

---

# **kRPC**

***Release 0.4.0***

**Oct 15, 2017**



## CONTENTS



kRPC allows you to control Kerbal Space Program from scripts running outside of the game. It comes with client libraries for many popular languages including *C#*, *C++*, *Java*, *Lua* and *Python*. Clients, made by others, are also available for [Ruby](#) and [Haskell](#).

- *Getting Started Guide*
- *Tutorials and Examples*
- *Clients, services and tools made by others*

The mod exposes most of KSP's API for controlling and interacting with rockets, and also includes support several popular mods including Ferram Aerospace Research, Kerbal Alarm Clock and Infernal Robotics.

This functionality is provided to client programs via a server running in the game. Client scripts connect to this server and use it to execute 'remote procedures'. This communication can be done on local machine only, over a local network, or even over the wider internet if configured correctly. The server is extensible - additional remote procedures (grouped into "services") can be added to the server using the *Service API*.



## GETTING STARTED

This short guide explains the basics for getting the kRPC server set up and running, and writing a basic Python script to interact with the game.

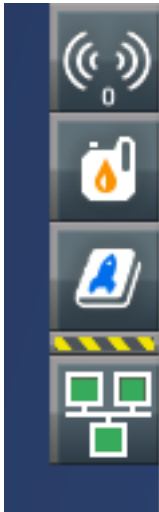
### 1.1 The Server Plugin

#### 1.1.1 Installation

1. Download and install the kRPC server plugin from one of these locations:
  - [Github](#)
  - [SpaceDock](#)
  - [Curse](#)
  - Or the install it using [CKAN](#)
2. Start up KSP and load a save game.
3. You should be greeted by the server window:



4. Click "Start server" to, erm... start the server! If all goes well, the light should turn a happy green color.
5. You can hide the window by clicking the close button in the top right. The window can also be shown/hidden by clicking on the icon in the top right:



This icon will also turn green when the server is online.

### 1.1.2 Configuration

The server is configured by clicking edit on the window displayed in-game:



1. **Protocol:** this is the protocol used by the server. This affects type of client can connect to the server. For Python, and most other clients that communicate over TCP/IP you want to select “Protobuf over TCP”.
2. **Address:** this is the IP address that the server will listen on. To only allow connections from the local machine, select ‘localhost’ (the default). To allow connections over a network, either select the local IP address of your machine, or choose ‘Manual’ and enter the local IP address manually.
3. **RPC and Stream port numbers:** These need to be set to port numbers that are available on your machine. In most cases, they can just be left as the default.

There are also several advanced settings, which are hidden by default, but can be revealed by checking “Show advanced settings”:

1. **Auto-start server:** When enabled, the server will start automatically when the game loads.
2. **Auto-accept new clients:** When enabled, new client connections are automatically allowed. When disabled, a pop-up is displayed asking whether the new client connection should be allowed.

The other advanced settings control the *performance of the server*.

## 1.2 The Python Client

---

**Note:** kRPC supports both Python 2.7 and Python 3.x.

---

### 1.2.1 On Windows

1. If you don’t already have python installed, download the python installer and run it: <https://www.python.org/downloads/windows> When running the installer, make sure that pip is installed as well.
2. Install the kRPC python module, by opening command prompt and running the following command:  
`C:\Python27\Scripts\pip.exe install krpc` You might need to replace C:\Python27 with the location of your python installation.
3. Run Python IDLE (or your favorite editor) and start coding!

### 1.2.2 On Linux

1. Your linux distribution likely already comes with python installed. If not, install python using your favorite package manager, or get it from here: <https://www.python.org/downloads>
2. You also need to install pip, either using your package manager, or from here: <https://pypi.python.org/pypi/pip>
3. Install the kRPC python module by running the following from a terminal: `sudo pip install krpc`
4. Start coding!

## 1.3 ‘Hello World’ Script

Run KSP and start the server with the default settings. Then run the following python script.

```

1 import krpc
2 conn = krpc.connect(name='Hello World')
3 vessel = conn.space_center.active_vessel
4 print(vessel.name)

```

This does the following: line 1 loads the kRPC python module, line 2 opens a new connection to the server, line 3 gets the active vessel and line 4 prints out the name of the vessel. You should see something like the following:



Congratulations! You've written your first script that communicates with KSP.

## 1.4 Going further...

- For some more interesting examples of what you can do with kRPC, check out the *tutorials*.
- Client libraries are available for other languages too, including *C#*, *C++*, *Java* and *Lua*.
- It is also possible to *communicate with the server manually* from any language you like.

## TUTORIALS AND EXAMPLES

This collection of tutorials and example scripts explain how to use the features of kRPC.

### 2.1 Sub-Orbital Flight

This introductory tutorial uses kRPC to send some Kerbals on a sub-orbital flight, and (hopefully) returns them safely back to Kerbin. It covers the following topics:

- Controlling a rocket (activating stages, setting the throttle)
- Using the auto pilot to point the vessel in a specific direction
- Using events to wait for things to happen in game
- Tracking the amount of resources in the vessel
- Tracking flight and orbital data (such as altitude and apoapsis altitude)

---

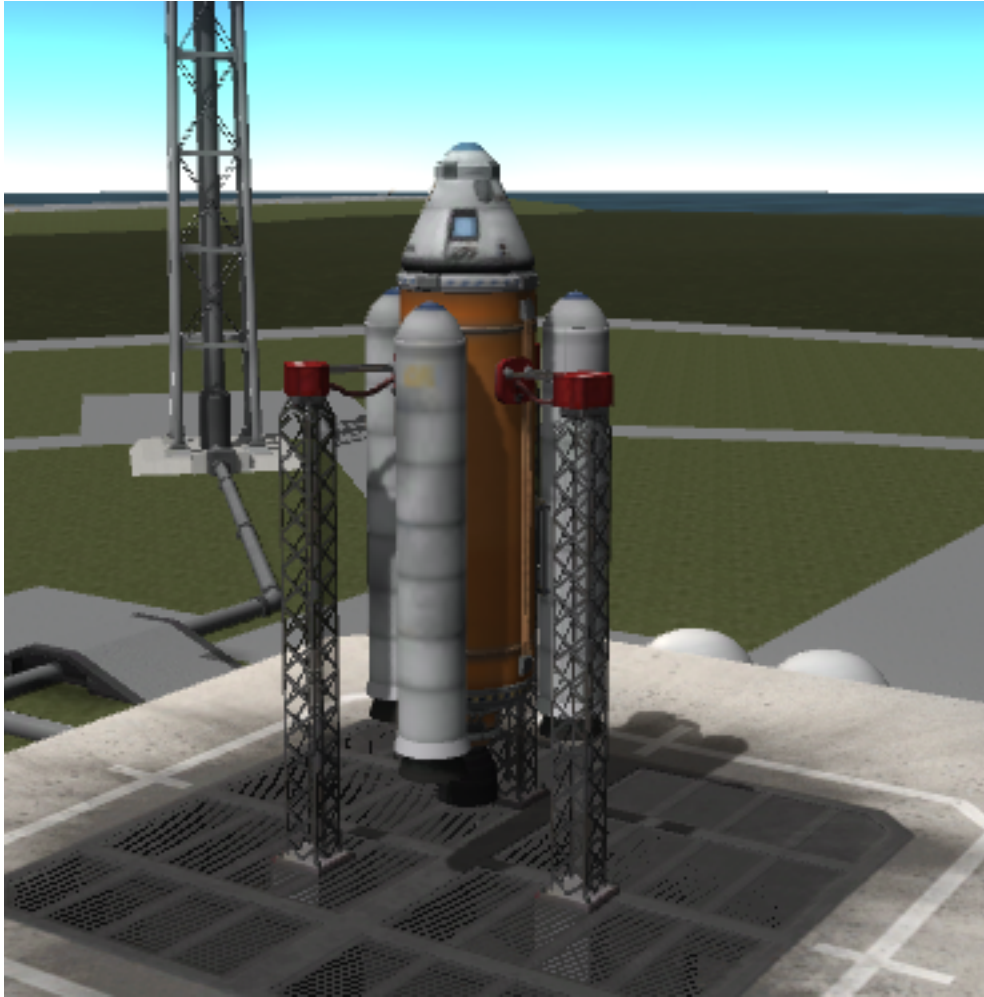
**Note:** For details on how to write scripts and connect to kRPC, see the *Getting Started* guide.

---

This tutorial uses the two stage rocket pictured below. The craft file for this rocket can be downloaded [here](#).

This tutorial includes source code examples for the main client languages that kRPC supports. The entire program, for your chosen language can be downloaded from [here](#):

C#, C++, Java, Lua, Python



### 2.1.1 Part One: Preparing for Launch

The first thing we need to do is open a connection to the server. We can also pass a descriptive name for our script that will appear in the server window in game:

C#

C++

Java

Lua

Python

```
10  var conn = new Connection ("Sub-orbital flight");
```

```
9   krpc::Client conn = krpc::connect("Sub-orbital flight");
10  krpc::services::KRPC krpc(&conn);
11  krpc::services::SpaceCenter space_center(&conn);
```

```

20 Connection connection = Connection.newInstance("Sub-orbital flight");
21 KRPC krpc = KRPC.newInstance(connection);
22 SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);

```

```

1 local krpc = require 'krpc'
2 local platform = require 'krpc.platform'
3 local conn = krpc.connect('Sub-orbital flight')

```

```

3 conn = krpc.connect(name='Sub-orbital flight')

```

Next we need to get an object representing the active vessel. It's via this object that we will send instructions to the rocket:

C#

C++

Java

Lua

Python

```

12 var vessel = conn.SpaceCenter ().ActiveVessel;

```

```

13 auto vessel = space_center.active_vessel();

```

```

24 SpaceCenter.Vessel vessel = spaceCenter.getActiveVessel();

```

```

5 local vessel = conn.space_center.active_vessel

```

```

5 vessel = conn.space_center.active_vessel

```

We then need to prepare the rocket for launch. The following code sets the throttle to maximum and instructs the auto-pilot to hold a pitch and heading of 90° (vertically upwards). It then waits for 1 second for these settings to take effect.

C#

C++

Java

Lua

Python

```

14 vessel.AutoPilot.TargetPitchAndHeading (90, 90);
15 vessel.AutoPilot.Engage ();
16 vessel.Control.Throttle = 1;
17 System.Threading.Thread.Sleep (1000);

```

```

15 vessel.auto_pilot().target_pitch_and_heading(90, 90);
16 vessel.auto_pilot().engage();
17 vessel.control().set_throttle(1);
18 std::this_thread::sleep_for(std::chrono::seconds(1));

```

```
26     vessel.getAutoPilot().targetPitchAndHeading(90, 90);
27     vessel.getAutoPilot().engage();
28     vessel.getControl().setThrottle(1);
29     Thread.sleep(1000);
```

```
7  vessel.auto_pilot:target_pitch_and_heading(90, 90)
8  vessel.auto_pilot:engage()
9  vessel.control.throttle = 1
10 platform.sleep(1)
```

```
7  vessel.auto_pilot.target_pitch_and_heading(90, 90)
8  vessel.auto_pilot.engage()
9  vessel.control.throttle = 1
10 time.sleep(1)
```

### 2.1.2 Part Two: Lift-off!

We're now ready to launch by activating the first stage (equivalent to pressing the space bar):

C#

C++

Java

Lua

Python

```
19     Console.WriteLine ("Launch!");
20     vessel.Control.ActivateNextStage ();
```

```
20     std::cout << "Launch!" << std::endl;
21     vessel.control().activate_next_stage();
```

```
31     System.out.println("Launch!");
32     vessel.getControl().activateNextStage();
```

```
12     print('Launch!')
13     vessel.control:activate_next_stage()
```

```
12     print('Launch!')
13     vessel.control.activate_next_stage()
```

The rocket has a solid fuel stage that will quickly run out, and will need to be jettisoned. We can monitor the amount of solid fuel in the rocket using an event that is triggered when there is very little solid fuel left in the rocket. When the event is triggered, we can activate the next stage to jettison the boosters:

C#

C++

Java

Lua

Python

```

23         var solidFuel = Connection.GetCall(() => vessel.Resources.Amount(
↳ "SolidFuel"));
24         var expr = Expression.LessThan(
25             conn, Expression.Call(conn, solidFuel), Expression.ConstantFloat(conn,
↳ 0.1f));
26         var evnt = conn.KRPC().AddEvent(expr);
27         lock (evnt.Condition) {
28             evnt.Wait();
29         }

```

```

26     auto solid_fuel = vessel.resources().amount_call("SolidFuel");
27     auto expr = Expr::less_than(
28         conn, Expr::call(conn, solid_fuel), Expr::constant_float(conn, 0.1));
29     auto event = krpc.add_event(expr);
30     event.acquire();
31     event.wait();
32     event.release();

```

```

34     {
35         ProcedureCall solidFuel = connection.getCall(vessel.getResources(), "amount",
↳ "SolidFuel");
36         Expression expr = Expression.lessThan(
37             connection,
38             Expression.call(connection, solidFuel),
39             Expression.constantFloat(connection, 0.1f));
40         Event event = krpc.addEvent(expr);
41         synchronized (event.getCondition()) {
42             event.waitFor();
43         }
44     }
45
46     System.out.println("Booster separation");
47     vessel.getControl().activateNextStage();

```

```

15 while vessel.resources:amount('SolidFuel') > 0.1 do
16     platform.sleep(1)
17 end
18 print('Booster separation')
19 vessel.control:activate_next_stage()

```

```

15 fuel_amount = conn.get_call(vessel.resources.amount, 'SolidFuel')
16 expr = conn.krpc.Expression.less_than(
17     conn.krpc.Expression.call(fuel_amount),
18     conn.krpc.Expression.constant_float(0.1))
19 event = conn.krpc.add_event(expr)
20 with event.condition:
21     event.wait()
22 print('Booster separation')
23 vessel.control.activate_next_stage()

```

In this bit of code, `vessel.resources` returns a *Resources* object that is used to get information about the resources in the rocket. The code creates the expression `vessel.resources.amount('SolidFuel') < 0.1` on the server, using the expression API. This expression is then used to drive an event, which is triggered when the expression returns true.

### 2.1.3 Part Three: Reaching Apoapsis

Next we will execute a gravity turn when the rocket reaches a sufficiently high altitude. The following uses an event to wait until the altitude of the rocket reaches 10km:

C#

C++

Java

Lua

Python

```

36         var meanAltitude = Connection.GetCall(() => vessel.Flight(null).
↪MeanAltitude);
37         var expr = Expression.GreaterThan(
38             conn, Expression.Call(conn, meanAltitude), Expression.
↪ConstantDouble(conn, 10000));
39         var evnt = conn.KRPC().AddEvent(expr);
40         lock (evnt.Condition) {
41             evnt.Wait();
42         }

```

```

39     auto mean_altitude = vessel.flight().mean_altitude_call();
40     auto expr = Expr::greater_than(
41         conn, Expr::call(conn, mean_altitude), Expr::constant_double(conn, 10000));
42     auto event = krpc.add_event(expr);
43     event.acquire();
44     event.wait();
45     event.release();

```

```

50     ProcedureCall meanAltitude = connection.getCall(vessel.flight(null),
↪"getMeanAltitude");
51     Expression expr = Expression.greaterThan(
52         connection,
53         Expression.call(connection, meanAltitude),
54         Expression.constantDouble(connection, 10000));
55     Event event = krpc.addEvent(expr);
56     synchronized (event.getCondition()) {
57         event.waitFor();
58     }

```

```

21 while vessel:flight().mean_altitude < 10000 do
22     platform.sleep(1)
23 end

```

```

25 mean_altitude = conn.get_call(getattr, vessel.flight(), 'mean_altitude')
26 expr = conn.krpc.Expression.greater_than(
27     conn.krpc.Expression.call(mean_altitude),
28     conn.krpc.Expression.constant_double(10000))
29 event = conn.krpc.add_event(expr)
30 with event.condition:
31     event.wait()

```

In this bit of code, calling `vessel.flight()` returns a *Flight* object that is used to get all sorts of information about the rocket, such as the direction it is pointing in and its velocity.



Now we need to angle the rocket over to a pitch of 60° and maintain a heading of 90° (west). To do this, we simply reconfigure the auto-pilot:

C#

C++

Java

Lua

Python

```
45 Console.WriteLine ("Gravity turn");
46 vessel.AutoPilot.TargetPitchAndHeading (60, 90);
```

```
48 std::cout << "Gravity turn" << std::endl;
49 vessel.auto_pilot().target_pitch_and_heading(60, 90);
```

```
61 System.out.println("Gravity turn");
62 vessel.getAutoPilot().targetPitchAndHeading(60, 90);
```

```
25 print('Gravity turn')
26 vessel.auto_pilot:target_pitch_and_heading(60, 90)
```

```
33 print('Gravity turn')
34 vessel.auto_pilot.target_pitch_and_heading(60, 90)
```

Now we wait until the apoapsis reaches 100km (again, using an event), then reduce the throttle to zero, jettison the launch stage and turn off the auto-pilot:

C#

C++

Java

Lua

Python

```
32 {
33     var apoapsisAltitude = Connection.GetCall(() => vessel.Orbit.
↪ApoapsisAltitude);
34     var expr = Expression.GreaterThan(
35         conn, Expression.Call(conn, apoapsisAltitude), Expression.
↪ConstantDouble(conn, 100000));
36     var evnt = conn.KRPC().AddEvent(expr);
37     lock (evnt.Condition) {
38         evnt.Wait();
39     }
40 }
41
42 Console.WriteLine ("Launch stage separation");
43 vessel.Control.Throttle = 0;
44 System.Threading.Thread.Sleep (1000);
45 vessel.Control.ActivateNextStage ();
46 vessel.AutoPilot.Disengage ();
```

```

51 {
52     auto apoapsis_altitude = vessel.orbit().apoapsis_altitude_call();
53     auto expr = Expr::greater_than(
54         conn, Expr::call(conn, apoapsis_altitude), Expr::constant_double(conn, 100000));
55     auto event = krpc.add_event(expr);
56     event.acquire();
57     event.wait();
58     event.release();
59 }
60
61 std::cout << "Launch stage separation" << std::endl;
62 vessel.control().set_throttle(0);
63 std::this_thread::sleep_for(std::chrono::seconds(1));
64 vessel.control().activate_next_stage();
65 vessel.auto_pilot().disengage();

```

```

64 {
65     ProcedureCall apoapsisAltitude = connection.getCall(
66         vessel.getOrbit(), "getApoapsisAltitude");
67     Expression expr = Expression.greaterThan(
68         connection,
69         Expression.call(connection, apoapsisAltitude),
70         Expression.constantDouble(connection, 100000));
71     Event event = krpc.addEvent(expr);
72     synchronized (event.getCondition()) {
73         event.waitFor();
74     }
75 }
76
77 System.out.println("Launch stage separation");
78 vessel.getControl().setThrottle(0);
79 Thread.sleep(1000);
80 vessel.getControl().activateNextStage();
81 vessel.getAutoPilot().disengage();

```

```

28 while vessel.orbit.apoapsis_altitude < 100000 do
29     platform.sleep(1)
30 end
31 print('Launch stage separation')
32 vessel.control.throttle = 0
33 platform.sleep(1)
34 vessel.control.activate_next_stage()
35 vessel.auto_pilot:disengage()

```

```

36 apoapsis_altitude = conn.get_call(getattr, vessel.orbit, 'apoapsis_altitude')
37 expr = conn.krpc.Expression.greater_than(
38     conn.krpc.Expression.call(apoapsis_altitude),
39     conn.krpc.Expression.constant_double(100000))
40 event = conn.krpc.add_event(expr)
41 with event.condition:
42     event.wait()
43
44 print('Launch stage separation')
45 vessel.control.throttle = 0
46 time.sleep(1)
47 vessel.control.activate_next_stage()
48 vessel.auto_pilot.disengage()

```

In this bit of code, `vessel.orbit` returns an *Orbit* object that contains all the information about the orbit of the rocket.

## 2.1.4 Part Four: Returning Safely to Kerbin

Our Kerbals are now heading on a sub-orbital trajectory and are on a collision course with the surface. All that remains to do is wait until they fall to 1km altitude above the surface, and then deploy the parachutes. If you like, you can use time acceleration to skip ahead to just before this happens - the script will continue to work.

C#

C++

Java

Lua

Python

```

64     {
65         var srfAltitude = Connection.GetCall(() => vessel.Flight(null).
↪SurfaceAltitude);
66         var expr = Expression.LessThan(
67             conn, Expression.Call(conn, srfAltitude), Expression.
↪ConstantDouble(conn, 1000));
68         var evnt = conn.KRPC().AddEvent(expr);
69         lock (evnt.Condition) {
70             evnt.Wait();
71         }
72     }
73
74     vessel.Control.ActivateNextStage ();

```

```

67     {
68         auto srf_altitude = vessel.flight().surface_altitude_call();
69         auto expr = Expr::less_than(
70             conn, Expr::call(conn, srf_altitude), Expr::constant_double(conn, 1000));
71         auto event = krpc.add_event(expr);
72         event.acquire();
73         event.wait();
74         event.release();
75     }
76
77     vessel.control().activate_next_stage();

```

```

83     {
84         ProcedureCall srfAltitude = connection.getCall(
85             vessel.flight(null), "getSurfaceAltitude");
86         Expression expr = Expression.lessThan(
87             connection,
88             Expression.call(connection, srfAltitude),
89             Expression.constantDouble(connection, 1000));
90         Event event = krpc.addEvent(expr);
91         synchronized (event.getCondition()) {
92             event.waitFor();
93         }
94     }

```

```
vessel.getControl().activateNextStage();
```

```
while vessel:flight().surface_altitude > 1000 do
    platform.sleep(1)
end
vessel.control:activate_next_stage()
```

```
srf_altitude = conn.get_call(getattr, vessel.flight(), 'surface_altitude')
expr = conn.krpc.Expression.less_than(
    conn.krpc.Expression.call(srf_altitude),
    conn.krpc.Expression.constant_double(1000))
event = conn.krpc.add_event(expr)
with event.condition:
    event.wait()
vessel.control.activate_next_stage()
```

The parachutes should have now been deployed. The next bit of code will repeatedly print out the altitude of the capsule until its speed reaches zero – which will happen when it lands:

C#

C++

Java

Lua

Python

```
while (vessel.Flight (vessel.Orbit.Body.ReferenceFrame).VerticalSpeed < -0.1)
{
    Console.WriteLine ("Altitude = {0:F1} meters", vessel.Flight ().
    SurfaceAltitude);
    System.Threading.Thread.Sleep (1000);
}
Console.WriteLine ("Landed!");
conn.Dispose();
```

```
while (vessel.flight(vessel.orbit().body().reference_frame()).vertical_speed() < -0.
1) {
    std::cout << "Altitude = " << vessel.flight().surface_altitude() << " meters" <<
std::endl;
    std::this_thread::sleep_for(std::chrono::seconds(1));
}
std::cout << "Landed!" << std::endl;
```

```
while (vessel.flight(vessel.getOrbit().getBody().getReferenceFrame()).
getVerticalSpeed() < -0.1) {
    System.out.printf("Altitude = %.1f meters\n", vessel.flight(null).
getSurfaceAltitude());
    Thread.sleep(1000);
}
System.out.println("Landed!");
connection.close();
```

```

42 while vessel:flight(vessel.orbit.body.reference_frame).vertical_speed < -0.1 do
43     print(string.format('Altitude = %.1f meters',
44                         vessel:flight().surface_altitude))
45     platform.sleep(1)
46 end
47 print('Landed!')

