
QML ROS Plugin

Stefan Fabian

Oct 19, 2021

TABLE OF CONTENTS

1	ActionClient	1
1.1	API	2
2	Arrays	5
2.1	API	5
3	Examples	7
3.1	Subscriber	7
3.2	Publisher	7
3.3	URDF Tutorial UI	7
3.4	Logging	8
4	ImageTransport	9
4.1	API	10
5	Logging	11
5.1	Output	11
5.2	Set Verbosity	11
6	Publishers	13
6.1	Simple Example	13
6.2	API	14
6.2.1	Publisher	14
7	Quickstart	15
7.1	Installation	15
7.1.1	From Source	15
7.2	Usage	15
7.2.1	Initialization	16
7.2.2	Shutdown	16
8	Ros Singleton	17
8.1	ROS Initialization	17
8.2	Query Topics	18
8.3	Create Empty Message	18
8.4	Package API	19
8.5	Console	19
8.6	IO	19
8.7	API	19
9	Services	25

9.1	API	26
10	Subscribers	27
10.1	Simple example	27
10.2	Full example	27
10.3	API	29
11	Tf Transforms	31
11.1	Component	31
11.2	Static	31
11.3	API	32
12	Time	35
12.1	API	35
Index		39

ACTIONCLIENT

An action client can be used to send goals to an ActionServer. One ActionClient has to have a single type of action but can send multiple goals simultaneously. Process can be tracked using either a goal handle manually or using callbacks.

An ActionClient can be created using the *Ros Singleton* as follows:

```
Item {  
    // ...  
    property var fibonacciClient: Ros.createActionClient("actionlib_tutorials/  
↳FibonacciAction", "fibonacci")  
    // ...  
}
```

In this example an action client is created using the `actionlib_tutorials/FibonacciAction` action (it is important to use the complete action type here, not any part like the `ActionGoal`) using the name `fibonacci` on which an ActionServer should be registered (You can use the `actionlib_tutorials` `fibonacci_server`).

To send a goal, you can use the `sendGoal` function:

```
goal_handle = fibonacciClient.sendGoal({ order: numberInput.value }, function (goal_  
↳handle) {  
    // Do something when the goal transitions, e.g., update the status  
    if (goal_handle.commState == ActionCommStates.DONE) {  
        // Goal is terminated, we can access the terminalState  
        switch (handle.terminalState.state) {  
            case ActionTerminalStates.RECALLED:  
                // Handle recalled  
                break  
            case ActionTerminalStates.REJECTED:  
                // Handle rejected  
                break  
            case ActionTerminalStates.PREEMPTED:  
                // Handle preempted  
                break  
            case ActionTerminalStates.ABORTED:  
                // Handle aborted  
                break  
            case ActionTerminalStates.SUCCEEDED:  
                // Handle succeeded  
                break  
            case ActionTerminalStates.LOST:  
                // Handle lost  
                break
```

(continues on next page)

(continued from previous page)

```

    }
  }
}, function (goal_handle, feedback) {
  // Feedback callback. Called whenever feedback is received from the action server
})

```

The `sendGoal` function takes 3 parameters: the goal, a transition callback and a feedback callback. Both callbacks are optional. It returns a `GoalHandle` which can be used query to state of the goal or to cancel the goal. The `goal_handle` passed to the callbacks and the one returned are the same.

1.1 API

```
class qml_ros_plugin::ActionClient : public QObjectRos
```

Public Functions

ActionClient(NodeHandle::Ptr nh, const QString &action_type, const QString &name)

True if the *ActionClient* is connected to the ActionServer, false otherwise.

The type of the action. Example: `actionlib_tutorials/FibonacciAction`

QObject ***sendGoal**(const QVariantMap &goal, QJSValue transition_cb = QJSValue(), QJSValue feedback_cb = QJSValue())

Sends a goal to the action server if it is connected.

Parameters

- **goal** – The goal that is sent to the action server.
- **transition_cb** – A callback that is called on every client state transition.
- **feedback_cb** – A callback that is called whenever feedback for this goal is received.

Returns null if the action server is not connected, otherwise a *GoalHandle* keeping track of the state of the goal.

void **cancelAllGoals**()

Cancels all goals that are currently tracked by this client.

void **cancelGoalsAtAndBeforeTime**(const QDateTime &time)

Cancels all goals that were sent at and before the given ROS time by this client. Use `Time.now()` to obtain the current ROS time which can differ from the actual time.

Signals

void **connectedChanged**()

Emitted when the connected status changes, e.g., when the client connected to the server.

```
class qml_ros_plugin::GoalHandle : public QObjectRos
```

Public Functions

explicit **GoalHandle**(std::shared_ptr<actionlib::ActionClient<ros_babel_fish::BabelFishAction>> client,
 const actionlib::ActionClient<ros_babel_fish::BabelFishAction>::GoalHandle &handle)
 True if this handle is not tracking a goal, false otherwise.

The terminal state of this goal. Only valid if commstate == ActionCommStates.DONE. The state of this goal's communication state machine. Possible states: WAITING_FOR_GOAL_ACK, PENDING, ACTIVE, WAITING_FOR_RESULT, WAITING_FOR_CANCEL_ACK, RECALLING, PREEMPTING, DONE

void **cancel**()
 Sends a cancellation request to the ActionServer. Transitions to WAITING_FOR_CANCEL_ACK.

void **resend**()
 Resends this goal with the same id to the ActionServer. Useful if you have reason to think that the goal was lost in transit.

QVariant **getResult**()
 Can be used to obtain the result returned by the ActionServer. Empty if no result received.

class qml_ros_plugin::TerminalState

Public Functions

inline **TerminalState**()
 An optional text that was returned by the ActionServer in combination with the terminal state.

The terminal state in form of an ActionTerminalStates enum value: RECALLED, REJECTED, PRE-EMPTED, ABORTED, SUCCEEDED, LOST

ARRAYS

Due to the lazy copy mechanism, arrays differ from the standard access above. Because the array is not copied into a QML compatible array container, access happens with methods.

Example: Instead of `path.to.array[1].someproperty`, you would write `path.to.array.at(1).someproperty`.

If you need the array as a javascript array, you can use `toArray` to copy the entire array and return it as a javascript array. The copy is only performed on the first call, subsequent calls should have less overhead.

2.1 API

class `qml_ros_plugin::Array`

View on an array field of a message. This allows access on array elements with lazy copy mechanism. Copies of an `Array` point to the same data and modifications of one array will be mirrored by the other.

Public Functions

`Array()`

The length of the array, i.e., the number of elements.

QVariant `at(int index)` const

If the index is out of the bounds of the array, an empty QVariant is returned.

Parameters `index` – Index of the retrieved element.

