the data or resources in `sharedObj.UserData` as required to progress towards the goal.

Example: `@stepExecution`

Data Types: `function_handle`

`CreateFeedbackFcn` — Callback function to construct feedback message each iteration
empty (default) | function handle

Callback function to construct a feedback message each iteration for the action client, specified as a function handle. In the default framework, `CreateFeedbackFcn` is empty and no feedback will be sent to the client. When specifying the handle for a custom function, the function must have two input arguments: a shared object containing `UserData` as the first, and the goal message as the second. This is an example function header signature:

```
function feedback = createFeedback(sharedObj,defaultFeedbackMsg)
```

You can use the data or resources in `sharedObj.UserData` as required to construct the feedback message to send to the action client.

Example: `@createFeedback`

Data Types: `function_handle`

`CreatePreemptedResultFcn` — Callback function to construct result message if goal is preempted
function handle

Callback function to construct the result message if the goal is preempted, specified as a function handle. The result message is then sent to the action client. In the default framework, this function always returns the default result message. When specifying the handle for a custom function, the function must have two input arguments: a shared object containing `UserData` as the first, and the goal message as the second. This is an example function header signature:

```
function result = createPreemptedResult(sharedObj,defaultResultMsg)
```

You can use the data or resources in `sharedObj.UserData` as required to construct the result message reflecting the incomplete goal execution.

Example: `@createPreemptedResult`

Data Types: `function_handle`

`CreateSuccessfulResultFcn` — Callback function to construct result message if goal is reached successfully
function handle

Callback function to construct the result message if the goal is reached successfully, specified as a function handle. The result message is then sent to the action client. In the default framework, this function always returns the default result message. When specifying the handle for a custom function, the function must have two input arguments: a shared object containing `UserData` as the first, and the goal message as the second. This is an example function header signature:

```
function result = createSuccessfulResult(sharedObj,defaultResultMsg)
```

You can use the data or resources in `sharedObj.UserData` as required to construct the result message that reflects successful goal execution.

Example: `@createSuccessfulResult`

Data Types: `function_handle`

`StepDelay` — Number of seconds to pause each iteration
0.01 seconds (default) | nonnegative scalar

Number of seconds to pause each iteration, specified as a nonnegative scalar. Provide a nonzero value to allow:

- Execution of other callbacks
- ROS action client to react to the received feedback

Example: 0.1

`UserData` — Data for use and modification during goal execution shared by all callbacks
`[]` (default) | scalar | array | structure

Data for use and modification during goal execution shared by all callbacks, specified as a scalar, array, or a structure. This data is stored in an object passed to all the callbacks in the framework. This enables all tasks to share the same data during goal execution, and any modifications made during one task are reflected in subsequent tasks.

Example: `eye(3)`

Output Arguments

`cb` — Callback function
function handle

Callback for use as the goal execution callback in a ROS action server, returned as a function handle. You can specify `cb` as the value of the `ExecuteGoalFcn` name-value argument when you create a `rosactionserver` object.

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The `UserData` specified for the callback created using `rosActionServerExecuteGoalFcn` function must be a 1-D array.

See Also

`rosactionserver` | `getFeedbackMessage` | `isPreemptRequested` | `sendFeedback`

Topics
"ROS Actions Overview"

waitForServer

Wait for ROS 2 action server to be ready for sending goals

Syntax

```
waitForServer(client)
waitForServer(client,Timeout=timeoutperiod)
[status,statustext] = waitForServer( ___ )
```

Description

`waitForServer(client)` waits until the action server is started up and available to send goals. The `IsServerConnected` property of the `ros2actionClient` object shows the status of the server connection. Press **Ctrl+C** to cancel the wait.

`waitForServer(client,Timeout=timeoutperiod)` specifies a timeout period in seconds using the name-value argument `Timeout=timeoutperiod`. If the action server does not start up in the timeout period, this function displays an error and lets MATLAB continue running the current program. The default value of `inf` prevents MATLAB from running the current program until the action client receives a response.

`[status,statustext] = waitForServer( ___ )` returns a `status` indicating whether the action server is available, and a `statustext` that captures additional information about the `status`, using any of the arguments from the previous syntaxes. If the server is not available within the `Timeout`, `status` will be `false`, and this function will not display an error.

Examples

Set Up ROS 2 Action Client and Execute an Action

This example shows how to create a ROS 2 action client and execute the action. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

List the actions available on the network. The `/fibonacci` action must be on the list.

```
ros2 action list
```

```
/fibonacci
```

Get the action type for the `/fibonacci` action.

```
ros2 action type /fibonacci
```

```
action_tutorials_interfaces/Fibonacci
```

Create a ROS 2 node.

```
node = ros2node("/node_1");
```

Create an action client by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
```

Wait for the action client to connect to the server.

```
status = waitForServer(client)
```

```
status = logical
   1
```

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. Set the order to an `int32` value of 8. If the input requires a 1-D array, set it as a column vector.

```
goalMsg.order = int32(8);
```

Send Goal and Execute Action

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response and the final result using the name-value arguments.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback,ResultFcn=@helperRes
```

Send the goal to the action server using the `sendGoal` function. Specify the callback options. During goal execution, you see outputs from the feedback and result callbacks displayed on the MATLAB® command window.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
```

```
Goal with GoalUUID 3d10ab880f960666fde5666f45f621a accepted by server, waiting for result!
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5  8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Full sequence result for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8  13
```

Get the status of goal execution.

```
exStatus = getStatus(client,goalHandle)
```

```
exStatus = int8
    2
```

Get the result using the action client and goal handle inputs. Display the result. The `getResult` function returns the sequence as a column vector.

```
resultMsg = getResult(client,goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Alternatively, you can only use the goal handle as input to get the result.

```
resultMsg = getResult(goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
    seq = feedbackMsg.partial_sequence;
    disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperResultCallback` defines the callback function to execute when the client receives the result message from the action server.

```
function helperResultCallback(goalHandle,wrappedResultMsg)
    seq = wrappedResultMsg.result.sequence;
    disp(['Full sequence result for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order `8`. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
```

```
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);
```

```
Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
```

```
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Input Arguments

`client` — ROS 2 action client
`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

Output Arguments

`status` — Status of the action server start up
`logical` scalar

Status of the action server start up, returned as a `logical` scalar. If the server is not available within the timeout period, `status` will be `false`.

Note Use the `status` output argument when you use waitForServer for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

`statustext` — Status text associated with the action server status
character vector

Status text associated with the action server status, returned as one of the following:

- `'success'` — The server is available.
- `'input'` — The input to the function is invalid.
- `'timeout'` — The server did not become available before the timeout period expired.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ros2actionclient` | `sendGoal` | `getResult` | `getStatus` | `ros2ActionSendGoalOptions` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait` | `cancelAllGoals` | `cancelAllGoalsAndWait` | `ActionClientGoalHandle`

sendGoal

Send goal message to ROS 2 action server

Syntax

```
goalHandle = sendGoal(client,goalMsg)
goalHandle = sendGoal(client,goalMsg,callbackOptions)
```

Description

`goalHandle = 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 the goal handle object, `goalHandle` immediately.

`goalHandle = sendGoal(client,goalMsg,callbackOptions)` specifies customized functions for goal response, feedback, and result callbacks using the structure `callbackOptions`. Use the `ros2ActionSendGoalOptions` function to customize callbacks and get the associated `callbackOptions` structure.

Examples

Set Up ROS 2 Action Client and Execute an Action

This example shows how to create a ROS 2 action client and execute the action. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

List the actions available on the network. The `/fibonacci` action must be on the list.

```
ros2 action list
```

```
/fibonacci
```

Get the action type for the `/fibonacci` action.

```
ros2 action type /fibonacci
```

```
action_tutorials_interfaces/Fibonacci
```

Create a ROS 2 node.

```
node = ros2node("/node_1");
```

Create an action client by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
```

Wait for the action client to connect to the server.

```
status = waitForServer(client)
```

```
status = logical
   1
```

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. Set the order to an `int32` value of 8. If the input requires a 1-D array, set it as a column vector.

```
goalMsg.order = int32(8);
```

Send Goal and Execute Action

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response and the final result using the name-value arguments.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback,ResultFcn=@helperRes
```

Send the goal to the action server using the `sendGoal` function. Specify the callback options. During goal execution, you see outputs from the feedback and result callbacks displayed on the MATLAB® command window.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
```

```
Goal with GoalUUID 3d10ab880f960666fde5666f45f621a accepted by server, waiting for result!
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5  8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Full sequence result for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8  13
```

Get the status of goal execution.

```
exStatus = getStatus(client,goalHandle)
```

```
exStatus = int8
    2
```

Get the result using the action client and goal handle inputs. Display the result. The `getResult` function returns the sequence as a column vector.

```
resultMsg = getResult(client,goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Alternatively, you can only use the goal handle as input to get the result.

```
resultMsg = getResult(goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
    seq = feedbackMsg.partial_sequence;
    disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperResultCallback` defines the callback function to execute when the client receives the result message from the action server.

```
function helperResultCallback(goalHandle,wrappedResultMsg)
    seq = wrappedResultMsg.result.sequence;
    disp(['Full sequence result for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order `8`. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);
```

```
Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Input Arguments

`client` — ROS 2 action client

`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

`goalMsg` — ROS 2 action goal message

structure

ROS 2 action goal message, specified as a message structure. Update this message with your goal details and send it to the ROS 2 action client using the `sendGoal` function.

`callbackOptions` — Callback options framework
structure

Callback options framework, specified as a structure. The fields of the structure specify the function handles for goal response, feedback and result callbacks along with the required data for each of the callbacks. Use the `ros2ActionSendGoalOptions` function to define custom functions for each of the callbacks and get the callback options structure.

Output Arguments

`goalHandle` — Action client goal handle
`ActionClientGoalHandle` object

Action client goal handle, returned as an `ActionClientGoalHandle` object. You can use the properties of the `ActionClientGoalHandle` object to track the goal execution asynchronously. To get the goal execution result synchronously, use the `getResult` object function.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The callback functions used to send and cancel goals must be specified while creating the `ros2actionclient` object using these name-value arguments.

  - `'SendGoalOptions'` — Specify a cell array of callback options structures for sending goals that you create using the `ros2ActionSendGoalOptions` function.
  - `'CancelFcn'` — Specify the cancel response callback that you use with the `cancelGoal` function.
  - `'CancelAllFcn'` — Specify the cancel response callback that you use with the `cancelAllGoals` function.
  - `'CancelBeforeFcn'` — Specify the cancel response callback that you use with the `cancelGoalsBefore` function.

  Note that these arguments are used for definition of `ros2actionclient` only and must be specified again at the time of usage in the respective function.

```
callbackOpts1 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback,ResultFcn=@glResult);
callbackOpts2 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback2,ResultFcn=@glResult2);
[client] = ros2actionClient(node,"my_action","my_action_type",SendGoalOptions={callbackOpts1,c
goalMsg = ros2message(client);
goalHandle = sendGoal(client,goalMsg,callbackOpts1);
cancelGoal(client,goalHandle,CancelFcn=@cancelGoalCB);
```

See Also

`ros2actionclient` | `ros2ActionSendGoalOptions` | `getResult` | `getStatus` | `ActionClientGoalHandle` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait` | `cancelAllGoals` | `cancelAllGoalsAndWait`

cancelGoal

Cancel specific goal sent by ROS 2 action client

Syntax

```
cancelGoal(client,goalHandle)
cancelGoal(client,goalHandle,CancelFcn=@cancelCallback)
```

Description

`cancelGoal(client,goalHandle)` sends a cancel request for the goal associated with the goal handle object `goalHandle`, sent by the ROS 2 action client, `client`. The function does not wait for the goal to be cancelled and returns immediately.

`cancelGoal(client,goalHandle,CancelFcn=@cancelCallback)` specifies a callback function to execute when the cancel response reaches the ROS 2 action client using the name-value argument `CancelFcn=@cancelCallback`. The callback function must have two input arguments: a `ros2ActionGoalHandle` object associated with the goal as the first, and the received cancel response message as the second. You can provide additional data to the callback using multiple subsequent input arguments. This is indicated by the `varargin` variable. This is an example function header signature:

```
function cancelCallback(goalHandle,cancelMsg,varargin)
```

Note To supply additional data to the callback function, you can specify one additional input argument. You must include both the callback function and the additional input argument as elements of a cell array as elements of a cell array while defining the `CancelFcn` name-value argument. For example:

```
cancelGoal(client,goalHandle,CancelFcn={@cancelCallback,4.5})
```

Examples

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order `8`. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);

Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)

cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Input Arguments

`client` — ROS 2 action client
`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

`goalHandle` — Action client goal handle
`ActionClientGoalHandle` object

Action client goal handle, specified as an `ActionClientGoalHandle` object. You can use the properties of the `ActionClientGoalHandle` object to track the goal execution asynchronously. To get the goal execution result synchronously, use the `getResult` object function.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The callback functions used to send and cancel goals must be specified while creating the `ros2actionclient` object using these name-value arguments.

  - `'SendGoalOptions'` — Specify a cell array of callback options structures for sending goals that you create using the `ros2ActionSendGoalOptions` function.
  - `'CancelFcn'` — Specify the cancel response callback that you use with the `cancelGoal` function.
  - `'CancelAllFcn'` — Specify the cancel response callback that you use with the `cancelAllGoals` function.
  - `'CancelBeforeFcn'` — Specify the cancel response callback that you use with the `cancelGoalsBefore` function.

  Note that these arguments are used for definition of `ros2actionclient` only and must be specified again at the time of usage in the respective function.

  ```
  callbackOpts1 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback,ResultFcn=@glResult);
  callbackOpts2 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback2,ResultFcn=@glResult2);
  [client] = ros2actionClient(node,"my_action","my_action_type",SendGoalOptions={callbackOpts1,c
  goalMsg = ros2message(client);
  goalHandle = sendGoal(client,goalMsg,callbackOpts1);
  cancelGoal(client,goalHandle,CancelFcn=@cancelGoalCB);
  ```

See Also

`ros2actionclient` | `sendGoal` | `getResult` | `getStatus` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait` | `cancelAllGoals` | `cancelAllGoalsAndWait` | `ActionClientGoalHandle`

cancelGoalAndWait

Cancel specific goal sent by ROS 2 action client and wait for cancel response

Syntax

```
cancelResponse = cancelGoalAndWait(client,goalHandle)
cancelResponse = cancelGoalAndWait(client,goalHandle,Timeout=timeoutperiod)
[cancelResponse,status,statustext] = cancelGoalAndWait( ___ )
```

Description

`cancelResponse = cancelGoalAndWait(client,goalHandle)` sends a cancel request for the goal associated with the goal handle object `goalHandle`, sent by the ROS 2 action client, `client`. This syntax blocks MATLAB from running the current program until the action server returns the cancel response `cancelResponse`. Press **Ctrl+C** to cancel the wait.

`cancelResponse = cancelGoalAndWait(client,goalHandle,Timeout=timeoutperiod)` specifies a timeout period in seconds using the name-value argument `Timeout=timeoutperiod`. If the action server does not return the cancel response in the timeout period, this function displays an error and lets MATLAB continue running the current program. The default value of `inf` prevents MATLAB from running the current program until the action client receives a cancel response.

`[cancelResponse,status,statustext] = cancelGoalAndWait( ___ )` returns a `status` indicating whether the action client received the cancel response, and a `statustext` that captures additional information about the `status`, using any of the arguments from the previous syntaxes. If the server is not available within the `Timeout`, `status` will be `false`, and this function will not display an error.

Examples

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order 8. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);

Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)

cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Input Arguments

`client` — ROS 2 action client

`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

`goalHandle` — Action client goal handle

`ActionClientGoalHandle` object

Action client goal handle, specified as an `ActionClientGoalHandle` object. You can use the properties of the `ActionClientGoalHandle` object to track the goal execution asynchronously. To get the goal execution result synchronously, use the `getResult` object function.

Output Arguments

`cancelResponse` — Cancel response received from the action server
structure

Cancel response received from the action server, returned as a ROS 2 message structure.

`status` — Status of the cancel response receipt
`logical` scalar

Status of the cancel response receipt, returned as a `logical` scalar. If the action client does not receive a cancel response within the timeout period, `status` will be `false`.

Note Use the `status` output argument when you use cancelGoalAndWait for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

`statustext` — Status text associated with the cancel response receipt
character vector

Status text associated with the cancel response receipt, returned as one of the following:

- `'success'` — The cancel response was successfully received.
- `'input'` — The input to the function is invalid.
- `'timeout'` — The cancel response was not received before the timeout period expired.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ros2actionclient` | `sendGoal` | `getResult` | `getStatus` | `ActionClientGoalHandle` | `cancelGoal` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait` | `cancelAllGoals` | `cancelAllGoalsAndWait`

cancelGoalsBefore

Cancel goals sent by ROS 2 action client before timestamp

Syntax

```
cancelGoalsBefore(client,timestamp)
cancelGoalsBefore(client,timestamp,CancelFcn=@cancelCallback)
```

Description

`cancelGoalsBefore(client,timestamp)` sends a cancel request for all the active goals sent by the ROS 2 action client `client` at or before the specified timestamp, `timestamp`. The function does not wait for the goals to be cancelled and returns immediately.

`cancelGoalsBefore(client,timestamp,CancelFcn=@cancelCallback)` specifies a callback function to execute when the cancel response reaches the ROS 2 action client using the name-value argument `CancelFcn=@cancelCallback`. The callback function must have the received cancel response message as the first input argument. You can provide additional data to the callback using multiple subsequent input arguments. This is indicated by the `varargin` variable. This is an example function header signature:

```
function cancelCallback(cancelMsg,varargin)
```

Note To supply additional data to the callback function, you can specify one additional input argument. You must include both the callback function and the additional input argument as elements of a cell array while defining the `CancelFcn` name-value argument. For example:

```
cancelgoalsBefore(client,goalHandle,CancelFcn={@cancelCallback,4.5})
```

Examples

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order 8. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);

Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)

cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Input Arguments

`client` — ROS 2 action client

`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

`timestamp` — Timestamp at or before which the goals sent by the action client must be cancelled

`builtin_interfaces/Time` message structure

Timestamp at or before which the goals sent by the action client must be cancelled, specified as a `builtin_interfaces/Time` message structure. You can use the `ros2time` function to create the message structure with a desired timestamp.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The callback functions used to send and cancel goals must be specified while creating the `ros2actionclient` object using these name-value arguments.

  - `'SendGoalOptions'` — Specify a cell array of callback options structures for sending goals that you create using the `ros2ActionSendGoalOptions` function.
  - `'CancelFcn'` — Specify the cancel response callback that you use with the `cancelGoal` function.
  - `'CancelAllFcn'` — Specify the cancel response callback that you use with the `cancelAllGoals` function.
  - `'CancelBeforeFcn'` — Specify the cancel response callback that you use with the `cancelGoalsBefore` function.

  Note that these arguments are used for definition of `ros2actionclient` only and must be specified again at the time of usage in the respective function.

  ```
  callbackOpts1 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback,ResultFcn=@glResult);
  callbackOpts2 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback2,ResultFcn=@glResult2);
  [client] = ros2actionClient(node,"my_action","my_action_type",SendGoalOptions={callbackOpts1,c
  goalMsg = ros2message(client);
  goalHandle = sendGoal(client,goalMsg,callbackOpts1);
  cancelGoal(client,goalHandle,CancelFcn=@cancelGoalCB);
  ```

See Also

`ros2actionclient` | `sendGoal` | `getResult` | `getStatus` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBeforeAndWait` | `cancelAllGoals` | `cancelAllGoalsAndWait`

cancelGoalsBeforeAndWait

Cancel goals sent by ROS 2 action client before timestamp and wait for cancel response

Syntax

```
cancelResponse = cancelGoalsBeforeAndWait(client,timestamp)
cancelResponse = cancelGoalsBeforeAndWait(client,
timestamp,Timeout=timeoutperiod)
[cancelResponse,status,statustext] = cancelGoalsBeforeAndWait( ___ )
```

Description

`cancelResponse = cancelGoalsBeforeAndWait(client,timestamp)` sends a cancel request for all the active goals sent by the ROS 2 action client `client` at or before the specified timestamp, `timestamp`. This syntax blocks MATLAB from running the current program until the action server returns the cancel response `cancelResponse`. Press **Ctrl+C** to cancel the wait.

`cancelResponse = cancelGoalsBeforeAndWait(client, timestamp,Timeout=timeoutperiod)` specifies a timeout period in seconds using the name-value argument `Timeout=timeoutperiod`. If the action server does not return the cancel response in the timeout period, this function displays an error and lets MATLAB continue running the current program. The default value of `inf` prevents MATLAB from running the current program until the action client receives a cancel response.

`[cancelResponse,status,statustext] = cancelGoalsBeforeAndWait( ___ )` returns a `status` indicating whether the action client received the cancel response, and a `statustext` that captures additional information about the `status`, using any of the arguments from the previous syntaxes. If the server is not available within the `Timeout`, `status` will be `false`, and this function will not display an error.

Examples

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order 8. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);

Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)

cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Input Arguments

`client` — ROS 2 action client

`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

`timestamp` — Timestamp at or before which the goals sent by the action client must be cancelled

`builtin_interfaces/Time` message structure

Timestamp at or before which the goals sent by the action client must be cancelled, specified as a `builtin_interfaces/Time` message structure. You can use the `ros2time` function to create the message structure with a desired timestamp.

Output Arguments

`cancelResponse` — Cancel response received from the action server
structure

Cancel response received from the action server, returned as a ROS 2 message structure.

`status` — Status of the cancel response receipt
`logical` scalar

Status of the cancel response receipt, returned as a `logical` scalar. If the action client does not receive a cancel response within the timeout period, `status` will be `false`.

Note Use the `status` output argument when you use cancelGoalsBeforeAndWait for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

`statustext` — Status text associated with the cancel response receipt
character vector

Status text associated with the cancel response receipt, returned as one of the following:

- `'success'` — The cancel response was successfully received.
- `'input'` — The input to the function is invalid.
- `'timeout'` — The cancel response was not received before the timeout period expired.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ros2actionclient` | `sendGoal` | `getResult` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelAllGoals` | `cancelAllGoalsAndWait`

cancelAllGoals

Cancel all active goals sent by ROS 2 action client

Syntax

```
cancelAllGoals(client)
cancelAllGoals(client,CancelFcn=@cancelCallback)
```

Description

`cancelAllGoals(client)` sends a cancel request for all the active goals sent by the ROS 2 action client, `client`. The function does not wait for the goals to be cancelled and returns immediately.

`cancelAllGoals(client,CancelFcn=@cancelCallback)` specifies a callback function to execute when the cancel response reaches the ROS 2 action client using the name-value argument `CancelFcn=@cancelCallback`. The callback function must have the received cancel response message as the first input argument. You can provide additional data to the callback using multiple subsequent input arguments. This is indicated by the `varargin` variable. This is an example function header signature:

```
function cancelCallback(cancelMsg,varargin)
```

Note To supply additional data to the callback function, you can specify one additional input argument. You must include both the callback function and the additional input argument as elements of a cell array while defining the `CancelFcn` name-value argument. For example:

```
cancelAllGoals(client,goalHandle,CancelFcn={@cancelCallback,4.5})
```

Examples

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order 8. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);
```

```
Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Input Arguments

`client` — ROS 2 action client

`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- The callback functions used to send and cancel goals must be specified while creating the `ros2actionclient` object using these name-value arguments.

  - `'SendGoalOptions'` — Specify a cell array of callback options structures for sending goals that you create using the `ros2ActionSendGoalOptions` function.
  - `'CancelFcn'` — Specify the cancel response callback that you use with the `cancelGoal` function.
  - `'CancelAllFcn'` — Specify the cancel response callback that you use with the `cancelAllGoals` function.
  - `'CancelBeforeFcn'` — Specify the cancel response callback that you use with the `cancelGoalsBefore` function.

  Note that these arguments are used for definition of `ros2actionclient` only and must be specified again at the time of usage in the respective function.

  ```
  callbackOpts1 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback,ResultFcn=@glResult);
  callbackOpts2 = ros2ActionSendGoalOptions(FeedbackFcn=@glfeedback2,ResultFcn=@glResult2);
  [client] = ros2actionClient(node,"my_action","my_action_type",SendGoalOptions={callbackOpts1,c
  goalMsg = ros2message(client);
  goalHandle = sendGoal(client,goalMsg,callbackOpts1);
  cancelGoal(client,goalHandle,CancelFcn=@cancelGoalCB);
  ```

See Also
`ros2actionclient` | `sendGoal` | `getResult` | `getStatus` | `cancelAllGoalsAndWait` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait`

cancelAllGoalsAndWait

Cancel all active goals sent by ROS 2 action client and wait for cancel response

Syntax

```
cancelResponse = cancelAllGoalsAndWait(client)
cancelResponse = cancelAllGoalsAndWait(client,Timeout=timeoutperiod)
[cancelResponse,status,statustext] = cancelAllGoalsAndWait( ___ )
```

Description

`cancelResponse = cancelAllGoalsAndWait(client)` sends a cancel request for all the active goals sent by the ROS 2 action client, `client`. This syntax blocks MATLAB from running the current program until the action server returns the cancel response `cancelResponse`. Press **Ctrl+C** to cancel the wait.

`cancelResponse = cancelAllGoalsAndWait(client,Timeout=timeoutperiod)` specifies a timeout period in seconds using the name-value argument `Timeout=timeoutperiod`. If the action server does not return the cancel response in the timeout period, this function displays an error and lets MATLAB continue running the current program. The default value of `inf` prevents MATLAB from running the current program until the action client receives a cancel response.

`[cancelResponse,status,statustext] = cancelAllGoalsAndWait( ___ )` returns a `status` indicating whether the action client received the cancel response, and a `statustext` that captures additional information about the `status`, using any of the arguments from the previous syntaxes. If the server is not available within the `Timeout`, `status` will be `false`, and this function will not display an error.

Examples

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for /fibonacci action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an int32 value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an int32 value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the sendGoal function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order 8. Use the goal handle object associated with that goal as input to the cancelGoal function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order `8`. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);
```

```
Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Input Arguments

`client` — ROS 2 action client

`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

Output Arguments

`cancelResponse` — Cancel response received from the action server

structure

Cancel response received from the action server, returned as a ROS 2 message structure.

`status` — Status of the cancel response receipt
`logical` scalar

Status of the cancel response receipt, returned as a `logical` scalar. If the action client does not receive a cancel response within the timeout period, `status` will be `false`.

Note Use the `status` output argument when you use cancelAllGoalsAndWait for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

`statustext` — Status text associated with the cancel response receipt
character vector

Status text associated with the cancel response receipt, returned as one of the following:

- `'success'` — The cancel response was successfully received.
- `'input'` — The input to the function is invalid.
- `'timeout'` — The cancel response was not received before the timeout period expired.