```

```

60 while vessel:flight(vessel.orbit.body.reference_frame).vertical_speed < -0.1:
61     print('Altitude = %.1f meters' % vessel:flight().surface_altitude)
62     time.sleep(1)
63 print('Landed!')

```

This bit of code uses the `vessel:flight()` function, as before, but this time it is passed a *ReferenceFrame* parameter. We want to get the vertical speed of the capsule relative to the surface of Kerbin, so the values returned by the flight object need to be relative to the surface of Kerbin. We therefore pass `vessel.orbit.body.reference_frame` to `vessel:flight()` as this reference frame has its origin at the center of Kerbin and it rotates with the planet. For more information, check out the tutorial on *Reference Frames*.

Your Kerbals should now have safely landed back on the surface.

## 2.2 Reference Frames

- *Introduction*
  - *Origin Position and Axis Orientation*
    - \* *Celestial Body Reference Frame*
    - \* *Vessel Orbital Reference Frame*
    - \* *Vessel Surface Reference Frame*
  - *Linear Velocity and Angular Velocity*
- *Available Reference Frames*
- *Custom Reference Frames*
- *Converting Between Reference Frames*
- *Visual Debugging*
- *Examples*
  - *Nayball directions*
  - *Orbital directions*
  - *Surface ‘prograde’*
  - *Vessel Speed*
  - *Vessel Velocity*
  - *Angle of attack*
  - *Landing Site*

### 2.2.1 Introduction

All of the positions, directions, velocities and rotations in kRPC are relative to something, and *reference frames* define what that something is.

A reference frame specifies:

- The position of the origin at (0,0,0)
- the direction of the coordinate axes x, y, and z
- the linear velocity of the origin (if the reference frame moves)
- The angular velocity of the coordinate axes (the speed and direction of rotation of the axes)

---

**Note:** KSP and kRPC use a left handed coordinate system

---

#### Origin Position and Axis Orientation

The following gives some examples of the position of the origin and the orientation of the coordinate axes for various reference frames.

#### Celestial Body Reference Frame

The reference frame obtained by calling `CelestialBody.reference_frame` for Kerbin has the following properties:

- The origin is at the center of Kerbin,
- the y-axis points from the center of Kerbin to the north pole,
- the x-axis points from the center of Kerbin to the intersection of the prime meridian and equator (the surface position at 0° longitude, 0° latitude),
- the z-axis points from the center of Kerbin to the equator at 90°E longitude,
- and the axes rotate with the planet, i.e. the reference frame has the same rotational/angular velocity as Kerbin.

This means that the reference frame is *fixed* relative to Kerbin – it moves with the center of the planet, and also rotates with the planet. Therefore, positions in this reference frame are relative to the center of the planet. The following code prints out the position of the active vessel in Kerbin's reference frame:

C#

C++

Java

Lua

Python

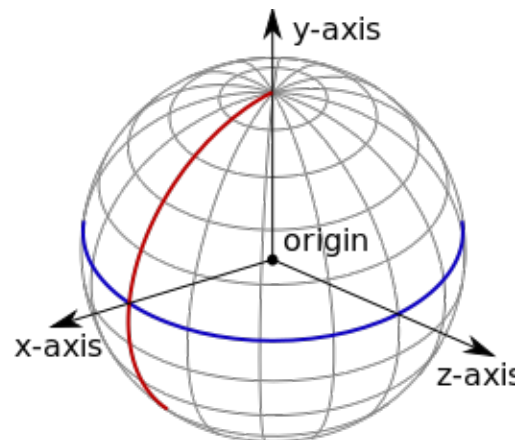


Fig. 2.1: The reference frame for a celestial body, such as Kerbin. The equator is shown in blue, and the prime meridian in red. The black arrows show the coordinate axes, and the origin is at the center of the planet.

```

using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class VesselPosition
{
    public static void Main ()
    {
        using (var connection = new Connection_
↪()) {
            var vessel = connection.SpaceCenter_
↪().ActiveVessel;
            var position = vessel.Position_
↪(vessel.Orbit.Body.ReferenceFrame);
            Console.WriteLine ("({0:F1}, {1:F1},
↪{2:F1})",
                                position.Item1,
↪position.Item2, position.Item3);
        }
    }
}

```

```

#include <iostream>
#include <iomanip>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client conn = krpc::connect();
    krpc::services::SpaceCenter spaceCenter(&conn);
    auto vessel = spaceCenter.active_vessel();
    auto position = vessel.position(vessel.orbit().
↪body().reference_frame());
    std::cout << std::fixed <<
↪std::setprecision(1);
    std::cout << std::get<0>(position) << ", "
                << std::get<1>(position) << ", "
                << std::get<2>(position) << std::endl;
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Triplet;

import java.io.IOException;

public class VesselPosition {
    public static void main(String[] args) throws
↪IOException, RPCException {
        Connection connection = Connection.
↪newInstance();
        SpaceCenter spaceCenter = SpaceCenter.
↪newInstance(connection);
        Vessel vessel = spaceCenter.
↪getActiveVessel();
    }
}

```

```

        Triplet<Double, Double, Double> position =
            vessel.position(vessel.getOrbit()).
            ↪getBody().getReferenceFrame());
        System.out.printf("(%.1f, %.1f, %.1f)\n",
                           position.getValue0(),
                           position.getValue1(),
                           position.getValue2());
        connection.close();
    }
}

```

```

local krpc = require 'krpc'
local conn = krpc.connect()
local vessel = conn.space_center.active_vessel
print(vessel:position(vessel.orbit.body.reference_
    ↪frame))

```

```

import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel
print('("%.1f, %.1f, %.1f)' % vessel.
    ↪position(vessel.orbit.body.reference_frame))

```

For a vessel sat on the launchpad, the magnitude of this position vector will be roughly 600,000 meters (equal to the radius of Kerbin). The position vector will also not change over time, because the vessel is sat on the surface of Kerbin and the reference frame also rotates with Kerbin.

## Vessel Orbital Reference Frame

Another example is the orbital reference frame for a vessel, obtained by calling *Vessel.orbital\_reference\_frame*. This is fixed to the vessel (the origin moves with the vessel) and is orientated so that the axes point in the orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel,
- the y-axis points in the prograde direction of the vessels orbit,
- the x-axis points in the anti-radial direction of the vessels orbit,
- the z-axis points in the normal direction of the vessels orbit,
- and the axes rotate to match any changes to the prograde/normal/radial directions, for example when the prograde direction changes as the vessel continues on its orbit.

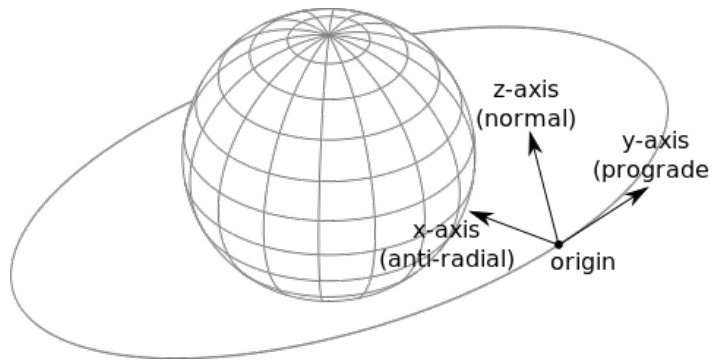


Fig. 2.2: The orbital reference frame for a vessel.

## Vessel Surface Reference Frame



Another example is `Vessel.reference_frame`. As with the previous example, it is fixed to the vessel (the origin moves with the vessel), however the orientation of the coordinate axes is different. They track the orientation of the vessel:

- The origin is at the center of mass of the vessel,
- the y-axis points in the same direction that the vessel is pointing,
- the x-axis points out of the right side of the vessel,
- the z-axis points downwards out of the bottom of the vessel,
- and the axes rotate with any changes to the direction of the vessel.

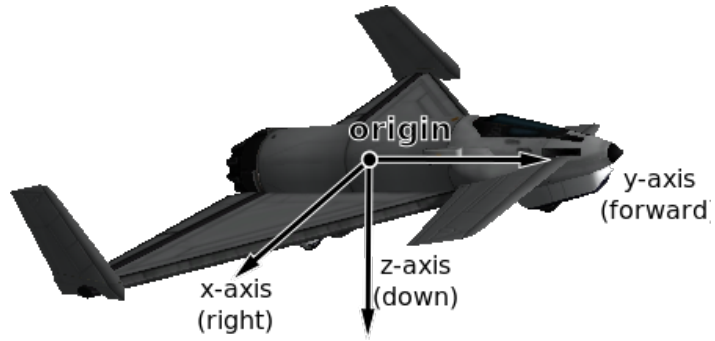


Fig. 2.3: The reference frame for an aircraft.

## Linear Velocity and Angular Velocity

Reference frames move and rotate relative to one another. For example, the reference frames discussed previously all have their origin position fixed to some object (such as a vessel or a planet). This means that they move and rotate to track the object, and so have a linear and angular velocity associated with them.

For example, the reference frame obtained by calling `CelestialBody.reference_frame` for Kerbin is fixed relative to Kerbin. This means the angular velocity of the reference frame is identical to Kerbin's angular velocity, and the linear velocity of the reference frame matches the current orbital velocity of Kerbin.

### 2.2.2 Available Reference Frames

kRPC provides the following reference frames:

C#

C++

Java

Lua

Python

- `Vessel.ReferenceFrame`
- `Vessel.OrbitalReferenceFrame`
- `Vessel.SurfaceReferenceFrame`
- `Vessel.SurfaceVelocityReferenceFrame`
- `CelestialBody.ReferenceFrame`
- `CelestialBody.NonRotatingReferenceFrame`
- `CelestialBody.OrbitalReferenceFrame`
- `Node.ReferenceFrame`

- *Node.OrbitalReferenceFrame*
- *Part.ReferenceFrame*
- *Part.CenterOfMassReferenceFrame*
- *DockingPort.ReferenceFrame*
- *Thruster.ThrustReferenceFrame*
- *Vessel::reference\_frame()*
- *Vessel::orbital\_reference\_frame()*
- *Vessel::surface\_reference\_frame()*
- *Vessel::surface\_velocity\_reference\_frame()*
- *CelestialBody::reference\_frame()*
- *CelestialBody::non\_rotating\_reference\_frame()*
- *CelestialBody::orbital\_reference\_frame()*
- *Node::reference\_frame()*
- *Node::orbital\_reference\_frame()*
- *Part::reference\_frame()*
- *Part::center\_of\_mass\_reference\_frame()*
- *DockingPort::reference\_frame()*
- *Thruster::thrust\_reference\_frame()*
- *Vessel.getReferenceFrame*
- *Vessel.getOrbitalReferenceFrame*
- *Vessel.getSurfaceReferenceFrame*
- *Vessel.getSurfaceVelocityReferenceFrame*
- *CelestialBody.getReferenceFrame*
- *CelestialBody.getNonRotatingReferenceFrame*
- *CelestialBody.getOrbitalReferenceFrame*
- *Node.getReferenceFrame*
- *Node.getOrbitalReferenceFrame*
- *Part.getReferenceFrame*
- *Part.getCenterOfMassReferenceFrame*
- *DockingPort.getReferenceFrame*
- *Thruster.getThrustReferenceFrame*
- *Vessel.reference\_frame*
- *Vessel.orbital\_reference\_frame*
- *Vessel.surface\_reference\_frame*
- *Vessel.surface\_velocity\_reference\_frame*
- *CelestialBody.reference\_frame*

- *CelestialBody.non\_rotating\_reference\_frame*
- *CelestialBody.orbital\_reference\_frame*
- *Node.reference\_frame*
- *Node.orbital\_reference\_frame*
- *Part.reference\_frame*
- *Part.center\_of\_mass\_reference\_frame*
- *DockingPort.reference\_frame*
- *Thruster.thrust\_reference\_frame*
- *Vessel.reference\_frame*
- *Vessel.orbital\_reference\_frame*
- *Vessel.surface\_reference\_frame*
- *Vessel.surface\_velocity\_reference\_frame*
- *CelestialBody.reference\_frame*
- *CelestialBody.non\_rotating\_reference\_frame*
- *CelestialBody.orbital\_reference\_frame*
- *Node.reference\_frame*
- *Node.orbital\_reference\_frame*
- *Part.reference\_frame*
- *Part.center\_of\_mass\_reference\_frame*
- *DockingPort.reference\_frame*
- *Thruster.thrust\_reference\_frame*

Relative and hybrid reference frames can also be constructed from the above.

### 2.2.3 Custom Reference Frames

Custom reference frames can be constructed from the built in frames listed above. They come in two varieties: ‘relative’ and ‘hybrid’.

A relative reference frame is constructed from a parent reference frame, a fixed position offset and a fixed rotation offset. For example, this could be used to construct a reference frame whose origin is 10m below the vessel as follows, by applying a position offset of 10 along the z-axis to *Vessel.reference\_frame*. Relative reference frames can be constructed by calling *ReferenceFrame.create\_relative()*.

A hybrid reference frame inherits its components (position, rotation, velocity and angular velocity) from the components of other reference frames. Note that these components need not be fixed. For example, you could construct a reference frame whose position is the center of mass of the vessel (inherited from *Vessel.reference\_frame*) and whose rotation is that of the planet being orbited (inherited from *CelestialBody.reference\_frame*). Relative reference frames can be constructed by calling *ReferenceFrame.create\_hybrid()*.

The parent reference frame(s) of a custom reference frame can also be other custom reference frames. For example, you could combine the two example frames from above: construct a hybrid reference frame, centered on the vessel and rotated with the planet being orbited, and then create a relative reference that offsets the position of this 10m along the z-axis. The resulting frame will have its origin 10m below the vessel, and will be rotated with the planet being orbited.

## 2.2.4 Converting Between Reference Frames

kRPC provides utility methods to convert positions, directions, rotations and velocities between the different reference frames:

C#

C++

Java

Lua

Python

- *SpaceCenter.TransformPosition*
- *SpaceCenter.TransformDirection*
- *SpaceCenter.TransformRotation*
- *SpaceCenter.TransformVelocity*
- *SpaceCenter::transform\_position()*
- *SpaceCenter::transform\_direction()*
- *SpaceCenter::transform\_rotation()*
- *SpaceCenter::transform\_velocity()*
- *SpaceCenter.transformPosition*
- *SpaceCenter.transformDirection*
- *SpaceCenter.transformRotation*
- *SpaceCenter.transformVelocity*
- *SpaceCenter.transform\_position()*
- *SpaceCenter.transform\_direction()*
- *SpaceCenter.transform\_rotation()*
- *SpaceCenter.transform\_velocity()*
- *SpaceCenter.transform\_position()*
- *SpaceCenter.transform\_direction()*
- *SpaceCenter.transform\_rotation()*
- *SpaceCenter.transform\_velocity()*

## 2.2.5 Visual Debugging

References frames can be confusing, and choosing the correct one is a challenge in itself. To aid debugging, kRPCs drawing functionality can be used to visualize direction vectors in-game.

*Drawing.add\_direction()* will draw a direction vector, starting from the origin of the given reference frame. For example, the following code draws the direction of the current vessels velocity relative to the surface of the body it is orbiting:

C#

C++

Java

Lua

Python

```
using System;
using KRPC.Client;
using KRPC.Client.Services.Drawing;
using KRPC.Client.Services.SpaceCenter;

class VisualDebugging
{
    public static void Main ()
    {
        var conn = new Connection ("Visual Debugging");
        var vessel = conn.SpaceCenter ().ActiveVessel;

        var refFrame = vessel.SurfaceVelocityReferenceFrame;
        conn.Drawing ().AddDirection(
            new Tuple<double, double, double>(0, 1, 0), refFrame);
        while (true) {
        }
    }
}
```

```
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>
#include <krpc/services/ui.hpp>
#include <krpc/services/drawing.hpp>

int main() {
    krpc::Client conn = krpc::connect("Visual Debugging");
    krpc::services::SpaceCenter space_center(&conn);
    krpc::services::Drawing drawing(&conn);
    auto vessel = space_center.active_vessel();

    auto ref_frame = vessel.surface_velocity_reference_frame();
    drawing.add_direction(std::make_tuple(0, 1, 0), ref_frame);
    while (true) {
    }
}
```

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.ReferenceFrame;
import krpc.client.services.Drawing;

import org.javatuples.Triplet;

import java.io.IOException;

public class VisualDebugging {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("Visual Debugging");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Drawing drawing = Drawing.newInstance(connection);
        SpaceCenter.Vessel vessel = spaceCenter.getActiveVessel();
    }
}
```

```
ReferenceFrame refFrame = vessel.getSurfaceVelocityReferenceFrame();
drawing.addDirection(
    new Triplet<Double, Double, Double>(0.0, 1.0, 0.0), refFrame, 10, true);
while (true) {
}
}
```

```
local krpc = require 'krpc'
local conn = krpc.connect('Visual Debugging')
local vessel = conn.space_center.active_vessel

local ref_frame = vessel.surface_velocity_reference_frame
conn.drawing.add_direction(List{0, 1, 0}, ref_frame)
while true do
end
```

```
import krpc
conn = krpc.connect(name='Visual Debugging')
vessel = conn.space_center.active_vessel

ref_frame = vessel.surface_velocity_reference_frame
conn.drawing.add_direction((0, 1, 0), ref_frame)
while True:
    pass
```

---

**Note:** The client must remain connected for the line to continue to be drawn, hence the infinite loop at the end of this example.

---

## 2.2.6 Examples

The following examples demonstrate various uses of reference frames.

### Navball directions

This example demonstrates how to make the vessel point in various directions on the navball:

C#

C++

Java

Lua

Python

```
using System;
using System.Collections.Generic;
using System.Net;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;
```

```

class NavballDirections
{
    public static void Main ()
    {
        using (var conn = new Connection ("Navball directions")) {
            var vessel = conn.SpaceCenter ().ActiveVessel;
            var ap = vessel.AutoPilot;
            ap.ReferenceFrame = vessel.SurfaceReferenceFrame;
            ap.Engage();

            // Point the vessel north on the navball, with a pitch of 0 degrees
            ap.TargetDirection = Tuple.Create (0.0, 1.0, 0.0);
            ap.Wait();

            // Point the vessel vertically upwards on the navball
            ap.TargetDirection = Tuple.Create (1.0, 0.0, 0.0);
            ap.Wait();

            // Point the vessel west (heading of 270 degrees), with a pitch of 0_
↪degrees
            ap.TargetDirection = Tuple.Create (0.0, 0.0, -1.0);
            ap.Wait();

            ap.Disengage();
        }
    }
}

```

```

#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client conn = krpc::connect("Navball directions");
    krpc::services::SpaceCenter space_center(&conn);
    auto vessel = space_center.active_vessel();
    auto ap = vessel.auto_pilot();
    ap.set_reference_frame(vessel.surface_reference_frame());
    ap.engage();

    // Point the vessel north on the navball, with a pitch of 0 degrees
    ap.set_target_direction(std::make_tuple(0, 1, 0));
    ap.wait();

    // Point the vessel vertically upwards on the navball
    ap.set_target_direction(std::make_tuple(1, 0, 0));
    ap.wait();

    // Point the vessel west (heading of 270 degrees), with a pitch of 0 degrees
    ap.set_target_direction(std::make_tuple(0, 0, -1));
    ap.wait();

    ap.disengage();
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;

```

```

import krpc.client.services.SpaceCenter.AutoPilot;
import krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Triplet;

import java.io.IOException;

public class NavballDirections {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("Navball directions");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        AutoPilot ap = vessel.getAutoPilot();
        ap.setReferenceFrame(vessel.getSurfaceReferenceFrame());
        ap.engage();

        // Point the vessel north on the navball, with a pitch of 0 degrees
        ap.setTargetDirection(new Triplet<Double,Double,Double> (0.0, 1.0, 0.0));
        ap.wait_();

        // Point the vessel vertically upwards on the navball
        ap.setTargetDirection(new Triplet<Double,Double,Double> (1.0, 0.0, 0.0));
        ap.wait_();

        // Point the vessel west (heading of 270 degrees), with a pitch of 0 degrees
        ap.setTargetDirection(new Triplet<Double,Double,Double> (0.0, 0.0, -1.0));
        ap.wait_();

        ap.disengage();
        connection.close();
    }
}

```

```

local krpc = require 'krpc'
local List = require 'pl.List'
local conn = krpc.connect('Navball directions')
local vessel = conn.space_center.active_vessel
local ap = vessel.auto_pilot
ap.reference_frame = vessel.surface_reference_frame
ap:engage()

-- Point the vessel north on the navball, with a pitch of 0 degrees
ap.target_direction = List{0, 1, 0}
ap:wait()

-- Point the vessel vertically upwards on the navball
ap.target_direction = List{1, 0, 0}
ap:wait()

-- Point the vessel west (heading of 270 degrees), with a pitch of 0 degrees
ap.target_direction = List{0, 0, -1}
ap:wait()

ap:disengage()

```

```

import krpc
conn = krpc.connect(name='Navball directions')

```



```

vessel = conn.space_center.active_vessel
ap = vessel.auto_pilot
ap.reference_frame = vessel.surface_reference_frame
ap.engage()