Returns The array element at the given index.

void `spliceList(int start, int delete_count, const QList &items)`

Changes the array content by removing `delete_count` elements at `index` and inserting the elements in `items`. This method can be used to remove, replace or add elements to the array.

<p>Warning: If the operation is not limited to the end of the array, it requires a deep copy of the message array.</p>

Parameters

- **start** – The index at which to start changing the array. If greater than the length of the array, `start` will be set to the length of the array. If negative, it will begin that many elements from the end of the array (with origin -1, meaning -n is the index of the nth last element)

and is therefore equivalent to the index of `array.length - n`). If the absolute value of `start` is greater than the length of the array, it will begin from index 0.

- **delete_count** – The number of elements to delete starting at index `start`. If `delete_count` is greater than the number of elements after `start`, all elements from `start` to the length of the array are removed. If `delete_count` is 0, no elements are removed, e.g., for a insert only operation.
- **items** – The items that will be inserted at `start`.

void **push**(const QVariant &value)

Adds the given value to the end of the array.

Parameters value – The item that is added.

void **unshift**(const QVariant &value)

Adds the given value to the front of the array.

Warning: This requires a deep copy of the message array on first call whereas appending can be done without copying the array.

Parameters value – The item that is added.

QVariant **pop**()

Removes the last element and returns it.

Returns The removed element or an empty QVariant if the array is empty.

QVariant **shift**()

Removes the first element and returns it.

Warning: This requires a deep copy of the message array on first call whereas appending can be done without copying the array.

Returns The removed element or an empty QVariant if the array is empty.

QVariantList **toArray**()

Converts the array to a QVariantList which can be used in place of a JS array in QML. This method performs a deep copy of the message array on the first call.

Returns The array as a QVariantList.

EXAMPLES

You can find the described example QML files in the [qml_ros_plugin repo examples directory](#).

3.1 Subscriber

The subscriber example demonstrates how to create a Subscriber in QML using the *QML ROS Plugin*.

You can run the example using the `qmlscene` executable:

```
qmlscene subscriber.qml
```

3.2 Publisher

The publisher example publishes an `std_msgs/Int32` message on the topic that the subscriber example subscribes to. Coincidentally, the two examples can very well be used together.

To run, run:

```
qmlscene publisher.qml
```

3.3 URDF Tutorial UI

This example combines several of the functionalities provided by this library and presents a user interface for the `urdf_sim_tutorial` diff drive example.

First, launch the example:

```
roslaunch urdf_sim_tutorial 13-diffdrive.launch
```

Next, launch the example UI:

```
qmlscene urdf_tutorial_combined.qml
```

It provides a top down view on the position of the robot and sliders to control the forward and angular movement.

3.4 Logging

This example demonstrates the logging functionality detailed in *Logging*. The “Output logging level” sets the minimum logging level that is printed whereas the “Message logging level” sets the level of the message that is logged when you click the *Log* button.

To run, run:

```
qmlscene logging.qml
```

IMAGETRANSPORT

Seeing what the robot sees is one of the most important features of any user interface. To enable this, this library provides the *ImageTransportSubscriber*. It allows easy subscription of camera messages and provides them in a QML native format as a VideoSource.

Example:

```
ImageTransportSubscriber {
  id: imageSubscriber
  // Enter a valid image topic here
  topic: "/front_rgbdcam/color/image_rect_color"
  // This is the default transport, change if compressed is not available
  defaultTransport: "compressed"
}

VideoOutput {
  anchors.fill: parent
  // Can be used in increments of 90 to rotate the video
  orientation: 90
  source: imageSubscriber
}
```

The *ImageTransportSubscriber* can be used as the source of a *VideoOutput* to display the camera images as they are received. Additionally, it can be configured to show a blank image after x milliseconds using the *timeout* property which is set to 3000ms (3s) by default. This can be disabled by setting the *timeout* to 0. If you do not want the full camera rate, you can throttle the rate by setting *throttleRate* to a value greater than 0 (which is the default and disables throttling). E.g. a rate of 0.2 would show a new frame every 5 seconds. Since there is no ROS functionality for a throttled subscription, this means the *image_transport::Subscriber* is shut down and newly subscribed for each frame. This comes at some overhead, hence, it should only be used to throttle to low rates <1. To avoid all throttled subscribers subscribing at the same time causing huge network spikes, the throttled rates are load balanced by default. This can be disabled globally using *ImageTransportManager::setLoadBalancingEnabled* which is available in QML using the singleton *ImageTransportManager*.

4.1 API

```
class qml_ros_plugin::ImageTransportSubscriber : public QObjectRos
```

Public Functions

```
ImageTransportSubscriber(NodeHandle::Ptr nh, QString topic, quint32 queue_size)
```

Interface for QML. This is the surface the images are passed to.

The image base topic (without image_raw etc.). This value may change once the subscriber is connected and private topic names or remappings were evaluated. The default transport passed as transport hint. May be overridden by a parameter. (Default: compressed) Whether or not this *ImageTransportSubscriber* is subscribed to the given topic The latency from the sender to the received time in ms not including the conversion latency before displaying. This latency is based on the ROS time of the sending and receiving machines, hence, they need to be synchronized. The latency (in ms) from the reception of the image until it is in a displayable format. The full latency (in ms) from the camera to your display excluding drawing time. The framerate of the received camera frames in frames per second. The timeout when no image is received until a blank frame is served. Set to 0 to disable and always show last frame. Default is 3000 ms. The update rate to throttle image receiving in images per second. Set to 0 to disable throttling. Default is 0 (disabled).

```
class qml_ros_plugin::ImageTransportManagerSingletonWrapper : public QObject
```

Public Functions

```
void setLoadBalancingEnabled(bool value)
```

Sets whether the manager should try to balance throttled subscriptions_ to ensure they don't update at the same time which would result in network spikes.

Logging is done using the *Ros Singleton*.

5.1 Output

To log a message you can use one of the following methods `debug`, `info`, `warn`, `error` and `fatal`.

```
Button {  
  // ...  
  onClicked: Ros.info("Button clicked.")  
}
```

This will produce the following output:

```
:: code-block:: bash
```

```
[ INFO] [1583062360.048922959]: Button clicked.
```

and publish the following on `/rosout` (unless `NoRosout` was specified in the `RosInitOptions`).

```
:: code-block:: bash
```

```
header: seq: 1 stamp:
```

```
secs: 1583062360 nsecs: 49001300
```

```
frame_id: ''
```

```
level:      2  name:      "/qml_logging_demo"  msg:      "Button clicked."  file:  
"/home/stefan/qml_ros_plugin/examples/logging.qml" function: "onClicked" line: 130 topics: [/rosout]
```

The file, function and line info is automatically extracted when you call the log function.