Version History
Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also
`ros2actionclient` | `sendGoal` | `getResult` | `getStatus` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait` | `cancelAllGoals`

ros2ActionSendGoalOptions

Return structure for ROS 2 action client callback

Syntax

```
cb = ros2ActionSendGoalOptions
cb = ros2ActionSendGoalOptions(Name=Value)
```

Description

`ros2ActionSendGoalOptions` provides a predefined callback framework for use as the send goal callback for ROS 2 action client. The callback framework is a set of callback functions, consisting of the tasks the action client must carry out based on these responses from the action server:

- Goal Response
- Feedback Response
- Result message

You can specify custom functions for these tasks by using the respective name-value arguments of `ros2ActionSendGoalOptions`.

`cb = ros2ActionSendGoalOptions` returns a callback options structure `cb`, with a predefined callback framework for action client to send goals. You can specify `cb` as value for the `callbackOptions` input argument when you send goals using the `sendGoal` function. When you use the function handle from this syntax to send goal, `cb` specifies a goal response callback to indicate that the goal has been accepted by the server immediately.

`cb = ros2ActionSendGoalOptions(Name=Value)` specifies additional options using one or more name-value arguments. To customize the behavior of the predefined callback framework, specify handles of custom functions using the corresponding name-value arguments. All custom functions must have the `ros2ActionGoalHandle` object associated with the goal as an input argument. Some functions have additional input arguments that you must specify. For more information about each function signature, see Name-Value Arguments on page 4-68.

Examples

Set Up ROS 2 Action Client and Execute an Action

This example shows how to create a ROS 2 action client and execute the action. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

List the actions available on the network. The `/fibonacci` action must be on the list.

```
ros2 action list
```

```
/fibonacci
```

Get the action type for the `/fibonacci` action.

```
ros2 action type /fibonacci
```

```
action_tutorials_interfaces/Fibonacci
```

Create a ROS 2 node.

```
node = ros2node("/node_1");
```

Create an action client by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
```

Wait for the action client to connect to the server.

```
status = waitForServer(client)
```

```
status = logical
   1
```

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. Set the order to an `int32` value of 8. If the input requires a 1-D array, set it as a column vector.

```
goalMsg.order = int32(8);
```

Send Goal and Execute Action

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response and the final result using the name-value arguments.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback,ResultFcn=@helperRes
```

Send the goal to the action server using the `sendGoal` function. Specify the callback options. During goal execution, you see outputs from the feedback and result callbacks displayed on the MATLAB® command window.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
```

```
Goal with GoalUUID 3d10ab880f960666fde5666f45f621a accepted by server, waiting for result!
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5  8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Full sequence result for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8  13
```

Get the status of goal execution.

```
exStatus = getStatus(client,goalHandle)
```

```
exStatus = int8
    2
```

Get the result using the action client and goal handle inputs. Display the result. The `getResult` function returns the sequence as a column vector.

```
resultMsg = getResult(client,goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Alternatively, you can only use the goal handle as input to get the result.

```
resultMsg = getResult(goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
    seq = feedbackMsg.partial_sequence;
    disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperResultCallback` defines the callback function to execute when the client receives the result message from the action server.

```
function helperResultCallback(goalHandle,wrappedResultMsg)
    seq = wrappedResultMsg.result.sequence;
    disp(['Full sequence result for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

Send and Cancel ROS 2 Action Goals

This example shows how to send and cancel ROS 2 action goals. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

Create a ROS 2 node .

```
node = ros2node("/node_1");
```

Create an action client for `/fibonacci` action by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.Wait for the action client to connect to the server.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
status = waitForServer(client)
```

```
status = logical
   1
```

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback);
```

Send and Cancel Goals

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. Set the order to an `int32` value of 8.

```
goalMsg.order = int32(8);
```

Create a new goal message and set the order to an `int32` value of 10.

```
goalMsg2 = ros2message(client);
goalMsg2.order = int32(10);
```

Send both the goals to the action server using the `sendGoal` function. Specify the same callback options for both goals.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
```

```
Goal with GoalUUID ca8dbca2b8608a6f2add01b298f6930 accepted by server, waiting for result!
Partial sequence feedback for goal ca8dbca2b8608a6f2add01b298f6930 is 0  1  1
Goal with GoalUUID f493913f4acd2224f31145ae74bbc35 accepted by server, waiting for result!
Partial sequence feedback for goal f493913f4acd2224f31145ae74bbc35 is 0  1  1
```

Cancel the specific goal associated with the sequence order 8. Use the goal handle object associated with that goal as input to the `cancelGoal` function, and specify the cancel callback to execute once the client receives the cancel response. This function returns immediately without waiting for the cancel response to arrive.

```
cancelGoal(client,goalHandle,CancelFcn=@helperCancelGoalCallback)
```

```
Goal ca8dbca2b8608a6f2add01b298f6930 is cancelled with return code 0
```

You can wait until the cancel response arrives from the server by using the `cancelGoalAndWait` function. Cancel the goal associated with the sequence order of 10 and wait until the client receives the cancel response.

```
cancelResponse = cancelGoalAndWait(client,goalHandle2)
```

```
cancelResponse = struct with fields:
              MessageType: 'action_msgs/CancelGoalResponse'
               ERROR_NONE: 0
           ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
              return_code: 0
          goals_canceling: [1×1 struct]
```

Cancel Goals Before Timestamp

Send the goal message with sequence order `10`. Note the timestamp in a ROS 2 message by using the `ros2time` function.

```
goalHandle = sendGoal(client,goalMsg2,callbackOpts);
timeStampMsg = ros2time(node,"now");
```

```
Goal with GoalUUID d8268c566b234e8784f0f1a8ec12b2 accepted by server, waiting for result!
Partial sequence feedback for goal d8268c566b234e8784f0f1a8ec12b2 is 0  1  1
```

Then, send a second goal message with sequence order 8. Note the timestamp.

```
goalHandle2 = sendGoal(client,goalMsg,callbackOpts);
timeStampMsg2 = ros2time(node,"now");
```

```
Goal with GoalUUID 9585bff2ba44bf60daa630a952b458be accepted by server, waiting for result!
Partial sequence feedback for goal 9585bff2ba44bf60daa630a952b458be is 0  1  1
```

Cancel the goal sent before the first time stamp using `cancelGoalsBefore` function.

```
cancelGoalsBefore(client,timeStampMsg,CancelFcn=@helperCancelGoalsCallback)
```

```
Goals cancelled with return code 0
```

Use the `cancelGoalsBeforeAndWait` function to cancel the goal sent before second time stamp and wait for the cancel response.

```
cancelResponse = cancelGoalsBeforeAndWait(client,timeStampMsg2)
```

```
cancelResponse = struct with fields:
             MessageType: 'action_msgs/CancelGoalResponse'
              ERROR_NONE: 0
          ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
             return_code: 0
         goals_canceling: [1×1 struct]
```

Cancel All Goals

Cancel all the active goals that the client sent.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
cancelAllGoals(client,CancelFcn=@helperCancelGoalsCallback);
```

```
Goals cancelled with return code 0
```

Cancel all the active goals that the client sent and wait for cancel response.

```
goalHandle2 = sendGoal(client,goalMsg2,callbackOpts);
cancelResponse = cancelAllGoalsAndWait(client)
```

```
cancelResponse = struct with fields:
             MessageType: 'action_msgs/CancelGoalResponse'
              ERROR_NONE: 0
          ERROR_REJECTED: 1
    ERROR_UNKNOWN_GOAL_ID: 2
    ERROR_GOAL_TERMINATED: 3
             return_code: 0
         goals_canceling: [1×1 struct]
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
seq = feedbackMsg.partial_sequence;
disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperCancelGoalCallback` defines the callback function to execute when the client receives a cancel response after canceling a specific goal.

```
function helperCancelGoalCallback(goalHandle,cancelMsg)
code = cancelMsg.return_code;
disp(['Goal ',goalHandle.GoalUUID,' is cancelled with return code ',num2str(code)])
end
```

`helperCancelGoalsCallback` defines the callback function to execute when the client receives a cancel response after canceling a set of goals.

```
function helperCancelGoalsCallback(cancelMsg)
code = cancelMsg.return_code;
```

```
disp(['Goals cancelled with return code ',num2str(code)])
end
```

Input Arguments

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `GoalRespFcn=@glResponse`

`GoalRespFcn` — Callback function called after receiving goal response from the server

function handle

Callback function called after receiving goal response from the server, specified as a function handle. In the default framework, this function specifies a callback to indicate that the goal has been accepted by the server immediately. When specifying the handle for a custom function, the function must have the `ros2ActionGoalHandle` object associated with the goal as the first input argument. This is an example function header signature:

```
function goalRespFcn(goalHandle,varargin)
```

You can provide additional data to the callback using multiple subsequent input arguments. This is indicated by the `varargin` variable.

Example: `@glResponse`

Data Types: `function_handle`

`FeedbackFcn` — Callback function called when receiving feedback response from the server

function handle

Callback function called when receiving feedback response from the server, specified as a function handle. In the default framework, this function specifies a callback to indicate that the goal has been accepted by the server immediately. When specifying the handle for a custom function, the function must have two input arguments: a `ros2ActionGoalHandle` object associated with the goal as the first, and the received feedback message as the second. This is an example function header signature:

```
function feedbackFcn(goalHandle,fbMsg,varargin)
```

You can provide additional data to the callback using multiple subsequent input arguments. This is indicated by the `varargin` variable.

Example: `@glFeedback`

Data Types: `function_handle`

`ResultFcn` — Callback function called when receiving result message from the server

function handle

Callback function called when receiving result message from the server, specified as a function handle. In the default framework, this function specifies a callback to indicate that the goal has been accepted by the server immediately. When specifying the handle for a custom function, the function must have two input arguments: a `ros2ActionGoalHandle` object associated with the goal as the first, and the received wrapped result message as the second. The wrapped result message contains these fields:

- `result` — Received result message
- `code` — Received result code
- `goalUUID` — Unique index of the goal associated with the result

This is an example function header signature:

```
function resultFcn(goalHandle,wrappedResultMsg,varargin)
```

You can provide additional data to the callback using multiple subsequent input arguments. This is indicated by the `varargin` variable.

Example: `@glResult`

Data Types: `function_handle`

Note To supply additional data to callback functions, you can specify one additional input argument. You must include both the callback function and the additional input argument as elements of a cell array while defining the call back options using `ros2ActionSendGoalOptions`. For example:

```
cb = ros2ActionSendGoalOptions(FeedbackFcn={@glFeedback,2.5},ResultFcn={@glResult,5.0});
```

Output Arguments

cb — Callback options framework
structure

Callback options framework, returned as a structure. The fields of the structure specify the function handles for goal response, feedback and result callbacks along with the required data for each of the callbacks. You can use `cb` as value for the `callbackOptions` input argument when you send a goal using the `sendGoal` function.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- When you create callback options to send goals using the `ros2ActionSendGoalOptions` function, any additional input arguments to `GoalRespFcn`, `FeedbackFcn` and `ResultFcn` callbacks are not supported. Only function signature with the goal handle and/or received message as input arguments is supported.

See Also
`ros2actionclient` | `sendGoal` | `getResult` | `getStatus` | `ActionClientGoalHandle` | `waitForServer` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait` | `cancelAllGoals` | `cancelAllGoalsAndWait`

getStatus

Get execution status of specific goal sent by ROS 2 action client

Syntax

```
status = getStatus(client,goalHandle)
```

Description

`status = getStatus(client,goalHandle)` returns the status of execution of the goal specified by `goalHandle`. The goal must be sent by the action client `client`.

Examples

Set Up ROS 2 Action Client and Execute an Action

This example shows how to create a ROS 2 action client and execute the action. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

List the actions available on the network. The `/fibonacci` action must be on the list.

```
ros2 action list
```

```
/fibonacci
```

Get the action type for the `/fibonacci` action.

```
ros2 action type /fibonacci
```

```
action_tutorials_interfaces/Fibonacci
```

Create a ROS 2 node.

```
node = ros2node("/node_1");
```

Create an action client by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
```

Wait for the action client to connect to the server.

```
status = waitForServer(client)
```

```
status = logical
   1
```

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. Set the order to an `int32` value of 8. If the input requires a 1-D array, set it as a column vector.

```
goalMsg.order = int32(8);
```

Send Goal and Execute Action

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response and the final result using the name-value arguments.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback,ResultFcn=@helperRes
```

Send the goal to the action server using the `sendGoal` function. Specify the callback options. During goal execution, you see outputs from the feedback and result callbacks displayed on the MATLAB® command window.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
```

```
Goal with GoalUUID 3d10ab880f960666fde5666f45f621a accepted by server, waiting for result!
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5  8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Full sequence result for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8  13
```

Get the status of goal execution.

```
exStatus = getStatus(client,goalHandle)
```

```
exStatus = int8
    2
```

Get the result using the action client and goal handle inputs. Display the result. The `getResult` function returns the sequence as a column vector.

```
resultMsg = getResult(client,goalHandle);
rosShowDetails(resultMsg)
```

```
ans =
    '
```

```
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Alternatively, you can only use the goal handle as input to get the result.

```
resultMsg = getResult(goalHandle);
rosShowDetails(resultMsg)
```

```
ans = 
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
    seq = feedbackMsg.partial_sequence;
    disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperResultCallback` defines the callback function to execute when the client receives the result message from the action server.

```
function helperResultCallback(goalHandle,wrappedResultMsg)
    seq = wrappedResultMsg.result.sequence;
    disp(['Full sequence result for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

Input Arguments

`client` — ROS 2 action client
`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

`goalHandle` — Action client goal handle
`ActionClientGoalHandle` object

Action client goal handle, specified as an `ActionClientGoalHandle` object.

Output Arguments

`status` — Execution status of associated goal
nonnegative integer

This property is read-only.

Execution status of the associated goal, returned as a nonnegative integer. Each integer denotes a specific status as defined in the `action_msgs/msg/GoalStatus` ROS 2 message definition:

- `0` — Unknown
- `1` — Accepted
- `2` — Executing
- `3` — Canceling
- `4` — Succeeded
- `5` — Canceled
- `6` — Aborted

Data Types: `int8`

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ros2actionclient` | `sendGoal` | `getResult` | `ros2ActionSendGoalOptions` | `waitForServer` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait` | `cancelAllGoals` | `cancelAllGoalsAndWait`

getResult

Get result of specific goal associated with goal handle

Syntax

```
resultMsg = getResult(client,goalHandle)
resultMsg = getResult(goalHandle)
resultMsg = getResult( ___ ,Timeout=timeoutperiod)
[resultMsg,status,statustext] = getResult( ___ )
```

Description

`resultMsg = getResult(client,goalHandle)` returns the result message `resultMsg` associated with the execution of the goal specified by `goalHandle`. The goal must be sent by the action client `client`. This syntax blocks MATLAB from running the current program until the action server provides `resultMsg`. You can Press **Ctrl+C** to cancel the wait.

`resultMsg = getResult(goalHandle)` returns the result message `resultMsg` associated with the execution of the goal specified by `goalHandle`. This syntax blocks MATLAB from running the current program until the action server provides `resultMsg`. You can Press **Ctrl+C** to cancel the wait.

`resultMsg = getResult( ___ ,Timeout=timeoutperiod)` specifies a timeout period in seconds using the name-value argument `Timeout=timeoutperiod`. If the action server does not return the result message in the timeout period, this function displays an error and lets MATLAB continue running the current program. The default value of `inf` prevents MATLAB from running the current program until the action client receives a cancel response.

`[resultMsg,status,statustext] = getResult( ___ )` returns a `status` indicating whether the action client received the result message, and a `statustext` that captures additional information about the `status`, using any of the arguments from the previous syntaxes. If the action server does not return the result message within the `Timeout`, `status` will be `false`, and this function will not display an error.

Examples

Set Up ROS 2 Action Client and Execute an Action

This example shows how to create a ROS 2 action client and execute the action. Action types must be set up beforehand with an action server running. This example uses the `/fibonacci` action. Follow these steps to set up the action server:

1. Create a ROS 2 package with the action definition. For instructions on setting up a `/fibonacci` action, see Creating an Action.
2. Create a ROS 2 package with the action server implementation. For more information on setting up the `/fibonacci` action server, see Writing an Action Server.
3. Use the `ros2genmsg` function for the ROS 2 package with action definition from Step 1, and generate action messages in MATLAB®.

To run the `/fibonacci` action server, use this command on the ROS 2 system:

```
ros2 run action_tutorials_cpp fibonacci_action_server
```

Set Up ROS 2 Action Client

List the actions available on the network. The `/fibonacci` action must be on the list.

```
ros2 action list
```

```
/fibonacci
```

Get the action type for the `/fibonacci` action.

```
ros2 action type /fibonacci
```

```
action_tutorials_interfaces/Fibonacci
```

Create a ROS 2 node.

```
node = ros2node("/node_1");
```

Create an action client by specifying the node, action name, and action type. Set the quality of service (QoS) parameters.

```
[client,goalMsg] = ros2actionclient(node,"fibonacci",...
"action_tutorials_interfaces/Fibonacci", ...
CancelServiceQoS=struct(Depth=200,History="keeplast"), ...
FeedbackTopicQoS=struct(Depth=200,History="keepall"));
```

Wait for the action client to connect to the server.

```
status = waitForServer(client)
```

```
status = logical
   1
```

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. Set the order to an `int32` value of 8. If the input requires a 1-D array, set it as a column vector.

```
goalMsg.order = int32(8);
```

Send Goal and Execute Action

Before sending the goal, define the callback options framework for the goal execution process. In this example, you specify a callback function to execute when the server returns a feedback response and the final result using the name-value arguments.

```
callbackOpts = ros2ActionSendGoalOptions(FeedbackFcn=@helperFeedbackCallback,ResultFcn=@helperRes
```

Send the goal to the action server using the `sendGoal` function. Specify the callback options. During goal execution, you see outputs from the feedback and result callbacks displayed on the MATLAB® command window.

```
goalHandle = sendGoal(client,goalMsg,callbackOpts);
```

```
Goal with GoalUUID 3d10ab880f960666fde5666f45f621a accepted by server, waiting for result!
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0  1  1  2  3  5  8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Partial sequence feedback for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8
Full sequence result for goal 3d10ab880f960666fde5666f45f621a is 0   1   1   2   3   5   8  13
```

Get the status of goal execution.

```
exStatus = getStatus(client,goalHandle)

exStatus = int8
    2
```

Get the result using the action client and goal handle inputs. Display the result. The `getResult` function returns the sequence as a column vector.

```
resultMsg = getResult(client,goalHandle);
rosShowDetails(resultMsg)

ans =
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Alternatively, you can only use the goal handle as input to get the result.

```
resultMsg = getResult(goalHandle);
rosShowDetails(resultMsg)

ans =
    '
       MessageType :  action_tutorials_interfaces/FibonacciResult
       sequence    :  [0, 1, 1, 2, 3, 5, 8, 13, 21]'
```

Helper Functions

`helperFeedbackCallback` defines the callback function to execute when the client receives a feedback response from the action server.

```
function helperFeedbackCallback(goalHandle,feedbackMsg)
    seq = feedbackMsg.partial_sequence;
    disp(['Partial sequence feedback for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

`helperResultCallback` defines the callback function to execute when the client receives the result message from the action server.

```
function helperResultCallback(goalHandle,wrappedResultMsg)
    seq = wrappedResultMsg.result.sequence;
    disp(['Full sequence result for goal ',goalHandle.GoalUUID,' is ',num2str(seq')])
end
```

Input Arguments

`client` — ROS 2 action client
`ros2actionclient` object handle

ROS 2 action client, specified as a `ros2actionclient` object handle.

`goalHandle` — Action client goal handle
`ActionClientGoalHandle` object

Action client goal handle, specified as an `ActionClientGoalHandle` object.

Output Arguments

`resultMsg` — Result message received from the action server
structure

Result message received from the action server, returned as a ROS 2 message structure.

`status` — Status of the result message receipt
`logical` scalar

Status of the result message receipt, returned as a `logical` scalar. If the action client does not receive a result message within the timeout period, `status` will be `false`.

Note Use the `status` output argument when you use getResult for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

`statustext` — Status text associated with the result message receipt
character vector

Status text associated with the cancel response receipt, returned as one of the following:

- `'unknown'` — Result message receipt failed for unknown reason.
- `'succeeded'` — The result message was successfully received.
- `'failed'` — Result message receipt failed because the goal was canceled.
- `'aborted'` — Result message receipt failed because the goal was aborted.
- `'input'` — The input to the function is invalid.
- `'timeout'` — The result message was not received before the timeout period expired.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ActionClientGoalHandle` | `sendGoal` | `getStatus` | `ros2actionclient` | `ros2ActionSendGoalOptions` | `waitForServer` | `cancelGoal` | `cancelGoalAndWait` | `cancelGoalsBefore` | `cancelGoalsBeforeAndWait` | `cancelAllGoals` | `cancelAllGoalsAndWait`

waitfor

Package: `ros`

Pause code execution to achieve desired execution rate

Syntax

```
waitfor(rate)
numMisses = waitfor(rate)
```

Description

`waitfor(rate)` pauses execution of the ROS 2 loop execution object rate until the code reaches the desired execution rate. The function accounts for the time 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 Using `ros2rate`

Create a ROS 2 node.

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

Create a publisher to publish a standard integer message.

```
pub = ros2publisher(node,"/my_int","std_msgs/Int64");
```

Create a `ros2rate` object that runs at 2 Hz.

```
r = ros2rate(node,2);
```

Start loop that prints the current 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: 0.008841
Iteration: 2 - Time Elapsed: 0.519071
Iteration: 3 - Time Elapsed: 1.015046
Iteration: 4 - Time Elapsed: 1.512195
Iteration: 5 - Time Elapsed: 2.012841
```

```
Iteration: 6 - Time Elapsed: 2.510505
Iteration: 7 - Time Elapsed: 3.002018
Iteration: 8 - Time Elapsed: 3.500703
Iteration: 9 - Time Elapsed: 4.014428
Iteration: 10 - Time Elapsed: 4.500222
```

Input Arguments

`rate` — `ros2rate` object
handle

`ros2rate` object, specified as a handle. This object contains the information for the desired rate and other information about the execution. See `ros2rate` 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 desired time steps missed based on the `LastPeriod` and `DesiredRate` properties of the `ros2rate` object `rate`. For example, if the desired rate is 1 Hz and the last period was 3.2 seconds, `numMisses` is 3.

Version History

Introduced in R2022b

See Also

`ros2rate`

reset

Package: `ros`

Reset `ros2rate` object

Syntax

```
reset(rate)
```

Description

`reset(rate)` resets the state of the `ros2rate` 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` — ROS 2 loop execution object
`ros2rate` object

ROS 2 loop execution object, specified as `ros2rate` object. This object contains the information for the `DesiredRate` and other info about the execution. See `ros2rate` for more information.

Version History

Introduced in R2022b

See Also

`ros2rate` | `waitfor`

statistics

Package: ros

Statistics of past execution periods

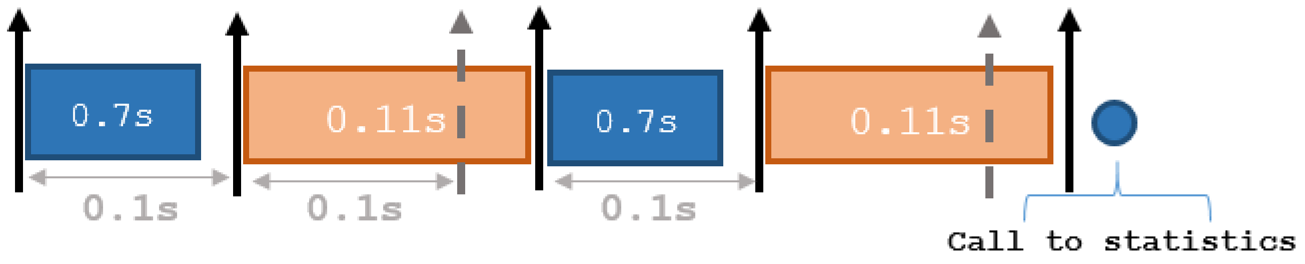
Syntax

```
stats = statistics(rate)
```

Description

`stats = statistics(rate)` returns statistics for the previous periods of code execution. `stats` is a structure 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 `ros2rate` 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 — ROS 2 loop execution object
`ros2rate` object

ROS 2 loop execution object, specified as `ros2rate` object. This object contains the information for the `DesiredRate` and other info about the execution. See `ros2rate` 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

Version History
Introduced in R2022b

See Also
`ros2rate` | `waitfor`

select

Select subset of messages in rosbag

Syntax

```
bagreadersel = select(bagreader)
bagreadersel = select(bagreader,Name=Value)
```

Description

`bagreadersel = select(bagreader)` returns a `rosbagreader` object `bagreadersel`, that contains all of the messages in the `rosbagreader` object `bagreader`.

This function creates a copy of the `rosbagreader` object or returns a new `rosbagreader` object that contains the specified message selection.

`bagreadersel = select(bagreader,Name=Value)` specifies additional parameters using one or more name-value arguments. For example `Topic="/odom"` selects a subset of the messages, filtered by the topic `/odom`.

Examples

Create rosbag Selection Using `rosbagreader` Object

Load a rosbag log file and parse out specific messages based on the selected criteria.

Create a `rosbagreader` object of all the messages in the rosbag log file.

```
bagMsgs = rosbagreader("ros_multi_topics.bag")
```

```
bagMsgs = 
  rosbagreader with properties:

           FilePath: 'B:\matlab\toolbox\robotics\robotexamples\ros\data\bags\ros_multi_topics.bag
          StartTime: 201.3400
            EndTime: 321.3400
        NumMessages: 36963
    AvailableTopics: [4x3 table]
    AvailableFrames: {0x1 cell}
        MessageList: [36963x4 table]
```

Select a subset of the messages based on their timestamp and topic.

```
bagMsgs2 = select(bagMsgs,...
    Time=[bagMsgs.StartTime bagMsgs.StartTime + 1],...
    Topic='/odom')
```

```
bagMsgs2 = 
  rosbagreader with properties:
```

```
              FilePath: 'B:\matlab\toolbox\robotics\robotexamples\ros\data\bags\ros_multi_topics.bag
             StartTime: 201.3400
               EndTime: 202.3200
           NumMessages: 99
       AvailableTopics: [1x3 table]
       AvailableFrames: {0x1 cell}
           MessageList: [99x4 table]
```

Retrieve the messages in the selection as a cell array.

```
msgs = readMessages(bagMsgs2)
```

```
msgs=99×1 cell array
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
    {1x1 Odometry}
      ⋮
```

Return certain message properties as a time series.

```
ts = timeseries(bagMsgs2,...
    'Pose.Pose.Position.X', ...
    'Twist.Twist.Angular.Y')
```

```
  timeseries

  Timeseries contains duplicate times.

  Common Properties:
            Name: '/odom Properties'
            Time: [99x1 double]
        TimeInfo: tsdata.timemetadata
            Data: [99x2 double]
        DataInfo: tsdata.datametadata
```

Input Arguments

bagreader — Index of messages in rosbag
`rosbagreader` object

Index of the messages in the rosbag, specified as a `rosbagreader` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Example: `Topic="/odom"` selects a subset of the messages, filtered by the topic `/odom`.