# Point the vessel north on the navball, with a pitch of 0 degrees
ap.target_direction = (0, 1, 0)
ap.wait()

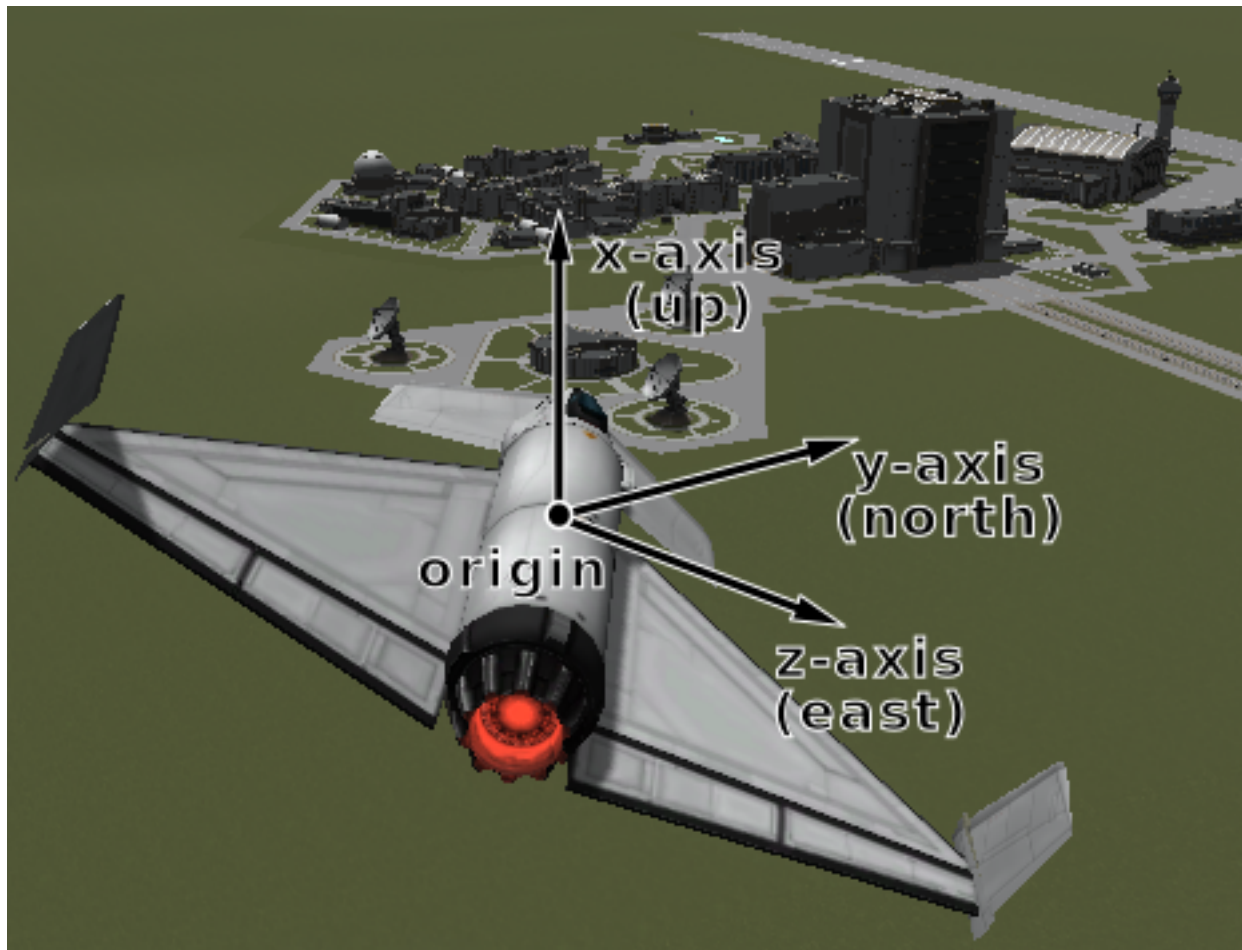
# Point the vessel vertically upwards on the navball
ap.target_direction = (1, 0, 0)
ap.wait()

# Point the vessel west (heading of 270 degrees), with a pitch of 0 degrees
ap.target_direction = (0, 0, -1)
ap.wait()

ap.disengage()

```

The code uses the vessel's surface reference frame (`Vessel.surface_reference_frame`), pictured below:



The first part instructs the auto-pilot to point in direction  $(0, 1, 0)$  (i.e. along the y-axis) in the vessel's surface reference frame. The y-axis of the reference frame points in the north direction, as required.

The second part instructs the auto-pilot to point in direction  $(1, 0, 0)$  (along the x-axis) in the vessel's surface

reference frame. This x-axis of the reference frame points upwards (away from the planet) as required.

Finally, the code instructs the auto-pilot to point in direction  $(0, 0, -1)$  (along the negative z axis). The z-axis of the reference frame points east, so the requested direction points west – as required.

## Orbital directions

This example demonstrates how to make the vessel point in the various orbital directions, as seen on the navball when it is in ‘orbit’ mode. It uses *Vessel.orbital\_reference\_frame*.

C#

C++

Java

Lua

Python

```
using System;
using System.Collections.Generic;
using System.Net;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class NavballDirections
{
    public static void Main ()
    {
        using (var conn = new Connection ("Orbital directions")) {
            var vessel = conn.SpaceCenter ().ActiveVessel;
            var ap = vessel.AutoPilot;
            ap.ReferenceFrame = vessel.OrbitalReferenceFrame;
            ap.Engage();

            // Point the vessel in the prograde direction
            ap.TargetDirection = Tuple.Create (0.0, 1.0, 0.0);
            ap.Wait();

            // Point the vessel in the orbit normal direction
            ap.TargetDirection = Tuple.Create (0.0, 0.0, 1.0);
            ap.Wait();

            // Point the vessel in the orbit radial direction
            ap.TargetDirection = Tuple.Create (-1.0, 0.0, 0.0);
            ap.Wait();

            ap.Disengage();
        }
    }
}
```

```
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client conn = krpc::connect("Orbital directions");
    krpc::services::SpaceCenter space_center(&conn);
}
```

```

auto vessel = space_center.active_vessel();
auto ap = vessel.auto_pilot();
ap.set_reference_frame(vessel.orbital_reference_frame());
ap.engage();

// Point the vessel in the prograde direction
ap.set_target_direction(std::make_tuple(0, 1, 0));
ap.wait();

// Point the vessel in the orbit normal direction
ap.set_target_direction(std::make_tuple(0, 0, 1));
ap.wait();

// Point the vessel in the orbit radial direction
ap.set_target_direction(std::make_tuple(-1, 0, 0));
ap.wait();

ap.disengage();
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.AutoPilot;
import krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Triplet;

import java.io.IOException;

public class OrbitalDirections {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("Orbital directions");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        AutoPilot ap = vessel.getAutoPilot();
        ap.setReferenceFrame(vessel.getOrbitalReferenceFrame());
        ap.engage();

        // Point the vessel in the prograde direction
        ap.setTargetDirection(new Triplet<Double,Double,Double> (0.0, 1.0, 0.0));
        ap.wait_();

        // Point the vessel in the orbit normal direction
        ap.setTargetDirection(new Triplet<Double,Double,Double> (0.0, 0.0, 1.0));
        ap.wait_();

        // Point the vessel in the orbit radial direction
        ap.setTargetDirection(new Triplet<Double,Double,Double> (-1.0, 0.0, 0.0));
        ap.wait_();

        ap.disengage();
        connection.close();
    }
}

```

```
local krpc = require 'krpc'
local List = require 'pl.List'
local conn = krpc.connect('Orbital directions')
local vessel = conn.space_center.active_vessel
local ap = vessel.auto_pilot
ap.reference_frame = vessel.orbital_reference_frame
ap:engage()

-- Point the vessel in the prograde direction
ap.target_direction = List{0, 1, 0}
ap:wait()

-- Point the vessel in the orbit normal direction
ap.target_direction = List{0, 0, 1}
ap:wait()

-- Point the vessel in the orbit radial direction
ap.target_direction = List{-1, 0, 0}
ap:wait()

ap:disengage()
```

```
import krpc
conn = krpc.connect(name='Orbital directions')
vessel = conn.space_center.active_vessel
ap = vessel.auto_pilot
ap.reference_frame = vessel.orbital_reference_frame
ap.engage()

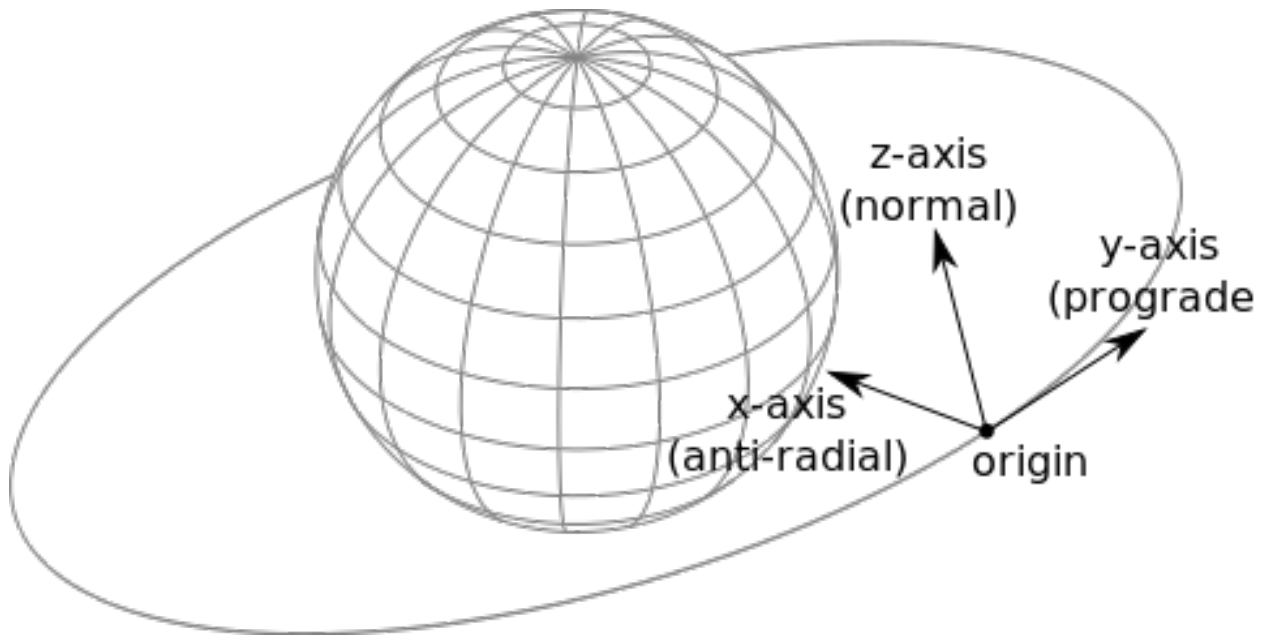
# Point the vessel in the prograde direction
ap.target_direction = (0, 1, 0)
ap.wait()

# Point the vessel in the orbit normal direction
ap.target_direction = (0, 0, 1)
ap.wait()

# Point the vessel in the orbit radial direction
ap.target_direction = (-1, 0, 0)
ap.wait()

ap.disengage()
```

This code uses the vessel's orbital reference frame, pictured below:



### Surface ‘prograde’

This example demonstrates how to point the vessel in the ‘prograde’ direction on the navball, when in ‘surface’ mode. This is the direction of the vessels velocity relative to the surface:

C#

C++

Java

Lua

Python

```
using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class SurfacePrograde
{
    public static void Main ()
    {
        using (var connection = new Connection (name : "Surface prograde")) {
            var vessel = connection.SpaceCenter ().ActiveVessel;
            var ap = vessel.AutoPilot;

            ap.ReferenceFrame = vessel.SurfaceVelocityReferenceFrame;
            ap.TargetDirection = new Tuple<double, double, double> (0, 1, 0);
            ap.Engage ();
            ap.Wait ();
            ap.Disengage ();
        }
    }
}
```

```
#include <iostream>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client conn = krpc::connect("Surface prograde");
    krpc::services::SpaceCenter spaceCenter(&conn);
    auto vessel = spaceCenter.active_vessel();
    auto ap = vessel.auto_pilot();

    ap.set_reference_frame(vessel.surface_velocity_reference_frame());
    ap.set_target_direction(std::make_tuple(0.0, 1.0, 0.0));
    ap.engage();
    ap.wait();
    ap.disengage();
}
```

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.AutoPilot;
import krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Triplet;

import java.io.IOException;

public class SurfacePrograde {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("Surface prograde");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        AutoPilot ap = vessel.getAutoPilot();

        ap.setReferenceFrame(vessel.getSurfaceVelocityReferenceFrame());
        ap.setTargetDirection(new Triplet<Double,Double,Double>(0.0, 1.0, 0.0));
        ap.engage();
        ap.wait_();
        ap.disengage();
        connection.close();
    }
}
```

```
local krpc = require 'krpc'
local List = require 'pl.List'
local conn = krpc.connect('Surface prograde')
local vessel = conn.space_center.active_vessel
local ap = vessel.auto_pilot

ap.reference_frame = vessel.surface_velocity_reference_frame
ap.target_direction = List{0, 1, 0}
ap:engage()
ap:wait()
ap:disengage()
```

```
import krpc
conn = krpc.connect(name='Surface prograde')
```

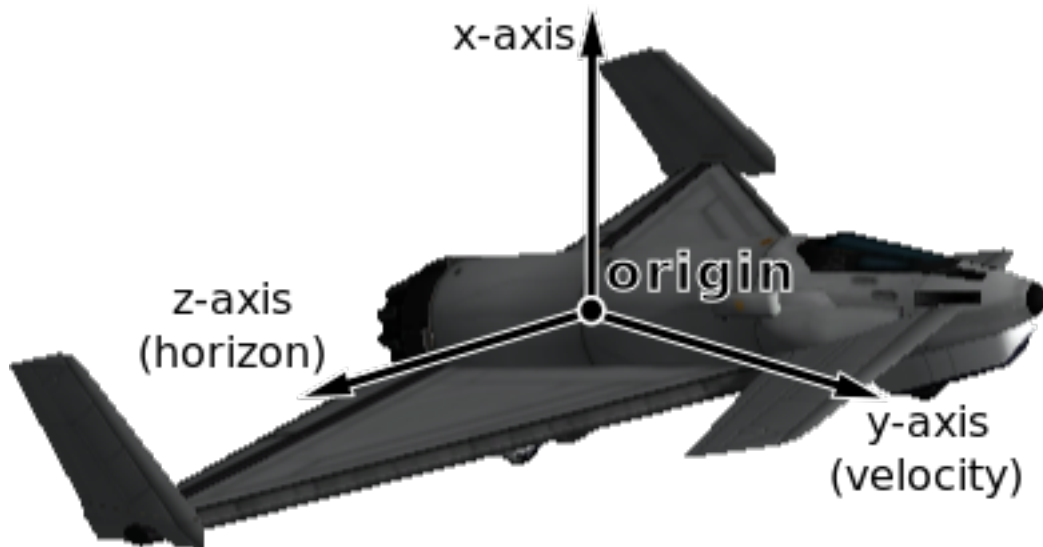
```

vessel = conn.space_center.active_vessel
ap = vessel.auto_pilot

ap.reference_frame = vessel.surface_velocity_reference_frame
ap.target_direction = (0, 1, 0)
ap.engage()
ap.wait()
ap.disengage()

```

This code uses the `Vessel.surface_velocity_reference_frame`, pictured below:



## Vessel Speed

This example demonstrates how to get the orbital and surface speeds of the vessel, equivalent to the values displayed by the navball.

To compute the orbital speed of a vessel, you need to get the velocity relative to the planet's *non-rotating* reference frame (`CelestialBody.non_rotating_reference_frame`). This reference frame is fixed relative to the body, but does not rotate.

For the surface speed, the planet's reference frame (`CelestialBody.reference_frame`) is required, as this reference frame rotates with the body.

C#

C++

Java

Lua

Python

```

using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class VesselSpeed
{
    public static void Main ()

```

```

{
    var connection = new Connection (name : "Vessel speed");
    var vessel = connection.SpaceCenter ().ActiveVessel;
    var obtFrame = vessel.Orbit.Body.NonRotatingReferenceFrame;
    var srfFrame = vessel.Orbit.Body.ReferenceFrame;
    while (true) {
        var obtSpeed = vessel.Flight (obtFrame).Speed;
        var srfSpeed = vessel.Flight (srfFrame).Speed;
        Console.WriteLine (
            "Orbital speed = {0:F1} m/s, Surface speed = {1:F1} m/s",
            obtSpeed, srfSpeed);
        System.Threading.Thread.Sleep (1000);
    }
}

```

```

#include <iostream>
#include <iomanip>
#include <thread>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client conn = krpc::connect("Vessel speed");
    krpc::services::SpaceCenter spaceCenter(&conn);
    auto vessel = spaceCenter.active_vessel();
    auto obt_frame = vessel.orbit().body().non_rotating_reference_frame();
    auto srf_frame = vessel.orbit().body().reference_frame();

    while (true) {
        auto obt_speed = vessel.flight(obt_frame).speed();
        auto srf_speed = vessel.flight(srf_frame).speed();
        std::cout << std::fixed << std::setprecision(1)
            << "Orbital speed = " << obt_speed << " m/s, "
            << "Surface speed = " << srf_speed << " m/s" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.ReferenceFrame;
import krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Triplet;

import java.io.IOException;

public class VesselSpeed {
    public static void main(String[] args)
        throws IOException, RPCException, InterruptedException {
        Connection connection = Connection.newInstance("Vessel speed");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        ReferenceFrame obtFrame = vessel.getOrbit().getBody().
        ↪getNonRotatingReferenceFrame();
    }
}

```



```

ReferenceFrame srfFrame = vessel.getOrbit().getBody().getReferenceFrame();
while (true) {
    double obtSpeed = vessel.flight(obtFrame).getSpeed();
    double srfSpeed = vessel.flight(srfFrame).getSpeed();
    System.out.printf(
        "Orbital speed = %.1f m/s, Surface speed = %.1f m/s\n",
        obtSpeed, srfSpeed);
    Thread.sleep(1000);
}
}
}

```

```

local krpc = require 'krpc'
local platform = require 'krpc.platform'
local conn = krpc.connect('Vessel speed')
local vessel = conn.space_center.active_vessel
local obt_frame = vessel.orbit.body.non_rotating_reference_frame
local srf_frame = vessel.orbit.body.reference_frame

while true do
    obt_speed = vessel:flight(obt_frame).speed
    srf_speed = vessel:flight(srf_frame).speed
    print(string.format(
        'Orbital speed = %.1f m/s, Surface speed = %.1f m/s',
        obt_speed, srf_speed))
    platform.sleep(1)
end

```

```

import time
import krpc

conn = krpc.connect(name='Vessel speed')
vessel = conn.space_center.active_vessel
obt_frame = vessel.orbit.body.non_rotating_reference_frame
srf_frame = vessel.orbit.body.reference_frame

while True:
    obt_speed = vessel.flight(obt_frame).speed
    srf_speed = vessel.flight(srf_frame).speed
    print('Orbital speed = %.1f m/s, Surface speed = %.1f m/s' %
        (obt_speed, srf_speed))
    time.sleep(1)

```

## Vessel Velocity

This example demonstrates how to get the velocity of the vessel (as a vector), relative to the surface of the body being orbited.

To do this, a hybrid reference frame is required. This is because we want a reference frame that is centered on the vessel, but whose linear velocity is fixed relative to the ground.

We therefore create a hybrid reference frame with its rotation set to the vessel's surface reference frame (*Vessel.surface\_reference\_frame*), and all other properties (including position and velocity) set to the body's reference frame (*CelestialBody.reference\_frame*) – which rotates with the body.

C#

C++

Java

Lua

Python

```
using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class VesselVelocity
{
    public static void Main ()
    {
        var connection = new Connection (name : "Vessel velocity");
        var vessel = connection.SpaceCenter ().ActiveVessel;
        var refFrame = ReferenceFrame.CreateHybrid(
            connection,
            vessel.Orbit.Body.ReferenceFrame,
            vessel.SurfaceReferenceFrame);

        while (true) {
            var velocity = vessel.Flight (refFrame).Velocity;
            Console.WriteLine ("Surface velocity = ({0:F1}, {1:F1}, {2:F1})",
                                velocity.Item1, velocity.Item2, velocity.Item3);
            System.Threading.Thread.Sleep (1000);
        }
    }
}
```

```
#include <iostream>
#include <iomanip>
#include <thread>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client connection = krpc::connect("Vessel velocity");
    krpc::services::SpaceCenter spaceCenter(&connection);
    auto vessel = spaceCenter.active_vessel();
    auto ref_frame = krpc::services::SpaceCenter::ReferenceFrame::create_hybrid(
        connection,
        vessel.orbit().body().reference_frame(),
        vessel.surface_reference_frame()
    );

    while (true) {
        auto velocity = vessel.flight(ref_frame).velocity();
        std::cout
            << std::fixed << std::setprecision(1)
            << "Surface velocity = ("
            << std::get<0>(velocity) << ", "
            << std::get<1>(velocity) << ", "
            << std::get<2>(velocity)
            << ")" << std::endl;
        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}
```

```
}
```

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.ReferenceFrame;
import krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Triplet;

import java.io.IOException;

public class VesselVelocity {
    public static void main(String[] args)
        throws IOException, RPCException, InterruptedException {
        Connection connection = Connection.newInstance("Vessel velocity");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        ReferenceFrame refFrame = ReferenceFrame.createHybrid(
            connection,
            vessel.getOrbit().getBody().getReferenceFrame(),
            vessel.getSurfaceReferenceFrame(),
            vessel.getOrbit().getBody().getReferenceFrame(),
            vessel.getOrbit().getBody().getReferenceFrame());
        while (true) {
            Triplet<Double,Double,Double> velocity =
                vessel.flight(refFrame).getVelocity();
            System.out.printf("Surface velocity = (%.1f, %.1f, %.1f)\n",
                velocity.getValue0(),
                velocity.getValue1(),
                velocity.getValue2());
            Thread.sleep(1000);
        }
    }
}
```

```
local krpc = require 'krpc'
local platform = require 'krpc.platform'
local conn = krpc.connect('Orbital speed')
local vessel = conn.space_center.active_vessel
local ref_frame = conn.SpaceCenter.ReferenceFrame.CreateHybrid(
    vessel.orbit.body.reference_frame,
    vessel.surface_reference_frame)

while true do
    velocity = vessel:flight(ref_frame).velocity
    print(string.format('Surface velocity = (%.1f, %.1f, %.1f)',
        velocity[1], velocity[2], velocity[3]))
    platform.sleep(1)
end
```

```
import time
import krpc

conn = krpc.connect(name='Orbital speed')
vessel = conn.space_center.active_vessel
ref_frame = conn.space_center.ReferenceFrame.create_hybrid(
```

```
position=vessel.orbit.body.reference_frame,
rotation=vessel.surface_reference_frame)

while True:
    velocity = vessel.flight(ref_frame).velocity
    print('Surface velocity = (%.1f, %.1f, %.1f)' % velocity)
    time.sleep(1)
```

## Angle of attack

This example computes the angle between the direction the vessel is pointing in, and the direction that the vessel is moving in (relative to the surface):

C#

C++

Java

Lua

Python

```
using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class AngleOfAttack
{
    public static void Main ()
    {
        var conn = new Connection ("Angle of attack");
        var vessel = conn.SpaceCenter ().ActiveVessel;

        while (true) {
            var d = vessel.Direction (vessel.Orbit.Body.ReferenceFrame);
            var v = vessel.Velocity (vessel.Orbit.Body.ReferenceFrame);

            // Compute the dot product of d and v
            var dotProd = d.Item1 * v.Item1 + d.Item2 * v.Item2 + d.Item3 * v.Item3;

            // Compute the magnitude of v
            var vMag = Math.Sqrt (
                v.Item1 * v.Item1 + v.Item2 * v.Item2 + v.Item3 * v.Item3);
            // Note: don't need to magnitude of d as it is a unit vector

            // Compute the angle between the vectors
            double angle = 0;
            if (dotProd > 0)
                angle = Math.Abs (Math.Acos (dotProd / vMag) * (180.0 / Math.PI));