5.2 Set Verbosity

You can change the verbosity, i.e., the minimal level of logging message that is printed (and published if enabled), using `Ros.console.setLogLevel`. By default the logging level is set to *Info*. To enable debug messages you can set it to *Debug* as follows:

```
Ros.console.setLogLevel(Ros.console.defaultName, RosConsoleLevels.Debug);
```

The first argument to that method is the name of the console to which the logging is printed. These are identifiers used by ros to enable you to change the verbosity of a submodule of your node using `rqt_console`.

You can optionally change to which console you're writing by passing a second argument to the logging function, e.g., `debug("Some message", "ros.my_pkg.my_submodule")`.

This name should contain only letters, numbers, dots and underscores.

Important: The name has to start with "ros.".

By default the value of `Ros.console.defaultName` is used which evaluates to `ros.qml_ros_plugin`.

Possible values for the console level are: Debug, Info, Warn, Error and Fatal.

PUBLISHERS

A Publisher is used to publish messages on a given topic for delivery to subscribers.

6.1 Simple Example

Contrary to Subscribers, a Publisher can not be instantiated but can be created using a factory method of the Ros singleton.

```
1  /* ... */
2  ApplicationWindow {
3    property var intPublisher: Ros.advertise("std_msgs/Int32", "/intval", 10, false)
4    /* ... */
5  }
```

In order, the arguments are the type, the topic, the queueSize and whether or not the topic isLatched.

To publish a message using our Publisher, we can simply use the `intPublisher` variable.

```
1  SpinBox {
2    id: numberInput
3  }
4
5  Button {
6    onClicked: {
7      intPublisher.publish({ data: numberInput.value })
8    }
9  }
```

where we pass an object with a data field containing the (integer) number of the SpinBox. This is according to the `std_msgs/Int32` message definition.

6.2 API

6.2.1 Publisher

```
class qml_ros_plugin::Publisher : public QObject
```

Public Functions

Publisher(NodeHandle::Ptr nh, QString type, QString topic, uint32_t queue_size, bool latch)

The type of the published messages, e.g., geometry_msgs/Pose.

The topic this *Publisher* publishes messages on. This property is only valid if the publisher is already advertised! Whether or not this *Publisher* is latched. A latched *Publisher* always sends the last message to new subscribers. The queue size of this *Publisher*. This is the maximum number of messages that are queued for delivery to subscribers at a time. Whether or not this publisher has advertised its existence on its topic. Reasons for not being advertised include ROS not being initialized yet.

unsigned int **getNumSubscribers**()

Returns The number of subscribers currently connected to this *Publisher*.

bool **publish**(const QVariantMap &msg)

Sends a message to subscribers currently connected to this *Publisher*.

Parameters *msg* – The message that is published.

Returns True if the message was sent successfully, false otherwise.

Signals

void **advertised**()

Fired once this *Publisher* was advertised. This is either done at construction or immediately after ROS is initialized. Since this is only fired once, you should check if the *Publisher* is already advertised using the `isAdvertised` property.

void **connected**()

Fired whenever a new subscriber connects to this *Publisher*.

void **disconnected**()

Fired whenever a subscriber disconnects from this *Publisher*.

QUICKSTART

This library provides convenient access of ROS concepts and functionalities in QML.

7.1 Installation

7.1.1 From Source

To install `qml_ros_plugin` from source, clone the [repo](#). Next, open a terminal and `cd` into the repo folder. To install create a build folder, `cd` into that folder and run `cmake ..` followed by `sudo make install`.

```
mkdir build && cd build
cmake ..
sudo make install
```

7.2 Usage

To use the plugin import Ros in QML.

```
import Ros 1.0
```

Now, you can use the provided components such as `Subscriber` and `TfTransform` and the Ros singleton to create a `Publisher` or the `Service` singleton to call a service.

As a simple example, a `Subscriber` can be created as follows:

```
1 Subscriber {
2   id: mySubscriber
3   topic: "/intval"
4 }
```

For more in-depth examples, check out the [Examples](#) section.

7.2.1 Initialization

Before a `Subscriber` can receive messages, a `Publisher` can publish messages, etc. the node has to be initialized. If your application calls `ros::init` on startup, you don't have to do anything. However, in cases where you can't call `ros::init` from the C++ entry function, you can use the `init` function of the `Ros` singleton:

```
1 ApplicationWindow {
2     /* ... */
3     Component.onCompleted: {
4         Ros.init("node_name");
5     }
6 }
```

7.2.2 Shutdown

To make your application quit when ROS shuts down, e.g., because of a `Ctrl+C` in the console or a `rostopic kill` request, you can connect to the `Shutdown` signal:

```
1 ApplicationWindow {
2     Connections {
3         target: Ros
4         onShutdown: Qt.quit()
5     }
6     /* ... */
7 }
```

For more on that, check out the [Ros Singleton](#).

ROS SINGLETON

The Ros singleton provides interfaces to static methods and convenience methods.

In QML it is available as `Ros`, e.g.:

```
if (Ros.ok()) console.log("Ros is ok!")
```

8.1 ROS Initialization

If you can not or for whatever reason do not want to initialize ROS in your C++ entry, you can also do it in QML, e.g., in the `onCompleted` handler:

```
Component.onCompleted: {  
    Ros.init("node_name")  
}
```

You can also conditionally initialize by checking if it was already initialized using `Ros.isInitialized`. As described in the API documentation for `Ros.init`, you can pass either just the node name or additionally use provided command line args instead of the command line args provided to your executable. In both cases, you can also pass the following `RosInitOptions` options:

- **NoSigintHandler:**
Don't install a SIGINT (e.g., Ctrl+C on terminal) handler.
- **AnonymousName:**
Anonymize the node name. Adds a random number to the node's name to make it unique.
- **NoRosout:**
Don't broadcast roscout output to the `/rosout` topic. See [Logging](#)

```
Component.onCompleted: {  
    Ros.init("node_name", RosInitOptions.AnonymousName | RosInitOptions.NoRosout)  
}
```

8.2 Query Topics

You can also use the Ros singleton to query the available topics. Currently, three methods are provided:

- `QStringList queryTopics(const QString &datatype = QString())`
Queries a list of topics with the given datatype or all topics if no type provided.
- `QList<TopicInfo> queryTopicInfo()`
Retrieves a list of all advertised topics including their datatype. See `TopicInfo`
- `QString queryTopicType(const QString &name)`
Retrieves the datatype for a given topic.