`MessageType` — ROS message type
string scalar | character vector | cell array of string scalars | cell array of character vectors

ROS message type, specified as a string scalar, character vector, cell array of string scalars, or cell array of character vectors. Specify multiple message types by using a cell array.

Example: `select(bagreader,MessageType={"nav_msgs/Odometry","rosgraph_msgs/Clock"})`

Data Types: `char` | `string` | `cell`

`Time` — Start and end times of rosbag selection
n-by-2 vector

Start and end times of the rosbag selection, specified as an *n*-by-2 vector.

Example: `select(bagreader,Time=[bag.StartTime,bag.StartTime+1])`

Data Types: `double`

`Topic` — ROS topic name
string scalar | character vector | cell array of string scalars | cell array of character vectors

ROS topic name, specified as a string scalar, character vector, cell array of string scalars, or cell array of character vectors. Specify multiple topic names by using a cell array.

Example: `select(bagreader,Topic={"/odom","/clock"})`

Data Types: `char` | `string` | `cell`

Output Arguments

`bagreadersel` — Copy or subset of rosbag messages
`rosbagreader` object

Copy or subset of rosbag messages, returned as a `rosbagreader` object.

Version History
Introduced in R2021b

See Also
`rosbagreader` | `readMessages` | `timeseries` | `canTransform` | `getTransform`

delete

Remove rosbag writer object from memory

Syntax

`delete(bagWriter)`

Description

`delete(bagWriter)` removes the `rosbagwriter` object from memory. The function closes the opened rosbag file before deleting the object.

If multiple references to the `rosbagwriter` object exist in the workspace, deleting the `rosbagwriter` object invalidates the remaining reference. Use the `clear` command to delete the remaining references to the object from the workspace.

Note The `rosbagwriter` object locks the created bag file for use, it is necessary to delete and clear the `rosbagwriter` object in order to use the bag file with a reader or perform other operations.

Examples

Write Log to rosbag File Using `rosbagwriter` Object

Retrieve all the information from the rosbag log file.

```
rosbag('info','path_record.bag')
```

```
Path:     C:\TEMP\Bdoc23a_2213998_3568\ib570499\10\tpf8d9c23d\ros-ex73035957\path_record.bag
Version:  2.0
Duration: 10.5s
Start:    Jul 05 2021 08:09:52.86 (1625486992.86)
End:      Jul 05 2021 08:10:03.40 (1625487003.40)
Size:     13.3 KB
Messages: 102
Types:    geometry_msgs/Point [4a842b65f413084dc2b10fb484ea7f17]
Topics:   /circle   51 msgs   : geometry_msgs/Point
          /line     51 msgs   : geometry_msgs/Point
```

Create a `rosbagreader` object of all the messages in the rosbag log file.

```
reader = rosbagreader('path_record.bag');
```

Select all the messages related to the topic `'/circle'`.

```
bagSelCircle = select(reader,'Topic','/circle');
```

Retrieve the list of timestamps from the topic.

```
timeStamps = bagSelCircle.MessageList.Time;
```

Retrieve the messages in the selection as a cell array.

```
messages = readMessages(bagSelCircle);
```

Create a `rosbagwriter` object to write the messages to a new rosbag file.

```
circleWriter = rosbagwriter('circular_path_record.bag');
```

Write all the messages related to the topic `'/circle'` to the new rosbag file.

```
write(circleWriter,'/circle',timeStamps,messages);
```

Remove the `rosbagwriter` object from memory and clear the associated object.

```
delete(circleWriter)
clear circleWriter
```

Retrieve all the information from the new rosbag log file.

```
rosbag('info','circular_path_record.bag')
```

```
Path:     C:\TEMP\Bdoc23a_2213998_3568\ib570499\10\tpf8d9c23d\ros-ex73035957\circular_path_recor
Version:  2.0
Duration: 10.4s
Start:    Jul 05 2021 08:09:52.86 (1625486992.86)
End:      Jul 05 2021 08:10:03.29 (1625487003.29)
Size:     8.8 KB
Messages: 51
Types:    geometry_msgs/Point [4a842b65f413084dc2b10fb484ea7f17]
Topics:   /circle   51 msgs  : geometry_msgs/Point
```

Load the new rosbag log file.

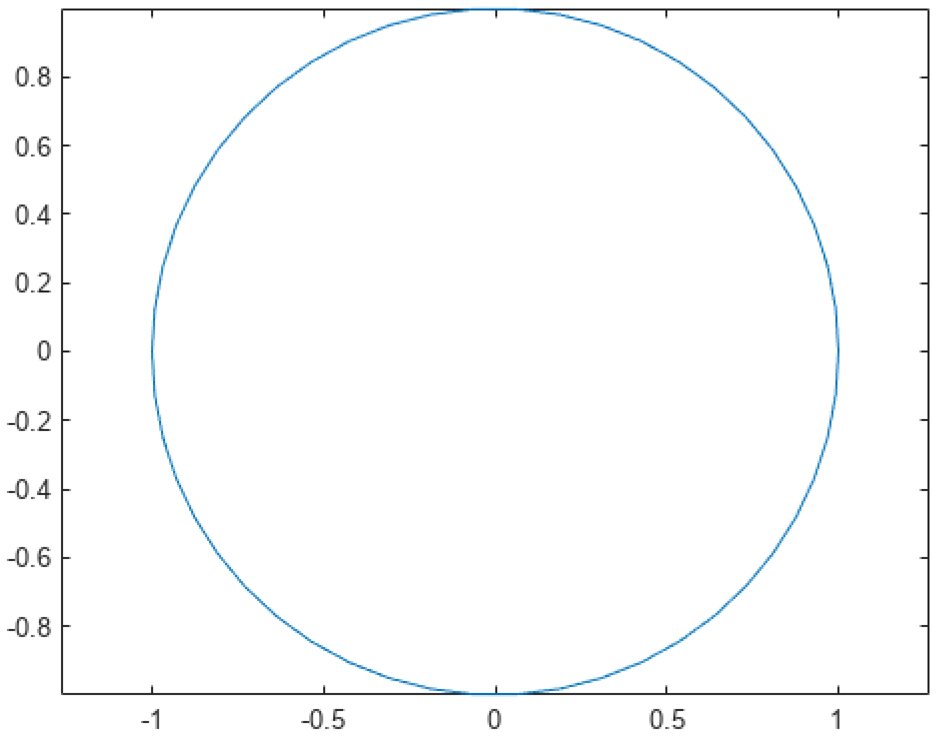
```
readerCircle = rosbagreader('circular_path_record.bag');
```

Create a time series for the coordinates.

```
tsCircle = timeseries(readerCircle,'X','Y');
```

Plot the coordinates.

```
plot(tsCircle.Data(:,1),tsCircle.Data(:,2))
axis equal
```



Create rosbag File Using `rosbagwriter` Object

Create a `rosbagwriter` object and a rosbag file in the current working directory. Specify the compression format of the message chunks and the size of each message chunk.

```
bagwriter = rosbagwriter("bagfile.bag", ...
    "Compression","lz4",...
    "ChunkSize",1500)
```

```
bagwriter = 
  rosbagwriter with properties:

       FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex26181333\bagfile.bag
      StartTime: 0
        EndTime: 0
    NumMessages: 0
    Compression: 'lz4'
      ChunkSize: 1500
       FileSize: 4117
```

Start node and connect to ROS master.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.1985 seconds.
Initializing ROS master on http://172.30.131.134:54539.
Initializing global node /matlab_global_node_21996 with NodeURI http://bat6234win64:64034/ and Ma
```

Write a single log to the rosbag file.

```
timeStamp = rostime("now");
rosMessage = rosmessage("nav_msgs/Odometry");
write(bagwriter,"/odom",timeStamp,rosMessage);
bagwriter
```

```
bagwriter = 
  rosbagwriter with properties:

       FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex26181333\bagfile.bag
      StartTime: 1.6779e+09
        EndTime: 1.6779e+09
    NumMessages: 1
    Compression: 'lz4'
      ChunkSize: 1500
       FileSize: 4172
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_21996 with NodeURI http://bat6234win64:64034/ and M
Shutting down ROS master on http://172.30.131.134:54539.
```

Remove rosbag writer object from memory and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Create a `rosbagreader` object and load all the messages in the rosbag log file. Verify the recently written log.

```
bagreader = rosbagreader('bagfile.bag')
```

```
bagreader = 
  rosbagreader with properties:

           FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex26181333\bagfile
          StartTime: 1.6779e+09
            EndTime: 1.6779e+09
        NumMessages: 1
    AvailableTopics: [1x3 table]
    AvailableFrames: {0x1 cell}
        MessageList: [1x4 table]
```

```
bagreader.AvailableTopics
```

```
ans=1×3 table
             NumMessages       MessageType             MessageDefinition
             ___________    _________________    _____________________________
```

```
    /odom         1         nav_msgs/Odometry    {'std_msgs/Header Header...'}
```

Input Arguments

`bagWriter` — ROS log file writer
`rosbagwriter` object

ROS log file writer, specified as a `rosbagwriter` object.

Version History
Introduced in R2021b

See Also

Objects
`rosbagwriter` | `rosbagreader`

Functions
`write`

write

Write logs to rosbag log file

Syntax

`write(bagwriter,topic,timestamp,message)`

Description

`write(bagwriter,topic,timestamp,message)` writes a single or multiple logs to a rosbag log file. A log contains a topic, its corresponding timestamp, and a ROS message.

Examples

Write Log to rosbag File Using `rosbagwriter` Object

Retrieve all the information from the rosbag log file.

```
rosbag('info','path_record.bag')
```

```
Path:     C:\TEMP\Bdoc23a_2213998_3568\ib570499\10\tpf8d9c23d\ros-ex73035957\path_record.bag
Version:  2.0
Duration: 10.5s
Start:    Jul 05 2021 08:09:52.86 (1625486992.86)
End:      Jul 05 2021 08:10:03.40 (1625487003.40)
Size:     13.3 KB
Messages: 102
Types:    geometry_msgs/Point [4a842b65f413084dc2b10fb484ea7f17]
Topics:   /circle   51 msgs   : geometry_msgs/Point
          /line     51 msgs   : geometry_msgs/Point
```

Create a `rosbagreader` object of all the messages in the rosbag log file.

```
reader = rosbagreader('path_record.bag');
```

Select all the messages related to the topic `'/circle'`.

```
bagSelCircle = select(reader,'Topic','/circle');
```

Retrieve the list of timestamps from the topic.

```
timeStamps = bagSelCircle.MessageList.Time;
```

Retrieve the messages in the selection as a cell array.

```
messages = readMessages(bagSelCircle);
```

Create a `rosbagwriter` object to write the messages to a new rosbag file.

```
circleWriter = rosbagwriter('circular_path_record.bag');
```

Write all the messages related to the topic `'/circle'` to the new rosbag file.

```
write(circleWriter,'/circle',timeStamps,messages);
```

Remove the `rosbagwriter` object from memory and clear the associated object.

```
delete(circleWriter)
clear circleWriter
```

Retrieve all the information from the new rosbag log file.

```
rosbag('info','circular_path_record.bag')
```

```
Path:     C:\TEMP\Bdoc23a_2213998_3568\ib570499\10\tpf8d9c23d\ros-ex73035957\circular_path_recor
Version:  2.0
Duration: 10.4s
Start:    Jul 05 2021 08:09:52.86 (1625486992.86)
End:      Jul 05 2021 08:10:03.29 (1625487003.29)
Size:     8.8 KB
Messages: 51
Types:    geometry_msgs/Point [4a842b65f413084dc2b10fb484ea7f17]
Topics:   /circle  51 msgs  : geometry_msgs/Point
```

Load the new rosbag log file.

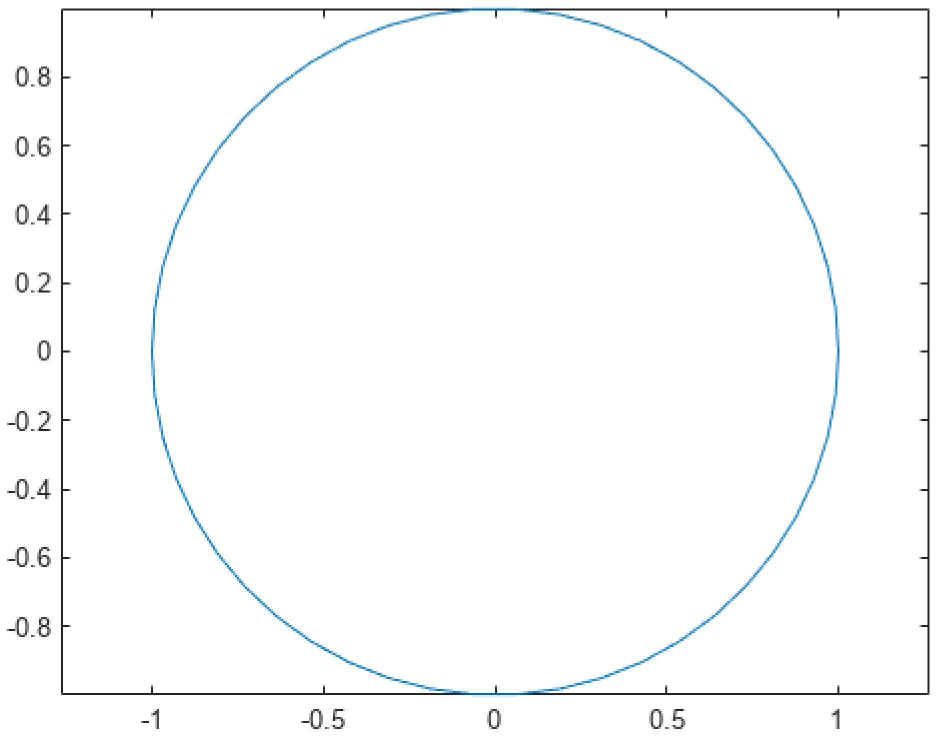
```
readerCircle = rosbagreader('circular_path_record.bag');
```

Create a time series for the coordinates.

```
tsCircle = timeseries(readerCircle,'X','Y');
```

Plot the coordinates.

```
plot(tsCircle.Data(:,1),tsCircle.Data(:,2))
axis equal
```



Create rosbag File Using `rosbagwriter` Object

Create a `rosbagwriter` object and a rosbag file in the current working directory. Specify the compression format of the message chunks and the size of each message chunk.

```
bagwriter = rosbagwriter("bagfile.bag", ...
    "Compression","lz4",...
    "ChunkSize",1500)
```

```
bagwriter = 
  rosbagwriter with properties:

       FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex26181333\bagfile.bag
      StartTime: 0
        EndTime: 0
    NumMessages: 0
    Compression: 'lz4'
      ChunkSize: 1500
       FileSize: 4117
```

Start node and connect to ROS master.

```
rosinit
```

```
Launching ROS Core...
..Done in 2.1985 seconds.
Initializing ROS master on http://172.30.131.134:54539.
Initializing global node /matlab_global_node_21996 with NodeURI http://bat6234win64:64034/ and Ma
```

Write a single log to the rosbag file.

```
timeStamp = rostime("now");
rosMessage = rosmessage("nav_msgs/Odometry");
write(bagwriter,"/odom",timeStamp,rosMessage);
bagwriter
```

```
bagwriter = 
  rosbagwriter with properties:

       FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex26181333\bagfile.bag
      StartTime: 1.6779e+09
        EndTime: 1.6779e+09
    NumMessages: 1
    Compression: 'lz4'
      ChunkSize: 1500
       FileSize: 4172
```

Shut down the ROS network.

```
rosshutdown
```

```
Shutting down global node /matlab_global_node_21996 with NodeURI http://bat6234win64:64034/ and M
Shutting down ROS master on http://172.30.131.134:54539.
```

Remove rosbag writer object from memory and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Create a `rosbagreader` object and load all the messages in the rosbag log file. Verify the recently written log.

```
bagreader = rosbagreader('bagfile.bag')
```

```
bagreader = 
  rosbagreader with properties:

           FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\14\tp5baae83e\ros-ex26181333\bagfile
          StartTime: 1.6779e+09
            EndTime: 1.6779e+09
        NumMessages: 1
    AvailableTopics: [1x3 table]
    AvailableFrames: {0x1 cell}
        MessageList: [1x4 table]
```

```
bagreader.AvailableTopics
```

```
ans=1×3 table
             NumMessages       MessageType              MessageDefinition
             ___________    _________________    _____________________________
```

```
    /odom         1         nav_msgs/Odometry    {'std_msgs/Header Header...'}
```

Input Arguments

`bagwriter` — ROS log file writer
`rosbagwriter` object

ROS log file writer, specified as a `rosbagwriter` object.

`topic` — ROS topic name
string scalar | character vector | cell array of string scalars | cell array of character vectors

ROS topic name, specified as a string scalar, character vector, cell array of string scalars, or cell array of character vectors. Specify multiple topic names by using a cell array.

Example: `"/odom"`

Example: `{"/odom","cmd_vel"}`

`timestamp` — Timestamp of ROS message
`Time` object handle | numeric scalar | structure | cell array of `Time` object handles | cell array of numeric scalars | cell array of structures

Timestamp of the ROS message, specified as a `Time` object handle, numeric scalar, structure, cell array of `Time` object handles, cell array of numeric scalars, or cell array of structures. Specify multiple timestamps by using a cell array. Create a `Time` object using `rostime`.

Example: `1625559291`

Example: `rostime("now")`

Example: `rostime("now","DataFormat","struct")`

Example: `{1625559291,1625559292}`

Example: `{rostime("now"),rostime("now")+1}`

`message` — ROS message
`Message` object handle | structure | cell array of `Message` object handles | cell array of structures

ROS message, specified as a `Message` object handle, structure, cell array of `Message` object handles, or cell array of structures. Specify multiple messages by using a cell array. Create a `Message` object using `rosmessage`.

Example: `rosmessage("nav_msgs/Odometry")`

Example: `rosmessage("nav_msgs/Odometry","DataFormat","struct")`

Example: `{rosmessage("nav_msgs/Odometry"),rosmessage("geometry_msgs/Twist")}`

Version History

Introduced in R2021b

See Also

Objects
`rosbagwriter` | `rosbagreader`

Functions
`delete`

delete

Remove ros2bagwriter object from memory

Syntax

```
delete(bagwriter)
```

Description

`delete(bagwriter)` removes the `ros2bagwriter` object from memory. The function closes the opened ROS 2 bag file before deleting the object.

If multiple references to the `ros2bagwriter` object exist in the workspace, deleting the `ros2bagwriter` object invalidates the remaining reference. Use the `clear` command to delete the remaining references to the object from the workspace.

Note The `ros2bagwriter` object locks the created bag file. Delete and clear the `ros2bagwriter` object to use the ROS 2 bag file.

Examples

Write Log Using `ros2bagwriter` Object by Reading Messages from ROS 2 Bag File

Extract the zip file that contains the ROS 2 bag log file and specify the full path to the log folder.

```
unzip('ros2_netwrk_bag.zip');
folderPath = fullfile(pwd,'ros2_netwrk_bag');
```

Get all the information from the ROS 2 bag log file.

```
bag2info = ros2("bag","info",folderPath);
```

Create a `ros2bagreader` object that contains all messages in the log file.

```
bag = ros2bagreader(folderPath);
bag.AvailableTopics
```

```
ans=4×3 table
                NumMessages         MessageType
                ___________    _____________________    ________________________________________

    /clock       1.607e+05     rosgraph_msgs/Clock      {'%...'
    /cmd_vel             3     geometry_msgs/Twist      {'...'
    /odom             5275     nav_msgs/Odometry        {'% The pose in this message should be sp
    /scan              892     sensor_msgs/LaserScan    {'%...'
```

Select a subset of the messages, by applying filters to the topic and timestamp.

```
start = bag.StartTime;
odomBagSel = select(bag,"Time",[start start + 30],"Topic","/odom")
```

```
odomBagSel = 
  ros2bagreader with properties:

           FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\10\tpf8d9c23d\ros-ex95368813\ros2_ne
          StartTime: 1.6020e+09
            EndTime: 1.6020e+09
    AvailableTopics: [1x3 table]
        MessageList: [801x3 table]
        NumMessages: 801
```

Get the messages in the selection.

```
odomMsgs = readMessages(odomBagSel);
```

Retrieve the list of timestamps from the topic.

```
timestamps = odomBagSel.MessageList.Time;
```

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder.

```
bagwriter = ros2bagwriter("myRos2bag");
```

Write the messages related to the topic '`/odom`' to the ROS 2 bag file.

```
write(bagwriter,"/odom",timestamps,odomMsgs)
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Load the new ROS 2 bag log file.

```
bagOdom = ros2bagreader("myRos2bag");
```

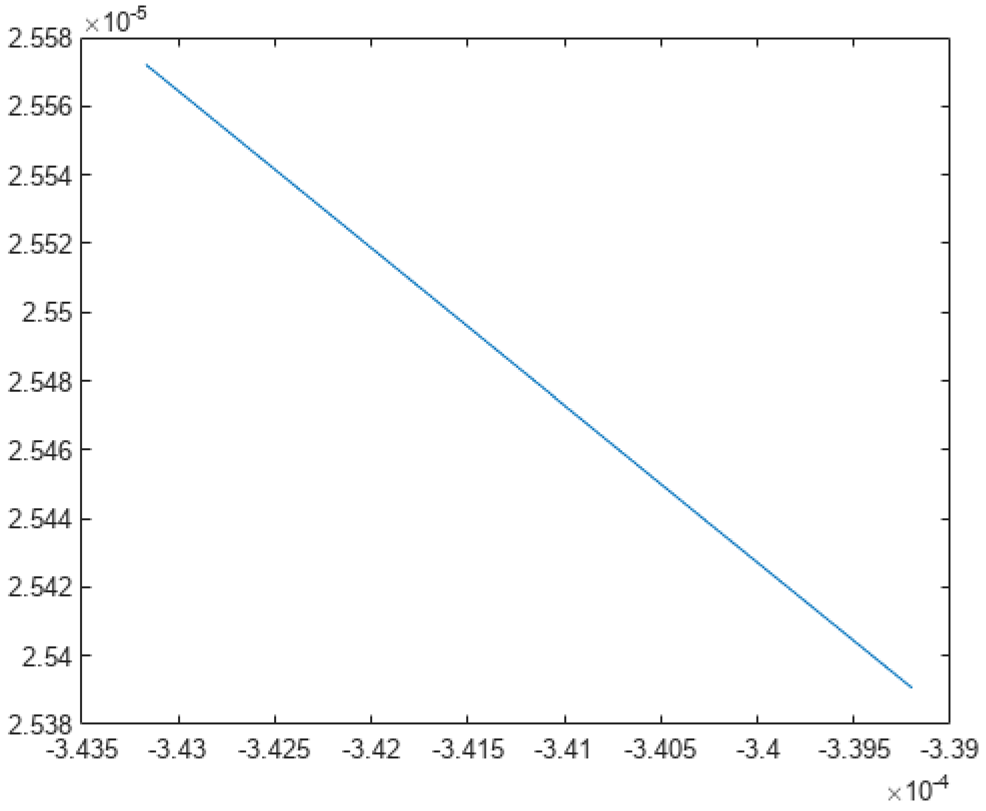
Retrieve messages from the ROS 2 bag log file.

```
msgs = readMessages(bagOdom);
```

Plot the coordinates for the messages in the ROS 2 bag log file.

Remove the `myRos2bag` file and the `ros2_netwrk_bag` file from memory to run the example again.

```
plot(cellfun(@(msg) msg.pose.pose.position.x,msgs),cellfun(@(msg) msg.twist.twist.angular.z,msgs
```



Create Single Log and Write to ROS 2 Bag File

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder.

```
bagwriter = ros2bagwriter("myRos2bag");
```

Write a single log to the ROS 2 bag file.

```
topic = "/odom";
message = ros2message("nav_msgs/Odometry");
timestamp = ros2time(1.6170e+09);
write(bagwriter,topic,timestamp,message)
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Create Multiple Logs and Write to ROS 2 Bag File

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder. Specify the cache size for each message.

```
bagwriter = ros2bagwriter("bag_files/my_bag_file",CacheSize=1500);
```

Write multiple logs to the ROS 2 bag file.

```
message1 = ros2message("nav_msgs/Odometry");
message2 = ros2message("geometry_msgs/Twist");
message3 = ros2message("sensor_msgs/Image");
write(bagwriter, ...
      ["/odom","cmd_vel","/camera/rgb/image_raw"], ...
      {ros2time(1.6160e+09),ros2time(1.6170e+09),ros2time(1.6180e+09)}, ...
      {message1,message2,message3})
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Create Multiple Logs for Same Topic and Write to ROS 2 Bag File

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder.

```
bagwriter = ros2bagwriter("myBag");
```

Write multiple logs for the same topic to the ROS 2 bag file.

```
pointMsg1 = ros2message("geometry_msgs/Point");
pointMsg2 = pointMsg1;
pointMsg3 = pointMsg1;
pointMsg1.x = 1;
pointMsg2.x = 2;
pointMsg3.x = 3;
write(bagwriter, ...
      "/point", ...
      {1.6190e+09, 1.6200e+09,1.6210e+09}, ...
      {pointMsg1,pointMsg2,pointMsg3})
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Input Arguments

`bagwriter` — ROS 2 log file writer
`ros2bagwriter` object

ROS 2 log file writer, specified as a `ros2bagwriter` object.

Version History

Introduced in R2022b

See Also

Objects
`ros2bagwriter`

Functions
`write`

Topics
"Write Log to rosbag File Using rosbagwriter Object" on page 4-88

write

Write logs to ROS 2 bag log file

Syntax

```
write(bagwriter,topic,timestamp,message)
```

Description

`write(bagwriter,topic,timestamp,message)` writes logs to the ROS 2 bag log file. A log contains a topic, its corresponding timestamp, and a ROS message.

Examples

Write Log Using `ros2bagwriter` Object by Reading Messages from ROS 2 Bag File

Extract the zip file that contains the ROS 2 bag log file and specify the full path to the log folder.