            Console.WriteLine (
                "Angle of attack = " + Math.Round (angle, 2) + " degrees");

            System.Threading.Thread.Sleep (1000);
        }
    }
}
```

```

#include <iostream>
#include <iomanip>
#include <cmath>
#include <chrono>
#include <thread>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

static const double pi = 3.1415926535897;

int main() {
    krpc::Client conn = krpc::connect("Angle of attack");
    krpc::services::SpaceCenter space_center(&conn);
    auto vessel = space_center.active_vessel();

    while (true) {
        auto d = vessel.direction(vessel.orbit().body().reference_frame());
        auto v = vessel.velocity(vessel.orbit().body().reference_frame());

        // Compute the dot product of d and v
        double dotProd =
            std::get<0>(d)*std::get<0>(v) +
            std::get<1>(d)*std::get<1>(v) +
            std::get<2>(d)*std::get<2>(v);

        // Compute the magnitude of v
        double vMag = sqrt(
            std::get<0>(v)*std::get<0>(v) +
            std::get<1>(v)*std::get<1>(v) +
            std::get<2>(v)*std::get<2>(v));
        // Note: don't need to magnitude of d as it is a unit vector

        // Compute the angle between the vectors
        double angle = 0;
        if (dotProd > 0)
            angle = fabs(acos(dotProd / vMag) * (180.0 / pi));

        std::cout << "Angle of attack = "
                  << std::fixed << std::setprecision(1)
                  << angle << " degrees" << std::endl;

        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;

import org.javatuples.Triplet;

import java.io.IOException;
import java.lang.Math;

public class AngleOfAttack {
    public static void main(String[] args)
        throws IOException, RPCException, InterruptedException {
        Connection connection = Connection.newInstance("Angle of attack");
    }
}

```

```

SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
SpaceCenter.Vessel vessel = spaceCenter.getActiveVessel();

while (true) {
    Triplet<Double,Double,Double> d =
        vessel.direction(vessel.getOrbit().getBody().getReferenceFrame());
    Triplet<Double,Double,Double> v =
        vessel.velocity(vessel.getOrbit().getBody().getReferenceFrame());

    // Compute the dot product of d and v
    double dotProd =
        d.getValue0() * v.getValue0()
        + d.getValue1() * v.getValue1()
        + d.getValue2() * v.getValue2();

    // Compute the magnitude of v
    double vMag = Math.sqrt(
        v.getValue0() * v.getValue0()
        + v.getValue1() * v.getValue1()
        + v.getValue2() * v.getValue2()
    );
    // Note: don't need to magnitude of d as it is a unit vector

    // Compute the angle between the vectors
    double angle = 0;
    if (dotProd > 0) {
        angle = Math.abs(Math.acos(dotProd / vMag) * (180.0 / Math.PI));
    }

    System.out.printf("Angle of attack = %.1f degrees\n", angle);

    Thread.sleep(1000);
}
}

```

```

local krpc = require 'krpc'
local platform = require 'krpc.platform'
local math = require 'math'
local conn = krpc.connect('Angle of attack')
local vessel = conn.space_center.active_vessel

while true do

    d = vessel:direction(vessel.orbit.body.reference_frame)
    v = vessel:velocity(vessel.orbit.body.reference_frame)

    -- Compute the dot product of d and v
    dotprod = d[1]*v[1] + d[2]*v[2] + d[3]*v[3]

    -- Compute the magnitude of v
    vmag = math.sqrt(v[1]*v[1] + v[2]*v[2] + v[3]*v[3])
    -- Note: don't need to magnitude of d as it is a unit vector

    -- Compute the angle between the vectors
    angle = 0
    if dotprod > 0 then
        angle = math.abs(math.acos (dotprod / vmag) * (180. / math.pi))
    end
end

```

```

end

print(string.format('Angle of attack = %.1f', angle))

platform.sleep(1)

end

```

```

import math
import time
import krpc

conn = krpc.connect(name='Angle of attack')
vessel = conn.space_center.active_vessel

while True:

    d = vessel.direction(vessel.orbit.body.reference_frame)
    v = vessel.velocity(vessel.orbit.body.reference_frame)

    # Compute the dot product of d and v
    dotprod = d[0]*v[0] + d[1]*v[1] + d[2]*v[2]

    # Compute the magnitude of v
    vmag = math.sqrt(v[0]**2 + v[1]**2 + v[2]**2)
    # Note: don't need to magnitude of d as it is a unit vector

    # Compute the angle between the vectors
    angle = 0
    if dotprod > 0:
        angle = abs(math.acos(dotprod / vmag) * (180.0 / math.pi))

    print('Angle of attack = %.1f degrees' % angle)

    time.sleep(1)

```

Note that the orientation of the reference frame used to get the direction and velocity vectors does not matter, as the angle between two vectors is the same regardless of the orientation of the axes. However, if we were to use a reference frame that moves with the vessel, the velocity would return  $(0, 0, 0)$ . We therefore need a reference frame that is not fixed relative to the vessel. *CelestialBody.reference\_frame* fits these requirements.

## Landing Site

This example computes a reference frame that is located on the surface of a body at a given altitude, which could be used as the target for a landing auto pilot.

C#

C++

Java

Lua

Python

```

using System;
using KRPC.Client;

```

```

using KRPC.Client.Services.SpaceCenter;
using KRPC.Client.Services.Drawing;

class LandingSite
{
    public static void Main ()
    {
        var conn = new Connection ("Landing Site");
        var vessel = conn.SpaceCenter ().ActiveVessel;
        var body = vessel.Orbit.Body;

        // Define the landing site as the top of the VAB
        double landingLatitude = -(0.0+(5.0/60.0)+(48.38/60.0/60.0));
        double landingLongitude = -(74.0+(37.0/60.0)+(12.2/60.0/60.0));
        double landingAltitude = 111;

        // Determine landing site reference frame
        // (orientation: x=zenith, y=north, z=east)
        var landingPosition = body.SurfacePosition(
            landingLatitude, landingLongitude, body.ReferenceFrame);
        var qLong = Tuple.Create(
            0.0,
            Math.Sin(-landingLongitude * 0.5 * Math.PI / 180.0),
            0.0,
            Math.Cos(-landingLongitude * 0.5 * Math.PI / 180.0)
        );
        var qLat = Tuple.Create(
            0.0,
            0.0,
            Math.Sin(landingLatitude * 0.5 * Math.PI / 180.0),
            Math.Cos(landingLatitude * 0.5 * Math.PI / 180.0)
        );
        var landingReferenceFrame =
            ReferenceFrame.CreateRelative(
                conn,
                ReferenceFrame.CreateRelative(
                    conn,
                    ReferenceFrame.CreateRelative(
                        conn,
                        body.ReferenceFrame,
                        landingPosition,
                        qLong),
                    Tuple.Create(0.0, 0.0, 0.0),
                    qLat),
                Tuple.Create(landingAltitude, 0.0, 0.0));

        // Draw axes
        var zero = Tuple.Create(0.0, 0.0, 0.0);
        conn.Drawing().AddLine(
            zero, Tuple.Create(1.0, 0.0, 0.0), landingReferenceFrame);
        conn.Drawing().AddLine(
            zero, Tuple.Create(0.0, 1.0, 0.0), landingReferenceFrame);
        conn.Drawing().AddLine(
            zero, Tuple.Create(0.0, 0.0, 1.0), landingReferenceFrame);

        while (true)
            System.Threading.Thread.Sleep (1000);
    }
}

```



}

```

#include <iostream>
#include <iomanip>
#include <cmath>
#include <chrono>
#include <thread>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>
#include <krpc/services/ui.hpp>
#include <krpc/services/drawing.hpp>

static const double pi = 3.1415926535897;

int main() {
    krpc::Client conn = krpc::connect("Landing Site");
    krpc::services::SpaceCenter space_center(&conn);
    krpc::services::Drawing drawing(&conn);
    auto vessel = space_center.active_vessel();
    auto body = vessel.orbit().body();

    // Define the landing site as the top of the VAB
    double landing_latitude = -(0.0+(5.0/60.0)+(48.38/60.0/60.0));
    double landing_longitude = -(74.0+(37.0/60.0)+(12.2/60.0/60.0));
    double landing_altitude = 111;

    // Determine landing site reference frame
    // (orientation: x=zenith, y=north, z=east)
    auto landing_position = body.surface_position(
        landing_latitude, landing_longitude, body.reference_frame());
    auto q_long = std::make_tuple(
        0.0,
        sin(-landing_longitude * 0.5 * pi / 180.0),
        0.0,
        cos(-landing_longitude * 0.5 * pi / 180.0)
    );
    auto q_lat = std::make_tuple(
        0.0,
        0.0,
        sin(landing_latitude * 0.5 * pi / 180.0),
        cos(landing_latitude * 0.5 * pi / 180.0)
    );
    auto landing_reference_frame =
        krpc::services::SpaceCenter::ReferenceFrame::create_relative(
            conn,
            krpc::services::SpaceCenter::ReferenceFrame::create_relative(
                conn,
                krpc::services::SpaceCenter::ReferenceFrame::create_relative(
                    conn,
                    body.reference_frame(),
                    landing_position,
                    q_long),
                std::make_tuple(0, 0, 0),
                q_lat),
            std::make_tuple(landing_altitude, 0, 0));

    // Draw axes
    drawing.add_line(

```

```

        std::make_tuple(0, 0, 0), std::make_tuple(1, 0, 0), landing_reference_frame);
drawing.add_line(
    std::make_tuple(0, 0, 0), std::make_tuple(0, 1, 0), landing_reference_frame);
drawing.add_line(
    std::make_tuple(0, 0, 0), std::make_tuple(0, 0, 1), landing_reference_frame);

while (true)
    std::this_thread::sleep_for(std::chrono::seconds(1));
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.Drawing;
import krpc.client.services.SpaceCenter;

import org.javatuples.Triplet;
import org.javatuples.Quartet;

import java.io.IOException;
import java.lang.Math;

public class LandingSite {
    public static void main(String[] args)
        throws IOException, RPCException, InterruptedException {
        Connection connection = Connection.newInstance("Landing Site");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Drawing drawing = Drawing.newInstance(connection);
        SpaceCenter.Vessel vessel = spaceCenter.getActiveVessel();
        SpaceCenter.CelestialBody body = vessel.getOrbit().getBody();

        // Define the landing site as the top of the VAB
        double landingLatitude = -(0.0+(5.0/60.0)+(48.38/60.0/60.0));
        double landingLongitude = -(74.0+(37.0/60.0)+(12.2/60.0/60.0));
        double landingAltitude = 111;

        // Determine landing site reference frame
        // (orientation: x=zenith, y=north, z=east)
        Triplet<Double, Double, Double> landingPosition = body.surfacePosition(
            landingLatitude, landingLongitude, body.getReferenceFrame());
        Quartet<Double, Double, Double, Double> qLong =
            new Quartet<Double, Double, Double, Double>(
                0.0,
                Math.sin(-landingLongitude * 0.5 * Math.PI / 180.0),
                0.0,
                Math.cos(-landingLongitude * 0.5 * Math.PI / 180.0));
        Quartet<Double, Double, Double, Double> qLat =
            new Quartet<Double, Double, Double, Double>(
                0.0,
                0.0,
                Math.sin(landingLatitude * 0.5 * Math.PI / 180.0),
                Math.cos(landingLatitude * 0.5 * Math.PI / 180.0));
        Quartet<Double, Double, Double, Double> qIdentity =
            new Quartet<Double, Double, Double, Double>(0.0, 0.0, 0.0, 1.0);
        Triplet<Double, Double, Double> zero =
            new Triplet<Double, Double, Double>(0.0, 0.0, 0.0);
        SpaceCenter.ReferenceFrame landingReferenceFrame =
            SpaceCenter.ReferenceFrame.createRelative(
                connection,

```

```

        SpaceCenter.ReferenceFrame.createRelative(
            connection,
            SpaceCenter.ReferenceFrame.createRelative(
                connection,
                body.getReferenceFrame(),
                landingPosition, qLong, zero, zero),
                zero, qLat, zero, zero),
        new Triplet<Double, Double, Double>(landingAltitude, 0.0, 0.0),
        qIdentity, zero, zero);

    // Draw axes
    drawing.addLine(
        zero, new Triplet<Double, Double, Double>(1.0, 0.0, 0.0),
        landingReferenceFrame, true);
    drawing.addLine(
        zero, new Triplet<Double, Double, Double>(0.0, 1.0, 0.0),
        landingReferenceFrame, true);
    drawing.addLine(
        zero, new Triplet<Double, Double, Double>(0.0, 0.0, 1.0),
        landingReferenceFrame, true);

    while (true)
        Thread.sleep(1000);
}

```

```

local krpc = require 'krpc'
local platform = require 'krpc.platform'
local List = require 'pl.List'
local math = require 'math'

local conn = krpc.connect('Landing Site')
local vessel = conn.space_center.active_vessel
local body = vessel.orbit.body
local ReferenceFrame = conn.space_center.ReferenceFrame

-- Define the landing site as the top of the VAB
local landing_latitude = -(0+(5.0/60)+(48.38/60/60))
local landing_longitude = -(74+(37.0/60)+(12.2/60/60))
local landing_altitude = 111

-- Determine landing site reference frame
-- (orientation: x=zenith, y=north, z=east)
local landing_position = body:surface_position(
    landing_latitude, landing_longitude, body.reference_frame)
local q_long = List{
    0,
    math.sin(-landing_longitude * 0.5 * math.pi / 180),
    0,
    math.cos(-landing_longitude * 0.5 * math.pi / 180)
}
local q_lat = List{
    0,
    0,
    math.sin(landing_latitude * 0.5 * math.pi / 180),
    math.cos(landing_latitude * 0.5 * math.pi / 180)
}
local landing_reference_frame =

```

```
ReferenceFrame.create_relative(
    ReferenceFrame.create_relative(
        ReferenceFrame.create_relative(
            body.reference_frame,
            landing_position,
            q_long),
        List{0, 0, 0},
        q_lat),
    List{landing_altitude, 0, 0})

-- Draw axes
conn.drawing.add_line(List{0, 0, 0}, List{1, 0, 0}, landing_reference_frame)
conn.drawing.add_line(List{0, 0, 0}, List{0, 1, 0}, landing_reference_frame)
conn.drawing.add_line(List{0, 0, 0}, List{0, 0, 1}, landing_reference_frame)

while true do
    platform.sleep(1)
end
```

```
import time
from math import sin, cos, pi
import krpc

conn = krpc.connect(name='Landing Site')
vessel = conn.space_center.active_vessel
body = vessel.orbit.body
create_relative = conn.space_center.ReferenceFrame.create_relative

# Define the landing site as the top of the VAB
landing_latitude = -(0+(5.0/60)+(48.38/60/60))
landing_longitude = -(74+(37.0/60)+(12.2/60/60))
landing_altitude = 111

# Determine landing site reference frame
# (orientation: x=zenith, y=north, z=east)
landing_position = body.surface_position(
    landing_latitude, landing_longitude, body.reference_frame)
q_long = (
    0,
    sin(-landing_longitude * 0.5 * pi / 180),
    0,
    cos(-landing_longitude * 0.5 * pi / 180)
)
q_lat = (
    0,
    0,
    sin(landing_latitude * 0.5 * pi / 180),
    cos(landing_latitude * 0.5 * pi / 180)
)
landing_reference_frame = \
    create_relative(
        create_relative(
            create_relative(
                body.reference_frame,
                landing_position,
                q_long),
            (0, 0, 0),
            q_lat),
```

```

        (landing_altitude, 0, 0))

# Draw axes
conn.drawing.add_line((0, 0, 0), (1, 0, 0), landing_reference_frame)
conn.drawing.add_line((0, 0, 0), (0, 1, 0), landing_reference_frame)
conn.drawing.add_line((0, 0, 0), (0, 0, 1), landing_reference_frame)

while True:
    time.sleep(1)

```

## 2.3 Launch into Orbit

This tutorial launches a two-stage rocket into a 150km circular orbit. The program assumes you are using this craft file.

The program is available in a variety of languages:

C#, C++, Java, Lua, Python

The following code connects to the server, gets the active vessel, sets up a bunch of streams to get flight telemetry then prepares the rocket for launch.

C#

C++

Java

Lua

Python

```

1  using System;
2  using System.Collections.Generic;
3  using System.Net;
4  using KRPC.Client;
5  using KRPC.Client.Services.SpaceCenter;
6
7  class LaunchIntoOrbit
8  {
9      public static void Main ()
10     {
11         var conn = new Connection ("Launch into orbit");
12         var vessel = conn.SpaceCenter ().ActiveVessel;
13
14         float turnStartAltitude = 250;
15         float turnEndAltitude = 45000;
16         float targetAltitude = 150000;
17
18         // Set up streams for telemetry
19         var ut = conn.AddStream (() => conn.SpaceCenter ().UT);
20         var flight = vessel.Flight ();
21         var altitude = conn.AddStream (() => flight.MeanAltitude);
22         var apoapsis = conn.AddStream (() => vessel.Orbit.ApoapsisAltitude);
23         var stage2Resources =
24             vessel.ResourcesInDecoupleStage (stage: 2, cumulative: false);
25         var srbFuel = conn.AddStream (() => stage2Resources.Amount ("SolidFuel"));
26

```

```

27 // Pre-launch setup
28 vessel.Control.SAS = false;
29 vessel.Control.RCS = false;
30 vessel.Control.Throttle = 1;
31
32 // Countdown...
33 Console.WriteLine ("3...");
34 System.Threading.Thread.Sleep (1000);
35 Console.WriteLine ("2...");
36 System.Threading.Thread.Sleep (1000);
37 Console.WriteLine ("1...");
38 System.Threading.Thread.Sleep (1000);
39 Console.WriteLine ("Launch!");

```

```

1  #include <iostream>
2  #include <chrono>
3  #include <thread>
4  #include <krpc.hpp>
5  #include <krpc/services/space_center.hpp>
6
7  int main() {
8      krpc::Client conn = krpc::connect("Launch into orbit");
9      krpc::services::SpaceCenter space_center(&conn);
10     auto vessel = space_center.active_vessel();
11
12     float turn_start_altitude = 250;
13     float turn_end_altitude = 45000;
14     float target_altitude = 150000;
15
16     // Set up streams for telemetry
17     auto ut = space_center.ut_stream();
18     auto altitude = vessel.flight().mean_altitude_stream();
19     auto apoapsis = vessel.orbit().apoapsis_altitude_stream();
20     auto stage_2_resources = vessel.resources_in_decouple_stage(2, false);
21     auto srb_fuel = stage_2_resources.amount_stream("SolidFuel");
22
23     // Pre-launch setup
24     vessel.control().set_sas(false);
25     vessel.control().set_rcs(false);
26     vessel.control().set_throttle(1);
27
28     // Countdown...
29     std::cout << "3..." << std::endl;
30     std::this_thread::sleep_for(std::chrono::seconds(1));
31     std::cout << "2..." << std::endl;
32     std::this_thread::sleep_for(std::chrono::seconds(1));
33     std::cout << "1..." << std::endl;
34     std::this_thread::sleep_for(std::chrono::seconds(1));
35     std::cout << "Launch!" << std::endl;

```

```

1  import krpc.client.Connection;
2  import krpc.client.RPCException;
3  import krpc.client.Stream;
4  import krpc.client.StreamException;
5  import krpc.client.services.SpaceCenter;
6  import krpc.client.services.SpaceCenter.Flight;
7  import krpc.client.services.SpaceCenter.Node;

```

```

8 import krpc.client.services.SpaceCenter.ReferenceFrame;
9 import krpc.client.services.SpaceCenter.Resources;
10
11 import org.javatuples.Triplet;
12
13 import java.io.IOException;
14 import java.lang.Math;
15
16 public class LaunchIntoOrbit {
17     public static void main(String[] args)
18         throws IOException, RPCException, InterruptedException, StreamException {
19         Connection connection = Connection.newInstance("Launch into orbit");
20         SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
21         SpaceCenter.Vessel vessel = spaceCenter.getActiveVessel();
22
23         float turnStartAltitude = 250;
24         float turnEndAltitude = 45000;
25         float targetAltitude = 150000;
26
27         // Set up streams for telemetry
28         spaceCenter.getUT();
29         Stream<Double> ut = connection.addStream(SpaceCenter.class, "getUT");
30         ReferenceFrame refFrame = vessel.getSurfaceReferenceFrame();
31         Flight flight = vessel.flight(refFrame);
32         Stream<Double> altitude = connection.addStream(flight, "getMeanAltitude");
33         Stream<Double> apoapsis =
34             connection.addStream(vessel.getOrbit(), "getApoapsisAltitude");
35         Resources stage2Resources = vessel.resourcesInDecoupleStage(2, false);
36         Stream<Float> srbFuel =
37             connection.addStream(stage2Resources, "amount", "SolidFuel");
38
39         // Pre-launch setup
40         vessel.getControl().setSAS(false);
41         vessel.getControl().setRCS(false);
42         vessel.getControl().setThrottle(1);
43
44         // Countdown...
45         System.out.println("3...");
46         Thread.sleep(1000);
47         System.out.println("2...");
48         Thread.sleep(1000);
49         System.out.println("1...");
50         Thread.sleep(1000);

```

```

1 local krpc = require 'krpc'
2 local platform = require 'krpc.platform'
3 local math = require 'math'
4 local List = require 'pl.List'
5
6 local turn_start_altitude = 250
7 local turn_end_altitude = 45000
8 local target_altitude = 150000
9
10 local conn = krpc.connect('Launch into orbit')
11 local vessel = conn.space_center.active_vessel
12
13 flight = vessel:flight()
14 stage_2_resources = vessel:resources_in_decouple_stage(2, False)

```

```
15
16  -- Pre-launch setup
17  vessel.control.sas = false
18  vessel.control.rcs = false
19  vessel.control.throttle = 1
20
21  -- Countdown...
22  print('3...')
23  platform.sleep(1)
24  print('2...')
25  platform.sleep(1)
26  print('1...')
27  platform.sleep(1)
28  print('Launch!')
```

```
1  import math
2  import time
3  import krpc
4
5  turn_start_altitude = 250
6  turn_end_altitude = 45000
7  target_altitude = 150000
8
9  conn = krpc.connect(name='Launch into orbit')
10 vessel = conn.space_center.active_vessel
11
12  # Set up streams for telemetry
13  ut = conn.add_stream(getattr, conn.space_center, 'ut')
14  altitude = conn.add_stream(getattr, vessel.flight(), 'mean_altitude')
15  apoapsis = conn.add_stream(getattr, vessel.orbit, 'apoapsis_altitude')
16  stage_2_resources = vessel.resources_in_decouple_stage(stage=2, cumulative=False)
17  srb_fuel = conn.add_stream(stage_2_resources.amount, 'SolidFuel')
18
19  # Pre-launch setup
20  vessel.control.sas = False
21  vessel.control.rcs = False
22  vessel.control.throttle = 1.0
23
24  # Countdown...
25  print('3...')
26  time.sleep(1)
27  print('2...')
28  time.sleep(1)
29  print('1...')
30  time.sleep(1)
31  print('Launch!')
```

The next part of the program launches the rocket. The main loop continuously updates the auto-pilot heading to gradually pitch the rocket towards the horizon. It also monitors the amount of solid fuel remaining in the boosters, separating them when they run dry. The loop exits when the rockets apoapsis is close to the target apoapsis.

C#

C++

Java

Lua



## Python