Example:

```
// Retrieves a list of topics with the type sensor_msgs/Image
var topics = Ros.queryTopics("sensor_msgs/Image")
// Another slower and less clean method of this would be
var cameraTopics = []
var topics = Ros.queryTopicInfo()
for (var i = 0; i < topics.length; ++i) {
  if (topics[i].datatype == "sensor_msgs/Image") cameraTopics.push(topics[i].name)
}
// The type of a specific topic can be retrieved as follows
var datatype = Ros.queryTopicType("/topic/that/i/care/about")
// Using this we can make an even worse implementation of the same functionality
var cameraTopics = []
var topics = Ros.queryTopics() // Gets all topics
for (var i = 0; i < topics.length; ++i) {
  if (Ros.queryTopicType(topics[i]) == "sensor_msgs/Image") cameraTopics.push(topics[i])
}
```

8.3 Create Empty Message

You can also create empty messages and service requests as javascript objects using the Ros singleton.

```
var message = Ros.createEmptyMessage("geometry_msgs/Point")
// This creates an empty instance of the message, we can override the fields
message.x = 1; message.y = 2; message.z = 1
// However, note that we do not call custom message constructors, hence, if the message_
↳ has different default values
// they will not be set here. This is a rarely known feature and not used often in ROS 1,
↳ though.

// Same can be done with service requests
var serviceRequest = Ros.createEmptyServiceRequest("std_srvs/SetBool")
// This creates an empty instance of the service request with all members set to their_
↳ default, we can override the fields
serviceRequest.data = true
```

8.4 Package API

The package property provides a wrapper for `ros::package`.

```
// Retrieve a list of all packages
var packages = Ros.package.getAll()
// Get the fully-qualified path to a specific package
var path = Ros.package.getPath("some_pkg")
// Get plugins for a package as a map [package_name -> [values]]
var plugins = Ros.package.getPlugins("rviz", "plugin")
```

8.5 Console

The Ros singleton also provides access to the Ros logging functionality. See [Logging](#).

8.6 IO

You can also save and read data that can be serialized in the yaml format using:

```
var obj = {"key": [1, 2, 3], "other": "value"}
if (!Ros.io.writeYaml("/home/user/file.yaml", obj))
  Ros.error("Could not write file!")
// and read it back
obj = Ros.io.readYaml("/home/user/file.yaml")
if (!obj) Ros.error("Failed to load file!")
```

8.7 API

class `qml_ros_plugin::Package`
A wrapper for `ros::package`

Public Functions

QString **command**(const QString &cmd)

Runs a command of the form 'rospack <cmd>'. (This does not make a subprocess call!)

Parameters `cmd` – The command passed to rospack.

Returns The output of the command as a string.

QString **getPath**(const QString &package_name)

Queries the path to a package.

Parameters `package_name` – The name of the package.

Returns The fully-qualified path to the given package or an empty string if the package is not found.

QStringList **getAll**()

Returns A list of all packages.

QVariantMap **getPlugins**(const QString &name, const QString &attribute, bool force_recrawl = false)
Queries for all plugins exported for a given package.

```
<export>
  <name attribute="value"/>
  <rviz plugin="{prefix}/plugin_description.xml"/>
</export>
```

To query for rviz plugins you would pass 'rviz' as the name and 'plugin' as the attribute.

Parameters

- **name** – The name of the export tag.
- **attribute** – The name of the attribute for the value is obtained.
- **force_recrawl** – Forces rospack to rediscover everything on the system before running the search.

Returns A map with the name of the package exporting something for name as the key (string) and a QStringList containing all exported values as the value.

```
class qml_ros_plugin::TopicInfo
```

Public Functions

TopicInfo() = default

The name of the topic, e.g., /front_camera/image_raw.

The datatype of the topic, e.g., sensor_msgs/Image

```
class qml_ros_plugin::IO
```

Public Functions

bool **writeYaml**(QString path, const QVariant &value)

Writes the given value to the given path in the yaml format.

Parameters

- **path** – The path to the file.
- **value** – The value to write.

Returns True if successful, false otherwise.

QVariant **readYaml**(QString path)

Reads a yaml file and returns the content in a QML compatible structure of 'QVariantMap's and 'QVariantList's.

Parameters **path** – The path to the file.

Returns A variant containing the file content or false.

```
class qml_ros_plugin::RosQmlSingletonWrapper : public QObject
```


Public Functions

bool **isInitialized()** const

Checks whether ROS is initialized.

Returns True if ROS is initialized, false otherwise.

void **init**(const QString &name, quint32 options = 0)

Initializes the ros node with the given name and the command line arguments passed from the command line.

Parameters

- **name** – The name of the ROS node.
- **options** – The options passed to ROS, see `ros_init_options::RosInitOption`.

void **init**(const QStringList &args, const QString &name, quint32 options = 0)

Initializes the ros node with the given args.

Parameters

- **args** – The args that are passed to ROS. Normally, these would be the command line arguments see *init(const QString &, quint32)*
- **name** – The name of the ROS node.
- **options** – The options passed to ROS, see `ros_init_options::RosInitOption`.

bool **ok()** const

Can be used to query the state of ROS.

Returns False if it's time to exit, true if still ok.

void **spinOnce()**

Processes a single round of callbacks. This call is non-blocking as the queue will be called on a detached thread. Not needed unless you disable the AsyncSpinner using *setThreads(int)* with the argument 0.

void **setThreads**(int count)

Sets the thread count for the AsyncSpinner. If asynchronous spinning is disabled, you have to manually call *spinOnce()* to receive and publish messages.

Parameters **count** – How many threads the AsyncSpinner will use. Set to zero to disable spinning. Default: 1

QString **getName()**

Returns the name of the node. Returns empty string before ROS node was initialized.

QString **getNamespace()**

Returns the namespace of the node. Returns empty string before ROS node was initialized.

QStringList **queryTopics**(const QString &datatype = QString()) const

Queries the ROS master for its topics or using the optional datatype parameter for all topics with the given type.

Parameters **datatype** – The message type to filter topics for, e.g., `sensor_msgs/Image`. Omit to query for all topics.

Returns A list of topics that matches the given datatype or all topics if no datatype provided.

QList<TopicInfo> **queryTopicInfo()** const

Queries the ROS master for its topics and their type.

Returns A list of *TopicInfo*.

QString **queryTopicType**(const QString &name) const
Queries the ROS master for a topic with the given name.

Parameters **name** – The name of the topic, e.g., `/front_camera/image_raw`.

Returns The type of the topic if found, otherwise an empty string.

QVariant **createEmptyMessage**(const QString &datatype) const
Creates an empty message for the given message type, e.g., “`geometry_msgs/Point`”. If the message type is known, an empty message with all members set to their default is returned. If the message type is not found on the current machine, a warning message is printed and null is returned.

Parameters **datatype** – The message datatype.

Returns A message with all members set to their default.

QVariant **createEmptyServiceRequest**(const QString &datatype) const
Creates an empty service request for the given service type, e.g., “`std_srvs/SetBool`”. If the service type is known, an empty request is returned with all members of the request message set to their default values. If the service type is not found on the current machine, a warning message is printed and null is returned.

Parameters **datatype** – The service datatype. NOT the request datatype.

Returns A request message with all members set to their default.