```
unzip('ros2_netwrk_bag.zip');
folderPath = fullfile(pwd,'ros2_netwrk_bag');
```

Get all the information from the ROS 2 bag log file.

```
bag2info = ros2("bag","info",folderPath);
```

Create a `ros2bagreader` object that contains all messages in the log file.

```
bag = ros2bagreader(folderPath);
bag.AvailableTopics
```

```
ans=4×3 table
                NumMessages         MessageType
                ___________    _____________________    ________________________________________

    /clock       1.607e+05     rosgraph_msgs/Clock      {'%...'
    /cmd_vel             3     geometry_msgs/Twist      {'...'
    /odom             5275     nav_msgs/Odometry        {'% The pose in this message should be sp
    /scan              892     sensor_msgs/LaserScan    {'%...'
```

Select a subset of the messages, by applying filters to the topic and timestamp.

```
start = bag.StartTime;
odomBagSel = select(bag,"Time",[start start + 30],"Topic","/odom")
```

```
odomBagSel = 
  ros2bagreader with properties:

           FilePath: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\10\tpf8d9c23d\ros-ex95368813\ros2_ne
          StartTime: 1.6020e+09
            EndTime: 1.6020e+09
    AvailableTopics: [1x3 table]
```

```
        MessageList: [801x3 table]
        NumMessages: 801
```

Get the messages in the selection.

```
odomMsgs = readMessages(odomBagSel);
```

Retrieve the list of timestamps from the topic.

```
timestamps = odomBagSel.MessageList.Time;
```

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder.

```
bagwriter = ros2bagwriter("myRos2bag");
```

Write the messages related to the topic '`/odom`' to the ROS 2 bag file.

```
write(bagwriter,"/odom",timestamps,odomMsgs)
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Load the new ROS 2 bag log file.

```
bagOdom = ros2bagreader("myRos2bag");
```

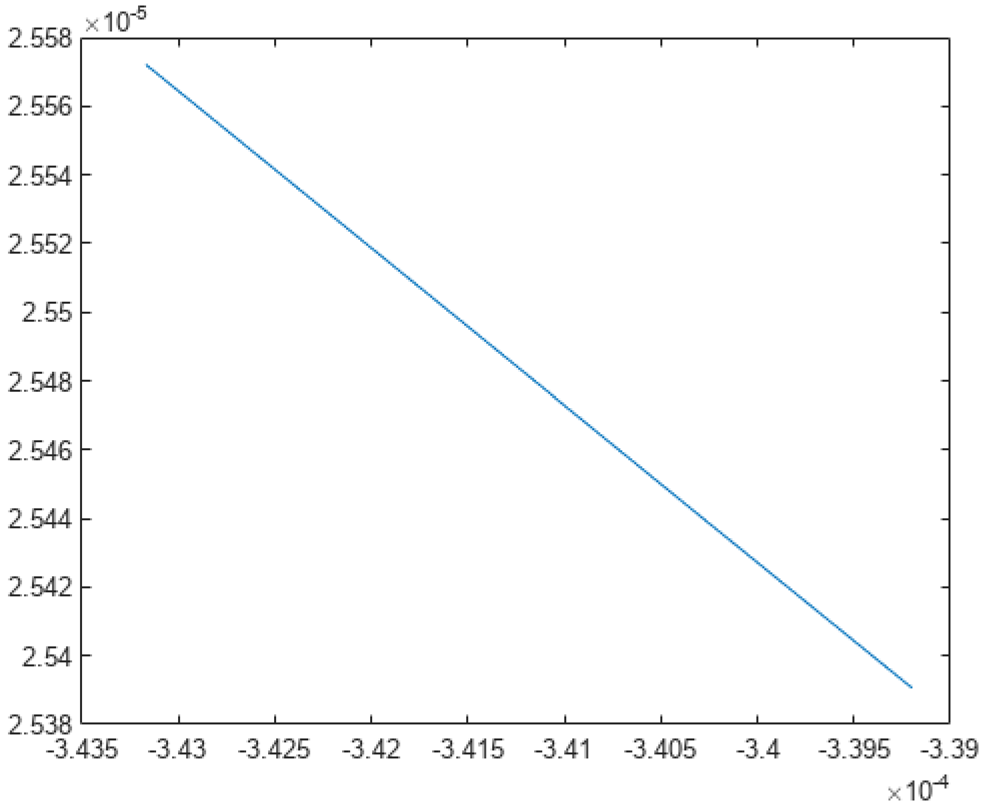
Retrieve messages from the ROS 2 bag log file.

```
msgs = readMessages(bagOdom);
```

Plot the coordinates for the messages in the ROS 2 bag log file.

Remove the `myRos2bag` file and the `ros2_netwrk_bag` file from memory to run the example again.

```
plot(cellfun(@(msg) msg.pose.pose.position.x,msgs),cellfun(@(msg) msg.twist.twist.angular.z,msgs
```



Create Single Log and Write to ROS 2 Bag File

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder.

```
bagwriter = ros2bagwriter("myRos2bag");
```

Write a single log to the ROS 2 bag file.

```
topic = "/odom";
message = ros2message("nav_msgs/Odometry");
timestamp = ros2time(1.6170e+09);
write(bagwriter,topic,timestamp,message)
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Create Multiple Logs and Write to ROS 2 Bag File

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder. Specify the cache size for each message.

```
bagwriter = ros2bagwriter("bag_files/my_bag_file",CacheSize=1500);
```

Write multiple logs to the ROS 2 bag file.

```
message1 = ros2message("nav_msgs/Odometry");
message2 = ros2message("geometry_msgs/Twist");
message3 = ros2message("sensor_msgs/Image");
write(bagwriter, ...
      ["/odom","cmd_vel","/camera/rgb/image_raw"], ...
      {ros2time(1.6160e+09),ros2time(1.6170e+09),ros2time(1.6180e+09)}, ...
      {message1,message2,message3})
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Create Multiple Logs for Same Topic and Write to ROS 2 Bag File

Create a `ros2bagwriter` object and a ROS 2 bag file in the specified folder.

```
bagwriter = ros2bagwriter("myBag");
```

Write multiple logs for the same topic to the ROS 2 bag file.

```
pointMsg1 = ros2message("geometry_msgs/Point");
pointMsg2 = pointMsg1;
pointMsg3 = pointMsg1;
pointMsg1.x = 1;
pointMsg2.x = 2;
pointMsg3.x = 3;
write(bagwriter, ...
      "/point", ...
      {1.6190e+09, 1.6200e+09,1.6210e+09}, ...
      {pointMsg1,pointMsg2,pointMsg3})
```

Close the bag file, remove the `ros2bagwriter` object from memory, and clear the associated object.

```
delete(bagwriter)
clear bagwriter
```

Input Arguments

`bagwriter` — ROS 2 log file writer

`ros2bagwriter` object

ROS 2 log file writer, specified as a `ros2bagwriter` object.

`topic` — ROS 2 topic names

string scalar | character vector | cell array of string scalars | cell array of character vectors

ROS 2 topic names, specified as a string scalar, character vector, cell array of string scalars, or cell array of character vectors. Specify multiple topic names by using a cell array.

Example: `"/odom"`

Example: `{"/odom","cmd_vel"}`

`timestamp` — Timestamps of ROS 2 messages
`Time` object | numeric scalar | structure | cell array of `Time` objects | cell array of nonnegative numeric scalars | cell array of structures

Timestamps of the ROS 2 messages, specified as `Time` objects, numeric scalar, structure, cell array of `Time` objects, cell array of nonnegative numeric scalars, or cell array of structures. Specify multiple timestamps by using a cell array. Create a `Time` object using `ros2time`.

Example: `1625559291`

Example: `ros2time(node, "now")`

Example: `{1625559291,1625559292}`

`message` — ROS 2 messages
`Message` object | structure | cell array of `Message` objects | cell array of structures

ROS 2 messages, specified as a `Message` object, structure, cell array of `Message` objects, or cell array of structures. Specify multiple messages by using a cell array. Create a `Message` object using `ros2message`.

Example: `ros2message("nav_msgs/Odometry")`

Example: `ros2message("nav_msgs/Odometry",DataFormat="struct")`

Example: `{ros2message("nav_msgs/Odometry"),ros2message("geometry_msgs/Twist")}`

Version History
Introduced in R2022b

See Also

Objects
`ros2bagwriter`

Functions
`delete`

Topics
“Write Log to rosbag File Using rosbagwriter Object” on page 4-93

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.

Version History

Introduced in R2019b

See Also

`rosrate` | `rateControl` | `waitfor`

Topics

"Execute Code at a Fixed-Rate" (Robotics System Toolbox)

statistics

Statistics of past execution periods

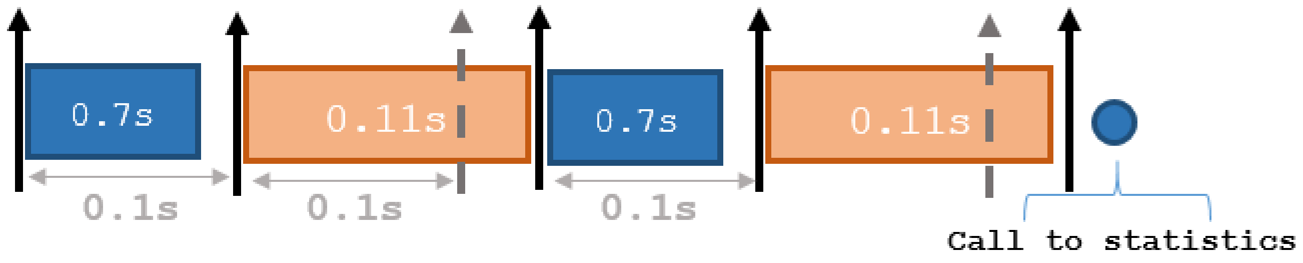
Syntax

```
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

Version History

Introduced in R2019b

See Also

`rosrate` | `waitfor` | `rateControl`

Topics

"Execute Code at a Fixed-Rate" (Robotics System Toolbox)

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.

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.

Version History

Introduced in R2019b

See Also

`rosrate` | `rateControl`

Topics
“Execute Code at a Fixed-Rate” (Robotics System Toolbox)

readMessages

Read messages from ros2bagreader object

Syntax

```
msgs = readMessages(bag)
msgs = readMessages(bag,rows)
```

Description

`msgs = readMessages(bag)` returns data from all of the messages in the `ros2bagreader` object `bag`. The messages are returned as a cell array of structures.

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

Examples

Read Messages from ROS 2 Bag Log File

Extract the zip file that contains the ROS 2 bag log file and specify the full path to the log folder.

```
unzip('ros2_netwrk_bag.zip');
folderPath = fullfile(pwd,'ros2_netwrk_bag');
```

Create a `ros2bagreader` object that contains all messages in the log file.

```
bag = ros2bagreader(folderPath);
```

Get information on the contents of the `ros2bagreader` object.

```
baginfo = ros2("bag","info",folderPath)
```

```
baginfo = struct with fields:
         Path: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\26\tp4cf343c3\ros-ex96596996\ros2_netwrk_ba
      Version: '1'
    StorageId: 'sqlite3'
     Duration: 207.9020
        Start: [1x1 struct]
          End: [1x1 struct]
         Size: 16839538
     Messages: 166867
        Types: [4x1 struct]
       Topics: [4x1 struct]
```

Get all the messages in the `ros2bagreader` object.

```
msgs = readMessages(bag);
```

Select a subset of the messages, filtered by topic.

```
bagSel = select(bag,"Topic","/odom");
```

Get the messages in the selection.

```
msgsFiltered = readMessages(bagSel);
```

Input Arguments

bag — Messages in ros2bagreader object
`ros2bagreader` object

Messages in the `ros2bagreader` object, specified as a `ros2bagreader` object.

rows — Rows of ros2bagreader object
n-element vector

Rows of the `ros2bagreader` object, specified as an *n*-element vector. *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.NumMessages`].

Output Arguments

msgs — ROS 2 message data
cell array of structures

ROS 2 message data, returned as a cell array of structures.

Version History

Introduced in R2021a

See Also

Objects
`ros2bagreader`

Functions
`select`

select

Select subset of messages in ros2bagreader

Syntax

```
bagsel = select(bag)
bagsel = select(bag,Name,Value)
```

Description

`bagsel = select(bag)` returns a `ros2bagreader` object, `bagsel`, that contains all of the messages in the `ros2bagreader` object, `bag`.

This function creates a copy of the `ros2bagreader` object or returns a new `ros2bagreader` object that contains the specified message selection.

`bagsel = select(bag,Name,Value)` provides additional options specified by one or more name-value pair arguments. For example, `"Topic","/scan"` selects a subset of the messages, filtered by the topic `/scan`.

Examples

Read Messages from ROS 2 Bag Log File

Extract the zip file that contains the ROS 2 bag log file and specify the full path to the log folder.

```
unzip('ros2_netwrk_bag.zip');
folderPath = fullfile(pwd,'ros2_netwrk_bag');
```

Create a `ros2bagreader` object that contains all messages in the log file.

```
bag = ros2bagreader(folderPath);
```

Get information on the contents of the `ros2bagreader` object.

```
baginfo = ros2("bag","info",folderPath)
```

```
baginfo = struct with fields:
         Path: 'C:\TEMP\Bdoc23a_2213998_3568\ib570499\26\tp4cf343c3\ros-ex96596996\ros2_netwrk_ba
      Version: '1'
    StorageId: 'sqlite3'
     Duration: 207.9020
        Start: [1x1 struct]
          End: [1x1 struct]
         Size: 16839538
     Messages: 166867
        Types: [4x1 struct]
       Topics: [4x1 struct]
```

Get all the messages in the `ros2bagreader` object.

```
msgs = readMessages(bag);
```

Select a subset of the messages, filtered by topic.

```
bagSel = select(bag,"Topic","/odom");
```

Get the messages in the selection.

```
msgsFiltered = readMessages(bagSel);
```

Input Arguments

bag — Messages in `ros2bagreader` object
`ros2bagreader` object

Messages in the `ros2bagreader` object, specified as a `ros2bagreader` object.

Name-Value Pair Arguments

Specify optional pairs of arguments as `Name1=Value1,...,NameN=ValueN`, where `Name` is the argument name and `Value` is the corresponding value. Name-value arguments must appear after other arguments, but the order of the pairs does not matter.

Before R2021a, use commas to separate each name and value, and enclose `Name` *in quotes.*

Example: `select(bag,"Topic","/scan")` selects a subset of the messages, filtered by the topic `/scan`.

`MessageType` — ROS 2 message type
string scalar | character vector | cell array of string scalars | cell array of character vectors

ROS 2 message type, specified as a string scalar, character vector, cell array of string scalars, or cell array of character vectors. Multiple message types can be specified with a cell array.

Example: `select(bag,"MessageType",{"sensor_msgs/CameraInfo","sensor_msgs/LaserScan"})`

Data Types: `char` | `string` | `cell`

`Time` — Start and end times of ROS 2 bag selection
n-by-2 vector

Start and end times of the ROS 2 bag selection, specified as an *n*-by-2 vector.

Example: `select(bag,"Time",[bag.MessageList(1,1).Time,bag.MessageList(2,1).Time])`

Data Types: `double`

`Topic` — ROS 2 topic name
string scalar | character vector | cell array of string scalars | cell array of character vectors

ROS 2 topic name, specified as a string scalar, character vector, cell array of string scalars, or cell array of character vectors. Multiple topic names can be specified with a cell array.

Example: `select(bag,"Topic",{"/scan","/clock"})`

Data Types: `char` | `string` | `cell`

Output Arguments

`bagsel` — Copy or subset of ROS 2 bag messages
`ros2bagreader` object

Copy or subset of ROS 2 bag messages, returned as a `ros2bagreader` object.

Version History
Introduced in R2021a

See Also

Objects
`ros2bagreader`

Functions
`readMessages`

get

Get value of parameter in associated ROS 2 node

Syntax

```
paramValue = get(paramObj,paramName)
[paramValue,status] = get(paramObj,paramName)
```

Description

`paramValue = get(paramObj,paramName)` returns `paramValue`, that contains the value of the specified parameter `paramName` in the ROS 2 node associated with the parameter object, `paramObj`. If it fails to return the parameter value, this syntax displays an error.

`[paramValue,status] = get(paramObj,paramName)` returns a `status` indicating whether the function was able to successfully return the parameter value. If it fails to return the parameter value, this syntax returns the `paramValue` as an empty `double`, and the `status` as `false` without displaying an error.

Examples

Interact with Parameters of ROS 2 Node

Create a ROS 2 node with parameters.

```
nodeParams.my_double = 2.0;
nodeParams.my_namespace.my_int = int64(1);
nodeParams.my_double_array = [1.1 2.2 3.3];
nodeParams.my_string = "Keyparams";
node1 = ros2node("/node1",Parameters=nodeParams);
```

Create a `ros2param` object to interact with the parameters of the ROS 2 node, `/node1`.

```
paramObj = ros2param("/node1");
```

Use the `set` function to change the value of the parameter `my_string`.

```
set(paramObj,"my_string","Newparams");
```

Use the `get` function to obtain the new value of `my_string`.

```
stringVal = get(paramObj,"my_string")
```

```
stringVal = 
'Newparams'
```

Use the `has` function to check if the parameter `my_char` exists in the ROS 2 node, `/node1`.

```
flag = has(paramObj,"my_char")
flag = logical
   0
```

Use the `search` function to search for names of all the parameters that contain the string "`my_d`". Obtain the values of the matching parameters.

```
[pNames,pVals] = search(paramObj,"my_d")
```

```
pNames = 2x1 cell
    {'my_double'      }
    {'my_double_array'}


pVals=2×1 cell array
    {[                  2]}
    {[1.1000 2.2000 3.3000]}
```

Use the `list` function to list the names of all parameters in the ROS 2 node.

```
pList = list(paramObj)
```

```
pList = 5x1 cell
    {'my_double'          }
    {'my_double_array'    }
    {'my_namespace.my_int'}
    {'my_string'          }
    {'use_sim_time'       }
```

Input Arguments

`paramObj` — ROS 2 parameter object

handle (default)

ROS 2 parameter object, specified as a `ros2param` object handle.

`paramName` — Name of parameter

string scalar | character vector

Name of the parameter, specified as a string scalar or a character vector.

Output Arguments

`paramValue` — Value of parameter

scalar | array

Value of the parameter, returned as a scalar or an array.

Data Types: `int64` | `logical` | `char` | `string` | `double` | `cell`

`status` — Status of parameter return

logical scalar

Status of the parameter return, returned as a logical scalar.

Version History

Introduced in R2022b

See Also

`ros2param` | `set`

has

Check if parameter exists in ROS 2 node

Syntax

```
exists = has(paramObj,paramName)
```

Description

`exists = has(paramObj,paramName)` checks if the parameter `paramName` exists in the ROS 2 node associated with the parameter object `paramObj`. This function returns `exists` as `true` only if the parameter exists.

Examples

Interact with Parameters of ROS 2 Node

Create a ROS 2 node with parameters.

```
nodeParams.my_double = 2.0;
nodeParams.my_namespace.my_int = int64(1);
nodeParams.my_double_array = [1.1 2.2 3.3];
nodeParams.my_string = "Keyparams";
node1 = ros2node("/node1",Parameters=nodeParams);
```

Create a `ros2param` object to interact with the parameters of the ROS 2 node, `/node1`.

```
paramObj = ros2param("/node1");
```

Use the `set` function to change the value of the parameter `my_string`.

```
set(paramObj,"my_string","Newparams");
```

Use the `get` function to obtain the new value of `my_string`.

```
stringVal = get(paramObj,"my_string")
```

```
stringVal =
'Newparams'
```

Use the `has` function to check if the parameter `my_char` exists in the ROS 2 node, `/node1`.

```
flag = has(paramObj,"my_char")
```

```
flag = logical
   0
```

Use the `search` function to search for names of all the parameters that contain the string "`my_d`". Obtain the values of the matching parameters.

```
[pNames,pVals] = search(paramObj,"my_d")
```

```
pNames = 2x1 cell
    {'my_double'      }
    {'my_double_array'}


pVals=2×1 cell array
    {[                  2]}
    {[1.1000 2.2000 3.3000]}
```

Use the `list` function to list the names of all parameters in the ROS 2 node.

```
pList = list(paramObj)
```

```
pList = 5x1 cell
    {'my_double'          }
    {'my_double_array'    }
    {'my_namespace.my_int'}
    {'my_string'          }
    {'use_sim_time'       }
```

Input Arguments

`paramObj` — ROS 2 parameter object
handle (default)

ROS 2 parameter object, specified as a `ros2param` object handle.

`paramName` — Name of the parameter
string scalar | character vector

Name of the parameter, specified as a string scalar or a character vector.

Output Arguments

`exists` — Flag indicating whether parameter exists
logical scalar

Flag indicating whether the parameter exists, returned as a logical scalar.

Version History

Introduced in R2022b

See Also

`ros2param`

list

List all parameters in associated ROS 2 node

Syntax

```
paramList = list(paramObj)
```

Description

`paramList = list(paramObj)` returns `paramList` which contains the list of all the parameters in the ROS 2 node associated with the parameter object, `paramObj`

Examples

Interact with Parameters of ROS 2 Node

Create a ROS 2 node with parameters.

```
nodeParams.my_double = 2.0;
nodeParams.my_namespace.my_int = int64(1);
nodeParams.my_double_array = [1.1 2.2 3.3];
nodeParams.my_string = "Keyparams";
node1 = ros2node("/node1",Parameters=nodeParams);
```

Create a `ros2param` object to interact with the parameters of the ROS 2 node, `/node1`.

```
paramObj = ros2param("/node1");
```

Use the `set` function to change the value of the parameter `my_string`.

```
set(paramObj,"my_string","Newparams");
```

Use the `get` function to obtain the new value of `my_string`.

```
stringVal = get(paramObj,"my_string")
```

```
stringVal = 
'Newparams'
```

Use the `has` function to check if the parameter `my_char` exists in the ROS 2 node, `/node1`.

```
flag = has(paramObj,"my_char")
```

```
flag = logical
   0
```

Use the `search` function to search for names of all the parameters that contain the string "`my_d`". Obtain the values of the matching parameters.

```
[pNames,pVals] = search(paramObj,"my_d")
```

```
pNames = 2x1 cell
    {'my_double'      }
    {'my_double_array'}


pVals=2×1 cell array
    {[                 2]}
    {[1.1000 2.2000 3.3000]}
```

Use the `list` function to list the names of all parameters in the ROS 2 node.

```
pList = list(paramObj)
```

```
pList = 5x1 cell
    {'my_double'          }
    {'my_double_array'    }
    {'my_namespace.my_int'}
    {'my_string'          }
    {'use_sim_time'       }
```

Input Arguments

`paramObj` — ROS 2 parameter object
handle (default)

ROS 2 parameter object, specified as a `ros2param` object handle.

Output Arguments

`paramList` — List of all parameter names in associated ROS 2 node
cell array

List of all parameter names in the associated ROS 2 node, returned as a cell array.

Version History

Introduced in R2022b

See Also

`ros2param`

search

Search for parameter names in ROS 2 node

Syntax

```
paramNames = search(paramObj,searchStr)
[paramNames,paramValues] = search(paramObj,searchStr)
```

Description

`paramNames = search(paramObj,searchStr)` searches for parameter names in the ROS 2 node associated with the parameter object `paramObj`, which contain the string `searchStr`. The function returns the matching parameter names in `pNames`.

`[paramNames,paramValues] = search(paramObj,searchStr)` also returns the corresponding values `pValues` of the matching parameters in `pNames`.

Examples

Interact with Parameters of ROS 2 Node

Create a ROS 2 node with parameters.

```
nodeParams.my_double = 2.0;
nodeParams.my_namespace.my_int = int64(1);
nodeParams.my_double_array = [1.1 2.2 3.3];
nodeParams.my_string = "Keyparams";
node1 = ros2node("/node1",Parameters=nodeParams);
```

Create a `ros2param` object to interact with the parameters of the ROS 2 node, `/node1`.

```
paramObj = ros2param("/node1");
```

Use the `set` function to change the value of the parameter `my_string`.

```
set(paramObj,"my_string","Newparams");
```

Use the `get` function to obtain the new value of `my_string`.

```
stringVal = get(paramObj,"my_string")
```

```
stringVal = 
'Newparams'
```

Use the `has` function to check if the parameter `my_char` exists in the ROS 2 node, `/node1`.

```
flag = has(paramObj,"my_char")
```

```
flag = logical
   0
```

Use the `search` function to search for names of all the parameters that contain the string "`my_d`". Obtain the values of the matching parameters.

```
[pNames,pVals] = search(paramObj,"my_d")
```

```
pNames = 2x1 cell
    {'my_double'      }
    {'my_double_array'}


pVals=2×1 cell array
    {[                 2]}
    {[1.1000 2.2000 3.3000]}
```

Use the `list` function to list the names of all parameters in the ROS 2 node.

```
pList = list(paramObj)
```

```
pList = 5x1 cell
    {'my_double'          }
    {'my_double_array'    }
    {'my_namespace.my_int'}
    {'my_string'          }
    {'use_sim_time'       }
```

Input Arguments

`paramObj` — ROS 2 parameter object
handle (default)

ROS 2 parameter object, specified as a `ros2param` object handle.

`searchStr` — ROS 2 parameter search string
string scalar | character vector

ROS 2 parameter search string specified as a string scalar or character vector. The `search` function returns all parameters that contain this search string.

Output Arguments

`paramNames` — Matching parameter names
cell array of character vectors

Matching parameter names, returned as a cell array of character vectors.

`paramValues` — Values of the matching parameters
cell array

Values of the matching parameters, returned as a cell array.

Data Types: `int64` | `logical` | `char` | `string` | `double` | `cell`

Version History

Introduced in R2022b

See Also

`ros2param`

set

Set value of parameter in associated ROS 2 node

Syntax

```
set(paramObj,paramName,paramValue)
```

Description

`set(paramObj,paramName,paramValue)` sets the value `paramValue` for the parameter `paramName` in the ROS 2 node associated with the parameter object `paramObj`. If `paramName` does not exist in the ROS 2 node, this syntax throws an error.

Examples

Interact with Parameters of ROS 2 Node

Create a ROS 2 node with parameters.

```
nodeParams.my_double = 2.0;
nodeParams.my_namespace.my_int = int64(1);
nodeParams.my_double_array = [1.1 2.2 3.3];
nodeParams.my_string = "Keyparams";
node1 = ros2node("/node1",Parameters=nodeParams);
```

Create a `ros2param` object to interact with the parameters of the ROS 2 node, `/node1`.

```
paramObj = ros2param("/node1");
```

Use the `set` function to change the value of the parameter `my_string`.

```
set(paramObj,"my_string","Newparams");
```

Use the `get` function to obtain the new value of `my_string`.