```

41 // Activate the first stage
42 vessel.Control.ActivateNextStage ();
43 vessel.AutoPilot.Engage ();
44 vessel.AutoPilot.TargetPitchAndHeading (90, 90);
45
46 // Main ascent loop
47 bool srbsSeparated = false;
48 double turnAngle = 0;
49 while (true) {
50
51     // Gravity turn
52     if (altitude.Get () > turnStartAltitude &&
53         altitude.Get () < turnEndAltitude) {
54         double frac = (altitude.Get () - turnStartAltitude)
55                     / (turnEndAltitude - turnStartAltitude);
56         double newTurnAngle = frac * 90.0;
57         if (Math.Abs (newTurnAngle - turnAngle) > 0.5) {
58             turnAngle = newTurnAngle;
59             vessel.AutoPilot.TargetPitchAndHeading (
60                 (float) (90 - turnAngle), 90);
61         }
62     }
63
64     // Separate SRBs when finished
65     if (!srbsSeparated) {
66         if (srbFuel.Get () < 0.1) {
67             vessel.Control.ActivateNextStage ();
68             srbsSeparated = true;
69             Console.WriteLine ("SRBs separated");
70         }
71     }
72
73     // Decrease throttle when approaching target apoapsis
74     if (apoapsis.Get () > targetAltitude * 0.9) {
75         Console.WriteLine ("Approaching target apoapsis");
76         break;
77     }
78 }

```

```

37 // Activate the first stage
38 vessel.control().activate_next_stage();
39 vessel.auto_pilot().engage();
40 vessel.auto_pilot().target_pitch_and_heading(90, 90);
41
42 // Main ascent loop
43 bool srbs_separated = false;
44 double turn_angle = 0;
45 while (true) {
46     // Gravity turn
47     if (altitude() > turn_start_altitude && altitude() < turn_end_altitude) {
48         double frac = (altitude() - turn_start_altitude)
49                     / (turn_end_altitude - turn_start_altitude);
50         double new_turn_angle = frac * 90.0;
51         if (fabs(new_turn_angle - turn_angle) > 0.5) {
52             turn_angle = new_turn_angle;
53             vessel.auto_pilot().target_pitch_and_heading(90.0 - turn_angle, 90.0);

```

```

54     }
55 }
56
57 // Separate SRBs when finished
58 if (!srbs_separated) {
59     if (srb_fuel() < 0.1) {
60         vessel.control().activate_next_stage();
61         srbs_separated = true;
62         std::cout << "SRBs separated" << std::endl;
63     }
64 }
65
66 // Decrease throttle when approaching target apoapsis
67 if (apoapsis() > target_altitude * 0.9) {
68     std::cout << "Approaching target apoapsis" << std::endl;
69     break;
70 }
71 }

```

```

52 // Activate the first stage
53 vessel.getControl().activateNextStage();
54 vessel.getAutoPilot().engage();
55 vessel.getAutoPilot().targetPitchAndHeading(90, 90);
56
57 // Main ascent loop
58 boolean srbsSeparated = false;
59 double turnAngle = 0;
60 while (true) {
61
62     // Gravity turn
63     if (altitude.get() > turnStartAltitude &&
64         altitude.get() < turnEndAltitude) {
65         double frac = (altitude.get() - turnStartAltitude)
66             / (turnEndAltitude - turnStartAltitude);
67         double newTurnAngle = frac * 90.0;
68         if (Math.abs(newTurnAngle - turnAngle) > 0.5) {
69             turnAngle = newTurnAngle;
70             vessel.getAutoPilot().targetPitchAndHeading(
71                 (float)(90 - turnAngle), 90);
72         }
73     }
74
75     // Separate SRBs when finished
76     if (!srbsSeparated) {
77         if (srbFuel.get() < 0.1) {
78             vessel.getControl().activateNextStage();
79             srbsSeparated = true;
80             System.out.println("SRBs separated");
81         }
82     }
83
84     // Decrease throttle when approaching target apoapsis
85     if (apoapsis.get() > targetAltitude * 0.9) {
86         System.out.println("Approaching target apoapsis");
87         break;
88     }

```

```

30 -- Activate the first stage
31 vessel.control:activate_next_stage()
32 vessel.auto_pilot:engage()
33 vessel.auto_pilot:target_pitch_and_heading(90, 90)
34
35 -- Main ascent loop
36 local srbs_separated = false
37 local turn_angle = 0
38 while true do
39
40     -- Gravity turn
41     if flight.mean_altitude > turn_start_altitude and flight.mean_altitude < turn_end_
↪altitude then
42         frac = (flight.mean_altitude - turn_start_altitude) / (turn_end_altitude -
↪turn_start_altitude)
43         new_turn_angle = frac * 90
44         if math.abs(new_turn_angle - turn_angle) > 0.5 then
45             turn_angle = new_turn_angle
46             vessel.auto_pilot:target_pitch_and_heading(90-turn_angle, 90)
47         end
48     end
49
50     -- Separate SRBs when finished
51     if not srbs_separated then
52         if stage_2_resources:amount('SolidFuel') < 0.1 then
53             vessel.control:activate_next_stage()
54             srbs_separated = true
55             print('SRBs separated')
56         end
57     end
58
59     -- Decrease throttle when approaching target apoapsis
60     if vessel.orbit.apoapsis_altitude > target_altitude*0.9 then
61         print('Approaching target apoapsis')
62         break
63     end
64 end

```

```

33 # Activate the first stage
34 vessel.control.activate_next_stage()
35 vessel.auto_pilot.engage()
36 vessel.auto_pilot.target_pitch_and_heading(90, 90)
37
38 # Main ascent loop
39 srbs_separated = False
40 turn_angle = 0
41 while True:
42
43     # Gravity turn
44     if altitude() > turn_start_altitude and altitude() < turn_end_altitude:
45         frac = ((altitude() - turn_start_altitude) /
46                 (turn_end_altitude - turn_start_altitude))
47         new_turn_angle = frac * 90
48         if abs(new_turn_angle - turn_angle) > 0.5:
49             turn_angle = new_turn_angle
50             vessel.auto_pilot.target_pitch_and_heading(90-turn_angle, 90)
51
52     # Separate SRBs when finished

```

```
53     if not srbs_separated:
54         if srb_fuel() < 0.1:
55             vessel.control.activate_next_stage()
56             srbs_separated = True
57             print('SRBs separated')
58
59     # Decrease throttle when approaching target apoapsis
60     if apoapsis() > target_altitude*0.9:
61         print('Approaching target apoapsis')
62         break
```

Next, the program fine tunes the apoapsis, using 10% thrust, then waits until the rocket has left Kerbin's atmosphere.

C#

C++

Java

Lua

Python

```
80     // Disable engines when target apoapsis is reached
81     vessel.Control.Throttle = 0.25f;
82     while (apoapsis.Get () < targetAltitude) {
83     }
84     Console.WriteLine ("Target apoapsis reached");
85     vessel.Control.Throttle = 0;
86
87     // Wait until out of atmosphere
88     Console.WriteLine ("Coasting out of atmosphere");
89     while (altitude.Get () < 70500) {
90     }
```

```
73     // Disable engines when target apoapsis is reached
74     vessel.control().set_throttle(0.25);
75     while (apoapsis() < target_altitude) {
76     }
77     std::cout << "Target apoapsis reached" << std::endl;
78     vessel.control().set_throttle(0);
79
80     // Wait until out of atmosphere
81     std::cout << "Coasting out of atmosphere" << std::endl;
82     while (altitude() < 70500) {
83     }
```

```
91     // Disable engines when target apoapsis is reached
92     vessel.getControl().setThrottle(0.25f);
93     while (apoapsis.get() < targetAltitude) {
94     }
95     System.out.println("Target apoapsis reached");
96     vessel.getControl().setThrottle(0);
97
98     // Wait until out of atmosphere
99     System.out.println("Coasting out of atmosphere");
100    while (altitude.get() < 70500) {
```

```

66 -- Disable engines when target apoapsis is reached
67 vessel.control.throttle = 0.25
68 while vessel.orbit.apoapsis_altitude < target_altitude do
69 end
70 print('Target apoapsis reached')
71 vessel.control.throttle = 0
72
73 -- Wait until out of atmosphere
74 print('Coasting out of atmosphere')
75 while flight.mean_altitude < 70500 do
76 end

```

```

64 # Disable engines when target apoapsis is reached
65 vessel.control.throttle = 0.25
66 while apoapsis() < target_altitude:
67     pass
68 print('Target apoapsis reached')
69 vessel.control.throttle = 0.0
70
71 # Wait until out of atmosphere
72 print('Coasting out of atmosphere')
73 while altitude() < 70500:
74     pass

```

It is now time to plan the circularization burn. First, we calculate the delta-v required to circularize the orbit using the [vis-viva equation](#). We then calculate the burn time needed to achieve this delta-v, using the [Tsiolkovsky rocket equation](#).

C#

C++

Java

Lua

Python

```

92 // Plan circularization burn (using vis-viva equation)
93 Console.WriteLine ("Planning circularization burn");
94 double mu = vessel.Orbit.Body.GravitationalParameter;
95 double r = vessel.Orbit.Apoapsis;
96 double a1 = vessel.Orbit.SemiMajorAxis;
97 double a2 = r;
98 double v1 = Math.Sqrt (mu * ((2.0 / r) - (1.0 / a1)));
99 double v2 = Math.Sqrt (mu * ((2.0 / r) - (1.0 / a2)));
100 double deltaV = v2 - v1;
101 var node = vessel.Control.AddNode (
102     ut.Get () + vessel.Orbit.TimeToApoapsis, prograde: (float)deltaV);
103
104 // Calculate burn time (using rocket equation)
105 double F = vessel.AvailableThrust;
106 double Isp = vessel.SpecificImpulse * 9.82;
107 double m0 = vessel.Mass;
108 double m1 = m0 / Math.Exp (deltaV / Isp);
109 double flowRate = F / Isp;
110 double burnTime = (m0 - m1) / flowRate;

```

```

85 // Plan circularization burn (using vis-viva equation)
86 std::cout << "Planning circularization burn" << std::endl;
87 double mu = vessel.orbit().body().gravitational_parameter();
88 double r = vessel.orbit().apoapsis();
89 double a1 = vessel.orbit().semi_major_axis();
90 double a2 = r;
91 double v1 = sqrt(mu * ((2.0 / r) - (1.0 / a1)));
92 double v2 = sqrt(mu * ((2.0 / r) - (1.0 / a2)));
93 double delta_v = v2 - v1;
94 auto node = vessel.control().add_node(
95     ut() + vessel.orbit().time_to_apoapsis(), delta_v);
96
97 // Calculate burn time (using rocket equation)
98 double F = vessel.available_thrust();
99 double Isp = vessel.specific_impulse() * 9.82;
100 double m0 = vessel.mass();
101 double m1 = m0 / exp(delta_v / Isp);
102 double flow_rate = F / Isp;
103 double burn_time = (m0 - m1) / flow_rate;

```

```

103 // Plan circularization burn (using vis-viva equation)
104 System.out.println("Planning circularization burn");
105 double mu = vessel.getOrbit().getBody().getGravitationalParameter();
106 double r = vessel.getOrbit().getApoapsis();
107 double a1 = vessel.getOrbit().getSemiMajorAxis();
108 double a2 = r;
109 double v1 = Math.sqrt(mu * ((2.0 / r) - (1.0 / a1)));
110 double v2 = Math.sqrt(mu * ((2.0 / r) - (1.0 / a2)));
111 double deltaV = v2 - v1;
112 Node node = vessel.getControl().addNode(
113     ut.get() + vessel.getOrbit().getTimeToApoapsis(), (float)deltaV, 0, 0);
114
115 // Calculate burn time (using rocket equation)
116 double force = vessel.getAvailableThrust();
117 double isp = vessel.getSpecificImpulse() * 9.82;
118 double m0 = vessel.getMass();
119 double m1 = m0 / Math.exp(deltaV / isp);
120 double flowRate = force / isp;

```

```

78 ---- Plan circularization burn (using vis-viva equation)
79 print('Planning circularization burn')
80 local mu = vessel.orbit.body.gravitational_parameter
81 local r = vessel.orbit.apoapsis
82 local a1 = vessel.orbit.semi_major_axis
83 local a2 = r
84 local v1 = math.sqrt(mu*((2./r)-(1./a1)))
85 local v2 = math.sqrt(mu*((2./r)-(1./a2)))
86 local delta_v = v2 - v1
87 local node = vessel.control:add_node(conn.space_center.ut + vessel.orbit.time_to_
↪apoapsis, delta_v, 0, 0)
88
89 ---- Calculate burn time (using rocket equation)
90 local F = vessel.available_thrust
91 local Isp = vessel.specific_impulse * 9.82
92 local m0 = vessel.mass
93 local m1 = m0 / math.exp(delta_v/Isp)
94 local flow_rate = F / Isp

```

```

95 local burn_time = (m0 - m1) / flow_rate

76 # Plan circularization burn (using vis-viva equation)
77 print('Planning circularization burn')
78 mu = vessel.orbit.body.gravitational_parameter
79 r = vessel.orbit.apoapsis
80 a1 = vessel.orbit.semi_major_axis
81 a2 = r
82 v1 = math.sqrt(mu*((2./r)-(1./a1)))
83 v2 = math.sqrt(mu*((2./r)-(1./a2)))
84 delta_v = v2 - v1
85 node = vessel.control.add_node(
86     ut() + vessel.orbit.time_to_apoapsis, prograde=delta_v)
87
88 # Calculate burn time (using rocket equation)
89 F = vessel.available_thrust
90 Isp = vessel.specific_impulse * 9.82
91 m0 = vessel.mass
92 m1 = m0 / math.exp(delta_v/Isp)
93 flow_rate = F / Isp
94 burn_time = (m0 - m1) / flow_rate

```

Next, we need to rotate the craft and wait until the circularization burn. We orientate the ship along the y-axis of the maneuver node's reference frame (i.e. in the direction of the burn) then time warp to 5 seconds before the burn.

C#

C++

Java

Lua

Python

```

112 // Orientate ship
113 Console.WriteLine ("Orientating ship for circularization burn");
114 vessel.AutoPilot.ReferenceFrame = node.ReferenceFrame;
115 vessel.AutoPilot.TargetDirection = Tuple.Create (0.0, 1.0, 0.0);
116 vessel.AutoPilot.Wait ();
117
118 // Wait until burn
119 Console.WriteLine ("Waiting until circularization burn");
120 double burnUT = ut.Get () + vessel.Orbit.TimeToApoapsis - (burnTime / 2.0);
121 double leadTime = 5;
122 conn.SpaceCenter ().WarpTo (burnUT - leadTime);

```

```

105 // Orientate ship
106 std::cout << "Orientating ship for circularization burn" << std::endl;
107 vessel.auto_pilot().set_reference_frame(node.reference_frame());
108 vessel.auto_pilot().set_target_direction(std::make_tuple(0.0, 1.0, 0.0));
109 vessel.auto_pilot().wait();
110
111 // Wait until burn
112 std::cout << "Waiting until circularization burn" << std::endl;
113 double burn_ut = ut() + vessel.orbit().time_to_apoapsis() - (burn_time / 2.0);
114 double lead_time = 5;
115 space_center.warp_to(burn_ut - lead_time);

```

```

123 // Orientate ship
124 System.out.println("Orientating ship for circularization burn");
125 vessel.getAutoPilot().setReferenceFrame(node.getReferenceFrame());
126 vessel.getAutoPilot().setTargetDirection(
127     new Triplet<Double,Double,Double>(0.0, 1.0, 0.0));
128 vessel.getAutoPilot().wait_();
129
130 // Wait until burn
131 System.out.println("Waiting until circularization burn");
132 double burnUt =
133     ut.get() + vessel.getOrbit().getTimeToApoapsis() - (burnTime / 2.0);
134 double leadTime = 5;

```

```

97 -- Orientate ship
98 print('Orientating ship for circularization burn')
99 vessel.auto_pilot.reference_frame = node.reference_frame
100 vessel.auto_pilot.target_direction = List{0, 1, 0}
101 vessel.auto_pilot:wait()
102
103 -- Wait until burn
104 print('Waiting until circularization burn')
105 local burn_ut = conn.space_center.ut + vessel.orbit.time_to_apoapsis - (burn_time/2.)
106 local lead_time = 5
107 conn.space_center.warp_to(burn_ut - lead_time)

```

```

96 # Orientate ship
97 print('Orientating ship for circularization burn')
98 vessel.auto_pilot.reference_frame = node.reference_frame
99 vessel.auto_pilot.target_direction = (0, 1, 0)
100 vessel.auto_pilot.wait()
101
102 # Wait until burn
103 print('Waiting until circularization burn')
104 burn_ut = ut() + vessel.orbit.time_to_apoapsis - (burn_time/2.)
105 lead_time = 5
106 conn.space_center.warp_to(burn_ut - lead_time)

```

This next part executes the burn. It sets maximum throttle, then throttles down to 5% approximately a tenth of a second before the predicted end of the burn. It then monitors the remaining delta-v until it flips around to point retrograde (at which point the node has been executed).

C#

C++

Java

Lua

Python

```

124 // Execute burn
125 Console.WriteLine ("Ready to execute burn");
126 var timeToApoapsis = conn.AddStream (() => vessel.Orbit.TimeToApoapsis);
127 while (timeToApoapsis.Get () - (burnTime / 2.0) > 0) {
128     }
129 Console.WriteLine ("Executing burn");
130 vessel.Control.Throttle = 1;
131 System.Threading.Thread.Sleep ((int)((burnTime - 0.1) * 1000));

```



```

132     Console.WriteLine ("Fine tuning");
133     vessel.Control.Throttle = 0.05f;
134     var remainingBurn = conn.AddStream (
135         () => node.RemainingBurnVector (node.ReferenceFrame));
136     while (remainingBurn.Get ().Item1 > 0) {
137     }
138     vessel.Control.Throttle = 0;
139     node.Remove ();
140
141     Console.WriteLine ("Launch complete");
142     conn.Dispose ();
143 }
144 }

```

```

115 // Execute burn
116 std::cout << "Ready to execute burn" << std::endl;
117 auto time_to_apoapsis = vessel.orbit().time_to_apoapsis_stream();
118 while (time_to_apoapsis() - (burn_time / 2.0) > 0) {
119 }
120 std::cout << "Executing burn" << std::endl;
121 vessel.control().set_throttle(1);
122 std::this_thread::sleep_for(
123     std::chrono::milliseconds(static_cast<int>((burn_time - 0.1) * 1000)));
124 std::cout << "Fine tuning" << std::endl;
125 vessel.control().set_throttle(0.05);
126 auto remaining_burn = node.remaining_burn_vector_stream(node.reference_frame());
127 while (std::get<0>(remaining_burn()) > 0) {
128 }
129 vessel.control().set_throttle(0);
130 node.remove();
131
132 std::cout << "Launch complete" << std::endl;
133 }

```

```

137 // Execute burn
138 System.out.println("Ready to execute burn");
139 Stream<Double> timeToApoapsis =
140     connection.addStream(vessel.getOrbit(), "getTimeToApoapsis");
141 while (timeToApoapsis.get() - (burnTime / 2.0) > 0) {
142 }
143 System.out.println("Executing burn");
144 vessel.getControl().setThrottle(1);
145 Thread.sleep((int)((burnTime - 0.1) * 1000));
146 System.out.println("Fine tuning");
147 vessel.getControl().setThrottle(0.05f);
148 Stream<Triplet<Double,Double,Double>> remainingBurn =
149     connection.addStream(
150         node, "remainingBurnVector", node.getReferenceFrame());
151 while (remainingBurn.get().getValue1() > 0) {
152 }
153 vessel.getControl().setThrottle(0);
154 node.remove();
155
156 System.out.println("Launch complete");
157 connection.close();
158 }
159 }

```

```

109 -- Execute burn
110 print('Ready to execute burn')
111 while vessel.orbit.time_to_apoapsis - (burn_time/2.) > 0 do
112 end
113 print('Executing burn')
114 vessel.control.throttle = 1
115 platform.sleep(burn_time - 0.1)
116 print('Fine tuning')
117 vessel.control.throttle = 0.05
118 while node.remaining_burn_vector(node.reference_frame)[2] > 0 do
119 end
120 vessel.control.throttle = 0
121 node.remove()
122
123 print('Launch complete')

```

```

108 # Execute burn
109 print('Ready to execute burn')
110 time_to_apoapsis = conn.add_stream(getattr, vessel.orbit, 'time_to_apoapsis')
111 while time_to_apoapsis() - (burn_time/2.) > 0:
112     pass
113 print('Executing burn')
114 vessel.control.throttle = 1.0
115 time.sleep(burn_time - 0.1)
116 print('Fine tuning')
117 vessel.control.throttle = 0.05
118 remaining_burn = conn.add_stream(node.remaining_burn_vector, node.reference_frame)
119 while remaining_burn()[1] > 0:
120     pass
121 vessel.control.throttle = 0.0
122 node.remove()
123
124 print('Launch complete')

```

The rocket should now be in a circular 150km orbit above Kerbin.

## 2.4 Pitch, Heading and Roll

This example calculates the pitch, heading and rolls angles of the active vessel once per second.

C#

C++

Java

Lua

Python

```

using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;
using Vector3 = System.Tuple<double, double, double>;

class AngleOfAttack
{

```

```

static Vector3 CrossProduct (Vector3 u, Vector3 v)
{
    return new Vector3 (
        u.Item2 * v.Item3 - u.Item3 * v.Item2,
        u.Item3 * v.Item1 - u.Item1 * v.Item3,
        u.Item1 * v.Item2 - u.Item2 * v.Item1
    );
}

static double DotProduct (Vector3 u, Vector3 v)
{
    return u.Item1 * v.Item1 + u.Item2 * v.Item2 + u.Item3 * v.Item3;
}

static double Magnitude (Vector3 v)
{
    return Math.Sqrt (DotProduct (v, v));
}

// Compute the angle between vector x and y
static double AngleBetweenVectors (Vector3 u, Vector3 v)
{
    double dp = DotProduct (u, v);
    if (dp == 0)
        return 0;
    double um = Magnitude (u);
    double vm = Magnitude (v);
    return Math.Acos (dp / (um * vm)) * (180f / Math.PI);
}

public static void Main ()
{
    var conn = new Connection ("Angle of attack");
    var vessel = conn.SpaceCenter ().ActiveVessel;

    while (true) {
        var vesselDirection = vessel.Direction (vessel.SurfaceReferenceFrame);

        // Get the direction of the vessel in the horizon plane
        var horizonDirection = new Vector3 (
            0, vesselDirection.Item2, vesselDirection.Item3);