QJSValue **debug**()
Outputs a ROS debug message. The equivalent of calling `ROS_DEBUG` in C++. The signature in QML is `debug(msg, consoleName)` where *consoleName* is an optional parameter which defaults to `ros.qml_ros_plugin`.

Note: The msg is not a format string.

Example:

```
Ros.debug("The value of x is: " + x);
```

QJSValue **info**()
Outputs a ROS info message. The equivalent of calling `ROS_INFO` in C++. The signature in QML is `info(msg, consoleName)` where *consoleName* is an optional parameter which defaults to `ros.qml_ros_plugin`.

Note: The argument is not a format string.

Example:

```
Ros.info("The value of x is: " + x);
```

QJSValue **warn**()
Outputs a ROS warn message. The equivalent of calling `ROS_WARN` in C++. The signature in QML is `warn(msg, consoleName)` where *consoleName* is an optional parameter which defaults to `ros.qml_ros_plugin`.

Note: The argument is not a format string.

Example:

```
Ros.warn("The value of x is: " + x);
```

QJSValue **error**()
Outputs a ROS error message. The equivalent of calling `ROS_ERROR` in C++. The signature in QML is `error(msg, consoleName)` where *consoleName* is an optional parameter which defaults to `ros.qml_ros_plugin`.

Note: The argument is not a format string.

Example:

```
Ros.error("The value of x is: " + x);
```

QObject **fatal()**

Outputs a ROS fatal message. The equivalent of calling ROS_FATAL in C++. The signature in QML is `fatal(msg, consoleName)` where `consoleName` is an optional parameter which defaults to `ros.qml_ros_plugin`.

Note: The argument is not a format string.

Example:

```
Ros.fatal("The value of x is: " + x);
```

QObject ***advertise**(const QString &type, const QString &topic, quint32 queue_size, bool latch = false)
Creates a *Publisher* to publish ROS messages.

Parameters

- **type** – The type of the messages published using this publisher.
- **topic** – The topic on which the messages are published.
- **queue_size** – The maximum number of outgoing messages to be queued for delivery to subscribers.
- **latch** – Whether or not this publisher should latch, i.e., always send out the last message to new subscribers.

Returns A *Publisher* instance.

QObject ***advertise**(const QString &ns, const QString &type, const QString &topic, quint32 queue_size, bool latch = false)

Creates a *Publisher* to publish ROS messages.

Parameters

- **ns** – The namespace for this publisher.
- **type** – The type of the messages published using this publisher.
- **topic** – The topic on which the messages are published.
- **queue_size** – The maximum number of outgoing messages to be queued for delivery to subscribers.
- **latch** – Whether or not this publisher should latch, i.e., always send out the last message to new subscribers.

Returns A *Publisher* instance.

QObject ***subscribe**(const QString &topic, quint32 queue_size)

Creates a *Subscriber* to subscribe to ROS messages. Convenience function to create a subscriber in a single line.

Parameters

- **topic** – The topic to subscribe to.
- **queue_size** – The maximum number of incoming messages to be queued for processing.

Returns A *Subscriber* instance.

QObject ***subscribe**(const QString &ns, const QString &topic, quint32 queue_size)

Creates a *Subscriber* to subscribe to ROS messages. Convenience function to create a subscriber in a single line.

Parameters

- **ns** – The namespace for this *Subscriber*.
- **topic** – The topic to subscribe to.
- **queue_size** – The maximum number of incoming messages to be queued for processing.

Returns A *Subscriber* instance.

QObject ***createActionClient**(const QString &type, const QString &name)

Creates an *ActionClient* for the given type and the given name.

Parameters

- **type** – The type of the action (which always ends with ‘Action’). Example: `action-lib_tutorials/FibonacciAction`
- **name** – The name of the action server to connect to. This is essentially a base topic.

Returns An instance of *ActionClient*.

QObject ***createActionClient**(const QString &ns, const QString &type, const QString &name)

Creates an *ActionClient* for the given type and the given name.

Parameters

- **ns** – The namespace for this *ActionClient*.
- **type** – The type of the action (which always ends with ‘Action’). Example: `action-lib_tutorials/FibonacciAction`
- **name** – The name of the action server to connect to. This is essentially a base topic.

Returns An instance of *ActionClient*.

Signals

void **initialized**()

Emitted once when ROS was initialized.

void **shutdown**()

Emitted when this ROS node was shut down and it is time to exit.

SERVICES

Currently, there is no service client or service server implementation (It's on my todo-list, though). However, there is a `Service` singleton that can be used to call a service.

Here's a short modified example of the service example provided in the *Examples*.

```
Button {
  onClicked: {
    var result = Service.call("/add_two_ints", "roscpp_tutorials/TwoInts",
                             { a: inputA.value, b: inputB.value })
    textResult.text = !!result ? ("Result: " + result.sum) : "Failed"
  }
}
```

The first argument is the service that is called, the second is the type of the service, and the final argument is the request that is sent.

The service either returns `false` if the call failed, `true` if the call was successful but the service description has an empty return message, and the return message of the service otherwise.

A service call is blocking and it is usually not a good idea to make blocking work on the UI thread. For that reason, there is also the `callAsync` method which runs the service call on a new separate thread and is additionally passed an optional callback which is called after the service call completed.

```
Button {
  onClicked: {
    textResult.text = "Loading..."
    Service.callAsync(
      "/add_two_ints", "roscpp_tutorials/TwoInts",
      { a: inputA.value, b: inputB.value },
      function (result) {
        textResult.text = !!result ? ("Result: " + result.sum) : "Failed"
      })
  }
}
```

9.1 API

```
class qml_ros_plugin::Service : public QObject
```

Public Functions

QVariant **call**(const QString &service, const QString &type, const QVariantMap &req)

Calls a service and returns the result.

Parameters

- **service** – The service topic.
- **type** – The type of the service, e.g., “roscpp_tutorials/TwoInts”
- **req** – The service request, i.e., a filled request message of the service type, e.g., “roscpp_tutorials/TwoIntsRequest”

Returns False, if request was not successful, true if the response message is empty and the translated service response, e.g., “roscpp_tutorials/TwoIntsResponse”, otherwise.

void **callAsync**(const QString &service, const QString &type, const QVariantMap &req, const QJSValue &callback = QJSValue())

Calls a service asynchronously returning immediately. Once the service call finishes, the optional callback is called with the result if provided.

Parameters

- **service** – The service topic.
- **type** – The type of the service, e.g., “roscpp_tutorials/TwoInts”
- **req** – The service request, i.e., a filled request message of the service type, e.g., “roscpp_tutorials/TwoIntsRequest”
- **callback** – The callback that is called once the service has finished.

SUBSCRIBERS

A subscriber listens for messages on a given topic.

10.1 Simple example

First, let's start with a simple example:

```
1 Subscriber {
2   id: mySubscriber
3   topic: "/intval"
4 }
```

This creates a subscriber object that is now available and subscribed to `/intval`. Let's assume the topic publishes an `std_msgs/Int32` message.