```
stringVal = get(paramObj,"my_string")
```

```
stringVal =
'Newparams'
```

Use the `has` function to check if the parameter `my_char` exists in the ROS 2 node, `/node1`.

```
flag = has(paramObj,"my_char")
```

```
flag = logical
   0
```

Use the `search` function to search for names of all the parameters that contain the string "`my_d`". Obtain the values of the matching parameters.

```
[pNames,pVals] = search(paramObj,"my_d")
```

```
pNames = 2x1 cell
    {'my_double'      }
    {'my_double_array'}
```

```
pVals=2×1 cell array
    {[                  2]}
    {[1.1000 2.2000 3.3000]}
```

Use the `list` function to list the names of all parameters in the ROS 2 node.

```
pList = list(paramObj)
```

```
pList = 5x1 cell
    {'my_double'          }
    {'my_double_array'    }
    {'my_namespace.my_int'}
    {'my_string'          }
    {'use_sim_time'       }
```

Input Arguments

`paramObj` — ROS 2 parameter object
handle (default)

ROS 2 parameter object, specified as a `ros2param` object handle.

`paramName` — Name of parameter
string scalar | character vector

Name of the parameter, specified as a string scalar or a character vector.

`paramValue` — Value of parameter
scalar | array

Value of the parameter, specified as a scalar or an array.

Data Types: `int64` | `logical` | `char` | `string` | `double` | `cell`

Version History

Introduced in R2022b

See Also

`ros2param` | `get`

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

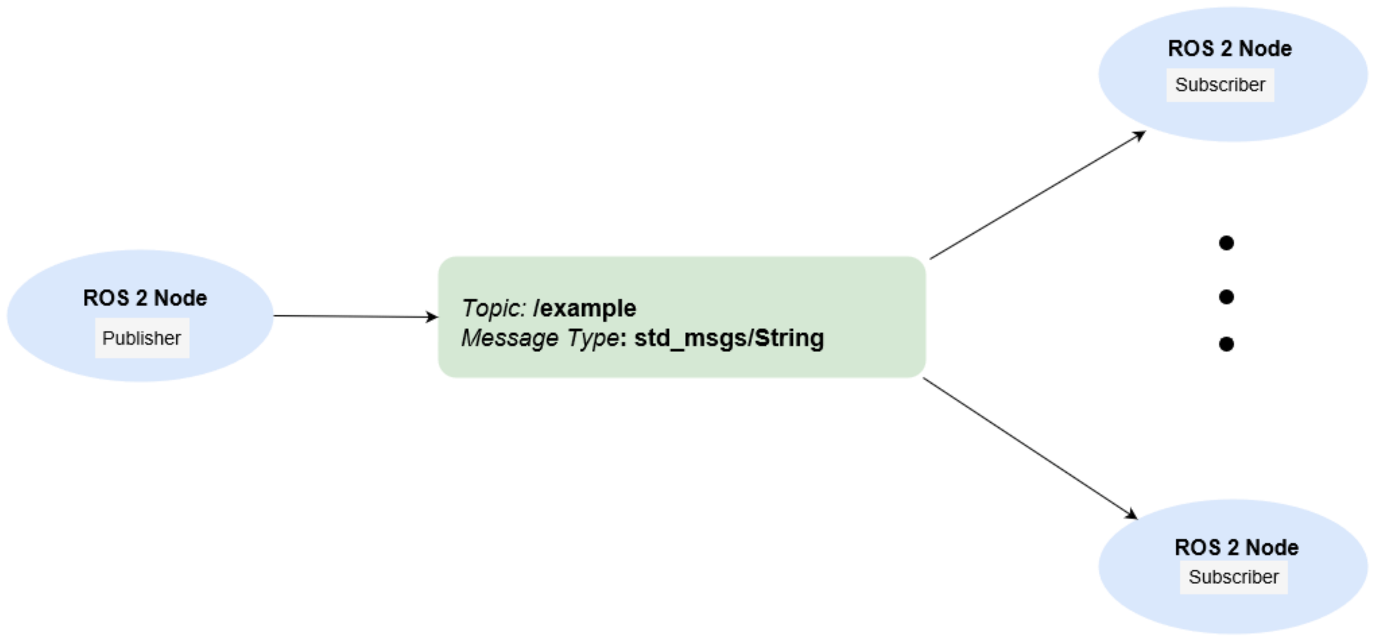
Exchange Data with ROS 2 Publishers and Subscribers

This example shows how to publish and subscribe to topics in a ROS 2 network.

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.



Besides how to publish and subscribe to topics in a ROS 2 network, this example 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
/rosout
/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(5)
laserSub = ros2subscriber(detectNode,"/scan");
pause(5)
```

Use `receive` to wait for a new message. Specify a timeout of 10 seconds. The output `scanData` contains the received message data. `status` indicates whether a message was received successfully and `statustext` provides additional information about the `status`.

```
[scanData,status,statustext] = 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");
pause(5)
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)
```

```
       0.00235920447111606       -0.0201184589892978        0.0203969078651195
```

```
disp(orient)
```

```
       -0.0118389124011118       0.00676849978014866        0.0387860955311228
```

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

"Create Callbacks for Graphics Objects" 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
/rosout
/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”
- “Generate ROS 2 Custom Messages in MATLAB”

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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ros2publisher`

Topics
“Exchange Data with ROS 2 Publishers and Subscribers”
“Manage Quality of Service Policies in ROS 2”

getParameter

Get value of parameter declared in ROS 2 node

Syntax

```
paramValue = getParameter(nodeObj,paramName)
[paramValue,status] = getParameter(nodeObj,paramName)
[paramValue,status] = getParameter(nodeObj,paramName,Datatype=dtype)
```

Description

`paramValue = getParameter(nodeObj,paramName)` returns `paramValue`, that contains the value of the specified parameter `paramName` associated with the ROS 2 node `nodeObj`. If it fails to return the parameter value, this syntax displays an error.

`[paramValue,status] = getParameter(nodeObj,paramName)` returns a `status` indicating whether the function was able to successfully return the parameter value. If it fails to return the parameter value, this syntax returns the `paramValue` as an empty `double`, and the `status` as `false` without displaying an error.

`[paramValue,status] = getParameter(nodeObj,paramName,Datatype=dtype)` specifies the expected return datatype of `paramValue` in the generated code using the name-value argument `Datatype`. You must specify this syntax for code generation. This syntax supports the returned datatype to be `int64`, `logical`, `string`, `char` or `double`.

Examples

Get and Set Parameters for ROS 2 Nodes

Create a structure that contains all the parameters for the ROS 2 node.

```
nodeParams.my_double = 2.0;
nodeParams.my_namespace.my_int = int64(1);
nodeParams.my_double_array = [1.1 2.2 3.3];
nodeParams.my_string = "Keyparams";
```

Create a ROS 2 node and specify `nodeParams` as the parameters.

```
node1 = ros2node("/node1",Parameters=nodeParams);
```

Set the parameter `my_double` to a new value.

```
setParameter(node1,"my_double",5.2);
```

Obtain the new value of the parameter `my_double`.

```
doubleValue = getParameter(node1,"my_double")
```

```
doubleValue = 5.2000
```

Input Arguments

`nodeObj` — ROS 2 node on network
handle (default)

A object on the network, specified as a `ros2node` object handle.

`paramName` — Name of the parameter
string scalar | character vector

Name of the parameter, specified as a string scalar or a character vector.

Output Arguments

`paramValue` — Value of the parameter
scalar | array

Value of the parameter, returned as a scalar or an array.

Data Types: `int64` | `logical` | `char` | `string` | `double` | `cell`

`status` — Status of parameter return
logical scalar

Status of the parameter return, returned as a logical scalar.

Version History
Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- You must use the syntax that specifies the `Datatype` name-value argument.

See Also
`setParameter` | `ros2node` | `ros2param`

setParameter

Set value of parameter declared in ROS 2 node

Syntax

```
setParameter(nodeObj,paramName,paramValue)
```

Description

`setParameter(nodeObj,paramName,paramValue)` sets the value of the parameter `paramName` associated with the ROS 2 node object `nodeObj` to the value, `paramValue`. If `paramName` does not exist in the ROS 2 node, this syntax throws an error.

Examples

Get and Set Parameters for ROS 2 Nodes

Create a structure that contains all the parameters for the ROS 2 node.

```
nodeParams.my_double = 2.0;
nodeParams.my_namespace.my_int = int64(1);
nodeParams.my_double_array = [1.1 2.2 3.3];
nodeParams.my_string = "Keyparams";
```

Create a ROS 2 node and specify `nodeParams` as the parameters.

```
node1 = ros2node("/node1",Parameters=nodeParams);
```

Set the parameter `my_double` to a new value.

```
setParameter(node1,"my_double",5.2);
```

Obtain the new value of the parameter `my_double`.

```
doubleValue = getParameter(node1,"my_double")
```

```
doubleValue = 5.2000
```

Input Arguments

`nodeObj` — ROS 2 node on network
handle (default)

A object on the network, specified as a `ros2node` object handle.

`paramName` — Name of the parameter
string scalar | character vector

Name of the parameter, specified as a string scalar or a character vector.

`paramValue` — Value of the parameter
scalar | array

Value of the parameter, specified as a scalar or an array.

Data Types: `int64` | `logical` | `char` | `string` | `double` | `cell`

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`getParameter` | `ros2node` | `ros2param`

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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

See Also

`ros2node`

receive

Wait for new message

Syntax

```
msg = receive(sub)
msg = receive(sub, timeout)
[msg,status,statustext] = receive( ___ )
```

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.

`[msg,status,statustext] = receive( ___ )` returns a `status` indicating whether a message has been received successfully, and a `statustext` that captures additional information about the `status`, using any of the arguments from the previous syntaxes. If an error condition occurs, such as no message received within the specified timeout, the `status` will be `false`, and this function will not display an error.

Examples

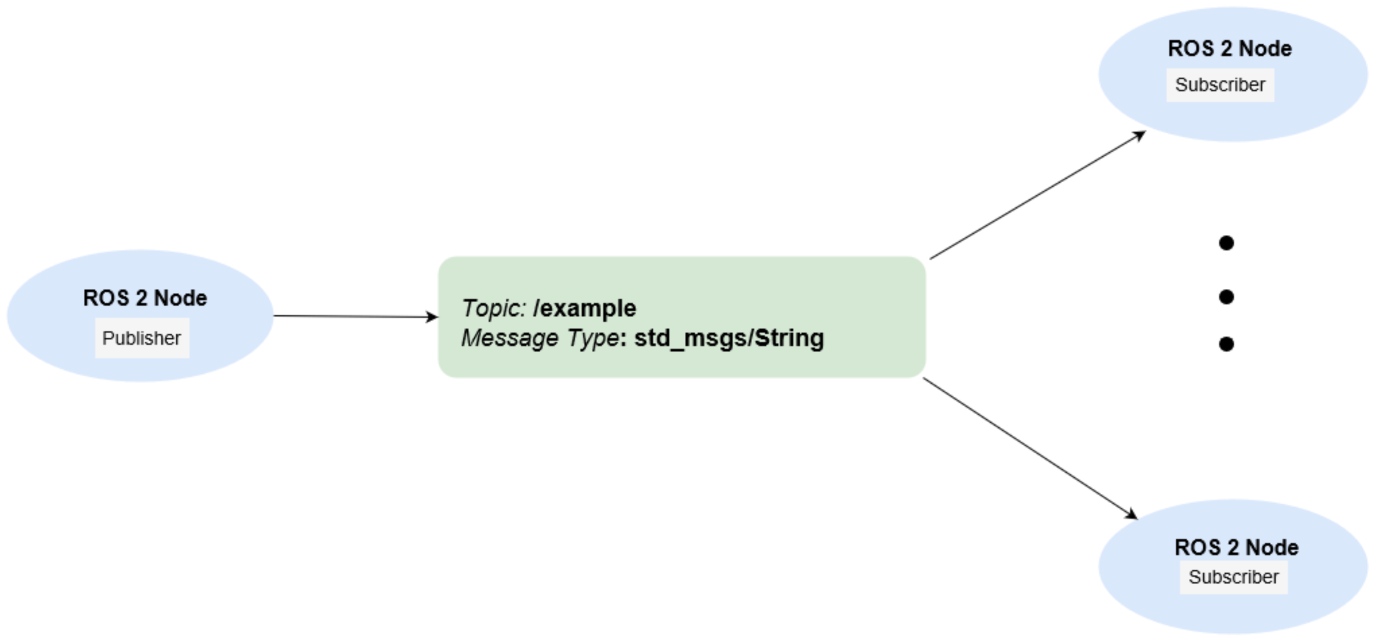
Exchange Data with ROS 2 Publishers and Subscribers

This example shows how to publish and subscribe to topics in a ROS 2 network.

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.



Besides how to publish and subscribe to topics in a ROS 2 network, this example 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
/rosout
/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(5)
laserSub = ros2subscriber(detectNode,"/scan");
pause(5)
```

Use `receive` to wait for a new message. Specify a timeout of 10 seconds. The output `scanData` contains the received message data. `status` indicates whether a message was received successfully and `statustext` provides additional information about the `status`.

```
[scanData,status,statustext] = 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");
pause(5)
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)
```

```
       0.00235920447111606       -0.0201184589892978        0.0203969078651195
```

```
disp(orient)
```

```
       -0.0118389124011118       0.00676849978014866        0.0387860955311228
```

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

“Create Callbacks for Graphics Objects” 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
/rosout
/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”
- “Generate ROS 2 Custom Messages in MATLAB”

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.

`status` — Status of the message reception
`logical` scalar

Status of the message reception, returned as a `logical` scalar. If no message is received, status will be `false`.

Note Use the `status` output argument when you use receive for code generation. This will avoid runtime errors and outputs the status instead, which can be reacted to in the calling code.

`statustext` — Status text associated with the message reception status
character vector

Status text associated with the message reception, returned as one of the following:

- `'success'` — The message was successfully received.
- `'timeout'` — The message was not received within the specified timeout.
- `'unknown'` — The message was not received due to unknown errors.

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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- To monitor the message reception status and react in the calling code, use the `status` output argument. This will avoid runtime errors when no message is received.

See Also

`ros2subscriber`

getTransform

Return the transformation between two coordinate frames

Syntax

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

Description

`tf = getTransform(tftree,targetframe,sourceframe)` gets and returns the latest known transformation between two coordinate frames. `tf` represents the transformation that takes coordinates in the `sourceframe` into the corresponding coordinates in the `targetframe`. The returned transformation `tf` is empty if it does not exist in the tree.

`tf = getTransform(tftree,targetframe,sourceframe,sourcetime)` returns the transformation at the time `sourcetime`. An error is displayed if the transformation at that time is not available.

`tf = getTransform(tftree,targetframe,sourceframe,Timeout=timeout)` specifies a timeout period in seconds, to wait until the transformation between two coordinates frames is available. Use `timeout` as `Inf` to wait indefinitely. If the transformation does not become available in the timeout period, MATLAB displays an error. Use this syntax with any of the input arguments in previous syntaxes.

Examples

Create a ROS 2 Transformation Tree

This example assumes that a ROS 2 node publishes transformations between `robot_base` and `camera_center`. For example, a real or simulated TurtleBot would do that.

Create a ROS 2 node on domain ID 25. Use the example helper function to publish transformation data.

```
node = ros2node("/matlabNode",25);
exampleHelperROS2StartTfPublisher
```

Retrieve the transformation tree object.

```
tftree = ros2tf(node);
pause(1)
```

Use the `AvailableFrames` property to see the transformation frames available. These transformations were specified separately prior to connecting to the network.

```
frames = tftree.AvailableFrames
```

```
frames = 3×1 cell
    {'camera_center' }
    {'mounting_point'}
    {'robot_base'    }
```

Use the `LastUpdateTime` property to see the time when the last transformation was received.

```
updateTime = tftree.LastUpdateTime
```

```
updateTime = struct with fields:
    MessageType: 'builtin_interfaces/Time'
            sec: 1670925404
        nanosec: 440832800
```

Wait for the transformation that takes data from `camera_center` to `robot_base`. It waits for the transformation to be valid within 5 seconds.

```
getTransform(tftree,'robot_base','camera_center', Timeout=5)
```

```
ans = struct with fields:
       MessageType: 'geometry_msgs/TransformStamped'
            header: [1×1 struct]
    child_frame_id: 'camera_center'
         transform: [1×1 struct]
```

Define a point [3 1.5 0.2] in the camera's coordinate frame.

```
pt = ros2message('geometry_msgs/PointStamped');
pt.header.frame_id = 'camera_center';
pt.point.x = 3;
pt.point.y = 1.5;
pt.point.z = 0.2;
```

The transformation is now available, so transform the point into the `robot_base` frame.

```
tfpt = transform(tftree,'robot_base',pt)
```

```
tfpt = struct with fields:
    MessageType: 'geometry_msgs/PointStamped'
         header: [1×1 struct]
          point: [1×1 struct]
```

Display the transformed point coordinates.

```
tfpt.point
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Point'
              x: 1.2000
              y: 1.5000
              z: -2.5000
```

Stop the example transformation publisher.

```
exampleHelperROS2StopTfPublisher
```

Clear the node.

```
clear('node')
```

Input Arguments

`tftree` — ROS 2 transformation tree
`ros2tf` object handle

ROS 2 transformation tree, specified as `ros2tf` object handle. You can create a ROS 2 transformation tree by using the `ros2tf` object.

`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`.

`sourcetime` — ROS 2 or system time
`"struct"` | scalar

ROS 2 or system time, specified as a scalar or structure that resembles `ros2time`. The scalar input is converted into structure using `ros2time`. By default, `sourcetime` is the ROS 2 simulation time published on the `/clock` topic. If you set the `use_sim_time` ROS 2 parameter to true, `sourcetime` returns the system time.

Data Types: `struct` | `scalar`

`Timeout` — Timeout period
`0` (default) | numeric scalar in seconds

Timeout for receiving the transform, specified as a numeric scalar in seconds. If the transformation does not become available, MATLAB displays an error, but continues running the current program.

Output Arguments

`tf` — Transformation between coordinate frames
`"struct"`

Transformation between coordinate frames, returned as a structure that represents `geometry_msgs/TransformStamped`. Transformations are structured as a 3-D translation (three-element vector) and a 3-D rotation (quaternion).

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Accessing the last update time from the `ros2tf` object using its `LastUpdateTime` property is not supported when you are generating code using MATLAB Coder.
- Usage of the function inside MATLAB Function block in Simulink is not supported.

See Also
`ros2tf` | `transform` | `sendTransform` | `canTransform`

Topics
"Access the tf Transformation Tree in ROS 2"

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 that takes data from the coordinate frame of the`entity` to the `targetframe`. The transformation is then applied to the data in `entity`. `entity` is a ROS 2 message of a specific type and the transformed message is returned in `tfentity`. An error is displayed if the transformation does not exist.

`tfentity = transform(tftree,targetframe,entity,"msgtime")` uses the timestamp in the header of message `entity` as source time to retrieve and apply the transformation.

`tfentity = transform(tftree,targetframe,entity,sourcetime)` uses the time `sourcetime` to retrieve and apply the transformation to the message `entity`.

Examples

Create a ROS 2 Transformation Tree

This example assumes that a ROS 2 node publishes transformations between `robot_base` and `camera_center`. For example, a real or simulated TurtleBot would do that.

Create a ROS 2 node on domain ID 25. Use the example helper function to publish transformation data.

```
node = ros2node("/matlabNode",25);
exampleHelperROS2StartTfPublisher
```

Retrieve the transformation tree object.

```
tftree = ros2tf(node);
pause(1)
```

Use the `AvailableFrames` property to see the transformation frames available. These transformations were specified separately prior to connecting to the network.

```
frames = tftree.AvailableFrames
```

```
frames = 3×1 cell
    {'camera_center' }
    {'mounting_point'}
    {'robot_base'    }
```

Use the `LastUpdateTime` property to see the time when the last transformation was received.

```
updateTime = tftree.LastUpdateTime
```

```
updateTime = struct with fields:
    MessageType: 'builtin_interfaces/Time'
            sec: 1670925404
        nanosec: 440832800
```

Wait for the transformation that takes data from `camera_center` to `robot_base`. It waits for the transformation to be valid within 5 seconds.

```
getTransform(tftree,'robot_base','camera_center', Timeout=5)
```

```
ans = struct with fields:
       MessageType: 'geometry_msgs/TransformStamped'
            header: [1×1 struct]
    child_frame_id: 'camera_center'
         transform: [1×1 struct]
```

Define a point [3 1.5 0.2] in the camera's coordinate frame.

```
pt = ros2message('geometry_msgs/PointStamped');
pt.header.frame_id = 'camera_center';
pt.point.x = 3;
pt.point.y = 1.5;
pt.point.z = 0.2;
```

The transformation is now available, so transform the point into the `robot_base` frame.

```
tfpt = transform(tftree,'robot_base',pt)
```

```
tfpt = struct with fields:
    MessageType: 'geometry_msgs/PointStamped'
         header: [1×1 struct]
          point: [1×1 struct]
```

Display the transformed point coordinates.

```
tfpt.point
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Point'
              x: 1.2000
              y: 1.5000
              z: -2.5000
```

Stop the example transformation publisher.

```
exampleHelperROS2StopTfPublisher
```

Clear the node.

```
clear('node')
```

Input Arguments

`tftree` — ROS 2 transformation tree
`ros2tf` object handle

ROS 2 transformation tree, specified as `ros2tf` object handle. You can create a ROS 2 transformation tree by using the `ros2tf` object.

`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`.

`entity` — Initial message entity
`"struct"`

Initial message entity, specified as a structure which represents `ros2message`. This function determines the type of the input message `entity` and apply the appropriate transformation method.

Supported messages are:

- `geometry_msgs/PointStamped`
- `geometry_msgs/PoseStamped`
- `geometry_msgs/QuaternionStamped`
- `geometry_msgs/Vector3Stamped`
- `sensor_msgs/PointCloud2`

`sourcetime` — ROS 2 or system time
`"struct"` | scalar

ROS 2 or system time, specified as a scalar or structure that resembles `ros2time`. The scalar input is converted into structure using `ros2time`. By default, `sourcetime` is the ROS 2 simulation time published on the `/clock` topic. If you set the `use_sim_time` ROS 2 parameter to true, `sourcetime` returns the system time.

Data Types: `struct` | `scalar`

Output Arguments

`tfentity` — Transformed entity
`"struct"`

Transformed entity, returned as a structure that represents `ros2message`.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Accessing the last update time from the `ros2tf` object using its `LastUpdateTime` property is not supported when you are generating code using MATLAB Coder.
- Usage of the function inside MATLAB Function block in Simulink is not supported.

See Also
`ros2tf` | `getTransform` | `sendTransform` | `canTransform`

Topics
"Access the tf Transformation Tree in ROS 2"

sendTransform

Send a transformation to the ROS 2 network

Syntax

```
sendTransform(tftree,tf)
sendTransform(tftree,tf,UseStatic=false)
sendTransform(tftree,tf,UseStatic=true)
```

Description

`sendTransform(tftree,tf)` broadcasts a dynamic or static transformation `tf` to the ROS 2 network over `/tf` or `/tf_static` topic. `tf` is a scalar message or a message list of type `geometry_msgs/TransformStamped`.

`sendTransform(tftree,tf,UseStatic=false)` broadcasts a dynamic transformation to the ROS 2 network over `/tf` topic.

`sendTransform(tftree,tf,UseStatic=true)` broadcasts a static transformation to the ROS 2 network over `/tf_static` topic.

Examples

Access the tf Transformation Tree in ROS 2

The `tf` system in ROS 2 keeps track of multiple coordinate frames and maintains the relationship between them in a tree structure. `tf` is distributed, so that all coordinate frame information is available to every node in the ROS 2 network. MATLAB® enables you to access this transformation tree. This example familiarizes you with accessing the available coordinate frames, retrieving transformations between them, and transform points, vectors, and other entities between any two coordinate frames.

Create a ROS 2 node on domain ID 25.

```
node = ros2node("/matlabNode",25);
```

To create a realistic environment for this example, use `exampleHelperROS2StartTfPublisher` to broadcast several transformations. The transformations represent a camera that is mounted on a robot.

There are three coordinate frames that are defined in this transformation tree:

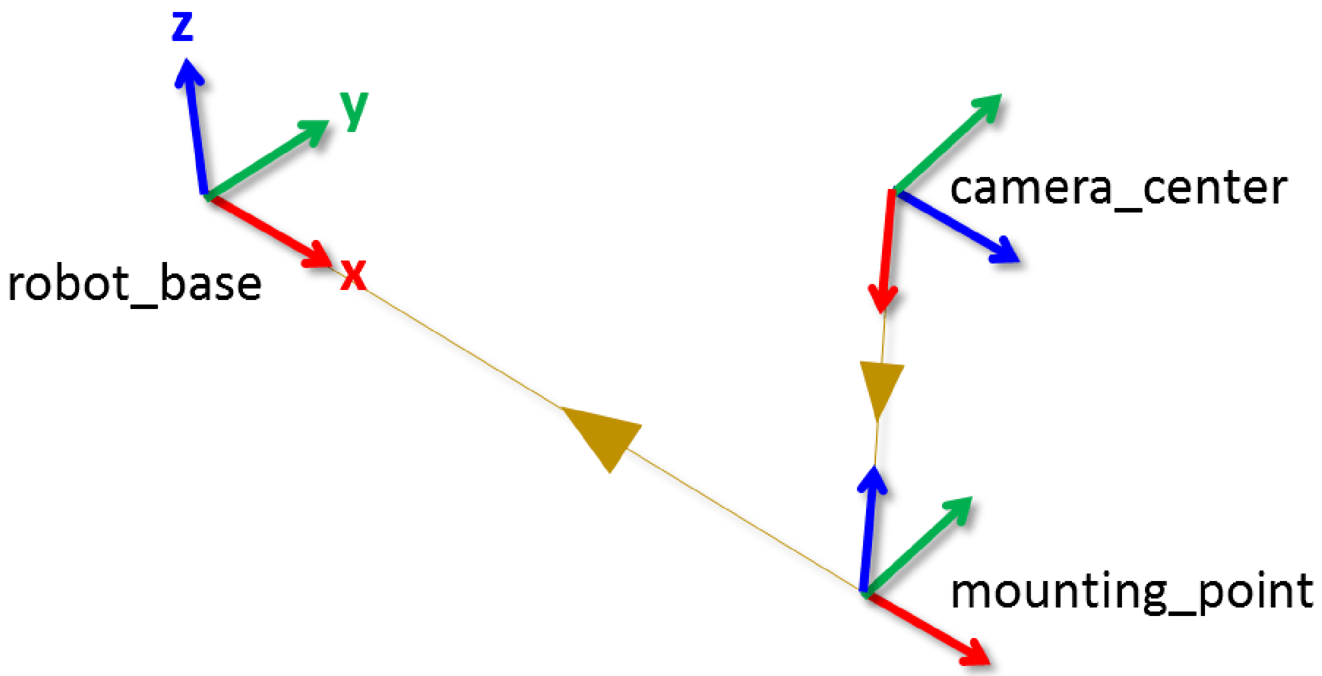
- the robot base frame (`robot_base`)
- the camera's mounting point (`mounting_point`)
- the optical center of the camera (`camera_center`)

Two transformations are being published:

- the transformation from the robot base to the camera's mounting point
- the transformation from the mounting point to the center of the camera

```
exampleHelperROS2StartTfPublisher
```

A visual representation of the three coordinate frames looks as follows.



Here, the x, y, and z axes of each frame are represented by red, green, and blue lines respectively. The parent-child relationship between the coordinate frames is shown through a brown arrow pointing from the child to its parent frame.

Create a new transformation tree using `ros2tf` object. You can use this object to access all available transformations and apply them to different entities.