        // Compute the pitch - the angle between the vessels direction and
        // the direction in the horizon plane
        double pitch = AngleBetweenVectors (vesselDirection, horizonDirection);
        if (vesselDirection.Item1 < 0)
            pitch = -pitch;

        // Compute the heading - the angle between north and
        // the direction in the horizon plane
        var north = new Vector3 (0, 1, 0);
        double heading = AngleBetweenVectors (north, horizonDirection);
        if (horizonDirection.Item3 < 0)
            heading = 360 - heading;

        // Compute the roll
        // Compute the plane running through the vessels direction
        // and the upwards direction
    }
}

```

```

var up = new Vector3 (1, 0, 0);
var planeNormal = CrossProduct (vesselDirection, up);
// Compute the upwards direction of the vessel
var vesselUp = conn.SpaceCenter ().TransformDirection (
    new Vector3 (0, 0, -1),
    vessel.ReferenceFrame, vessel.SurfaceReferenceFrame);
// Compute the angle between the upwards direction of
// the vessel and the plane normal
double roll = AngleBetweenVectors (vesselUp, planeNormal);
// Adjust so that the angle is between -180 and 180 and
// rolling right is +ve and left is -ve
if (vesselUp.Item1 > 0)
    roll *= -1;
else if (roll < 0)
    roll += 180;
else
    roll -= 180;

Console.WriteLine ("pitch = {0:F1}, heading = {1:F1}, roll = {2:F1}",
    pitch, heading, roll);

System.Threading.Thread.Sleep (1000);
}
}
}

```

```

#include <iostream>
#include <iomanip>
#include <tuple>
#include <thread>
#include <chrono>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

static const double pi = 3.1415926535897;
typedef std::tuple<double, double, double> vector3;

vector3 cross_product(const vector3& u, const vector3& v) {
    return std::make_tuple(
        std::get<1>(u)*std::get<2>(v) - std::get<2>(u)*std::get<1>(v),
        std::get<2>(u)*std::get<0>(v) - std::get<0>(u)*std::get<2>(v),
        std::get<0>(u)*std::get<1>(v) - std::get<1>(u)*std::get<0>(v));
}

double dot_product(const vector3& u, const vector3& v) {
    return
        std::get<0>(u)*std::get<0>(v) +
        std::get<1>(u)*std::get<1>(v) +
        std::get<2>(u)*std::get<2>(v);
}

double magnitude(const vector3& v) {
    return sqrt(dot_product(v, v));
}

// Compute the angle between vector u and v
double angle_between_vectors(const vector3& u, const vector3& v) {
    double dp = dot_product(u, v);

```

```

    if (dp == 0)
        return 0;
    double um = magnitude(u);
    double vm = magnitude(v);
    return acos(dp / (um*vm)) * (180.0 / pi);
}

int main() {
    krpc::Client conn = krpc::connect("Pitch/Heading/Roll");
    krpc::services::SpaceCenter space_center(&conn);
    auto vessel = space_center.active_vessel();

    while (true) {
        vector3 vessel_direction = vessel.direction(vessel.surface_reference_frame());

        // Get the direction of the vessel in the horizon plane
        vector3 horizon_direction {
            0, std::get<1>(vessel_direction), std::get<2>(vessel_direction)
        };

        // Compute the pitch - the angle between the vessels direction
        // and the direction in the horizon plane
        double pitch = angle_between_vectors(vessel_direction, horizon_direction);
        if (std::get<0>(vessel_direction) < 0)
            pitch = -pitch;

        // Compute the heading - the angle between north
        // and the direction in the horizon plane
        vector3 north {0, 1, 0};
        double heading = angle_between_vectors(north, horizon_direction);
        if (std::get<2>(horizon_direction) < 0)
            heading = 360 - heading;

        // Compute the roll
        // Compute the plane running through the vessels direction
        // and the upwards direction
        vector3 up {1, 0, 0};
        vector3 plane_normal = cross_product(vessel_direction, up);
        // Compute the upwards direction of the vessel
        vector3 vessel_up = space_center.transform_direction(
            std::make_tuple(0, 0, -1),
            vessel.reference_frame(),
            vessel.surface_reference_frame());
        // Compute the angle between the upwards direction of
        // the vessel and the plane normal
        double roll = angle_between_vectors(vessel_up, plane_normal);
        // Adjust so that the angle is between -180 and 180 and
        // rolling right is +ve and left is -ve
        if (std::get<0>(vessel_up) > 0)
            roll *= -1;
        else if (roll < 0)
            roll += 180;
        else
            roll -= 180;

        std::cout << std::fixed << std::setprecision(1);
        std::cout << "pitch = " << pitch << ", "
            << "heading = " << heading << ", "

```

```

        << "roll = " << roll << std::endl;

        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;

import org.javatuples.Triplet;

import java.io.IOException;
import java.lang.Math;

public class PitchHeadingRoll {

    static Triplet<Double,Double,Double> crossProduct(
        Triplet<Double,Double,Double> u, Triplet<Double,Double,Double> v) {
        return new Triplet<Double,Double,Double>(
            u.getValue1() * v.getValue2() - u.getValue2() * v.getValue1(),
            u.getValue2() * v.getValue0() - u.getValue0() * v.getValue2(),
            u.getValue0() * v.getValue1() - u.getValue1() * v.getValue0()
        );
    }

    static double dotProduct(Triplet<Double,Double,Double> u,
        Triplet<Double,Double,Double> v) {
        return u.getValue0() * v.getValue0() +
            u.getValue1() * v.getValue1() +
            u.getValue2() * v.getValue2();
    }

    static double magnitude(Triplet<Double,Double,Double> v) {
        return Math.sqrt(dotProduct(v, v));
    }

    // Compute the angle between vector x and y
    static double angleBetweenVectors(Triplet<Double,Double,Double> u,
        Triplet<Double,Double,Double> v) {
        double dp = dotProduct(u, v);
        if (dp == 0) {
            return 0;
        }
        double um = magnitude(u);
        double vm = magnitude(v);
        return Math.acos(dp / (um * vm)) * (180f / Math.PI);
    }

    public static void main(String[] args)
        throws IOException, RPCException, InterruptedException {
        Connection connection = Connection.newInstance();
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        SpaceCenter.Vessel vessel = spaceCenter.getActiveVessel();

        while (true) {
            Triplet<Double,Double,Double> vesselDirection =
                vessel.direction(vessel.getSurfaceReferenceFrame());

```

```

// Get the direction of the vessel in the horizon plane
Triplet<Double,Double,Double> horizonDirection =
    new Triplet<Double,Double,Double>(
        0.0, vesselDirection.getValue1(), vesselDirection.getValue2());

// Compute the pitch - the angle between the vessels direction
// and the direction in the horizon plane
double pitch = angleBetweenVectors(vesselDirection, horizonDirection);
if (vesselDirection.getValue0() < 0) {
    pitch = -pitch;
}

// Compute the heading - the angle between north
// and the direction in the horizon plane
Triplet<Double,Double,Double> north =
    new Triplet<Double,Double,Double>(0.0,1.0,0.0);
double heading = angleBetweenVectors(north, horizonDirection);
if (horizonDirection.getValue2() < 0) {
    heading = 360 - heading;
}

// Compute the roll
// Compute the plane running through the vessels direction
// and the upwards direction
Triplet<Double,Double,Double> up =
    new Triplet<Double,Double,Double>(1.0,0.0,0.0);
Triplet<Double,Double,Double> planeNormal =
    crossProduct(vesselDirection, up);
// Compute the upwards direction of the vessel
Triplet<Double,Double,Double> vesselUp = spaceCenter.transformDirection(
    new Triplet<Double,Double,Double>(0.0,0.0,-1.0),
    vessel.getReferenceFrame(), vessel.getSurfaceReferenceFrame());
// Compute the angle between the upwards direction
// of the vessel and the plane normal
double roll = angleBetweenVectors(vesselUp, planeNormal);
// Adjust so that the angle is between -180 and 180 and
// rolling right is +ve and left is -ve
if (vesselUp.getValue0() > 0) {
    roll *= -1;
} else if (roll < 0) {
    roll += 180;
} else {
    roll -= 180;
}

System.out.printf("pitch = %.1f, heading = %.1f, roll = %.1f\n",
    pitch, heading, roll);
Thread.sleep(1000);
}
}
}

```

```

local krpc = require 'krpc'
local platform = require 'krpc.platform'
local math = require 'math'
local List = require 'pl.List'
local conn = krpc.connect('Pitch/Heading/Roll')

```

```

local vessel = conn.space_center.active_vessel

function cross_product(u, v)
    return List{u[3]*v[3] - u[3]*v[2],
                u[1]*v[1] - u[1]*v[3],
                u[2]*v[2] - u[2]*v[1]}
end

function dot_product(u, v)
    return u[1]*v[1] + u[2]*v[2] + u[3]*v[3]
end

function magnitude(v)
    return math.sqrt(dot_product(v, v))
end

function angle_between_vectors(u, v)
    -- Compute the angle between vector u and v
    dp = dot_product(u, v)
    if dp == 0 then
        return 0
    end
    um = magnitude(u)
    vm = magnitude(v)
    return math.acos(dp / (um*vm)) * (180. / math.pi)
end

while true do

    local vessel_direction = vessel:direction(vessel.surface_reference_frame)

    -- Get the direction of the vessel in the horizon plane
    local horizon_direction = List{0, vessel_direction[2], vessel_direction[3]}

    -- Compute the pitch - the angle between the vessels direction and
    -- the direction in the horizon plane
    local pitch = angle_between_vectors(vessel_direction, horizon_direction)
    if vessel_direction[1] < 0 then
        pitch = -pitch
    end

    -- Compute the heading - the angle between north and
    -- the direction in the horizon plane
    local north = List{0, 1, 0}
    local heading = angle_between_vectors(north, horizon_direction)
    if horizon_direction[3] < 0 then
        heading = 360 - heading
    end

    -- Compute the roll
    -- Compute the plane running through the vessels direction
    -- and the upwards direction
    local up = List{1, 0, 0}
    local plane_normal = cross_product(vessel_direction, up)
    -- Compute the upwards direction of the vessel
    local vessel_up = conn.space_center.transform_direction(
        List{0, 0, -1}, vessel.reference_frame, vessel.surface_reference_frame)
    -- Compute the angle between the upwards direction of

```



```

-- the vessel and the plane normal
local roll = angle_between_vectors(vessel_up, plane_normal)
-- Adjust so that the angle is between -180 and 180 and
-- rolling right is +ve and left is -ve
if vessel_up[1] > 0 then
    roll = -roll
elseif roll < 0 then
    roll = roll + 180
else
    roll = roll - 180
end

print(string.format('pitch = %.1f, heading = %.1f, roll = %.1f',
                    pitch, heading, roll))

platform.sleep(1)

end

```

```

import math
import time
import krpc
conn = krpc.connect(name='Pitch/Heading/Roll')
vessel = conn.space_center.active_vessel

def cross_product(u, v):
    return (u[1]*v[2] - u[2]*v[1],
            u[2]*v[0] - u[0]*v[2],
            u[0]*v[1] - u[1]*v[0])

def dot_product(u, v):
    return u[0]*v[0] + u[1]*v[1] + u[2]*v[2]

def magnitude(v):
    return math.sqrt(dot_product(v, v))

def angle_between_vectors(u, v):
    """ Compute the angle between vector u and v """
    dp = dot_product(u, v)
    if dp == 0:
        return 0
    um = magnitude(u)
    vm = magnitude(v)
    return math.acos(dp / (um*vm)) * (180. / math.pi)

while True:

    vessel_direction = vessel.direction(vessel.surface_reference_frame)

    # Get the direction of the vessel in the horizon plane
    horizon_direction = (0, vessel_direction[1], vessel_direction[2])

    # Compute the pitch - the angle between the vessels direction and

```

```
# the direction in the horizon plane
pitch = angle_between_vectors(vessel_direction, horizon_direction)
if vessel_direction[0] < 0:
    pitch = -pitch

# Compute the heading - the angle between north and
# the direction in the horizon plane
north = (0, 1, 0)
heading = angle_between_vectors(north, horizon_direction)
if horizon_direction[2] < 0:
    heading = 360 - heading

# Compute the roll
# Compute the plane running through the vessels direction
# and the upwards direction
up = (1, 0, 0)
plane_normal = cross_product(vessel_direction, up)
# Compute the upwards direction of the vessel
vessel_up = conn.space_center.transform_direction(
    (0, 0, -1), vessel.reference_frame, vessel.surface_reference_frame)
# Compute the angle between the upwards direction of
# the vessel and the plane normal
roll = angle_between_vectors(vessel_up, plane_normal)
# Adjust so that the angle is between -180 and 180 and
# rolling right is +ve and left is -ve
if vessel_up[0] > 0:
    roll *= -1
elif roll < 0:
    roll += 180
else:
    roll -= 180

print('pitch = % 5.1f, heading = % 5.1f, roll = % 5.1f' %
      (pitch, heading, roll))

time.sleep(1)
```

## 2.5 Interacting with Parts

The following examples demonstrate use of the *Parts* functionality to achieve various tasks. More details on specific topics can also be found in the API documentation:

C#

C++

Java

Lua

Python

- *Trees of Parts*
- *Attachment Modes*
- *Fuel Lines*
- *Staging*

- *Trees of Parts*
- *Attachment Modes*
- *Fuel Lines*
- *Staging*
- *Trees of Parts*
- *Attachment Modes*
- *Fuel Lines*
- *Staging*
- *Trees of Parts*
- *Attachment Modes*
- *Fuel Lines*
- *Staging*
- *Trees of Parts*
- *Attachment Modes*
- *Fuel Lines*
- *Staging*

### 2.5.1 Deploying all Parachutes

Sometimes things go horribly wrong. The following script does its best to save your Kerbals by deploying all the parachutes:

C#

C++

Java

Lua

Python

```
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class DeployParachutes
{
    public static void Main ()
    {
        using (var connection = new Connection ()) {
            var vessel = connection.SpaceCenter ().ActiveVessel;
            foreach (var parachute in vessel.Parts.Parachutes)
                parachute.Deploy ();
        }
    }
}
```

```
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client conn = krpc::connect();
    krpc::services::SpaceCenter space_center(&conn);
    auto vessel = space_center.active_vessel();
    for (auto parachute : vessel.parts().parachutes())
        parachute.deploy();
}
```

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Parachute;
import krpc.client.services.SpaceCenter.Vessel;

import java.io.IOException;

public class DeployParachutes {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        for (Parachute parachute : vessel.getParts().getParachutes()) {
            parachute.deploy();
        }
        connection.close();
    }
}
```

```
local krpc = require 'krpc'
local conn = krpc.connect('Example')
local vessel = conn.space_center.active_vessel

for _, parachute in ipairs(vessel.parts.parachutes) do
    parachute:deploy()
end
```

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel

for parachute in vessel.parts.parachutes:
    parachute.deploy()
```

## 2.5.2 ‘Control From Here’ for Docking Ports

The following example will find a standard sized Clamp-O-Tron docking port, and control the vessel from it:

C#  
C++  
Java  
Lua

## Python

```

using System;
using System.Linq;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class ControlFromHere
{
    public static void Main ()
    {
        using (var conn = new Connection ()) {
            var vessel = conn.SpaceCenter ().ActiveVessel;
            var part = vessel.Parts.WithTitle ("Clamp-O-Tron Docking Port") [0];
            vessel.Parts.Controlling = part;
        }
    }
}

```

```

#include <iostream>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client conn = krpc::connect();
    krpc::services::SpaceCenter space_center(&conn);
    auto vessel = space_center.active_vessel();
    auto part = vessel.parts().with_title("Clamp-O-Tron Docking Port").front();
    vessel.parts().set_controlling(part);
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Part;
import krpc.client.services.SpaceCenter.Vessel;

import java.io.IOException;

public class ControlFromHere {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.newInstance(connection).getActiveVessel();
        Part part = vessel.getParts().withTitle("Clamp-O-Tron Docking Port").get(0);
        vessel.getParts().setControlling(part);
        connection.close();
    }
}

```

```

local krpc = require 'krpc'
local conn = krpc.connect()
local vessel = conn.space_center.active_vessel
local part = vessel.parts:with_title('Clamp-O-Tron Docking Port')[1]
vessel.parts.controlling = part

```

```

import krpc
conn = krpc.connect()

```

```
vessel = conn.space_center.active_vessel
part = vessel.parts.with_title('Clamp-O-Tron Docking Port')[0]
vessel.parts.controlling = part
```

## 2.5.3 Combined Specific Impulse

The following script calculates the combined specific impulse of all currently active and fueled engines on a rocket. See [here](https://wiki.kerbalspaceprogram.com/wiki/Specific_impulse#Multiple_engines) for a description of the maths: [https://wiki.kerbalspaceprogram.com/wiki/Specific\\_impulse#Multiple\\_engines](https://wiki.kerbalspaceprogram.com/wiki/Specific_impulse#Multiple_engines)

C#

C++

Java

Lua

Python

```
using System;
using System.Linq;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class CombinedIsp
{
    public static void Main ()
    {
        using (var connection = new Connection ()) {
            var vessel = connection.SpaceCenter ().ActiveVessel;

            var activeEngines = vessel.Parts.Engines
                .Where (e => e.Active && e.HasFuel).ToList ();

            Console.WriteLine ("Active engines:");
            foreach (var engine in activeEngines)
                Console.WriteLine ("    " + engine.Part.Title +
                                    " in stage " + engine.Part.Stage);

            double thrust = activeEngines.Sum (e => e.Thrust);
            double fuel_consumption =
                activeEngines.Sum (e => e.Thrust / e.SpecificImpulse);
            double isp = thrust / fuel_consumption;
            Console.WriteLine ("Combined vacuum Isp = {0:F0} seconds", isp);
        }
    }
}
```

```
#include <iostream>
#include <vector>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

using SpaceCenter = krpc::services::SpaceCenter;

int main() {
    auto conn = krpc::connect();
    SpaceCenter sc(&conn);
```

```

auto vessel = sc.active_vessel();

auto engines = vessel.parts().engines();

std::vector<SpaceCenter::Engine> active_engines;
for (auto engine : engines)
    if (engine.active() && engine.has_fuel())
        active_engines.push_back(engine);

std::cout << "Active engines:" << std::endl;
for (auto engine : active_engines)
    std::cout << "    " << engine.part().title() << " in stage "
                << engine.part().stage() << std::endl;

double thrust = 0;
double fuel_consumption = 0;
for (auto engine : active_engines) {
    thrust += engine.thrust();
    fuel_consumption += engine.thrust() / engine.specific_impulse();
}
double isp = thrust / fuel_consumption;
std::cout << "Combined vacuum Isp = " << isp << " seconds" << std::endl;
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Engine;
import krpc.client.services.SpaceCenter.Vessel;

import java.io.IOException;
import java.util.LinkedList;
import java.util.List;

public class CombinedIsp {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.newInstance(connection).getActiveVessel();

        List<Engine> engines = vessel.getParts().getEngines();
        List<Engine> activeEngines = new LinkedList<Engine>();
        for (Engine engine : engines) {
            if (engine.getActive() && engine.getHasFuel()) {
                activeEngines.add(engine);
            }
        }

        System.out.println("Active engines:");
        for (Engine engine : activeEngines) {
            System.out.println("    " + engine.getPart().getTitle() +
                               " in stage " + engine.getPart().getStage());
        }

        double thrust = 0;
        double fuelConsumption = 0;
        for (Engine engine : activeEngines) {
            thrust += engine.getThrust();
            fuelConsumption += engine.getThrust() / engine.getSpecificImpulse();
        }
    }
}

```

```
    }
    double isp = thrust / fuelConsumption;
    System.out.printf("Combined vacuum Isp = %.0f\n", isp);
    connection.close();
}
}
```

```
local krpc = require 'krpc'
local math = require 'math'
local conn = krpc.connect()
local vessel = conn.space_center.active_vessel

local active_engines = {}
for _, engine in ipairs(vessel.parts.engines) do
    if engine.active and engine.has_fuel then
        table.insert(active_engines, engine)
    end
end

print('Active engines:')
for _, engine in ipairs(active_engines) do
    print('    ' .. engine.part.title .. ' in stage ' .. engine.part.stage)
end

thrust = 0
fuel_consumption = 0
for _, engine in ipairs(active_engines) do
    thrust = thrust + engine.thrust
    fuel_consumption = fuel_consumption + engine.thrust / engine.specific_impulse
end
isp = thrust / fuel_consumption

print(string.format('Combined vacuum Isp = %.1f seconds', isp))
```

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel

active_engines = [e for e in vessel.parts.engines if e.active and e.has_fuel]

print('Active engines:')
for engine in active_engines:
    print('    %s in stage %d' % (engine.part.title, engine.part.stage))

thrust = sum(engine.thrust for engine in active_engines)
fuel_consumption = sum(engine.thrust / engine.specific_impulse
                        for engine in active_engines)
isp = thrust / fuel_consumption

print('Combined vacuum Isp = %d seconds' % isp)
```

## 2.6 Docking Guidance

The following script outputs docking guidance information. It waits until the vessel is being controlled from a docking port, and a docking port is set as the current target. It then prints out information about speeds and distances relative



to the docking axis.