The `std_msgs/Int32` message is defined as follows:

```
int32 data
```

We can display the published value using a text field:

```
1 Text {
2   text: "Published value was: " + mySubscriber.message.data
3 }
```

Whenever a new message is received on `/intval` the message property is updated and the change is propagated to the text field. Thus, the text field will always display the last received value.

10.2 Full example

In most cases, the above Subscriber is sufficient. However, the Subscriber has more properties to give you more fine-grained control.

```
1 Subscriber {
2   id: mySubscriber
3   ns: "~" // Namespace
4   topic: "/intval"
5   queueSize: 10
6   running: true
```

(continues on next page)

```
7   onNewMessage: doStuff(message)
8 }
```

The namespace `ns` property enables you to set the namespace of the `ros::NodeHandle` that is created to subscribe to the given topic.

The `queueSize` property controls how many incoming messages are queued for processing before the oldest are dropped.

Using the `running` property, the subscriber can be enabled and disabled. If the property is set to `false`, the subscriber is shut down until it is set to `true` again and subscribes to the topic again. For example, the state of a Subscriber can be toggled using a button:

```
1 Button {
2   id: myButton
3   state: "active"
4   onClicked: {
5     mySubscriber.running = !mySubscriber.running
6     state = state == "active" ? "paused" : "active"
7   }
8   states: [
9     State {
10      name: "active"
11      PropertyChanges {
12        target: myButton
13        text: "Unsubscribe"
14      }
15    },
16    State {
17      name: "paused"
18      PropertyChanges {
19        target: myButton
20        text: "Subscribe"
21      }
22    }
23  ]
24 }
```

Whenever a new message is received, the `newMessage` signal is emitted and the message is passed and can be accessed as `message` which technically refers to the received message and not the message property of the Subscriber. Untechnically, they are usually the same, though.

Finally, there's also the `messageType` property which holds the type of the last received message, e.g., `std_msgs/Int32`.

10.3 API

```
class qml_ros_plugin::Subscriber : public QObjectRos
```

Public Functions

Subscriber()

The topic this subscriber subscribes to.

The maximum number of messages that are queued for processing. Default: 10 The namespace of the NodeHandle created for this subscriber. The last message that was received by this subscriber. The type of the last received message, e.g., geometry_msgs/Pose. Controls whether or not the subscriber is currently running, i.e., able to receive messages. Default: true

```
unsigned int getNumPublishers()
```

Returns The number of publishers this subscriber is connected to.

Signals

```
void topicChanged()
```

Emitted when the topic changed.

```
void queueSizeChanged()
```

Emitted when the queueSize changed.

```
void nsChanged()
```

Emitted when the namespace ns changed.

```
void runningChanged()
```

Emitted if the running state of this subscriber changed.

```
void messageChanged()
```

Emitted whenever a new message was received.

```
void messageTypeChanged()
```

Emitted whenever the type of the last received message changed.

```
void newMessage(QVariant message)
```

Emitted whenever a new message was received.

Parameters `message` – The received message.

TF TRANSFORMS

There are two methods for looking up **tf2** transforms.

11.1 Component

The `TfTransform` component can be used to subscribe to transforms between two frames.

```
TfTransform {
  id: tfTransform
  active: true // This is the default, if false no updates will be received
  sourceFrame: "base_link"
  targetFrame: "odom"
}
```

11.2 Static

You can use the `TfTransformListener` singleton to look up transforms if you just need it once.

```
Button {
  text: "Look Up"
  onClicked: {
    var transformStamped = TfTransformListener.lookupTransform(inputTargetFrame.text,
↳inputSourceFrame.text)
    if (!transformStamped.valid)
    {
      transformResult.text = "Transform from '" + inputSourceFrame.text + "' to '" +
↳inputTargetFrame.text + "' was not valid!\n" +
      "Exception: " + transformStamped.exception + "\nMessage: "
↳+ transformStamped.message
      return
    }
    transformResult.text = "Position:\n" + printVector3(transformStamped.transform.
↳translation) + "\nOrientation:\n" + printRotation(transformStamped.transform.rotation)
  }
}
```

Use the provided `Time.now()` static methods to look up at specific time points. For the latest, you can pass `new Date(0)`. Be aware that in `ros::Duration` the double constructor argument represents seconds whereas here the duration is given in milliseconds.

Warning: Be aware that *canLookup* can return a boolean value or a string error message. You should explicitly test for that since strings are truthy, too.

11.3 API

```
class qml_ros_plugin::TfTransformListener : public QObjectRos
```

Public Functions

QVariant **canTransform**(const QString &target_frame, const QString &source_frame, const ros::Time &time = ros::Time(0), double timeout = 0) const

Checks if a transform is possible. Returns true if possible, otherwise either false or if available a message why the transform failed.

Parameters

- **target_frame** – The frame into which to transform.
- **source_frame** – The frame from which to transform.
- **time** – The time at which to transform in seconds.
- **timeout** – How long to block before failing in milliseconds. Set to 0 for no timeout.

Returns True if the transform is possible, otherwise an error message (string) if available, false if not.

QVariant **canTransform**(const QString &target_frame, const ros::Time &target_time, const QString &source_frame, const ros::Time &source_time, const QString &fixed_frame, double timeout = 0) const

Checks if a transform is possible. Returns true if possible, otherwise either false or if available a message why the transform failed.

Parameters

- **target_frame** – The frame into which to transform.
- **target_time** – The time into which to transform.
- **source_frame** – The frame from which to transform.
- **source_time** – The time from which to transform.
- **fixed_frame** – The frame in which to treat the transform as constant in time.
- **timeout** – How long to block before failing in milliseconds. Set to 0 for no timeout.

Returns True if the transform is possible, otherwise an error message (string) if available, false if not.

QVariantMap **lookupTransform**(const QString &target_frame, const QString &source_frame, const ros::Time &time = ros::Time(0), double timeout = 0)

Get the transform between two frames by frame id.

Parameters

- **target_frame** – The frame to which the data should be transformed.
- **source_frame** – The frame where the data originated.

- **time** – The time at which the value of the transform is desired. Set to 0 for latest.
- **timeout** – How long to block before failing in milliseconds. Set to 0 for no timeout.

Returns A map containing a boolean valid field. If valid is true it also contains the transform. If valid is false, it might contain more information, e.g., an exception field with the name of the exception and a message field containing more information about the reason of failure.

```
QVariantMap lookupTransform(const QString &target_frame, const ros::Time &target_time, const QString
                             &source_frame, const ros::Time &source_time, const QString
                             &fixed_frame, double timeout = 0)
```

Get the transform between two frames by frame id.