```
tftree = ros2tf(node,...
"DynamicBroadcasterQoS", struct('Depth', 50), ...
"DynamicListenerQoS", struct('Reliability','besteffort'), ...
"StaticBroadcasterQoS", struct('Depth',10), ...
"StaticListenerQoS", struct('Durability','volatile'));
```

Once the object is created, it starts receiving tf transformations and buffers them internally. Keep the `tftree` variable in the workspace so that it continues to receive data.

Pause for a little bit to make sure that all transformations are received.

```
pause(2);
```

You can see the names of all the available coordinate frames by accessing the `AvailableFrames` property.

```
tftree.AvailableFrames
```

```
ans = 3×1 cell
    {'camera_center' }
    {'mounting_point'}
    {'robot_base'    }
```

This should show the three coordinate frames that describe the relationship between the camera, its mounting point, and the robot.

Receive Transformations

Now that the transformations are available, you can inspect them. Any transformation is described by a ROS 2 `geometry_msgs/TransformStamped` message and has a translational and rotational component.

Retrieve the transformation that describes the relationship between the mounting point and the camera center. Use the `getTransform` function to do that.

```
mountToCamera = getTransform(tftree, 'mounting_point', 'camera_center');
mountToCameraTranslation = mountToCamera.transform.translation
```

```
mountToCameraTranslation = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              x: 0
              y: 0
              z: 0.5000
```

```
quat = mountToCamera.transform.rotation
```

```
quat = struct with fields:
    MessageType: 'geometry_msgs/Quaternion'
              x: 0
              y: 0.7071
              z: 0
              w: 0.7071
```

```
mountToCameraRotationAngles = rad2deg(quat2eul([quat.w quat.x quat.y quat.z]))
```

```
mountToCameraRotationAngles = 1×3

     0    90     0
```

Relative to the mounting point, the camera center is located 0.5 meters along the z-axis and is rotated by 90 degrees around the y-axis.

To inspect the relationship between the robot base and the camera's mounting point, call `getTransform` again.

```
baseToMount = getTransform(tftree, 'robot_base', 'mounting_point');
baseToMountTranslation = baseToMount.transform.translation
```

```
baseToMountTranslation = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              x: 1
```

```
          y: 0
          z: 0
```

```
baseToMountRotation = baseToMount.transform.rotation
```

```
baseToMountRotation = struct with fields:
    MessageType: 'geometry_msgs/Quaternion'
              x: 0
              y: 0
              z: 0
              w: 1
```

```
baseToMountRotationRotationAngles = rad2deg(quat2eul([baseToMountRotation.w baseToMountRotation.x
```

The mounting point is located at 1 meter along the robot base's x-axis.

Apply Transformations

Assume now that you have a 3D point that is defined in the `camera_center` coordinate frame and you want to calculate what the point coordinates in the `robot_base` frame are.

Use the `getTransform` function to wait until the transformation between the `camera_center` and `robot_base` coordinate frames becomes available. This call blocks until the transformation that takes data from `camera_center` to `robot_base` is valid and available in the transformation tree.

```
getTransform(tftree,'robot_base','camera_center','Timeout', Inf);
```

Now define a point at position `[3 1.5 0.2]` in the camera center's coordinate frame. You will subsequently transform this point into `robot_base` coordinates.

```
pt = ros2message('geometry_msgs/PointStamped');
pt.header.frame_id = 'camera_center';
pt.point.x = 3;
pt.point.y = 1.5;
pt.point.z = 0.2;
```

You can transform the point coordinates by calling the `transform` function on the transformation tree object. Specify what the target coordinate frame of this transformation is. In this example, use `robot_base`.

```
tfpt = transform(tftree, 'robot_base', pt)
```

```
tfpt = struct with fields:
    MessageType: 'geometry_msgs/PointStamped'
         header: [1×1 struct]
          point: [1×1 struct]
```

Besides `PointStamped` messages, the `transform` function allows you to transform other entities like poses (`geometry_msgs/PoseStamped`), quaternions (`geometry_msgs/QuaternionStamped`), and point clouds (`sensor_msgs/PointCloud2`).

If you want to store a transformation, you can retrieve it with the `getTransform` function.

```
robotToCamera = getTransform(tftree, 'robot_base', 'camera_center')
robotToCamera = struct with fields:
       MessageType: 'geometry_msgs/TransformStamped'
            header: [1×1 struct]
    child_frame_id: 'camera_center'
         transform: [1×1 struct]
```

This transformation can be used to transform coordinates in the `camera_center` frame into coordinates in the `robot_base` frame.

```
robotToCamera.transform.translation
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Vector3'
              x: 1
              y: 0
              z: 0.5000
```

```
robotToCamera.transform.rotation
```

```
ans = struct with fields:
    MessageType: 'geometry_msgs/Quaternion'
              x: 0
              y: 0.7071
              z: 0
              w: 0.7071
```

Send Transformations

You can also broadcast a new transformation to the ROS 2 network.

Assume that you have a simple transformation that describes the offset of the `wheel` coordinate frame relative to the `robot_base` coordinate frame. The wheel is mounted -0.2 meters along the y-axis and -0.3 along the z-axis. The wheel has a relative rotation of 30 degrees around the y-axis.

Create the corresponding `geometry_msgs/TransformStamped` message that describes this transformation. The source coordinate frame, `wheel`, is placed to the `child_frame_id` property. The target coordinate frame, `robot_base`, is added to the `header.FrameId` property.

```
tfStampedMsg = ros2message('geometry_msgs/TransformStamped');
tfStampedMsg.child_frame_id = 'wheel';
tfStampedMsg.header.frame_id = 'robot_base';
```

The transformation itself is described in the `Transform` property. It contains a `Translation` and a `Rotation`. Fill in the values that are listed above.

The `Rotation` is described as a quaternion. To convert the 30 degree rotation around the y-axis to a quaternion, you can use the `axang2quat` (Navigation Toolbox) function. The y-axis is described by the `[0 1 0]` vector and 30 degrees can be converted to radians with the `deg2rad` function.

```
tfStampedMsg.transform.translation.x = 0;
tfStampedMsg.transform.translation.y = -0.2;
tfStampedMsg.transform.translation.z = -0.3;

quatrot = axang2quat([0 1 0 deg2rad(30)]);
tfStampedMsg.transform.rotation.w = quatrot(1);
tfStampedMsg.transform.rotation.x = quatrot(2);
tfStampedMsg.transform.rotation.y = quatrot(3);
tfStampedMsg.transform.rotation.z = quatrot(4);
```

Use `ros2time` to retrieve the current system time and use that to timestamp the transformation. This lets the recipients know at which point in time this transformation was valid.

```
tfStampedMsg.header.stamp = ros2time(node,'now');
```

Broadcast this transformation over the ROS 2 network.

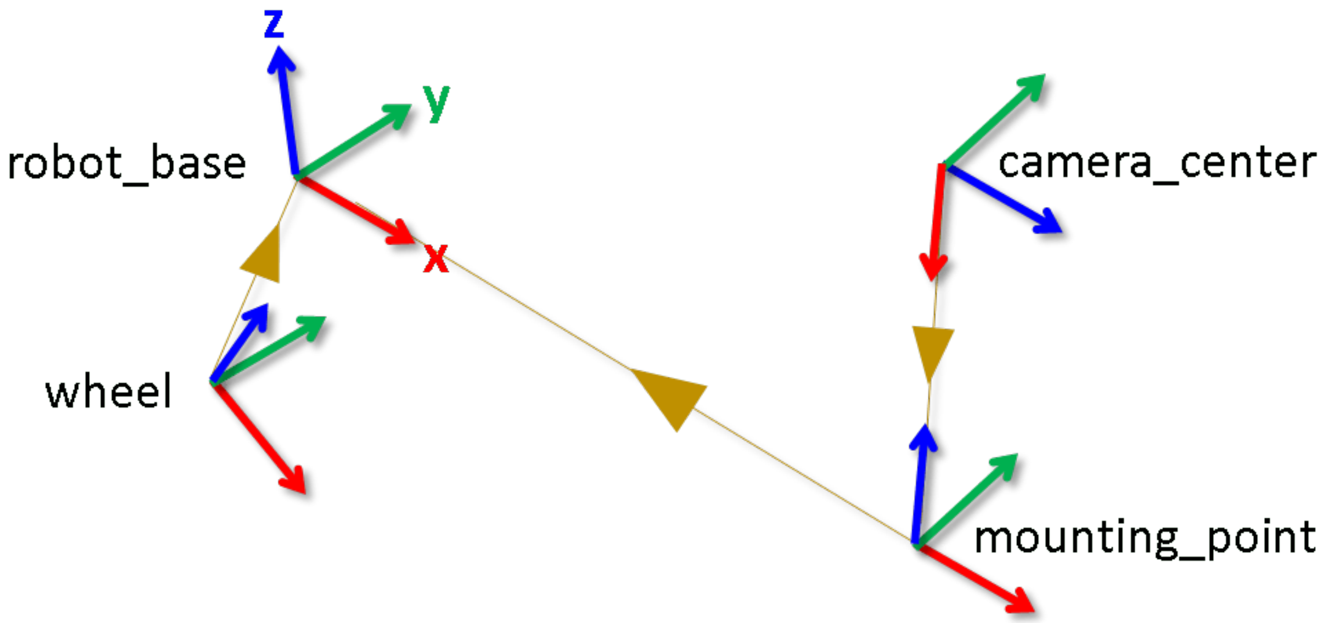
```
sendTransform(tftree, tfStampedMsg)
```

The new `wheel` coordinate frame is now also in the list of available coordinate frames.

```
tftree.AvailableFrames
```

```
ans = 4×1 cell
    {'camera_center' }
    {'mounting_point'}
    {'robot_base'    }
    {'wheel'         }
```

The final visual representation of all four coordinate frames looks as follows.



You can see that the `wheel` coordinate frame has the translation and rotation that you specified in sending the transformation.

Stop Example Publisher

Stop the example transformation publisher.

```
exampleHelperROS2StopTfPublisher
```

Clear the node.

```
clear('node')
```

Input Arguments

`tftree` — ROS 2 transformation tree
`ros2tf` object handle

ROS 2 transformation tree, specified as `ros2tf` object handle. You can create a ROS 2 transformation tree by using the `ros2tf` object.

`tf` — Transformation between coordinate frames
`"struct"`

Transformation between coordinate frames, returned as a structure that represents `geometry_msgs/TransformStamped`. Transformations are structured as a 3-D translation (three-element vector) and a 3-D rotation (quaternion).

`UseStatic` — Static transform broadcaster setting
`"false"` (default) | `"true"`

Static transform broadcaster setting, specified as `"true"` or `"false"`. It broadcasts a static or dynamic transformation over `/tf_static` or `/tf` topic.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Usage of the function inside MATLAB Function block in Simulink is not supported.

See Also

`getTransform` | `transform` | `canTransform`

Topics
“Create a ROS 2 Transformation Tree” on page 3-173
“Access the tf Transformation Tree in ROS 2”

canTransform

Verify if transformation is available

Syntax

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

Description

`isAvailable = canTransform(tftree,targetframe,sourceframe)` verifies if a transformation that takes coordinates in the `sourceframe` into the corresponding coordinates in the `targetframe` is available. `isAvailable` is `true` if that transformation is available and `false`. Use `getTransform` to retrieve the transformation.

`isAvailable = canTransform(tftree,targetframe,sourceframe,sourcetime)` verifies that the transformation is available for the time `sourcetime`. If `sourcetime` is outside of the buffer window for the transformation tree, the function returns `false`. Use `getTransform` with the `sourcetime` argument to retrieve the transformation.

Input Arguments

`tftree` — ROS 2 transformation tree
`ros2tf` object handle

ROS 2 transformation tree, specified as `ros2tf` object handle. You can create a ROS 2 transformation tree by using the `ros2tf` object.

`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`.

`sourcetime` — ROS 2 or system time
`"struct"` | scalar

ROS 2 or system time, specified as a scalar or structure that resembles `ros2time`. The scalar input is converted into structure using `ros2time`. By default, `sourcetime` is the ROS 2 simulation time published on the `/clock` topic. If you set the `use_sim_time` ROS 2 parameter to true, `sourcetime` returns the system time.

Data Types: `struct` | `scalar`

Output Arguments

`isAvailable` — Transformation verification entity
boolean

Transformation verification entity, returned as boolean. It verifies if a transformation is available between the `sourceframe` and `targetframe`.

The function returns "`false`" if:

- `sourcetime` is outside the buffer window for a `tftree` object.
- `sourcetime` is in the future.
- The transformation is not published yet.

Version History

Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using MATLAB® Coder™.

Usage notes and limitations:

- Supported only for the Build Type, `Executable`.
- Accessing the last update time from the `ros2tf` object using its `LastUpdateTime` property is not supported when you are generating code using MATLAB Coder.
- Usage of the function inside MATLAB Function block in Simulink is not supported.

See Also

`getTransform` | `ros2tf`

Topics
"Access the tf Transformation Tree in ROS 2"

hasFrame

Determine if another Velodyne point cloud is available in the ROS messages

Syntax

```
isAvailable = hasFrame(veloReader)
```

Description

`isAvailable = hasFrame(veloReader)` determines if another point cloud is available in the Velodyne ROS messages.

Examples

Work with Velodyne ROS Messages

This example shows how to handle `VelodyneScan` messages from a Velodyne LiDAR.

Velodyne ROS messages store data in a format that requires some interpretation before it can be used for further processing. MATLAB® can help you by formatting Velodyne ROS messages for easy use.

Prerequisites: “Work with Basic ROS Messages”

Load Sample Messages

Load sample Velodyne messages. These messages are populated with data gathered from Velodyne LiDAR sensor.

```
load("lidarData_ConstructionRoad.mat")
```

VelodyneScan Messages

`VelodyneScan` messages are ROS messages that contain Velodyne LIDAR scan packets. You can see the standard ROS format for a `VelodyneScan` message by creating an empty message of the appropriate type. Use messages in structure format for better performance.

```
emptyveloScan = rosmessage("velodyne_msgs/VelodyneScan","DataFormat","struct")
```

```
emptyveloScan = struct with fields:
    MessageType: 'velodyne_msgs/VelodyneScan'
         Header: [1×1 struct]
        Packets: [0×1 struct]
```

Since you created an empty message, `emptyveloScan` does not contain any meaningful data. For convenience, the loaded `lidarData_ConstructionRoad.mat` file contains a set of `VelodyneScan` messages that are fully populated and stored in the `msgs` variable. Each element in the `msgs` cell array is a `VelodyneScan` ROS message struct. The primary data in each `VelodyneScan` message is in the `Packets` property, it contains multiple `VelodynePacket` messages. You can see the standard ROS format for a VelodynePacket message by creating an empty message of the appropriate type.

```
emptyveloPkt = rosmessage("velodyne_msgs/VelodynePacket","DataFormat","struct")
```

```
emptyveloPkt = struct with fields:
    MessageType: 'velodyne_msgs/VelodynePacket'
          Stamp: [1×1 struct]
           Data: [1206×1 uint8]
```

Create Velodyne ROS Message Reader

The `velodyneROSMessageReader` object reads point cloud data from `VelodyneScan` ROS messages based on their specified model type. Note that providing an incorrect device model may result in improperly calibrated point clouds. This example uses messages from the `"HDL32E"` model.

```
veloReader = velodyneROSMessageReader(msgs,"HDL32E")
```

```
veloReader = 
  velodyneROSMessageReader with properties:

    VelodyneMessages: {28×1 cell}
         DeviceModel: 'HDL32E'
     CalibrationFile: 'M:\jobarchive\Bdoc21b\2021_06_16_h16m50s15_job1697727_pass\matlab\toolbox
      NumberOfFrames: 55
            Duration: 2.7477 sec
           StartTime: 1145.2 sec
             EndTime: 1147.9 sec
          Timestamps: [1145.2 sec    1145.2 sec    1145.3 sec    1145.3 sec    1145.4 sec    114
         CurrentTime: 1145.2 sec
```

Extract Point Clouds

You can extract point clouds from the raw packets message with the help of this `velodyneROSMessageReader` object. By providing a specific frame number or timestamp, one point cloud can be extracted from `velodyneROSMessageReader` object using the `readFrame` object function. If you call `readFrame` without a frame number or timestamp, it extracts the next point cloud in the sequence based on the `CurrentTime` property.

Create a duration scalar that represents one second after the first point cloud reading.

```
timeDuration = veloReader.StartTime + seconds(1);
```

Read the first point cloud recorded at or after the given time duration.

```
ptCloudObj = readFrame(veloReader,timeDuration);
```

Access `Location` data in the point cloud.

```
ptCloudLoc = ptCloudObj.Location;
```

Reset the `CurrentTime` property of `veloReader` to the default value

```
reset(veloReader)
```

Display All Point Clouds

You can also loop through all point clouds in the input Velodyne ROS messages.

Define *x*-, *y*-, and *z*-axes limits for `pcplayer` in meters. Label the axes.

```
xlimits = [-60 60];
ylimits = [-60 60];
zlimits = [-20 20];
```

Create the point cloud player.

```
player = pcplayer(xlimits,ylimits,zlimits);
```

Label the axes.

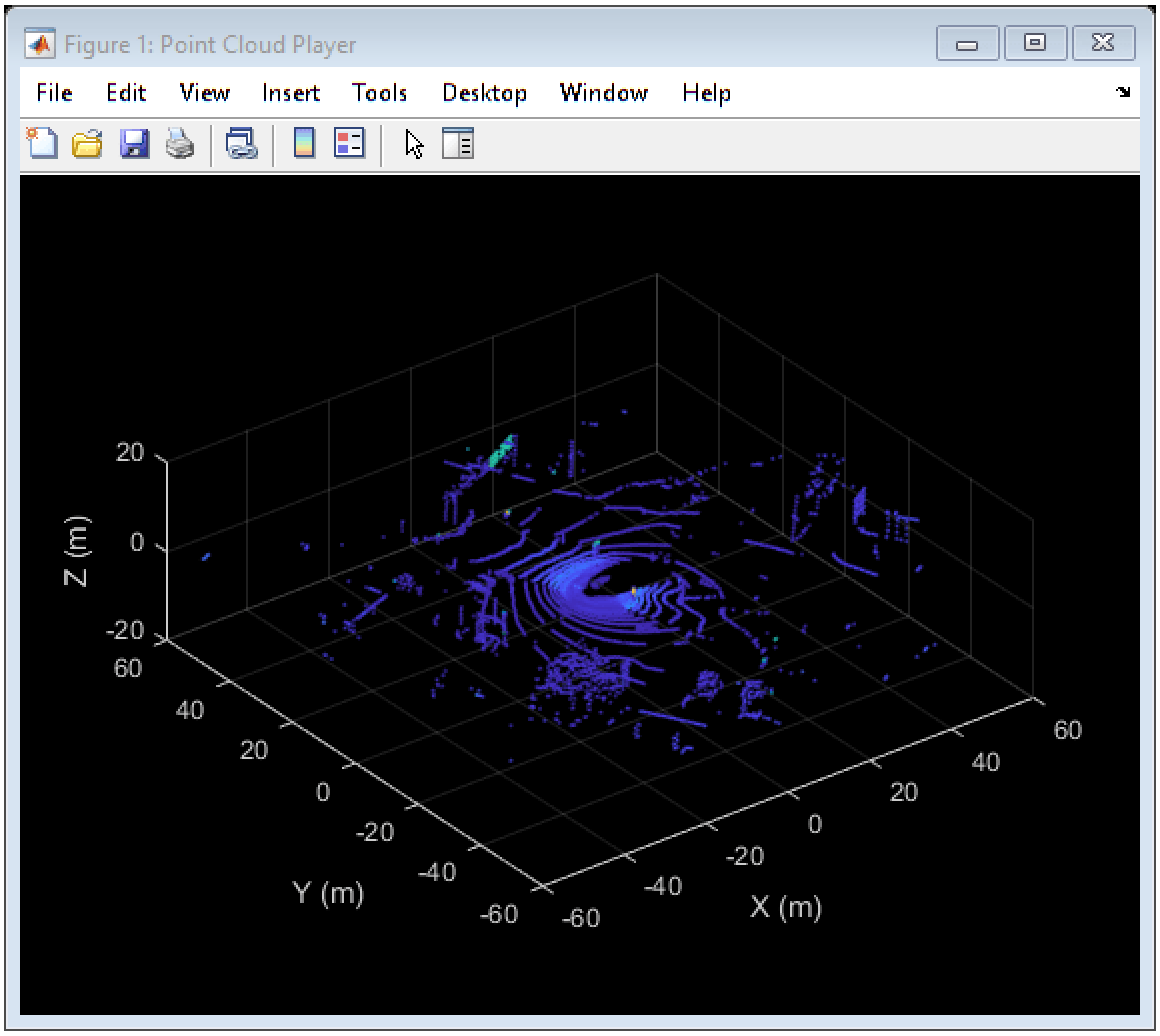
```
xlabel(player.Axes,"X (m)");
ylabel(player.Axes,"Y (m)");
zlabel(player.Axes,"Z (m)");
```

The first point cloud of interest is captured at 0.3 second into the input messages. Set the `CurrentTime` property to that time to begin reading point clouds from there.

```
veloReader.CurrentTime = veloReader.StartTime + seconds(0.3);
```

Display the point cloud stream for 2 seconds. To check if a new frame is available and continue past 2 seconds, remove the last `while` condition. Iterate through the file by calling `readFrame` to read in point clouds. Display them using the point cloud player.

```
while(hasFrame(veloReader) && isOpen(player) && (veloReader.CurrentTime < veloReader.StartTime +
    ptCloudObj = readFrame(veloReader);
    view(player,ptCloudObj.Location,ptCloudObj.Intensity);
    pause(0.1);
end
```



Input Arguments

veloReader — Velodyne ROS message reader
`velodyneROSMessageReader` object

Velodyne ROS message reader, specified as a `velodyneROSMessageReader` object.

Output Arguments

isAvailable — Indicator of frame availability
`true` or `1` | `false` or `0`

Indicator of frame availability, returned as a logical `1` (`true`) when a later frame is available or a logical `0` (`false`) when a later frame is not available.

Version History

Introduced in R2020b

See Also

`velodyneROSMessageReader` | `readFrame` | `reset`

readFrame

Read point cloud frame from ROS message

Syntax

```
ptCloud = readFrame(veloReader)
ptCloud = readFrame(veloReader,frameNumber)
ptCloud = readFrame(veloReader,frameTime)
```

Description

`ptCloud = readFrame(veloReader)` reads the next point cloud frame from the Velodyne ROS messages and returns a `pointCloud` object. The next point cloud frame is the first point cloud available at or after the value of the `CurrentTime` property of the Velodyne ROS message reader object `veloReader`.

`ptCloud = readFrame(veloReader,frameNumber)` reads the point cloud with the specified frame number from the Velodyne ROS messages.

`ptCloud = readFrame(veloReader,frameTime)` reads the first point cloud available at or after the specified timestamp `frameTime`.

Examples

Work with Velodyne ROS Messages

This example shows how to handle `VelodyneScan` messages from a Velodyne LiDAR.

Velodyne ROS messages store data in a format that requires some interpretation before it can be used for further processing. MATLAB® can help you by formatting Velodyne ROS messages for easy use.

Prerequisites: “Work with Basic ROS Messages”

Load Sample Messages

Load sample Velodyne messages. These messages are populated with data gathered from Velodyne LiDAR sensor.

```
load("lidarData_ConstructionRoad.mat")
```

VelodyneScan Messages

`VelodyneScan` messages are ROS messages that contain Velodyne LIDAR scan packets. You can see the standard ROS format for a `VelodyneScan` message by creating an empty message of the appropriate type. Use messages in structure format for better performance.

```
emptyveloScan = rosmessage("velodyne_msgs/VelodyneScan","DataFormat","struct")
```

```
emptyveloScan = struct with fields:
    MessageType: 'velodyne_msgs/VelodyneScan'
```

```
         Header: [1×1 struct]
        Packets: [0×1 struct]
```

Since you created an empty message, `emptyveloScan` does not contain any meaningful data. For convenience, the loaded `lidarData_ConstructionRoad.mat` file contains a set of `VelodyneScan` messages that are fully populated and stored in the `msgs` variable. Each element in the `msgs` cell array is a `VelodyneScan` ROS message struct. The primary data in each `VelodyneScan` message is in the `Packets` property, it contains multiple `VelodynePacket` messages. You can see the standard ROS format for a VelodynePacket message by creating an empty message of the appropriate type.

```
emptyveloPkt = rosmessage("velodyne_msgs/VelodynePacket","DataFormat","struct")
```

```
emptyveloPkt = struct with fields:
    MessageType: 'velodyne_msgs/VelodynePacket'
          Stamp: [1×1 struct]
           Data: [1206×1 uint8]
```

Create Velodyne ROS Message Reader

The `velodyneROSMessageReader` object reads point cloud data from `VelodyneScan` ROS messages based on their specified model type. Note that providing an incorrect device model may result in improperly calibrated point clouds. This example uses messages from the `"HDL32E"` model.

```
veloReader = velodyneROSMessageReader(msgs,"HDL32E")
```

```
veloReader = 
  velodyneROSMessageReader with properties:

    VelodyneMessages: {28×1 cell}
         DeviceModel: 'HDL32E'
     CalibrationFile: 'M:\jobarchive\Bdoc21b\2021_06_16_h16m50s15_job1697727_pass\matlab\toolbox
      NumberOfFrames: 55
            Duration: 2.7477 sec
           StartTime: 1145.2 sec
             EndTime: 1147.9 sec
          Timestamps: [1145.2 sec    1145.2 sec    1145.3 sec    1145.3 sec    1145.4 sec    114
         CurrentTime: 1145.2 sec
```

Extract Point Clouds

You can extract point clouds from the raw packets message with the help of this `velodyneROSMessageReader` object. By providing a specific frame number or timestamp, one point cloud can be extracted from `velodyneROSMessageReader` object using the `readFrame` object function. If you call `readFrame` without a frame number or timestamp, it extracts the next point cloud in the sequence based on the `CurrentTime` property.

Create a duration scalar that represents one second after the first point cloud reading.

