roscpp overview (/roscpp/Overview)*: Initialization and Shutdown (/roscpp/Overview/Initialization%20and%20Shutdown) | Basics (/roscpp/Overview/Messages) | Advanced: Traits [ROS C Turtle] (/roscpp/Overview/MessagesTraits) | Advanced: Custom Allocators [ROS C Turtle] (/roscpp/Overview/MessagesCustomAllocators) | Advanced: Serialization and Adapting Types [ROS C Turtle] (/roscpp/Overview/MessagesSerializationAndAdaptingTypes) | Publishers and Subscribers (/roscpp/Overview/Publishers%20and%20Subscribers) | Services (/roscpp/Overview/Services) | Parameter Server (/roscpp/Overview/Parameter%20Server) | Timers (Periodic Callbacks) (/roscpp/Overview/Timers) | NodeHandles (/roscpp/Overview/NodeHandles) | Callbacks and Spinning (/roscpp/Overview/Callbacks%20and%20Spinning) | Logging (/roscpp/Overview/Logging) | Names and Node Information (/roscpp/Overview/Names%20and%20Node%20Information) | Time | Exceptions (/roscpp/Overview/Exceptions) | Compilation Options (/roscpp/Overview/Compilation%20Options) | Advanced: Internals (/roscpp/Overview/Internals) | tf/Overview (/tf/Overview) | tf/Tutorials (/tf/Tutorials) | C++ Style Guide (/CppStyleGuide)*

Contents

1. Time and Duration

See also: ros::TimeBase API docs (http://docs.ros.org/latest/api/rostime/html/classros_1_1TimeBase.html), ros::DurationBase API docs (http://docs.ros.org/latest/api/rostime/html/classros_1_1DurationBase.html)

ROS has builtin `time` and `duration` primitive types, which roslib (/roslib) provides as the `ros::Time` and `ros::Duration` classes, respectively. A `Time` is a specific moment (e.g. "today at 5pm") whereas a `Duration` is a period of time (e.g. "5 hours"). Durations can be negative.

Times and durations have identical representations:

```
int32 sec
int32 nsec
```

ROS has the ability to setup a simulated Clock (/Clock) for nodes. Instead of using platform time routines, you should use roscpp's time routines for accessing the current time, which will work seamlessly with simulated Clock (/Clock) time as well as wall-clock time.

1.1 Getting the Current Time

`ros::Time::now()`

Get the current time as a `ros::Time` instance:

Toggle line numbers

```
ros::Time begin = ros::Time::now();
```

1.1.1 Time zero

When using simulated Clock (/Clock) time, `now()` returns time `0` until first message has been received on `/clock`, so `0` means essentially that the client does not know clock time yet. A value of `0` should therefore be treated differently, such as looping over `now()` until non-zero is returned.

1.2 Creating Time and Duration Instances

You can create a `Time` or `Duration` to a specific value as well, either floating-point seconds:

Toggle line numbers

```
ros::Time a_little_after_the_beginning(0.001);
ros::Duration five_seconds(5.0);
```

or through the two-integer constructor:

Toggle line numbers

```
ros::Time a_little_after_the_beginning(0, 1000000);
ros::Duration five_seconds(5, 0);
```

1.3 Converting Time and Duration Instances

`Time` and `Duration` objects can also be turned into floating point seconds:

Toggle line numbers

```
double secs =ros::Time::now().toSec();

ros::Duration d(0.5);
secs = d.toSec();
```

1.4 Time and Duration Arithmetic

Like other primitive types, you can perform arithmetic operations on `Time`s and `Duration`s. People are often initially confused on what arithmetic with these instances is like, so it's good to run through some examples:

1 hour + 1 hour = 2 hours (*duration* + *duration* = *duration*)

2 hours - 1 hour = 1 hour (*duration* - *duration* = *duration*)

Today + 1 day = tomorrow (*time* + *duration* = *time*)

Today - tomorrow = -1 day (*time* - *time* = *duration*)

Today + tomorrow = *error* (*time* + *time is undefined*)

Arithmetic with `Time` and `Duration` instances is similar to the above examples:

Toggle line numbers

```
ros::Duration two_hours = ros::Duration(60*60) + ros::Duration(60*60);
ros::Duration one_hour = ros::Duration(2*60*60) - ros::Duration(60*60);
ros::Time tomorrow = ros::Time::now() + ros::Duration(24*60*60);
ros::Duration negative_one_day = ros::Time::now() - tomorrow;
```

2. Sleeping and Rates

`bool ros::Duration::sleep()`

Sleep for the amount of time specified by the duration:

Toggle line numbers

```
ros::Duration(0.5).sleep(); // sleep for half a second

```

`ros::Rate`

roslib provides a `ros::Rate` convenience class which makes a best effort at maintaining a particular rate for a loop. For example:

```
ros::Rate r(10); // 10 hz
while (ros::ok())
{
  ... do some work ...
  r.sleep();
}
```

In the above example, the `Rate` instance will attempt to keep the loop at 10hz by accounting for the time used by the work done during the loop.

Note: It is generally recommended to use `Timers` instead of `Rate`. See the Timers Tutorial (/roscpp_tutorials/Tutorials/Timers) for details.

3. Wall Time

For cases where you want access to the actual wall-clock time even if running inside simulation, roslib provides `Wall` versions of all its time constructs, i.e. `ros::WallTime`, `ros::WallDuration`, and `ros::WallRate` which have identical interfaces to `ros::Time`, `ros::Duration`, and `ros::Rate` respectively.

Wiki: roscpp/Overview/Time (last edited 2017-07-10 22:17:20 by BryceWilley (/BryceWilley))