Parameters

- **target_frame** – The frame to which the data should be transformed.
- **target_time** – The time to which the data should be transformed. Set to 0 for latest.
- **source_frame** – The frame where the data originated.
- **source_time** – The time at which the source_frame should be evaluated. Set to 0 for latest.
- **fixed_frame** – The frame in which to assume the transform is constant in time.
- **timeout** – How long to block before failing in milliseconds. Set to 0 for no timeout.

Returns A map containing a boolean valid field. If valid is true it also contains the transform. If valid is false, it might contain more information, e.g., an exception field with the name of the exception and a message field containing more information about the reason of failure.

```
void unregisterWrapper()
```

If the count of wrappers gets to zero, the resources of this singleton will be freed.

Signals

```
void transformChanged()
```

Emitted whenever a new transform arrived. Warning this signal is not throttled!

```
class qml_ros_plugin::TfTransform : public QObject
    Represents a tf transform between source and target frame.
```

Public Functions

```
TfTransform()
```

The source frame of the tf transform, i.e., the frame where the data originated.

The target frame of the tf transform, i.e., the frame to which the data should be transformed. Whether this tf transform is enabled, i.e., receiving transform updates. Alias for enabled The last received transform as a geometry_msgs/TransformStamped with an added boolean valid field and optional error fields. See [TfTransformListener::lookupTransform](#) An alias for transform. The translation part of the tf transform as a vector with x, y, z fields. Zero if no valid transform available (yet). The rotation part of the tf transform as a quaternion with w, x, y, z fields. Identity if no valid transform available (yet). The maximum rate in Hz at which tf updates are processed and emitted as changed signals. Default: 60 Note: The rate can not exceed 1000. To disable rate limiting set to 0. Whether the current transform, i.e., the fields message, translation and rotation are valid.

TIME

To preserve the accuracy and allow for compatible serialization of message objects, custom *Time* and *WallTime* anonymous datatypes were introduced. This *Time* datatype is used for time fields in received messages and can be obtained using the *Time* singleton.

Example:

```
property var currentTime: Time.now()
property var currentWallTime: WallTime.now()
```

The *Time* singleton wraps most static methods of `ros::Time` whereas the instances obtained using either *Time.now()* or *Time.create()* contain the instance methods and properties for `sec` and `nsec`.

Both wrapper types can be converted to QML/JavaScript Date objects using the *toJSDate()* function at the cost of micro- and nanosecond accuracy.

Please note that due to limitations in QML and JavaScript mathematical operations for `Time` and `WallTime` are not possible.

12.1 API

```
class qml_ros_plugin::Time : public TimeWrapper<ros::Time>
    Represents a point in time of the robot or simulation.
```

Properties:

- `sec`: unsigned integer containing the seconds passed since 1970
- `nsec`: unsigned integer containing the nanoseconds since the last second

Public Functions

```
inline double toSec() const
    The time in seconds (since 1970) as a decimal value. (Possible loss in precision)
```

```
inline quint64 toNSec() const
    The time in nanoseconds (since 1970) as an unsigned integer.
```

```
inline bool isZero() const
    Whether the time represented by this instance is zero.
```

inline QVariant **toJSDate**() const

A JS Date representing the value stored in this instance. Since JS Dates only have millisecond accuracy, information about microseconds and nanoseconds are lost. The time is always rounded down to prevent the JS Date from being in the future.

class qml_ros_plugin::**WallTime** : public TimeWrapper<ros::WallTime>

Represents a point in time of the current system.

Properties:

- **sec**: unsigned integer containing the seconds passed since 1970
- **nsec**: unsigned integer containing the nanoseconds since the last second

Public Functions

inline double **toSec**() const

The time in seconds (since 1970) as a decimal value. (Possible loss in precision)

inline quint64 **toNSec**() const

The time in nanoseconds (since 1970) as an unsigned integer.

inline bool **isZero**() const

Whether the time represented by this instance is zero.

inline QVariant **toJSDate**() const

A JS Date representing the value stored in this instance. Since JS Dates only have millisecond accuracy, information about microseconds and nanoseconds are lost. The time is always rounded down to prevent the JS Date from being in the future.

class qml_ros_plugin::**TimeSingleton** : public QObjectRos

Public Functions

QVariant **now**()

Returns the ros::Time as *Time*. This can be either the simulation time or the system time.

Before the ros::Time got valid, this method returns a zero *Time*.

QVariant **create**(double t)

Creates a *Time* instance from the given time in seconds since 1970.

QVariant **create**(quint32 sec, quint32 nsec)

Creates a *Time* instance from the given time in seconds since 1970 and nanoseconds since the last full second.

bool **isSimTime**()

Whether the time obtained using *now()* is the simulation time.

bool **isSystemTime**()

Whether the time obtained using *now()* is the system time.

bool **isValid**()

Whether the time obtained with *now()* is currently valid.

class qml_ros_plugin::**WallTimeSingleton** : public QObject

Public Functions

QVariant **now**()

Returns the `ros::WallTime` as *WallTime*. This is always the system time.

QVariant **create**(double t)

Creates a *WallTime* instance from the given time in seconds since 1970.

QVariant **create**(quint32 sec, quint32 nsec)

Creates a *WallTime* instance from the given time in seconds since 1970 and nanoseconds since the last full second.

For instructions on how to setup the QML ROS plugin and a quick getting started guide, check the [Quickstart](#).

More in-depth examples can be found in the examples folder as described in [Examples](#).