```
timeDuration = veloReader.StartTime + seconds(1);
```

Read the first point cloud recorded at or after the given time duration.

```
ptCloudObj = readFrame(veloReader,timeDuration);
```

Access `Location` data in the point cloud.

```
ptCloudLoc = ptCloudObj.Location;
```

Reset the `CurrentTime` property of `veloReader` to the default value

```
reset(veloReader)
```

Display All Point Clouds

You can also loop through all point clouds in the input Velodyne ROS messages.

Define *x*-, *y*-, and *z*-axes limits for `pcplayer` in meters. Label the axes.

```
xlimits = [-60 60];
ylimits = [-60 60];
zlimits = [-20 20];
```

Create the point cloud player.

```
player = pcplayer(xlimits,ylimits,zlimits);
```

Label the axes.

```
xlabel(player.Axes,"X (m)");
ylabel(player.Axes,"Y (m)");
zlabel(player.Axes,"Z (m)");
```

The first point cloud of interest is captured at 0.3 second into the input messages. Set the `CurrentTime` property to that time to begin reading point clouds from there.

```
veloReader.CurrentTime = veloReader.StartTime + seconds(0.3);
```

Display the point cloud stream for 2 seconds. To check if a new frame is available and continue past 2 seconds, remove the last `while` condition. Iterate through the file by calling `readFrame` to read in point clouds. Display them using the point cloud player.

```
while(hasFrame(veloReader) && isOpen(player) && (veloReader.CurrentTime < veloReader.StartTime +
    ptCloudObj = readFrame(veloReader);
    view(player,ptCloudObj.Location,ptCloudObj.Intensity);
    pause(0.1);
end
```

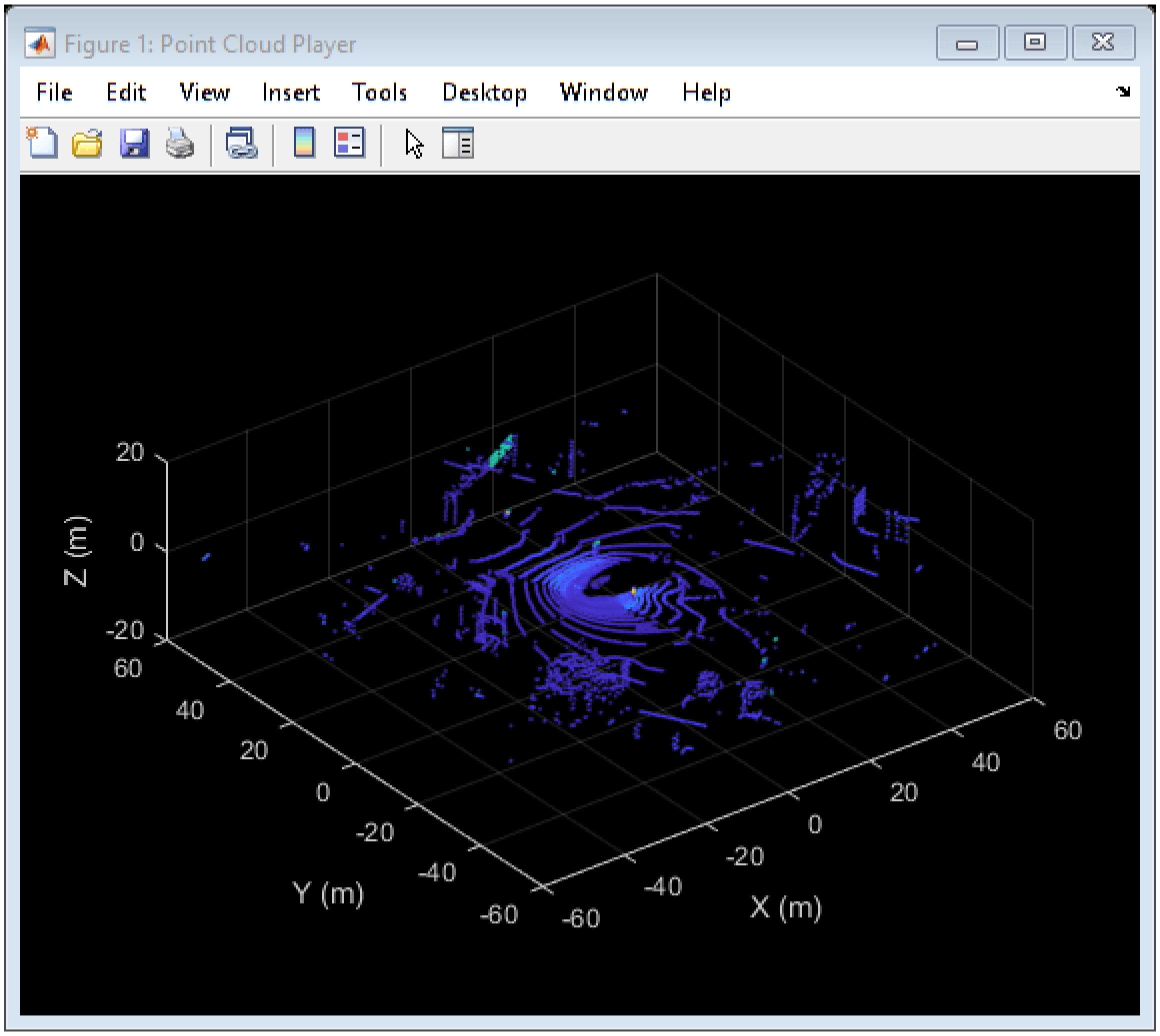


Figure 1: Point Cloud Player
File Edit View Insert Tools Desktop Window Help
Z (m)
Y (m)
X (m)
20
0
-20
60
40
20
0
-20
-40
-60
-60
-40
-20
0
20
40
60

Input Arguments

veloReader — Velodyne ROS message reader
`velodyneROSMesasgeReader` object

Velodyne ROS message reader, specified as a `velodyneROSMessageReader` object.

frameNumber — Frame number of desired point cloud
positive integer

Frame number of the desired point cloud, specified as a positive integer. Frame numbers are sequential in the `velodyneROSMessageReader` object.

frameTime — Frame time of desired point cloud
`duration` scalar

Frame time of the desired point cloud, specified as a `duration` scalar in seconds. The function return the first frame available at or after the specified timestamp.

Output Arguments

`ptCloud` — Point Cloud
`pointCloud` object

Point cloud, returned as `pointCloud` object.

Version History

Introduced in R2020b

See Also

`velodyneROSMessageReader` | `pointCloud` | `hasFrame` | `reset`

reset

Reset `CurrentTime` property of `velodyneROSMessageReader` object to default value

Syntax

```
reset(veloReader)
```

Description

`reset(veloReader)` resets the `CurrentTime` property of the specified `velodyneROSMessageReader` object to the default value. The default value is the value of the `StartTime` property of the `velodyneROSMessageReader` object.

Examples

Work with Velodyne ROS Messages

This example shows how to handle `VelodyneScan` messages from a Velodyne LiDAR.

Velodyne ROS messages store data in a format that requires some interpretation before it can be used for further processing. MATLAB® can help you by formatting Velodyne ROS messages for easy use.

Prerequisites: “Work with Basic ROS Messages”

Load Sample Messages

Load sample Velodyne messages. These messages are populated with data gathered from Velodyne LiDAR sensor.

```
load("lidarData_ConstructionRoad.mat")
```

VelodyneScan Messages

`VelodyneScan` messages are ROS messages that contain Velodyne LIDAR scan packets. You can see the standard ROS format for a `VelodyneScan` message by creating an empty message of the appropriate type. Use messages in structure format for better performance.

```
emptyveloScan = rosmessage("velodyne_msgs/VelodyneScan","DataFormat","struct")
```

```
emptyveloScan = struct with fields:
    MessageType: 'velodyne_msgs/VelodyneScan'
         Header: [1×1 struct]
        Packets: [0×1 struct]
```

Since you created an empty message, `emptyveloScan` does not contain any meaningful data. For convenience, the loaded `lidarData_ConstructionRoad.mat` file contains a set of `VelodyneScan` messages that are fully populated and stored in the `msgs` variable. Each element in the `msgs` cell array is a `VelodyneScan` ROS message struct. The primary data in each `VelodyneScan` message is in the Packets property, it contains multiple VelodynePacket messages. You can see the standard ROS format for a VelodynePacket message by creating an empty message of the appropriate type.

```
emptyveloPkt = rosmessage("velodyne_msgs/VelodynePacket","DataFormat","struct")
```

```
emptyveloPkt = struct with fields:
    MessageType: 'velodyne_msgs/VelodynePacket'
          Stamp: [1×1 struct]
           Data: [1206×1 uint8]
```

Create Velodyne ROS Message Reader

The velodyneROSMessageReader object reads point cloud data from VelodyneScan ROS messages based on their specified model type. Note that providing an incorrect device model may result in improperly calibrated point clouds. This example uses messages from the "HDL32E" model.

```
veloReader = velodyneROSMessageReader(msgs,"HDL32E")
```

```
veloReader = 
  velodyneROSMessageReader with properties:

    VelodyneMessages: {28×1 cell}
         DeviceModel: 'HDL32E'
     CalibrationFile: 'M:\jobarchive\Bdoc21b\2021_06_16_h16m50s15_job1697727_pass\matlab\toolbox
      NumberOfFrames: 55
            Duration: 2.7477 sec
           StartTime: 1145.2 sec
             EndTime: 1147.9 sec
          Timestamps: [1145.2 sec    1145.2 sec    1145.3 sec    1145.3 sec    1145.4 sec    114
         CurrentTime: 1145.2 sec
```

Extract Point Clouds

You can extract point clouds from the raw packets message with the help of this velodyneROSMessageReader object. By providing a specific frame number or timestamp, one point cloud can be extracted from velodyneROSMessageReader object using the readFrame object function. If you call readFrame without a frame number or timestamp, it extracts the next point cloud in the sequence based on the CurrentTime property.

Create a duration scalar that represents one second after the first point cloud reading.

```
timeDuration = veloReader.StartTime + seconds(1);
```

Read the first point cloud recorded at or after the given time duration.

```
ptCloudObj = readFrame(veloReader,timeDuration);
```

Access Location data in the point cloud.

```
ptCloudLoc = ptCloudObj.Location;
```

Reset the CurrentTime property of veloReader to the default value

```
reset(veloReader)
```

Display All Point Clouds

You can also loop through all point clouds in the input Velodyne ROS messages.

Define *x*-, *y*-, and *z*-axes limits for `pcplayer` in meters. Label the axes.

```
xlimits = [-60 60];
ylimits = [-60 60];
zlimits = [-20 20];
```

Create the point cloud player.

```
player = pcplayer(xlimits,ylimits,zlimits);
```

Label the axes.

```
xlabel(player.Axes,"X (m)");
ylabel(player.Axes,"Y (m)");
zlabel(player.Axes,"Z (m)");
```

The first point cloud of interest is captured at 0.3 second into the input messages. Set the `CurrentTime` property to that time to begin reading point clouds from there.

```
veloReader.CurrentTime = veloReader.StartTime + seconds(0.3);
```

Display the point cloud stream for 2 seconds. To check if a new frame is available and continue past 2 seconds, remove the last `while` condition. Iterate through the file by calling `readFrame` to read in point clouds. Display them using the point cloud player.

```
while(hasFrame(veloReader) && isOpen(player) && (veloReader.CurrentTime < veloReader.StartTime +
    ptCloudObj = readFrame(veloReader);
    view(player,ptCloudObj.Location,ptCloudObj.Intensity);
    pause(0.1);
end
```

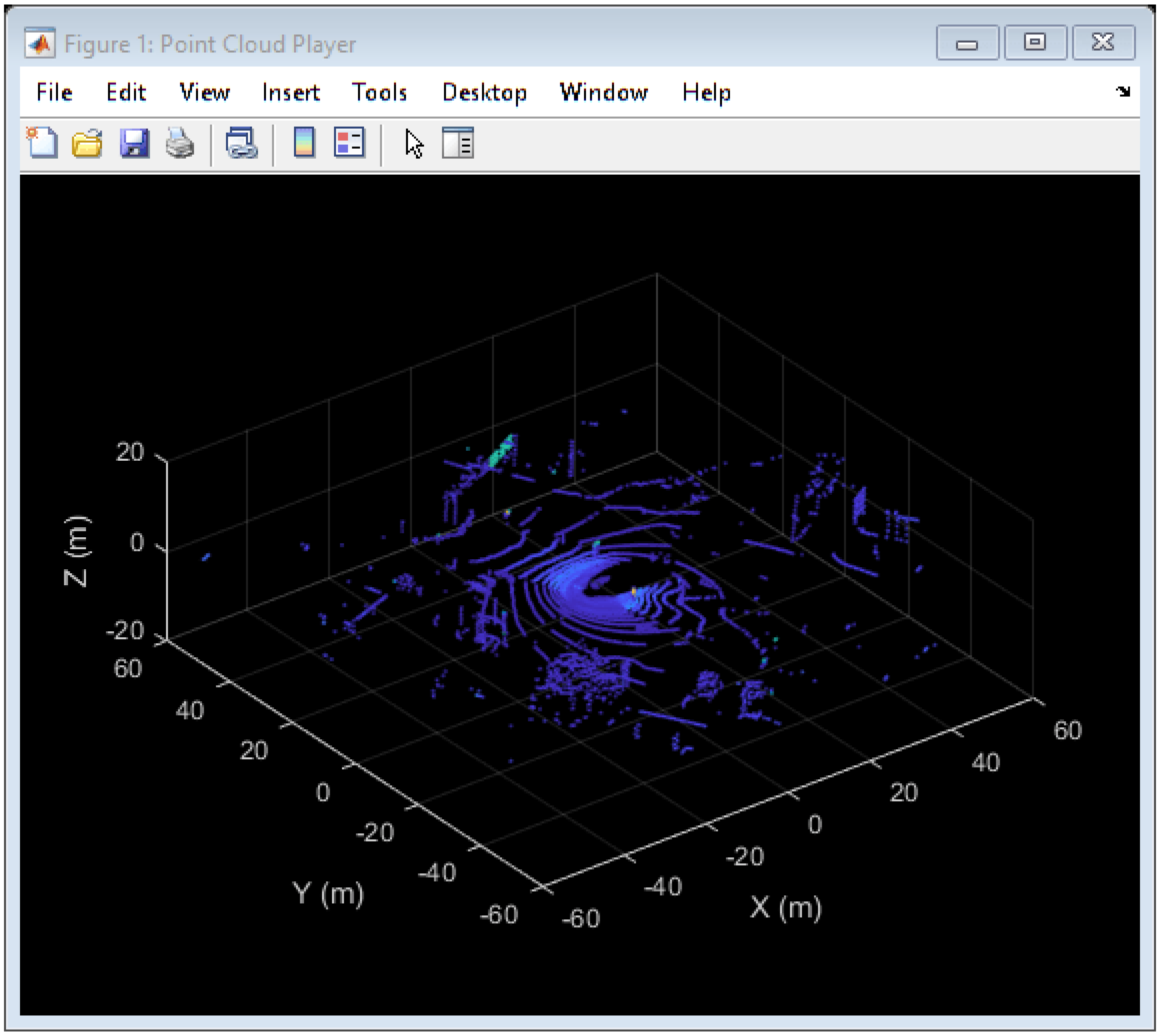


Figure 1: Point Cloud Player
File Edit View Insert Tools Desktop Window Help
Z (m)
20
0
-20
Y (m)
60
40
20
0
-20
-40
-60
X (m)
-60
-40
-20
0
20
40
60

Input Arguments

veloReader — Velodyne ROS message reader
`velodyneROSMessageReader` object

Velodyne ROS message reader, specified as a `velodyneROSMessageReader` object.

Version History

Introduced in R2020b

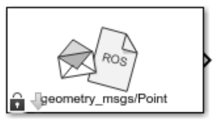
See Also

`velodyneROSMessageReader` | `readFrame` | `hasFrame`

Blocks

Blank Message

Create blank message using specified message type



Libraries:
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 **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).

Version History
Introduced in R2019b

Extended Capabilities

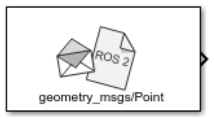
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Publish | Subscribe | Call Service

Topics
“Get Started with ROS in Simulink”
“Work with ROS Messages in Simulink”
“Connect to a ROS-enabled Robot from Simulink®”
“Composite Interface Guidelines” (Simulink)

Blank Message

Create blank ROS 2 message using specified message type



Libraries:
ROS Toolbox / ROS 2

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 and Call Service blocks.

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 **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 2 message
`Message` (default) | `Service Request` | `Service Response`

Class of ROS 2 message, specified as `Message`, `Service Request`, or `Service Response`. For basic publishing and subscribing, use the `Message` class. To create a service request message for Call Service input, use the `Service Request` class.

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).

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

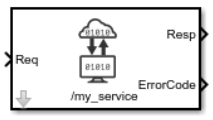
Generate C and C++ code using Simulink® Coder™.

See Also

Publish | Subscribe | Call Service

Call Service

Call service in ROS network



Libraries:
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.
`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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

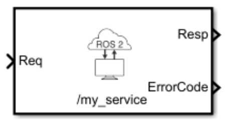
Blocks
Blank Message | Publish | Subscribe

Functions
`rosservice` | `rossvcclient` | `rossvcserver`

Topics
"Call and Provide ROS Services"
"Publish and Subscribe to ROS Messages in Simulink"

Call Service

Call service in ROS 2 network



Libraries:
ROS Toolbox / ROS 2

Description

The Call Service block takes a ROS 2 service request message, sends it to the ROS 2 service server, and waits for a response. A ROS 2 service server should be set up somewhere on the network before using this block. Check the available services on a ROS 2 network by executing `ros2 service list` command, which uses `ros2` function. Use `ros2svcserver` to set up a service server in MATLAB.

Specify the name for your ROS 2 service and the service type in the block mask. If connected to a ROS 2 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.

Always specify the quality of service (QoS) parameters in the block mask. QoS parameters for this block must be compatible with the service server to send requests and receive responses.

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.

Dependencies

This output is enabled when the **Show ErrorCode output port** check box is `on`.

Data Types: `uint8`

Parameters

Main

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 2 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 2 network, see `ros2`.

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.

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.

Quality of Service (QoS)

History — Mode of storing requests in the queue
`Keep last` (default) | `Keep all`

Determines the mode of storing requests in the queue. If the queue fills with requests waiting to be processed, then old requests will be dropped to make room for new. If set to `Keep last`, the queue stores the number of requests set by the `Depth` parameter. If set to `Keep all`, the queue stores all requests up to the MATLAB resource limits.

Depth — Size of the request queue
`1` (default) | positive scalar

Number of requests stored in the request queue when `History` is set to `Keep last`.

Reliability — Delivery guarantee of requests
`Reliable` (default) | `Best effort`

Affects the guarantee of request delivery. If `Reliable`, then delivery is guaranteed, but may retry multiple times. If `Best effort`, then attempt delivery and do not retry. `Reliable` setting is recommended for services.

Durability — Persistence of requests
`Volatile` (default) | `Transient local`

Affects persistence of requests, which allows late-starting servers to receive the number of old requests specified by `Depth`. If `Volatile`, then requests do not persist. If `Transient local`, then the block will persist most recent requests.

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

Blocks

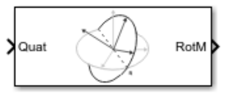
Blank Message | Publish | Subscribe

Functions

`ros2` | `ros2svcclient` | `ros2svcserver`

Coordinate Transformation Conversion

Convert to a specified coordinate transformation representation



Libraries:
Robotics System Toolbox / Utilities
Navigation Toolbox / Utilities
ROS Toolbox / Utilities
UAV 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, returned as a coordinate transformation with the specified representation. 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]`

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, returned 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.

Axis rotation sequence — Order of Euler angle axis rotations
ZYX (default) | ZYZ | XYZ

Order of the Euler angle axis rotations, specified as `ZYX`, `ZYZ`, or `XYZ`. The order of the angles in the input or output port `Eul` must match this rotation sequence. The default order `ZYX` specifies an orientation by:

- Rotating about the initial *z*-axis
- Rotating about the intermediate *y*-axis
- Rotating about the second intermediate *x*-axis

Dependencies

You must select `Euler Angles` for the `Representation` input or output parameter. The axis rotation sequence only applies to Euler angle rotations.

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.

Simulate using — Type of simulation to run
`Interpreted execution` (default) | `Code generation`

- `Interpreted execution` — Simulate model using the MATLAB interpreter. This option shortens startup time but has a slower simulation speed than `Code generation`. In this mode, you can debug the source code of the block.
- `Code generation` — Simulate model using generated C code. The first time you run a simulation, Simulink generates C code for the block. The C code is reused for subsequent simulations, as long as the model does not change. This option requires additional startup time, but the speed of the subsequent simulations is comparable to `Interpreted execution`.

Tunable: No

Version History

Introduced in R2017b

Extended Capabilities

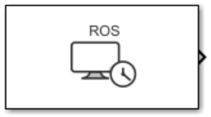
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

`rosmessage`

Current Time

Retrieve current ROS time or system time



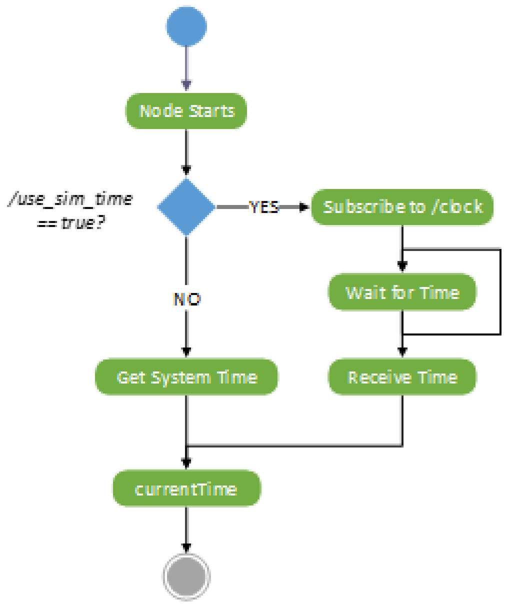
Libraries:
ROS Toolbox / ROS

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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

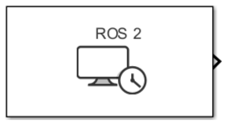
Blocks
Set Parameter | Publish | Get Parameter

Functions
`rostime` | `rosparam` | `get` | `rospublisher` | `set`

External Websites
ROS Time

Current Time

Retrieve current ROS 2 time or system time

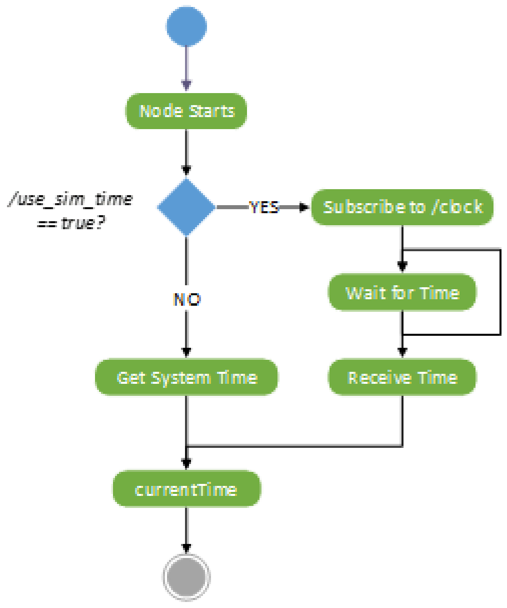


Libraries:
ROS Toolbox / ROS 2

Description

The Current Time block outputs the current ROS 2 time. ROS 2 time is based on the system clock of your computer or the `/clock` topic being published on the ROS 2 network.

If the `use_sim_time` ROS 2 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 2 time
bus | scalar

ROS 2 time, returned as a bus signal or a scalar. The `Output format` parameter determines the format of this output port. The bus represents a `builtin_interfaces/Time` ROS 2 message with `sec` and `nanosec` elements. The scalar is the ROS 2 time in seconds. If the block does not receive a time from the `/clock` topic, this output is `0`.

Data Types: `bus` | `double`

Parameters

Output format — Format of ROS 2 time
`bus` (default) | `double`

Format of ROS 2 `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).

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

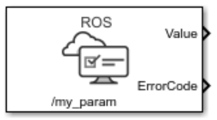
Blocks
Get Parameter | Publish

Functions
`ros2time` | `ros2duration`

Objects
`ros2rate` | `ros2publisher`

Get Parameter

Get values from ROS parameter server



Libraries:
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.

Version History
Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

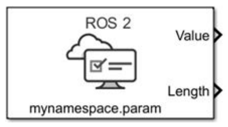
See Also
Set Parameter

Topics
“ROS Parameters in Simulink”

External Websites
ROS Parameter Server
ROS Graph Names

Get Parameter

Get ROS 2 parameter value



Libraries:
ROS Toolbox / ROS 2

Description

The Get Parameter block outputs the value of the specified ROS 2 parameter associated with the node of the Simulink model. Simulink creates this node when you run the model and deletes it when the model terminates. If the model does not have a parameter, the block creates one.

At the start of the simulation, the block initializes the ROS 2 parameter with the specified initial value. For each sample hit, the block checks the ROS 2 parameter and outputs its value.

Ports

Output

Value — Parameter value
scalar | array

Parameter value from the ROS 2 network, returned as a scalar or array. The type and size of this output depends on the **Data type** parameter.

Data Types: `double` | `int64` | `Boolean` | `uint8[]`

Length — Length of array parameter
scalar

Length of the array parameter, returned as an integer scalar. This length is the number of elements of the array or the number of characters in the string that you cast to `uint8`.

Note When getting string parameters from the ROS 2 network, an ASCII value of 13 returns an error due to its incompatible character type.

Dependencies

To enable this port, set the **Data type** parameter to `uint8[]`, `double[]` , `int64[]` or `boolean[]`.

Parameters

Name — Parameter name
`mynamespace.param` (default) | string

Parameter name to get from the ROS 2 network, specified as a string. Valid parameter 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 parameter
`double` (default) | `int32` | `boolean` | `uint8[]` | `double[]` | `int64[]` | `boolean[]`

Data type of parameter, specified as a string. Array data types such as `uint8[]`, `double[]`, `int64[]` and `boolean[]` enable the **Maximum length** parameter. If the data type changes during runtime, the block throws a warning.

Note The `uint8[]` 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.

Maximum length — Maximum length of array
16 (default) | numeric scalar

Maximum length of the array, specified as a scalar. If the array has a length greater than **Maximum length**, the block returns a warning.

Dependencies

To enable this parameter, set the **Data type** parameter to `uint8[]`, `double[]`, `int64[]` or `boolean[]`.

Initial value — Default parameter value output
`0.0` (default) | numeric scalar

Default parameter value output for when an error occurs and no valid value is received from the parameter. The specified value must be a valid value for the specified **Data type**.

Sample time — Interval between outputs
`-1` (default) | numeric scalar

Interval between outputs, specified as a numeric scalar. The default value of `-1` indicates that the block inherits the output sample time from the model. Otherwise, the block outputs a new blank message at each interval of `Sample time`.

For more information, see “Specify Sample Time” (Simulink).

Version History

Introduced in R2022b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

`ros2param`

Header Assignment

Update fields of ROS message header

Libraries:
ROS Toolbox / ROS

Description

The Header Assignment block updates the values in the header field of a ROS message. When a ROS message contains a header field of type `std_msgs/Header`, you can use this block to update the `frame_id` and `stamp` values in its header field. During each sample hit, the block updates the `frame_id` and `stamp` fields in the header. The accuracy of the timestamp depends on the step size of the solver. Smaller step sizes result in more accurate timestamps.

Ports

Input

InputMsg — ROS message to update
nonvirtual bus

ROS message with a `std_msgs/Header` field, specified as a nonvirtual bus.

Data Types: `bus`

Output

OutputMsg — ROS message with an updated header
nonvirtual bus

ROS message with an updated header, returned as a nonvirtual bus.

Data Types: `bus`

Parameters

Set Frame ID — Specify frame associated with message data
`off` (default) | `on`

Select this parameter to specify the frame that the message data is associated with. Specify the frame ID in the text box enabled when you select this parameter. The block populates the `frame_id` field of the ROS message header with the specified frame.

Example: `base_link`

Set Timestamp — Set timestamp in header
`off` (default) | `on`

Select this parameter to set the `stamp` value of the header to the current ROS system time. In order to use the custom time published on the `/clock` topic instead of the ROS system time, set the `use_sim_time` ROS parameter to `true`.

Header field name — Specify Header field name
`Use the default Header field name` (default) | `Specify your own`

Specify the name of the Header field as one of the following:

- `Use the default Header field name` — The block sets the name of the header field in the ROS message to the default value, `Header`.
- `Specify your own` — Enables a text box in which you can specify a custom name for the header field in the ROS message.

Example: `my_custom_header`

Version History

Introduced in R2021a

Extended Capabilities

C/C++ Code Generation

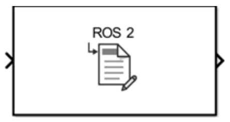
Generate C and C++ code using Simulink® Coder™.

See Also

Blank Message | Publish | Subscribe

Header Assignment

Update fields of ROS 2 message header



Libraries:
ROS Toolbox / ROS 2

Description

The Header Assignment block updates the values in the header field of a ROS 2 message. When a ROS 2 message contains a header field of type `std_msgs/header`, you can use this block to update the `frame_id` and `stamp` values in its header field. During each sample hit, the block updates the `frame_id` and `stamp` fields in the header. The accuracy of the timestamp depends on the step size of the solver. Smaller step sizes result in more accurate timestamps.

Ports

Input

InputMsg — ROS 2 message to update
nonvirtual bus

ROS 2 message with a `std_msgs/header` field, specified as a nonvirtual bus.

Data Types: `bus`

Output

OutputMsg — ROS 2 message with an updated header
nonvirtual bus

ROS 2 message with an updated header, returned as a nonvirtual bus.

Data Types: `bus`

Parameters

Set Frame ID — Specify frame associated with message data
`off` (default) | `on`

Select this parameter to specify the frame that the message data is associated with. Specify the frame ID in the text box enabled when you select this parameter. The block populates the `frame_id` field of the ROS 2 message header with the specified frame.

Example: `base_link`

Set Timestamp — Set timestamp in header
`off` (default) | `on`

Select this parameter to set the `stamp` value of the header to the current ROS 2 system time.

Header field name — Specify Header field name
`Use the default Header field name` (default) | `Specify your own`

Specify the name of the Header field as one of the following:

- `Use the default Header field name` — The block sets the name of the header field in the ROS 2 message to the default value, `header`.
- `Specify your own` — Enables a text box in which you can specify a custom name for the header field in the ROS 2 message.

Example: `my_custom_header`

Version History
Introduced in R2023a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Blank Message | Publish | Subscribe

Publish

Send messages to ROS network

ROS
/my_topic

Libraries:
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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Blank Message | Subscribe

Topics
"Composite Interface Guidelines" (Simulink)
"ROS Simulink Interaction"

Publish

Send messages to ROS 2 network

Libraries:
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.

Version History
Introduced in R2019b

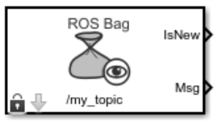
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Blank Message | Subscribe

Read Data

Play back data from log file



Libraries:
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).

Version History

Introduced in R2019b

See Also

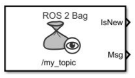
Blocks
Publish | Subscribe | Read Image | Read Point Cloud

Functions
`rosbag` | `readMessages` | `select`

Topics
“Work with ROS Messages in Simulink”
“Work with rosbag Logfiles”

Read Data

Play back data from ROS 2 log file



Libraries:
ROS Toolbox / ROS 2

Description

The Read Data block plays back ROS 2 bag log files by outputting the most recent message from the log file, based on the current simulation time. You must load a ROS 2 bag log file (`.db3`) and specify a topic, using the **Topic** parameter, to get a stream of messages from the file. The block outputs messages of this topic from the file in sync with the simulation time.

In the Read Data block mask, select **Load logfile data** to specify a ROS 2 bag log file (`.db3`) to load. In the Load Logfile dialog box, specify the full path to the log file, or select Browse and navigate to the logfile you want to load. To start playback at a certain point in the file, specify a **Start time offset**, in seconds. To specify how long the block plays back this file, from the specified start time, specify a **Duration**, in seconds. By default, the block outputs all messages for the specified topic in the file.

Ports

Output

IsNew — New message indicator
logical scalar

New message indicator, returned as a logical scalar. If the output is `1`, then the block loaded a new message from the ROS 2 bag log file at the corresponding time. This output can be used to trigger subsystems for processing new messages.

Data Types: `Boolean`

Msg — ROS 2 message
nonvirtual bus

ROS 2 message, returned as a nonvirtual bus. Messages are returned in the order they are stored in the ROS 2 bag log file and synced with the simulation time.

Data Types: `bus`

Parameters

Topic — Topic name to extract from log file
string scalar | character vector

Specify the name of the topic to extract from the log file. This topic must exist in the loaded ROS 2 bag log file. Click **Select** to inspect the available topics and select a specific topic.

Sample time — Interval between outputs
-1 (default) | scalar

Specify the interval between outputs. In simulation, the sample time follows simulation time rather than wall-clock time.

The default value indicates that the block sample time is inherited. For more information about the inherited sample time type, see “Specify Sample Time” (Simulink).

Version History

Introduced in R2021b

See Also

Blocks
Publish | Subscribe | Read Image | Read Point Cloud

Functions
`ros2bagreader` | `readMessages` | `select`

Read Image

Extract image from ROS Image message

Msg
Image
ErrorCode

Libraries:
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 for all message types in the model, from the **Prepare** section under **Simulation** tab, select **ROS Toolbox > Variable Size Messages**. Uncheck **Use default limits for this message type** and then in the **Maximum length** column, increase the length based on the number of pixels 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. Each element has a value in the range `[0,1]` that indicates the opacity of the 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 **Image Encoding** 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 `rosReadImage`.

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 **Image Encoding** 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).

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- Requires a valid version of OpenCV and `cv_bridge` package to be installed for deployment.

See Also

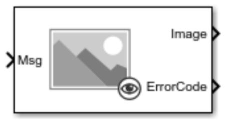
`rosReadImage` | Subscribe | Blank Message | `Image` | `CompressedImage`

Topics

“Manage Array Sizes for ROS Messages in Simulink”
“Variable-Size Signal Basics” (Simulink)

Read Image

Extract image from ROS 2 `Image` message



Libraries:
ROS Toolbox / ROS 2

Description

The Read Image block extracts an image from a ROS 2 `Image` or `CompressedImage` message. You can select the message parameters of a topic active on a live ROS 2 network, or specify the message parameters separately. The ROS 2 messages are specified as a nonvirtual bus. Use the Subscribe block output to receive a message from a ROS 2 network and input the message to the Read Image block.

Note When reading ROS 2 image messages from the network, the `Data` property of the message can exceed the maximum array length set in Simulink. Follow the steps below to increase the maximum array length for all message types in the model:

1. Enable ROS options by selecting the **Robot Operating System (ROS)** app under the **Apps** tab and configure the ROS network parameters appropriately.
2. From the **Prepare** section under **Simulation** tab, select **ROS Toolbox > Variable Size Messages**.
3. Uncheck **Use default limits for this message type** and then in the **Maximum length** column, increase the length based on the number of pixels in the image.

Ports

Input

Msg — ROS 2 `Image` or `CompressedImage` message
nonvirtual bus

ROS 2 `Image` or `CompressedImage` message, specified as a nonvirtual bus. You can use the Subscribe block to get a message from an active ROS 2 network.

Data Types: `bus`

Output

Image — Extracted image signal
M-by-*N*-by-3 matrix | *M*-by-*N* matrix

Extracted image signal from a ROS 2 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 2 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. Each element has a value in the range `[0,1]` that indicates the opacity of the 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.

Dependencies

Enable `Show Alpha output port` parameter.

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 **Image Encoding** 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.

Select **Configure using ROS 2...** to set this parameter automatically using an active topic on a ROS 2 network. You must be connected to the ROS 2 network.

Image encoding — Image encoding
`rgb8` (default) | `rgba8` | ...

Image encoding for the input **ImageMsg**. Select the encoding type that matches the `Encoding` property of the message. For more information about encoding types, see `rosReadImage`.

Show Alpha output port — AlphaChannel port toggle
`off` (default) | `on`

Toggle alpha channel output port on or off, if the selected encoding type supports alpha channels.

Dependencies

To enable this parameter, set the **Image Encoding** parameter to an encoding type that supports alpha channels.

Show ErrorCode output port — Toggle ErrorCode port
on (default) | `off`

Toggle the **ErrorCode** port on or off, to monitor errors.

Output variable-size signals — Variable-size signal output toggle
`off` (default) | on

Toggle whether to output a variable-size signal. Use variable-sized signals only if you expect the image size to change over time. For more information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

Usage notes and limitations:

- For remote deployment, OpenCV and `cv_bridge` ROS package must be installed on the remote device.
- Local host deployment is not supported.

See Also

`rosReadImage` | Subscribe | Blank Message

Topics

“Manage Array Sizes for ROS Messages in Simulink”
“Variable-Size Signal Basics” (Simulink)

Read Point Cloud

Extract point cloud from ROS PointCloud2 message

Msg
XYZ
ErrorCode

Libraries:
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 for all message types in the model, from the **Prepare** section under **Simulation** tab, select **ROS Toolbox > Variable Size Messages**. Uncheck **Use default limits for this message type** and then in the **Maximum length** column, increase the length 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 | array

x-, *y*-, and *z*- coordinates of each point in the point cloud data, returned as either an *N*-by-3 matrix or *h*-by-*w*-by-3 array. *N* is the number of points in the point cloud. *h* and *w* are the height and width of the image, in pixels. To get the *x*-, *y*-, and *z*- coordinates as an array, select the **Preserve point cloud structure** parameter.

Data Types: `single`

RGB — RGB values for each point
matrix | 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 array. *N* is the number of points in the point cloud. *h* and *w* are the height and width of the image in pixels. The RGB values specify the red, green, and blue color intensities in the range of `[0,1]`.To return the RGB values as an array, select the **Preserve point cloud structure** parameter.

Data Types: `double`

Intensity — Intensity values for each point
array | matrix

Intensity value for each point of the point cloud data, returned as either an array or a *h*-by-*w* matrix. *h* and *w* are the height and width of the image in pixels. To return the intensity values as a matrix, select the **Preserve point cloud structure** parameter.

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 the **Maximum point cloud size** parameter.
- `2` – One of the variable-length arrays in the incoming message was truncated. For more information on increasing the maximum length of the array, see “Manage Array Sizes for ROS Messages in Simulink”.
- `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. This error only occurs if you enable the **Show RGB output port** parameter.
- `5` – The point cloud does not contain any intensity data. This error only occurs if you enable the **Show Intensity output port** parameter.

For certain error codes, the block truncates the data or populates 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.

Select **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 — Point cloud data output shape preservation
`off` (default) | `on`

When this parameter is selected, the block preserves the point cloud data output shape for **XYZ**, **RGB**, and **Intensity** outputs. Each output corresponds to the resolution of the original image. The **XYZ** and **RGB** outputs become multidimensional arrays, and the **Intensity** output becomes a matrix.

Show RGB output port — RGB port toggle
`off` (default) | `on`

Select this parameter to enable the **RGB** port. If you enable this parameter, the message must contain RGB data or the block returns an error code.

Show Intensity output port — `Intensity` port toggle
`off` (default) | `on`

Select this parameter to enable the **Intensity** port. If you enable this parameter, the message must contain intensity data or the block returns an error code.

Show ErrorCode output port — `ErrorCode` port toggle
`on` (default) | `off`

Select this parameter to enable the **ErrorCode** port and monitor errors.

Output variable-size signals — Variable-size signal output toggle
`off` (default) | `on`

Toggle whether to output a variable-size signal. Use variable-sized signals only if you expect the image size to change over time. For more information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Subscribe | Blank Message | `PointCloud2`

Topics
“Manage Array Sizes for ROS Messages in Simulink”
“Variable-Size Signal Basics” (Simulink)

Read Point Cloud

Extract point cloud from ROS 2 `PointCloud2` message

Msg
XYZ
ErrorCode

Libraries:
ROS Toolbox / ROS 2

Description

The Read Point Cloud block extracts a point cloud from a ROS 2 `PointCloud2` message. You can select the message parameters of a topic active on a live ROS 2 network, or specify the message parameters separately. The ROS 2 messages are specified as a nonvirtual bus. Use the Subscribe block to receive a message from a ROS 2 network and input the message to the Read Point Cloud block.

Note When reading ROS 2 point cloud messages from the network, the `Data` property of the message can exceed the maximum array length set in Simulink. Follow the steps below to increase the maximum array length for all message types in the model:

1. Enable ROS options by selecting the **Robot Operating System (ROS)** app under the **Apps** tab and configure the ROS network parameters appropriately.
2. From the **Prepare** section under **Simulation** tab, select **ROS Toolbox > Variable Size Messages**.
3. Uncheck **Use default limits for this message type** and then in the **Maximum length** column, increase the length based on the number of points in the point cloud.

Ports

Input

Msg — ROS 2 `PointCloud2` message
nonvirtual bus

ROS 2 `PointCloud2` message, specified as a nonvirtual bus. You can use the Subscribe block to get a message from the ROS 2 network.

Data Types: `bus`

Output

XYZ — XYZ coordinates
matrix | array

x-, *y*-, and *z*- coordinates of each point in the point cloud data, returned as either an *N*-by-3 matrix or *h*-by-*w*-by-3 array. *N* is the number of points in the point cloud. *h* and *w* are the height and width of the image, in pixels. To get the *x*-, *y*-, and *z*- coordinates as an array, select the **Preserve point cloud structure** parameter.

Data Types: `single`

RGB — RGB values for each point
matrix | 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 array. *N* is the number of points in the point cloud. *h* and *w* are the height and width of the image in pixels. The RGB values specify the red, green, and blue color intensities in the range of `[0,1]`.To return the RGB values as an array, select the **Preserve point cloud structure** parameter.

Dependencies

Enable `Show RGB output port` parameter.

Data Types: `double`

Intensity — Intensity values for each point
array | matrix

Intensity value for each point of the point cloud data, returned as either an array or a *h*-by-*w* matrix. *h* and *w* are the height and width of the image in pixels. To return the intensity values as a matrix, select the **Preserve point cloud structure** parameter.

Dependencies

Enable `Show Intensity output port` parameter.

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 the **Maximum point cloud size** parameter.
- `2` – One of the variable-length arrays in the incoming message was truncated. For more information on increasing the maximum length of the array, see “Manage Array Sizes for ROS Messages in Simulink”.
- `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. This error only occurs if you enable the **Show RGB output port** parameter.
- `5` – The point cloud does not contain any intensity data. This error only occurs if you enable the **Show Intensity output port** parameter.

For certain error codes, the block truncates the data or populates 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.

Select **Configure using ROS 2...** to set this parameter automatically using an active topic on a ROS 2 network. You must be connected to the ROS 2 network.

Preserve point cloud structure — Point cloud data output shape preservation
`off` (default) | `on`

When this parameter is selected, the block preserves the point cloud data output shape for **XYZ**, **RGB**, and **Intensity** outputs. Each output corresponds to the resolution of the original image. The **XYZ** and **RGB** outputs become multidimensional arrays, and the **Intensity** output becomes a matrix.

Show RGB output port — RGB port toggle
`off` (default) | `on`

Select this parameter to enable the **RGB** port. If you enable this parameter, the message must contain RGB data or the block returns an error code.

Show Intensity output port — `Intensity` port toggle
`off` (default) | `on`

Select this parameter to enable the **Intensity** port. If you enable this parameter, the message must contain intensity data or the block returns an error code.

Show ErrorCode output port — `ErrorCode` port toggle
`on` (default) | `off`

Select this parameter to enable the **ErrorCode** port and monitor errors.

Output variable-size signals — Variable-size signal output toggle
`off` (default) | `on`

Toggle whether to output a variable-size signal. Use variable-sized signals only if you expect the image size to change over time. For more information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Version History

Introduced in R2021b

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

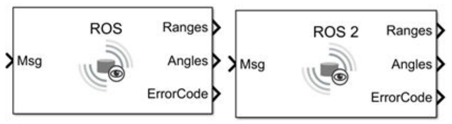
Subscribe | Blank Message

Topics

“Manage Array Sizes for ROS Messages in Simulink”
“Variable-Size Signal Basics” (Simulink)

ROS Read Scan, ROS 2 Read Scan

Extract scan data from ROS or ROS 2 laser scan message



Libraries:
ROS Toolbox / ROS
ROS Toolbox / ROS 2

Description

The Read Scan block extracts range, scan and intensity data from a ROS or ROS 2 laser scan message. You can select the message parameters of a topic active on a live ROS or ROS 2 network, or specify the message parameters separately. The input messages are specified as a nonvirtual bus. Use the ROS Subscribe or the ROS 2 Subscribe block to receive a message from the network and input the message to the Read Scan block. For ROS and ROS 2 models, you must use the blocks in the respective ROS and ROS 2 library.

Ports

Input

Msg — ROS or ROS 2 laser scan message
nonvirtual bus

ROS 2 laser scan message, specified as a nonvirtual bus. You can use the ROS Subscribe or the ROS 2 Subscribe blocks to get a message from the network.

Data Types: `bus`

Output

Ranges — Range values for each point
array

Range values for each point in the scan, returned as an array of length M.

Data Types: `single`

Angles — Angle values for each point
array

Angle values for each point in the scan, returned as an array of length M.

Dependencies

Enable `Show Angles output port` parameter.

Data Types: `single`

Intensities — Intensity value for each point
array

Intensity values for each point in the scan, returned as an array of length *M*.

Dependencies

Enable `Show Intensities output port` parameter.

Data Types: `single`

ErrorCode — Error code for message parsing
scalar

Error code for message parsing, returned as a scalar. The error code values are:

- `0` — Successfully parsed the laser scan message.
- `1` — The `Data` field of the laser scan message was truncated.

Dependencies

Enable `Show ErrorCode output port` parameter.

Data Types: `uint8`

Parameters

Maximum array length — Maximum length of the laser scan array
`100` (default) | positive scalar

Maximum length of the laser scan array, specified as a positive scalar.

Select **Configure using ROS...** to set this parameter automatically using an active topic on a ROS or ROS 2 network. You must be connected to the network.

Show Angles output port — RGB port toggle
`off` (default) | `on`

Select this parameter to enable the **Angles** port. If you enable this parameter, the message must contain RGB data or the block returns an error code.

Show Intensity output port — `Intensity` port toggle
`off` (default) | `on`

Select this parameter to enable the **Intensity** port. If you enable this parameter, the message must contain intensity data or the block returns an error code.

Show ErrorCode output port — `ErrorCode` port toggle
`on` (default) | `off`

Select this parameter to enable the **ErrorCode** port and monitor errors.

Output variable-size signals — Variable-size signal output toggle
`off` (default) | `on`

Toggle whether to output a variable-size signal. Use variable-sized signals only if you expect the image size to change over time. For more information about variable-size signals, see “Variable-Size Signal Basics” (Simulink).

Version History

Introduced in R2022a

Extended Capabilities

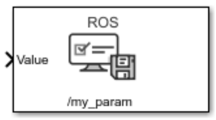
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

`rosReadScanAngles` | `rosReadCartesian` | `rosReadLidarScan` | `rosPlot`

Set Parameter

Set values on ROS parameter server



Libraries:
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`

Version History
Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
Get Parameter

Topics
“ROS Parameters in Simulink”

External Websites
ROS Parameter Servers
ROS Graph Names

Subscribe

Receive messages from ROS network

IsNew
Msg
ROS
/my_topic

Libraries:
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.

Version History

Introduced in R2019b

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

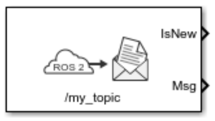
See Also

Blank Message | Publish

Topics
“Composite Interface Guidelines” (Simulink)
“ROS Simulink Interaction”

Subscribe

Receive messages from ROS 2 network



Libraries:
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.

Version History

Introduced in R2019b

Extended Capabilities

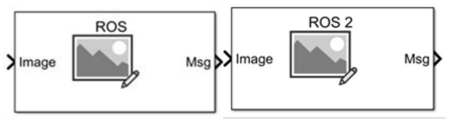
C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

Publish | Blank Message

ROS Write Image, ROS 2 Write Image

Write image data to a ROS or ROS 2 message



Libraries:
ROS Toolbox / ROS
ROS Toolbox / ROS 2

Description

The Write Image block writes image data to a ROS or ROS 2 image message. You can specify the encoding for the output image message. Use the ROS Publish or ROS 2 Publish block to publish the output image message to an active topic on the network.

Ports

Input

Image — Input image signal
M-by-*N*-by-3 matrix | *M*-by-*N* matrix

Image pixel data, specified as an *M*-by-*N*-by-3 matrix for color images, and an *M*-by-*N* matrix for grayscale images.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16`

AlphaChannel — Alpha channel for input image
M-by-*N* matrix

Alpha channel for image, specified as an *M*-by-*N* matrix. This matrix is the same height and width as the image input. Each element has a value in the range `[0,1]` that indicates the opacity of the corresponding pixel, with a value of `0` being completely transparent.

Dependencies

The selected image encoding must support alpha channel.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `uint8` | `uint16`

Output

Msg — ROS or ROS 2 image message
nonvirtual bus

ROS or ROS 2 image message, returned as a nonvirtual bus. You can use the ROS Publish or ROS 2 Publish block to publish the message to an active ROS or ROS 2 network respectively.

Data Types: `bus`

Parameters

Image Encoding — Image encoding
`rgb8` (default) | `rgba8` | ...

Image encoding for the input **Image**, specified as one of the supported options. For more information about encoding types, see `rosReadImage`.

Version History
Introduced in R2022a

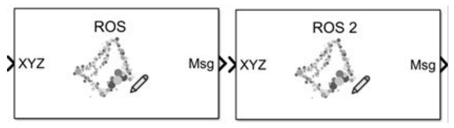
Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also
`rosWriteImage` | `rosReadImage`

ROS Write Point Cloud, ROS 2 Write Point Cloud

Write point cloud data to a ROS or ROS 2 message



Libraries:
ROS Toolbox / ROS
ROS Toolbox / ROS 2

Description

The Write Point Cloud block writes point cloud data to a ROS or ROS 2 point cloud message. You can specify the appropriate color encoding for the point cloud image and write the corresponding color and alpha values to the output message. You can also write intensity values to the output message. Use the ROS Publish or ROS 2 Publish block to publish the output image message to an active topic on the network.

Ports

Input

XYZ — XYZ coordinates
matrix | array

x-, *y*-, and *z*- coordinates of each point in the point cloud data, specified as either an *N*-by-3 matrix or *h*-by-*w*-by-3 array. *N* is the number of points in the point cloud. *h* and *w* are the height and width of the image, in pixels.

Data Types: `single`

RGB — RGB values for each point
matrix | array

RGB values for each point of the point cloud data, specified as either an *N*-by-3 matrix or *h*-by-*w*-by-3 array. *N* is the number of points in the point cloud. *h* and *w* are the height and width of the image in pixels. The RGB values specify the red, green, and blue color intensities in the range of `[0,1]`.

Dependencies

The selected color field must support rgb values.

Data Types: `double`

Alpha — Alpha channel for input image
N-by-1 vector | *h*-by-*w* matrix

Alpha channel for image, specified as an *N*-by-1 vector or *h*-by-*w* matrix. Each element has a value in the range `[0,1]` that indicates the opacity of the corresponding pixel, with a value of `0` being completely transparent.

Dependencies

The selected color field must support alpha channel.

Data Types: `double`

Intensity — Intensity values for each point
matrix | array

Intensity values for each point in the point cloud data, specified as either an *N*-by-3 matrix or *h*-by-*w*-by-3 array. *N* is the number of points in the point cloud. *h* and *w* are the height and width of the image, in pixels.

Dependencies

You must select the **Write to Intensity Field** parameter.

Data Types: `single` | `double`

Output

Msg — ROS or ROS 2 point cloud message
nonvirtual bus

ROS or ROS 2 point cloud message, retuned as a nonvirtual bus. You can use the ROS Publish or ROS 2 Publish block to publish the message to an active ROS or ROS 2 network respectively.

Data Types: `bus`

Parameters

Select color field — Color field for the point cloud image
`none` (default) | `rgb` | `rgba`

Color field of the point cloud image, specified as one of the supported options. Choose the appropriate option to specify color and alpha values corresponding to the point cloud image.

Write to Intensity field — Write intensity values
off (default) | on

Select whether to write intensity values to the output message.

Version History

Introduced in R2022a

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

See Also

`rosReadXYZ` | `rosReadRGB` | `rosPlot` | `rosReadField` | `rosReadAllFieldNames`