INDEX

Q

- qml_ros_plugin::ActionClient (C++ class), 2
- qml_ros_plugin::ActionClient::ActionClient (C++ function), 2
- qml_ros_plugin::ActionClient::cancelAllGoals (C++ function), 2
- qml_ros_plugin::ActionClient::cancelGoalsAtAndBeforeTime (C++ function), 2
- qml_ros_plugin::ActionClient::connectedChanged (C++ function), 2
- qml_ros_plugin::ActionClient::sendGoal (C++ function), 2
- qml_ros_plugin::Array (C++ class), 5
- qml_ros_plugin::Array::Array (C++ function), 5
- qml_ros_plugin::Array::at (C++ function), 5
- qml_ros_plugin::Array::pop (C++ function), 6
- qml_ros_plugin::Array::push (C++ function), 6
- qml_ros_plugin::Array::shift (C++ function), 6
- qml_ros_plugin::Array::spliceList (C++ function), 5
- qml_ros_plugin::Array::toArray (C++ function), 6
- qml_ros_plugin::Array::unshift (C++ function), 6
- qml_ros_plugin::GoalHandle (C++ class), 2
- qml_ros_plugin::GoalHandle::cancel (C++ function), 3
- qml_ros_plugin::GoalHandle::getResult (C++ function), 3
- qml_ros_plugin::GoalHandle::GoalHandle (C++ function), 3
- qml_ros_plugin::GoalHandle::resend (C++ function), 3
- qml_ros_plugin::ImageTransportManagerSingletonWrapper (C++ class), 10
- qml_ros_plugin::ImageTransportManagerSingletonWrapper::setLoadBalancingEnabled (C++ function), 10
- qml_ros_plugin::ImageTransportSubscriber (C++ class), 10
- qml_ros_plugin::ImageTransportSubscriber::ImageTransportSubscriber (C++ function), 10
- qml_ros_plugin::IO (C++ class), 20
- qml_ros_plugin::IO::readYaml (C++ function), 20
- qml_ros_plugin::IO::writeYaml (C++ function), 20
- qml_ros_plugin::Package (C++ class), 19
- qml_ros_plugin::Package::command (C++ function), 19
- qml_ros_plugin::Package::getAll (C++ function), 19
- qml_ros_plugin::Package::getPath (C++ function), 19
- qml_ros_plugin::Package::getPlugins (C++ function), 20
- qml_ros_plugin::Publisher (C++ class), 14
- qml_ros_plugin::Publisher::advertised (C++ function), 14
- qml_ros_plugin::Publisher::connected (C++ function), 14
- qml_ros_plugin::Publisher::disconnected (C++ function), 14
- qml_ros_plugin::Publisher::getNumSubscribers (C++ function), 14
- qml_ros_plugin::Publisher::publish (C++ function), 14
- qml_ros_plugin::Publisher::Publisher (C++ function), 14
- qml_ros_plugin::RosQmlSingletonWrapper (C++ class), 20
- qml_ros_plugin::RosQmlSingletonWrapper::advertise (C++ function), 23
- qml_ros_plugin::RosQmlSingletonWrapper::createActionClient (C++ function), 24
- qml_ros_plugin::RosQmlSingletonWrapper::createEmptyMessage (C++ function), 22
- qml_ros_plugin::RosQmlSingletonWrapper::createEmptyService (C++ function), 22
- qml_ros_plugin::RosQmlSingletonWrapper::debug (C++ function), 22
- qml_ros_plugin::RosQmlSingletonWrapper::error (C++ function), 22
- qml_ros_plugin::RosQmlSingletonWrapper::fatal (C++ function), 23
- qml_ros_plugin::RosQmlSingletonWrapper::getName (C++ function), 21
- qml_ros_plugin::RosQmlSingletonWrapper::getNamespace (C++ function), 21

qml_ros_plugin::RosQmlSingletonWrapper::info (C++ function), 22
 qml_ros_plugin::RosQmlSingletonWrapper::init (C++ function), 21
 qml_ros_plugin::RosQmlSingletonWrapper::initialized (C++ function), 24
 qml_ros_plugin::RosQmlSingletonWrapper::isInitialized (C++ function), 32, 33
 qml_ros_plugin::RosQmlSingletonWrapper::isInitialized (C++ function), 21
 qml_ros_plugin::RosQmlSingletonWrapper::ok (C++ function), 21
 qml_ros_plugin::RosQmlSingletonWrapper::queryTopicInfo (C++ function), 33
 qml_ros_plugin::RosQmlSingletonWrapper::queryTopicInfo (C++ function), 21
 qml_ros_plugin::RosQmlSingletonWrapper::queryTopicType (C++ function), 22
 qml_ros_plugin::RosQmlSingletonWrapper::setThreadName (C++ function), 21
 qml_ros_plugin::RosQmlSingletonWrapper::shutdown (C++ function), 24
 qml_ros_plugin::RosQmlSingletonWrapper::spinOnce (C++ function), 21
 qml_ros_plugin::RosQmlSingletonWrapper::subscribe (C++ function), 23
 qml_ros_plugin::RosQmlSingletonWrapper::warn (C++ function), 22
 qml_ros_plugin::Service (C++ class), 26
 qml_ros_plugin::Service::call (C++ function), 26
 qml_ros_plugin::Service::callAsync (C++ function), 26
 qml_ros_plugin::Subscriber (C++ class), 29
 qml_ros_plugin::Subscriber::getNumPublishers (C++ function), 29
 qml_ros_plugin::Subscriber::messageChanged (C++ function), 29
 qml_ros_plugin::Subscriber::messageTypeChanged (C++ function), 29
 qml_ros_plugin::Subscriber::newMessage (C++ function), 29
 qml_ros_plugin::Subscriber::nsChanged (C++ function), 29
 qml_ros_plugin::Subscriber::queueSizeChanged (C++ function), 29
 qml_ros_plugin::Subscriber::runningChanged (C++ function), 29
 qml_ros_plugin::Subscriber::Subscriber (C++ function), 29
 qml_ros_plugin::Subscriber::topicChanged (C++ function), 29
 qml_ros_plugin::TerminalState (C++ class), 3
 qml_ros_plugin::TerminalState::TerminalState (C++ function), 3
 qml_ros_plugin::TfTransform (C++ class), 33
 qml_ros_plugin::TfTransform::TfTransform (C++ function), 33
 qml_ros_plugin::TfTransformListener (C++ class), 32
 qml_ros_plugin::TfTransformListener::canTransform (C++ function), 32
 qml_ros_plugin::TfTransformListener::lookUpTransform (C++ function), 32
 qml_ros_plugin::TfTransformListener::transformChanged (C++ function), 33
 qml_ros_plugin::TfTransformListener::unregisterWrapper (C++ function), 33
 qml_ros_plugin::Time (C++ class), 35
 qml_ros_plugin::Time::isZero (C++ function), 35
 qml_ros_plugin::Time::toJSDate (C++ function), 35
 qml_ros_plugin::Time::toNSec (C++ function), 35
 qml_ros_plugin::Time::toSec (C++ function), 35
 qml_ros_plugin::TimeSingleton (C++ class), 36
 qml_ros_plugin::TimeSingleton::create (C++ function), 36
 qml_ros_plugin::TimeSingleton::isSimTime (C++ function), 36
 qml_ros_plugin::TimeSingleton::isSystemTime (C++ function), 36
 qml_ros_plugin::TimeSingleton::isValid (C++ function), 36
 qml_ros_plugin::TimeSingleton::now (C++ function), 36
 qml_ros_plugin::TopicInfo (C++ class), 20
 qml_ros_plugin::TopicInfo::TopicInfo (C++ function), 20
 qml_ros_plugin::WallTime (C++ class), 36
 qml_ros_plugin::WallTime::isZero (C++ function), 36
 qml_ros_plugin::WallTime::toJSDate (C++ function), 36
 qml_ros_plugin::WallTime::toNSec (C++ function), 36
 qml_ros_plugin::WallTime::toSec (C++ function), 36
 qml_ros_plugin::WallTimeSingleton (C++ class), 36
 qml_ros_plugin::WallTimeSingleton::create (C++ function), 37
 qml_ros_plugin::WallTimeSingleton::now (C++ function), 37