It uses `numpy` to do linear algebra on the vectors returned by kRPC – for example computing the dot product or length of a vector – and uses `curses` for terminal output.

```
import curses
import time
import numpy as np
import numpy.linalg as la
import krpc

# Set up curses
stdscr = curses.initscr()
curses.nocbreak()
stdscr.keypad(1)
curses.noecho()

try:
    # Connect to kRPC
    conn = krpc.connect(name='Docking Guidance')
    vessel = conn.space_center.active_vessel
    current = None
    target = None

    while True:
        stdscr.clear()
        stdscr.addstr(0, 0, '-- Docking Guidance --')

        current = conn.space_center.active_vessel.parts.controlling.docking_port
        target = conn.space_center.target_docking_port

        if current is None:
            stdscr.addstr(2, 0, 'Awaiting control from docking port...')

        elif target is None:
            stdscr.addstr(2, 0, 'Awaiting target docking port...')

        else:
            # Get positions, distances, velocities and
            # speeds relative to the target docking port
            current_position = current.position(target.reference_frame)
            velocity = current.part.velocity(target.reference_frame)
            displacement = np.array(current_position)
            distance = la.norm(displacement)
            speed = la.norm(np.array(velocity))

            # Get speeds and distances relative to the docking axis
            # (the direction the target docking port is facing in)

            # Axial = along the docking axis
            axial_displacement = np.copy(displacement)
            axial_displacement[0] = 0
            axial_displacement[2] = 0
            axial_distance = axial_displacement[1]
            axial_velocity = np.copy(velocity)
            axial_velocity[0] = 0
            axial_velocity[2] = 0
            axial_speed = axial_velocity[1]
            if axial_distance > 0:
```

```

        axial_speed *= -1

    # Radial = perpendicular to the docking axis
    radial_displacement = np.copy(displacement)
    radial_displacement[1] = 0
    radial_distance = la.norm(radial_displacement)
    radial_velocity = np.copy(velocity)
    radial_velocity[1] = 0
    radial_speed = la.norm(radial_velocity)
    if np.dot(radial_velocity, radial_displacement) > 0:
        radial_speed *= -1

    # Get the docking port state
    if current.state == conn.space_center.DockingPortState.ready:
        state = 'Ready to dock'
    elif current.state == conn.space_center.DockingPortState.docked:
        state = 'Docked'
    elif current.state == conn.space_center.DockingPortState.docking:
        state = 'Docking...'
    else:
        state = 'Unknown'

    # Output information
    stdscr.addstr(2, 0, 'Current ship: {:30}'.format(current.part.vessel.
↪name[:30]))
    stdscr.addstr(3, 0, 'Current port: {:30}'.format(current.part.title[:30]))
    stdscr.addstr(5, 0, 'Target ship:  {:30}'.format(target.part.vessel.
↪name[:30]))
    stdscr.addstr(6, 0, 'Target port:  {:30}'.format(target.part.title[:30]))
    stdscr.addstr(8, 0, 'Status: {:10}'.format(state))
    stdscr.addstr(10, 0, '          +-----+')
    stdscr.addstr(11, 0, '          | Distance | Speed      |')
    stdscr.addstr(12, 0, '+-----+-----+-----+')
    stdscr.addstr(13, 0, '|          | {:>+6.2f} m | {:>+6.2f} m/s |'
        .format(distance, speed))
    stdscr.addstr(14, 0, '| Axial | {:>+6.2f} m | {:>+6.2f} m/s |'
        .format(axial_distance, axial_speed))
    stdscr.addstr(15, 0, '| Radial | {:>+6.2f} m | {:>+6.2f} m/s |'
        .format(radial_distance, radial_speed))
    stdscr.addstr(16, 0, '+-----+-----+-----+')

    stdscr.refresh()
    time.sleep(0.25)

finally:
    # Shutdown curses
    curses.nocbreak()
    stdscr.keypad(0)
    curses.echo()
    curses.endwin()

```

## 2.7 User Interface

The following script demonstrates how to use the UI service to display text and handle basic user input. It adds a panel to the left side of the screen, displaying the current thrust produced by the vessel and a button to set the throttle to

maximum.

C#

C++

Java

Lua

Python

```
using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;
using KRPC.Client.Services.UI;

class UserInterface
{
    public static void Main ()
    {
        var conn = new Connection ("User Interface Example");
        var canvas = conn.UI ().StockCanvas;

        // Get the size of the game window in pixels
        var screenSize = canvas.RectTransform.Size;

        // Add a panel to contain the UI elements
        var panel = canvas.AddPanel ();

        // Position the panel on the left of the screen
        var rect = panel.RectTransform;
        rect.Size = Tuple.Create (200.0, 100.0);
        rect.Position = Tuple.Create ((110-(screenSize.Item1)/2), 0.0);

        // Add a button to set the throttle to maximum
        var button = panel.AddButton ("Full Throttle");
        button.RectTransform.Position = Tuple.Create (0.0, 20.0);

        // Add some text displaying the total engine thrust
        var text = panel.AddText ("Thrust: 0 kN");
        text.RectTransform.Position = Tuple.Create (0.0, -20.0);
        text.Color = Tuple.Create (1.0, 1.0, 1.0);
        text.Size = 18;

        // Set up a stream to monitor the throttle button
        var buttonClicked = conn.AddStream (() => button.Clicked);

        var vessel = conn.SpaceCenter ().ActiveVessel;
        while (true) {
            // Handle the throttle button being clicked
            if (buttonClicked.Get ()) {
                vessel.Control.Throttle = 1;
                button.Clicked = false;
            }

            // Update the thrust text
            text.Content = "Thrust: " + (vessel.Thrust/1000) + " kN";

            System.Threading.Thread.Sleep (1000);
        }
    }
}
```

```

    }
}

```

```

#include <chrono>
#include <thread>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>
#include <krpc/services/ui.hpp>

int main() {
    krpc::Client conn = krpc::connect("User Interface Example");
    krpc::services::SpaceCenter space_center(&conn);
    krpc::services::UI ui(&conn);
    auto canvas = ui.stock_canvas();

    // Get the size of the game window in pixels
    auto screen_size = canvas.rect_transform().size();

    // Add a panel to contain the UI elements
    auto panel = canvas.add_panel();

    // Position the panel on the left of the screen
    auto rect = panel.rect_transform();
    rect.set_size(std::make_tuple(200, 100));
    rect.set_position(std::make_tuple(110-(std::get<0>(screen_size)/2), 0));

    // Add a button to set the throttle to maximum
    auto button = panel.add_button("Full Throttle");
    button.rect_transform().set_position(std::make_tuple(0, 20));

    // Add some text displaying the total engine thrust
    auto text = panel.add_text("Thrust: 0 kN");
    text.rect_transform().set_position(std::make_tuple(0, -20));
    text.set_color(std::make_tuple(1, 1, 1));
    text.set_size(18);

    // Set up a stream to monitor the throttle button
    auto button_clicked = button.clicked_stream();

    auto vessel = space_center.active_vessel();
    while (true) {
        // Handle the throttle button being clicked
        if (button_clicked()) {
            vessel.control().set_throttle(1);
            button.set_clicked(false);
        }

        // Update the thrust text
        text.set_content("Thrust: " + std::to_string((int)(vessel.thrust()/1000)) + " kN
↪");

        std::this_thread::sleep_for(std::chrono::seconds(1));
    }
}

```

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.Stream;
import krpc.client.StreamException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;
import krpc.client.services.UI;
import krpc.client.services.UI.Button;
import krpc.client.services.UI.Canvas;
import krpc.client.services.UI.Panel;
import krpc.client.services.UI.RectTransform;
import krpc.client.services.UI.Text;

import org.javatuples.Pair;
import org.javatuples.Triplet;

import java.io.IOException;

public class UserInterface {
    public static void main(String[] args)
        throws IOException, RPCException, InterruptedException, StreamException {
        Connection connection = Connection.newInstance("User Interface Example");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        UI ui = UI.newInstance(connection);
        Canvas canvas = ui.getStockCanvas();

        // Get the size of the game window in pixels
        Pair<Double, Double> screenSize = canvas.getRectTransform().getSize();

        // Add a panel to contain the UI elements
        Panel panel = canvas.addPanel(true);

        // Position the panel on the left of the screen
        RectTransform rect = panel.getRectTransform();
        rect.setSize(new Pair<Double, Double>(200.0, 100.0));
        rect.setPosition(
            new Pair<Double, Double>((110-(screenSize.getValue0())/2), 0.0));

        // Add a button to set the throttle to maximum
        Button button = panel.addButton("Full Throttle", true);
        button.getRectTransform().setPosition(new Pair<Double, Double>(0.0, 20.0));

        // Add some text displaying the total engine thrust
        Text text = panel.addText("Thrust: 0 kN", true);
        text.getRectTransform().setPosition(new Pair<Double, Double>(0.0, -20.0));
        text.setColor(new Triplet<Double, Double, Double>(1.0, 1.0, 1.0));
        text.setSize(18);

        // Set up a stream to monitor the throttle button
        Stream<Boolean> buttonClicked = connection.addStream(button, "getClicked");

        Vessel vessel = spaceCenter.getActiveVessel();
        while (true) {
            // Handle the throttle button being clicked
            if (buttonClicked.get ()) {
                vessel.getControl().setThrottle(1);
                button.setClicked(false);
            }
        }
    }
}

```

```
        // Update the thrust text
        text.setContent(String.format("Thrust: %.0f kN", (vessel.getThrust()/
↪1000)));

        Thread.sleep(1000);
    }
}
}
```

```
local krpc = require 'krpc'
local platform = require 'krpc.platform'
local List = require 'pl.List'
local conn = krpc.connect('User Interface Example')
local canvas = conn.ui.stock_canvas

-- Get the size of the game window in pixels
local screen_size = canvas.rect_transform.size

-- Add a panel to contain the UI elements
local panel = canvas:add_panel()

-- Position the panel on the left of the screen
local rect = panel.rect_transform
rect.size = List{200, 100}
rect.position = List{110-(screen_size[1]/2), 0}

-- Add a button to set the throttle to maximum
local button = panel:add_button("Full Throttle")
button.rect_transform.position = List{0, 20}

-- Add some text displaying the total engine thrust
local text = panel:add_text("Thrust: 0 kN")
text.rect_transform.position = List{0, -20}
text.color = List{1, 1, 1}
text.size = 18

local vessel = conn.space_center.active_vessel
while true do
    -- Handle the throttle button being clicked
    if button.clicked then
        vessel.control.throttle = 1
        button.clicked = false
    end

    -- Update the thrust text
    text.content = string.format('Thrust: %.1f kN', vessel.thrust/1000)

    platform.sleep(0.1)
end
```

```
import time
import krpc

conn = krpc.connect(name='User Interface Example')
canvas = conn.ui.stock_canvas
```

```

# Get the size of the game window in pixels
screen_size = canvas.rect_transform.size

# Add a panel to contain the UI elements
panel = canvas.add_panel()

# Position the panel on the left of the screen
rect = panel.rect_transform
rect.size = (200, 100)
rect.position = (110-(screen_size[0]/2), 0)

# Add a button to set the throttle to maximum
button = panel.add_button("Full Throttle")
button.rect_transform.position = (0, 20)

# Add some text displaying the total engine thrust
text = panel.add_text("Thrust: 0 kN")
text.rect_transform.position = (0, -20)
text.color = (1, 1, 1)
text.size = 18

# Set up a stream to monitor the throttle button
button_clicked = conn.add_stream(getattr, button, 'clicked')

vessel = conn.space_center.active_vessel
while True:
    # Handle the throttle button being clicked
    if button_clicked():
        vessel.control.throttle = 1
        button.clicked = False

    # Update the thrust text
    text.content = 'Thrust: %d kN' % (vessel.thrust/1000)

    time.sleep(0.1)

```

## 2.8 AutoPilot

kRPC provides an autopilot that can be used to hold a vessel in a chosen orientation. It automatically tunes itself to cope with vessels of differing size and control authority. This tutorial explains how the autopilot works, how to configure it and mathematics behind it.

### 2.8.1 Overview

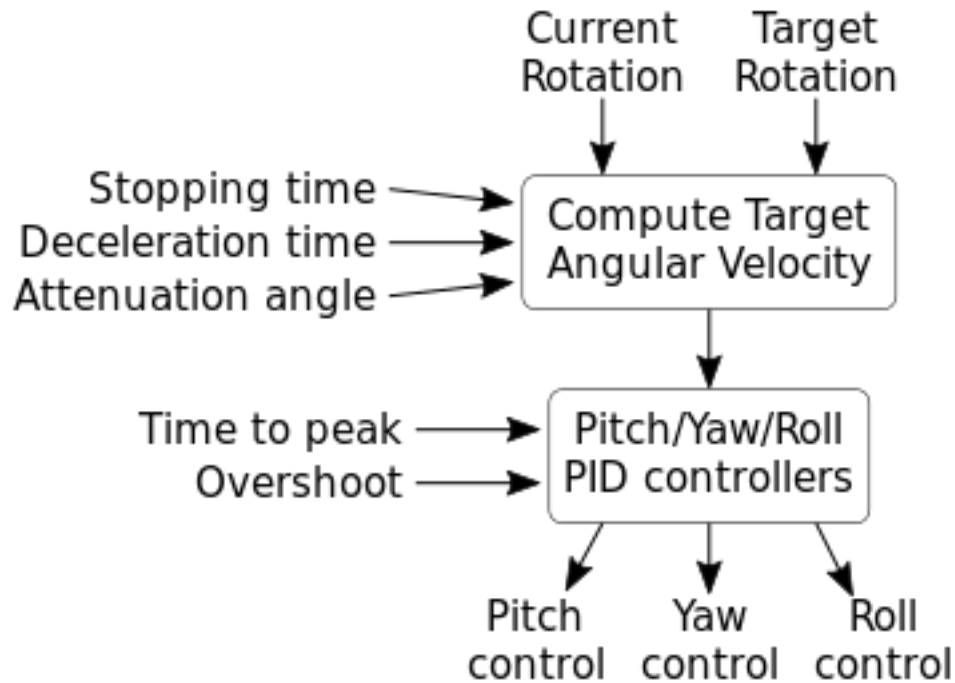
The inputs to the autopilot are:

- A reference frame defining where zero rotation is,
- target pitch and heading angles,
- and an (optional) target roll angle.

When a roll angle is not specified, the autopilot will try to zero out any rotation around the roll axis but will not try to hold a specific roll angle.

The diagram below shows a high level overview of the autopilot. First, the current rotation and target rotation are used to compute the *target angular velocity* that is needed to rotate the vessel to face the target. Next, the components of this angular velocity in the pitch, yaw and roll axes of the vessel are passed to three PID controllers. The outputs of these controllers are used as the control inputs for the vessel.

There are several parameters affecting the operation of the autopilot, shown the the left of the diagram. They are covered in the next section.



## 2.8.2 Configuring the AutoPilot

There are several parameters that affect the behavior of the autopilot. The default values for these should suffice in most cases, but they can be adjusted to fit your needs.

- The **stopping time** is the maximum amount of time that the vessel should need to come to a complete stop. This limits the maximum angular velocity of the vessel. It is a vector of three stopping times, one for each of the pitch, roll and yaw axes. The default value is 0.5 seconds for each axis.
- The **deceleration time** is the minimum time the autopilot should take to decelerate the vessel to a stop, as it approaches the target direction. This is a minimum value, as the time required may be higher if the vessel does not have sufficient angular acceleration. It is a vector of three deceleration times, in seconds, for each of the pitch, roll and yaw axes. The default value is 5 seconds for each axis. A smaller value will make the autopilot decelerate more aggressively, turning the vessel towards the target more quickly. However, decreasing the value too much could result in overshoot.
- In order to avoid overshoot, the stopping time should be smaller than the deceleration time. This gives the autopilot some 'spare' acceleration, to adjust for errors in the vessels rotation, for example due to changing aerodynamic forces.
- The **attenuation angle** sets the region in which the autopilot considers the vessel to be 'close' to the target direction. In this region, the target velocity is attenuated based on how close the vessel is to the target. It is an angle, in degrees, for each of the pitch, roll and yaw axes. The default value is 1 degree in each axis. This attenuation prevents the controls from oscillating when the vessel is pointing in the correct direction. If you find that the vessel still oscillates, try increasing this value.

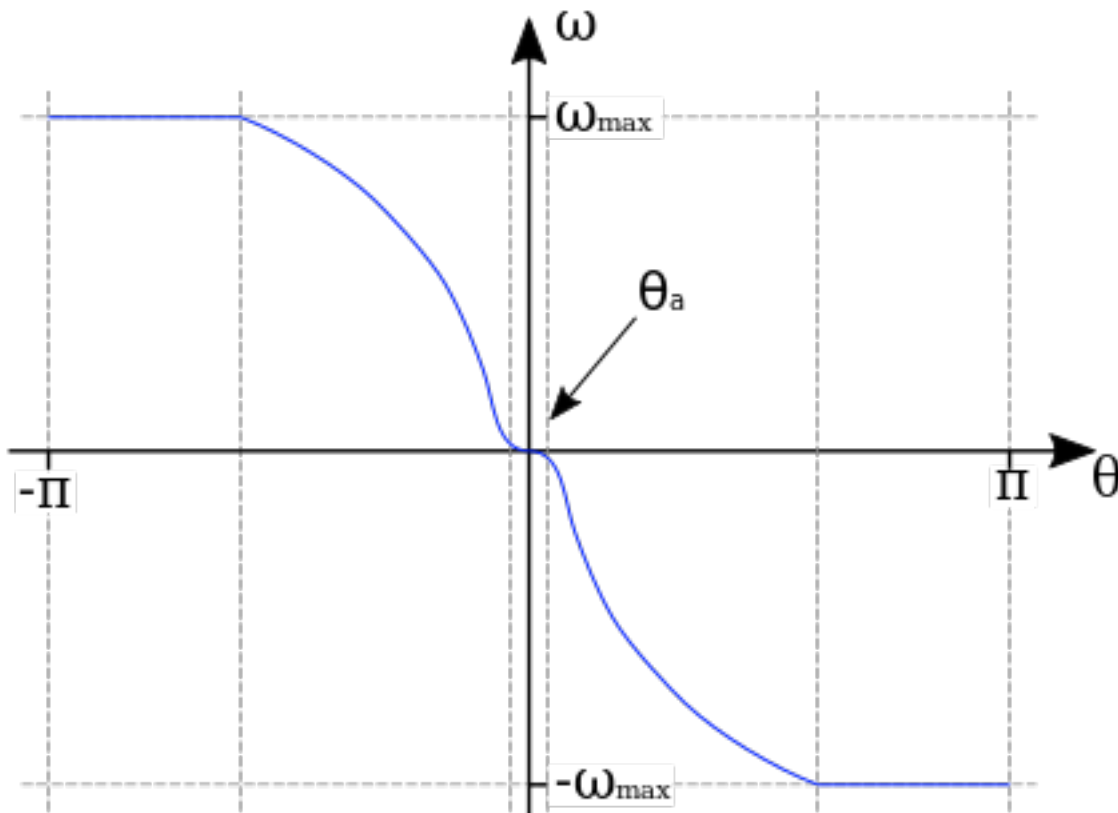


- The **time to peak**, in seconds, that the PID controllers take to adjust the angular velocity of the vessel to the target angular velocity. Decreasing this value will make the controllers try to match the target velocity more aggressively. It is a vector of three times, one for each of the pitch, roll and yaw axes. The default is 3 seconds in each axis.
- The **overshoot** is the percentage by which the PID controllers are allowed to overshoot the target angular velocity. Increasing this value will make the controllers try to match the target velocity more aggressively, but will cause more overshoot. It is a vector of three values, between 0 and 1, for each of the pitch, roll and yaw axes. The default is 0.01 in each axis.

### 2.8.3 Computing the Target Angular Velocity

The target angular velocity is the angular velocity needed to the vessel to rotate it towards the target direction. It is computed by summing a target angular speed for each of pitch, yaw and roll axes. If no roll angle is set, then the target angular velocity in the roll axis is simply set to 0.

The target angular speed  $\omega$  in a given axis is computed from the angular error  $\theta$  using the following function:



The equation for this function is:

$$\omega = -\frac{\theta}{|\theta|} \min(\omega_{max}, \sqrt{2\alpha|\theta|} \cdot f_a(\theta))$$

where

$$\alpha = \frac{\omega_{max}}{t_{decel}}$$

$$\omega_{max} = \frac{\tau_{max} t_{stop}}{I}$$

$$f_a(\theta) = \frac{1}{1 + e^{-6/\theta_a(|\theta| - \theta_a)}}$$

The reasoning and derivation for this is as follows:

- The vessel needs to rotate towards  $\theta = 0$ . This means that the target angular speed  $\omega$  needs to be positive when  $\theta$  is negative, and negative when  $\theta$  is positive. This is done by multiplying by the term  $-\frac{\theta}{|\theta|}$ , which is 1 when  $\theta < 0$  and -1 when  $\theta \geq 0$
- We want the vessel to rotate at a maximum angular speed  $\omega_{max}$ , which is determined by the stopping time  $t_{stop}$ . Using the equations of motion under constant acceleration we can derive it as follows:

$$\begin{aligned} \omega &= \alpha t \\ \Rightarrow \omega_{max} &= \alpha_{max} t_{stop} \\ &= \frac{\tau_{max} t_{stop}}{I} \end{aligned}$$

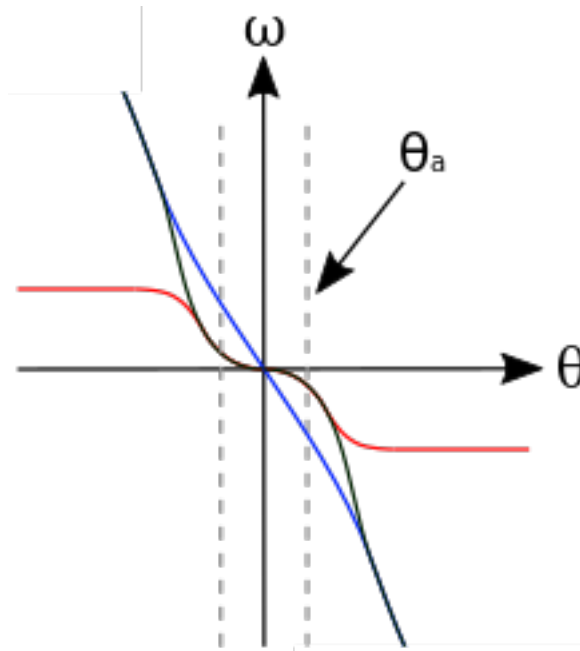
where  $\tau_{max}$  is the maximum torque the vessel can generate, and  $I$  is its moment of inertia.

- We want the vessel to take time  $t_{decel}$  (the deceleration time) to go from moving at speed  $\omega_{max}$  to rest, when facing the target. And we want it to do this using a constant acceleration  $\alpha$ . Using the equations of motion under constant acceleration we can derive the target velocity  $\omega$  in terms of the current angular error  $\theta$ :

$$\begin{aligned} \omega &= \alpha t \\ \Rightarrow \alpha &= \frac{\omega}{t} = \frac{\omega_{max}}{t_{decel}} \\ \theta &= \frac{1}{2} \alpha t^2 \Rightarrow t = \sqrt{\frac{2\theta}{\alpha}} \\ \Rightarrow \omega &= \alpha \sqrt{\frac{2\theta}{\alpha}} = \sqrt{2\alpha\theta} \end{aligned}$$

- To prevent the vessel from oscillating when it is pointing in the target direction, the gradient of the target angular speed curve at  $\theta = 0$  needs to be 0, and increase/decrease smoothly with increasing/decreasing  $\theta$ .

This is not the case for the target angular speed calculated above. To correct this, we multiply by an attenuation function which has the required shape. The following diagram shows the shape of the attenuation function (line in red), the target velocity as calculated previously (line in blue) and the result of multiplying these together (dashed line in black):



The formula for the attenuation function is a logistic function, with the following formula:

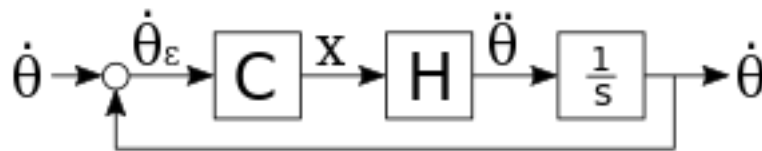
$$f_a(\theta) = \frac{1}{1 + e^{-6/\theta_a(|\theta| - \theta_a)}}$$

Note that the original function, derived from the equations of motion under constant acceleration, is only affected by the attenuation function close to the attenuation angle. This means that autopilot will use a constant acceleration to slow the vessel, until it gets close to the target direction.

## 2.8.4 Tuning the Controllers

Three PID controllers, one for each of the pitch, roll and yaw control axes, are used to control the vessel. Each controller takes the relevant component of the target angular velocity as input. The following describes how the gains for these controllers are automatically tuned based on the vessels available torque and moment of inertia.

The schematic for the entire system, in a single control axis, is as follows:



The input to the system is the angular speed around the control axis, denoted  $\omega$ . The error in the angular speed  $\omega_e$  is calculated from this and passed to controller  $C$ . This is a PID controller that we need to tune. The output of the controller is the control input,  $x$ , that is passed to the vessel. The plant  $H$  describes the physical system, i.e. how the control input affects the angular acceleration of the vessel. The derivative of this is computed to get the new angular speed of the vessel, which is then fed back to compute the new error.

For the controller,  $C$ , we use a proportional-integral controller. Note that the controller does not have a derivative term, so that the system behaves like a second order system and is therefore easy to tune.

The transfer function for the controller in the  $s$  domain is:

$$C(s) = K_P + K_I s^{-1}$$

From the schematic, the transfer function for the plant  $H$  is:

$$H(s) = \frac{\omega_\epsilon(s)}{X(s)}$$

$x$  is the control input to the vessel, which is the percentage of the available torque  $\tau_{max}$  that is being applied to the vessel. Call this the current torque, denoted  $\tau$ . This can be written mathematically as:

$$\tau = x\tau_{max}$$

Combining this with the angular equation of motion gives the angular acceleration in terms of the control input:

$$\begin{aligned} I &= \text{moment of inertia of the vessel} \\ \tau &= I\omega_\epsilon \\ \Rightarrow \omega_\epsilon &= \frac{x\tau_{max}}{I} \end{aligned}$$

Taking the laplace transform of this gives us:

$$\begin{aligned} \mathcal{L}(\omega_\epsilon(t)) &= s\omega_\epsilon(s) \\ &= \frac{sX(s)\tau_{max}}{I} \\ \Rightarrow \frac{\omega_\epsilon(s)}{X(s)} &= \frac{\tau_{max}}{I} \end{aligned}$$

We can now rewrite the transfer function for  $H$  as:

$$H(s) = \frac{\tau_{max}}{I}$$

The open loop transfer function for the entire system is:

$$\begin{aligned} G_{OL}(s) &= C(s) \cdot H(s) \cdot s^{-1} \\ &= (K_P + K_I s^{-1}) \frac{\tau_{max}}{I s} \end{aligned}$$

The closed loop transfer function is then:

$$\begin{aligned} G(s) &= \frac{G_{OL}(s)}{1 + G_{OL}(s)} \\ &= \frac{aK_P s + aK_I}{s^2 + aK_P s + aK_I} \text{ where } a = \frac{\tau_{max}}{I} \end{aligned}$$

The characteristic equation for the system is therefore:

$$\Phi = s^2 + \frac{\tau_{max}}{I} K_P s + \frac{\tau_{max}}{I} K_I$$

The characteristic equation for a standard second order system is:

$$\Phi_{standard} = s^2 + 2\zeta\omega_0 s + \omega_0^2$$

where  $\zeta$  is the damping ratio and  $\omega_0$  is the natural frequency of the system.

Equating coefficients between these equations, and rearranging, gives us the gains for the PI controller in terms of  $\zeta$  and  $\omega_0$ :

$$\begin{aligned} K_P &= \frac{2\zeta\omega_0 I}{\tau_{max}} \\ K_I &= \frac{I\omega_0^2}{\tau_{max}} \end{aligned}$$

We now need to choose some performance requirements to place on the system, which will allow us to determine the values of  $\zeta$  and  $\omega_0$ , and therefore the gains for the controller.

The percentage by which a second order system overshoots is:

$$O = e^{-\frac{\pi\zeta}{\sqrt{1-\zeta^2}}}$$

And the time it takes to reach the first peak in its output is:

$$T_P = \frac{\pi}{\omega_0\sqrt{1-\zeta^2}}$$

These can be rearranged to give us  $\zeta$  and  $\omega_0$  in terms of overshoot and time to peak:

$$\zeta = \sqrt{\frac{\ln^2(O)}{\pi^2 + \ln^2(O)}}$$

$$\omega_0 = \frac{\pi}{T_P\sqrt{1-\zeta^2}}$$

By default, kRPC uses the values  $O = 0.01$  and  $T_P = 3$ .

## 2.8.5 Corner Cases

### When sitting on the launchpad

In this situation, the autopilot cannot rotate the vessel. This means that the integral term in the controllers will build up to a large value. This is even true if the vessel is pointing in the correct direction, as small floating point variations in the computed error will also cause the integral term to increase. The integral terms are therefore fixed at zero to overcome this.

### When the available angular acceleration is zero

This could be caused, for example, by the reaction wheels on a vessel running out of electricity resulting in the vessel having no torque.

In this situation, the autopilot also has little or no control over the vessel. This means that the integral terms in the controllers will build up to a large value over time. This is overcome by fixing the integral terms to zero when the available angular acceleration falls below a small threshold.

This situation also causes an issue with the controller gain auto-tuning: as the available angular acceleration tends towards zero, the controller gains tend towards infinity. When it equals zero, the auto-tuning would cause a division by zero. Therefore, auto-tuning is also disabled when the available acceleration falls below the threshold. This leaves the controller gains at their current values until the available acceleration rises again.



## 3.1 C# Client

This client provides a C# API for interacting with a kRPC server. It is distributed as an assembly named `KRPC.Client.dll`.

### 3.1.1 Installing the Library

The C# client can be [installed using NuGet](#) or [downloaded from GitHub](#). Two versions of the client are provided: one compatible with .NET 4.5 and one for .NET 3.5.

You also need to [install Google.Protobuf](#) using NuGet.

---

**Note:** The copy of `Google.Protobuf.dll` in the `GameData` folder included with the kRPC server plugin is *not* the official release of this assembly. It is a modified version built for .NET 3.5 so that it works within KSP.

---

### 3.1.2 Connecting to the Server

To connect to a server, create a *Connection* object. All interaction with the server is done via this object. When constructed without any arguments, it will connect to the local machine on the default port numbers. You can specify different connection settings, and also a descriptive name for the connection, as follows:

```
using System;
using System.Net;
using KRPC.Client;
using KRPC.Client.Services.KRPC;

class Program {
    public static void Main() {
        using (var connection = new Connection(
            name: "My Example Program",
            address: IPAddress.Parse("192.168.0.10"),
            rpcPort: 1000,
            streamPort: 1001)) {
            var krpc = connection.KRPC();
            Console.WriteLine(krpc.GetStatus().Version);
        }
    }
}
```

The *Connection* object needs to be disposed of correctly when finished with, so that the network connection it manages can be released. This can be done with a `using` block (as in the example above) or by calling *Connection.Dispose* directly.

### 3.1.3 Calling Remote Procedures

The kRPC server provides *procedures* that a client can run. These procedures are arranged in groups called *services* to keep things organized. The functionality for the services are defined in the namespace `KRPC.Client.Services`. \*. For example, all of the functionality provided by the `SpaceCenter` service is contained in the namespace `KRPC.Client.Services.SpaceCenter`.

To interact with a service, you must first instantiate it. You can then call its methods and properties to invoke remote procedures. The following example demonstrates how to do this. It instantiates the `SpaceCenter` service and calls `KRPC.Client.Services.SpaceCenter.SpaceCenter.ActiveVessel` to get an object representing the active vessel (of type `KRPC.Client.Services.SpaceCenter.Vessel`). It sets the name of the vessel and then prints out its altitude:

```
using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class Program {
    public static void Main() {
        using (var connection = new Connection()) {
            var spaceCenter = connection.SpaceCenter();
            var vessel = spaceCenter.ActiveVessel;
            vessel.Name = "My Vessel";
            var flightInfo = vessel.Flight();
            Console.WriteLine(flightInfo.MeanAltitude);
        }
    }
}
```

### 3.1.4 Streaming Data from the Server

A common use case for kRPC is to continuously extract data from the game. The naive approach to do this would be to repeatedly call a remote procedure, such as in the following which repeatedly prints the position of the active vessel:

```
using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class Program {
    public static void Main() {
        var connection = new Connection();
        var spaceCenter = connection.SpaceCenter();
        var vessel = spaceCenter.ActiveVessel;
        var refFrame = vessel.Orbit.Body.ReferenceFrame;
        while (true)
            Console.WriteLine(vessel.Position(refFrame));
    }
}
```



This approach requires significant communication overhead as request/response messages are repeatedly sent between the client and server. kRPC provides a more efficient mechanism to achieve this, called *streams*.

A stream repeatedly executes a procedure on the server (with a fixed set of argument values) and sends the result to the client. It only requires a single message to be sent to the server to establish the stream, which will then continuously send data to the client until the stream is closed.

The following example does the same thing as above using streams:

```
using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class Program {
    public static void Main() {
        var connection = new Connection();
        var spaceCenter = connection.SpaceCenter();
        var vessel = spaceCenter.ActiveVessel;
        var refFrame = vessel.Orbit.Body.ReferenceFrame;
        var posStream = connection.AddStream(() => vessel.Position(refFrame));
        while (true)
            Console.WriteLine(posStream.Get());
    }
}
```

It calls *Connection.AddStream* once at the start of the program to create the stream, and then repeatedly prints the position returned by the stream. The stream is automatically closed when the client disconnects.

A stream can be created for any method call by calling *Connection.AddStream* and passing it a lambda expression that invokes the desired method. This lambda expression must take zero arguments and be either a method call expression or a parameter call expression. It returns a stream object of type *Stream*. The most recent value of the stream can be obtained by calling *Stream.Get*. A stream can be stopped and removed from the server by calling *Stream.Remove* on the stream object. All of a clients streams are automatically stopped when it disconnects.

### 3.1.5 Synchronizing with Stream Updates

A common use case for kRPC is to wait until the value returned by a method or attribute changes, and then take some action. kRPC provides two mechanisms to do this efficiently: *condition variables* and *callbacks*.

#### Condition Variables

Each stream has a condition variable associated with it, that is notified whenever the value of the stream changes. These can be used to block the current thread of execution until the value of the stream changes.

The following example waits until the abort button is pressed in game, by waiting for the value of *KRPC.Client.Services.SpaceCenter.Control.Abort* to change to true:

```
using System;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class Program {
    public static void Main() {
        var connection = new Connection();
        var spaceCenter = connection.SpaceCenter();
        var control = spaceCenter.ActiveVessel.Control;
        var abort = connection.AddStream(() => control.Abort);
    }
}
```

```
        lock (abort.Condition) {  
            while (!abort.Get())  
                abort.Wait();  
        }  
    }  
}
```

This code creates a stream, acquires a lock on the streams condition variable (by using a `lock` statement) and then repeatedly checks the value of `Abort`. It leaves the loop when it changes to true.

The body of the loop calls `Wait` on the stream, which causes the program to block until the value changes. This prevents the loop from ‘spinning’ and so it does not consume processing resources whilst waiting.

---

**Note:** The stream does not start receiving updates until the first call to `Wait`. This means that the example code will not miss any updates to the streams value, as it will have already locked the condition variable before the first stream update is received.

---

## Callbacks

Streams allow you to register callback functions that are called whenever the value of the stream changes. Callback functions should take a single argument, which is the new value of the stream, and should return nothing.

For example the following program registers two callbacks that are invoked when the value of `KRPC.Client.Services.SpaceCenter.Control.Abort` changes:

```
using System;  
using KRPC.Client;  
using KRPC.Client.Services.SpaceCenter;  
  
class Program {  
    public static void Main() {  
        var connection = new Connection();  
        var spaceCenter = connection.SpaceCenter();  
        var control = spaceCenter.ActiveVessel.Control;  
        var abort = connection.AddStream(() => control.Abort);  
  
        abort.AddCallback(  
            (bool x) => {  
                Console.WriteLine("Abort 1 called with a value of " + x);  
            }  
        );  
        abort.AddCallback(  
            (bool x) => {  
                Console.WriteLine("Abort 2 called with a value of " + x);  
            }  
        );  
        abort.Start();  
  
        // Keep the program running...  
        while (true) {  
        }  
    }  
}
```

---

**Note:** When a stream is created it does not start receiving updates until `Start` is called. This is implicitly called

when accessing the value of a stream, but as this example does not do this an explicit call to `Start` is required.

**Note:** The callbacks are registered before the call to `Start` so that stream updates are not missed.

**Note:** The callback function may be called from a different thread to that which created the stream. Any changes to shared state must therefore be protected with appropriate synchronization.

### 3.1.6 Custom Events

Some procedures return event objects of type *Event*. These allow you to wait until an event occurs, by calling *Event.Wait*. Under the hood, these are implemented using streams and condition variables.

Custom events can also be created. An expression API allows you to create code that runs on the server and these can be used to build a custom event. For example, the following creates the expression `MeanAltitude > 1000` and then creates an event that will be triggered when the expression returns true:

```
using System;
using KRPC.Client;
using KRPC.Client.Services.KRPC;
using KRPC.Client.Services.SpaceCenter;

class Program {
    public static void Main() {
        var connection = new Connection();
        var krpc = connection.KRPC();
        var spaceCenter = connection.SpaceCenter();
        var flight = spaceCenter.ActiveVessel.Flight();

        // Get the remote procedure call as a message object,
        // so it can be passed to the server
        var meanAltitude = Connection.GetCall(() => flight.MeanAltitude);

        // Create an expression on the server
        var expr = Expression.GreaterThan(connection,
            Expression.Call(connection, meanAltitude),
            Expression.ConstantDouble(connection, 1000));

        var evnt = krpc.AddEvent(expr);
        lock (evnt.Condition) {
            evnt.Wait();
            Console.WriteLine("Altitude reached 1000m");
        }
    }
}
```

### 3.1.7 Client API Reference

#### class `IConnection`

Interface implemented by the *Connection* class.

**class Connection**

A connection to the kRPC server. All interaction with kRPC is performed via an instance of this class.

**Connection** (*String name = ""*, *Net.IPAddress address = null*, *Int32 rpcPort = 50000*, *Int32 stream-Port = 50001*)  
Connect to a kRPC server.

**Parameters**

- **name** – A descriptive name for the connection. This is passed to the server and appears in the in-game server window.
- **address** – The address of the server to connect to. Defaults to 127.0.0.1.
- **rpc\_port** – The port number of the RPC Server. Defaults to 50000. This should match the RPC port number of the server you want to connect to.
- **stream\_port** – The port number of the Stream Server. Defaults to 50001. This should match the stream port number of the server you want to connect to.

*Stream<ReturnType>* **AddStream***<ReturnType>* (*LambdaExpression expression*)  
Create a new stream from the given lambda expression.

KRPC.Schema.KRPC.ProcedureCall **GetCall** (*LambdaExpression expression*)  
Returns a procedure call message for the given lambda expression. This allows descriptions of procedure calls to be passed to the server, for example when constructing custom events. See *Custom Events*.

void **Dispose** ()  
Closes the connection and frees the resources associated with it.

**class Stream<ReturnType>**

This class represents a stream. See *Streaming Data from the Server*.

Stream objects implement `GetHashCode`, `Equals`, `operator ==` and `operator !=` such that two stream objects are equal if they are bound to the same stream on the server.

void **Start** (*Boolean wait = true*)  
Starts the stream. When a stream is created by calling *Connection.AddStream* it does not start sending updates to the client until this method is called.

If *wait* is true, this method will block until at least one update has been received from the server.

If *wait* is false, the method starts the stream and returns immediately. Subsequent calls to *Get* may raise an *InvalidOperationException* if the stream does not yet contain a value.

*ReturnType* **Get** ()  
Returns the most recent value for the stream. If executing the remote procedure for the stream throws an exception, calling this method will rethrow the exception. Raises an *InvalidOperationException* if no update has been received from the server.

If the stream has not been started this method calls *Start(true)* to start the stream and wait until at least one update has been received.

object **Condition** { *get*; }  
A condition variable that is notified (using *Monitor.PulseAll*) whenever the value of the stream changes.

void **Wait** (*Double timeout = -1*)  
This method blocks until the value of the stream changes or the operation times out.

The streams condition variable must be locked before calling this method.

If *timeout* is specified and is greater than or equal to 0, it is the timeout in seconds for the operation.

If the stream has not been started this method calls `Start(false)` to start the stream (without waiting for at least one update to be received).

void **AddCallback** (Action<ReturnType> *callback*)

Adds a callback function that is invoked whenever the value of the stream changes. The callback function should take one argument, which is passed the new value of the stream.

---

**Note:** The callback function may be called from a different thread to that which created the stream. Any changes to shared state must therefore be protected with appropriate synchronization.

---

void **Remove** ()

Removes the stream from the server.

### class Event

This class represents an event. See *Custom Events*. It is wrapper around a `Boolean` that indicates when the event occurs.

Event objects implement `GetHashCode`, `Equals`, `operator ==` and `operator !=` such that two event objects are equal if they are bound to the same underlying stream on the server.

void **Start** ()

Starts the event. When an event is created, it will not receive updates from the server until this method is called.

object **Condition** { **get**; }

The condition variable that is notified (using `Monitor.PulseAll`) whenever the event occurs.

void **Wait** (Double *timeout* = -1)

This method blocks until the event occurs or the operation times out.

The events condition variable must be locked before calling this method.

If *timeout* is specified and is greater than or equal to 0, it is the timeout in seconds for the operation.

If the event has not been started this method calls `Start()` to start the underlying stream.

void **AddCallback** (Action *callback*)

Adds a callback function that is invoked whenever the event occurs. The callback function should be a function that takes zero arguments.

void **Remove** ()

Removes the event from the server.

Stream<Boolean> **Stream** { **get**; }

Returns the underlying stream for the event.

## 3.2 KRPC API

### 3.2.1 KRPC

None None None None

#### class KRPC

Main kRPC service, used by clients to interact with basic server functionality.

Byte[] **GetClientID** ()

Returns the identifier for the current client.

**String GetClientName ()**

Returns the name of the current client. This is an empty string if the client has no name.

**ILog<Tuple<Byte[], String, String>> Clients { get; }**

A list of RPC clients that are currently connected to the server. Each entry in the list is a clients identifier, name and address.

**KRPC.Schema.KRPC.Status GetStatus ()**

Returns some information about the server, such as the version.

**KRPC.Schema.KRPC.Services GetServices ()**

Returns information on all services, procedures, classes, properties etc. provided by the server. Can be used by client libraries to automatically create functionality such as stubs.

**GameScene CurrentGameScene { get; }**

Get the current game scene.

**Boolean Paused { get; set; }**

Whether the game is paused.

**enum GameScene**

The game scene. See *KRPC.CurrentGameScene*.

**SpaceCenter**

The game scene showing the Kerbal Space Center buildings.

**Flight**

The game scene showing a vessel in flight (or on the launchpad/runway).

**TrackingStation**

The tracking station.

**EditorVAB**

The Vehicle Assembly Building.

**EditorSPH**

The Space Plane Hangar.

**class InvalidOperationException**

A method call was made to a method that is invalid given the current state of the object.

**class ArgumentException**

A method was invoked where at least one of the passed arguments does not meet the parameter specification of the method.

**class ArgumentNullException**

A null reference was passed to a method that does not accept it as a valid argument.

**class ArgumentOutOfRangeException**

The value of an argument is outside the allowable range of values as defined by the invoked method.

## 3.2.2 Expressions

**class Expression**

A server side expression.

*static Expression ConstantDouble (IConnection connection, Double value)*

A constant value of type double.

**Parameters**

*static Expression* **ConstantFloat** (*ICollection connection, Single value*)

A constant value of type float.

**Parameters**

*static Expression* **ConstantInt** (*ICollection connection, Int32 value*)

A constant value of type int.

**Parameters**

*static Expression* **ConstantString** (*ICollection connection, String value*)

A constant value of type string.

**Parameters**

*static Expression* **Call** (*ICollection connection, KRPC.Schema.KRPC.ProcedureCall call*)

An RPC call.

**Parameters**

*static Expression* **Equal** (*ICollection connection, Expression arg0, Expression arg1*)

Equality comparison.

**Parameters**

*static Expression* **NotEqual** (*ICollection connection, Expression arg0, Expression arg1*)

Inequality comparison.

**Parameters**

*static Expression* **GreaterThan** (*ICollection connection, Expression arg0, Expression arg1*)

Greater than numerical comparison.

**Parameters**

*static Expression* **GreaterThanOrEqual** (*ICollection connection, Expression arg0, Expression arg1*)

Greater than or equal numerical comparison.

**Parameters**

*static Expression* **LessThan** (*ICollection connection, Expression arg0, Expression arg1*)

Less than numerical comparison.

**Parameters**

*static Expression* **LessThanOrEqual** (*ICollection connection, Expression arg0, Expression arg1*)

Less than or equal numerical comparison.

**Parameters**

*static Expression* **And** (*ICollection connection, Expression arg0, Expression arg1*)

Boolean and operator.

**Parameters**

*static Expression* **Or** (*ICollection connection, Expression arg0, Expression arg1*)

Boolean or operator.

**Parameters**

*static Expression* **ExclusiveOr** (*ICollection connection, Expression arg0, Expression arg1*)

Boolean exclusive-or operator.

**Parameters**

*static Expression* **Not** (*ICollection connection, Expression arg*)  
Boolean negation operator.

**Parameters**

*static Expression* **Add** (*ICollection connection, Expression arg0, Expression arg1*)  
Numerical addition.

**Parameters**

*static Expression* **Subtract** (*ICollection connection, Expression arg0, Expression arg1*)  
Numerical subtraction.

**Parameters**

*static Expression* **Multiply** (*ICollection connection, Expression arg0, Expression arg1*)  
Numerical multiplication.

**Parameters**

*static Expression* **Divide** (*ICollection connection, Expression arg0, Expression arg1*)  
Numerical division.

**Parameters**

*static Expression* **Modulo** (*ICollection connection, Expression arg0, Expression arg1*)  
Numerical modulo operator.

**Parameters**

**Returns** The remainder of arg0 divided by arg1

*static Expression* **Power** (*ICollection connection, Expression arg0, Expression arg1*)  
Numerical power operator.

**Parameters**

**Returns** arg0 raised to the power of arg1

*static Expression* **LeftShift** (*ICollection connection, Expression arg0, Expression arg1*)  
Bitwise left shift.

**Parameters**

*static Expression* **RightShift** (*ICollection connection, Expression arg0, Expression arg1*)  
Bitwise right shift.

**Parameters**

*static Expression* **ToDouble** (*ICollection connection, Expression arg*)  
Convert to a double type.

**Parameters**

*static Expression* **ToFloat** (*ICollection connection, Expression arg*)  
Convert to a float type.

**Parameters**

*static Expression* **ToInt** (*ICollection connection, Expression arg*)  
Convert to an int type.

**Parameters**



## 3.3 SpaceCenter API

### 3.3.1 SpaceCenter

#### **class SpaceCenter**

Provides functionality to interact with Kerbal Space Program. This includes controlling the active vessel, managing its resources, planning maneuver nodes and auto-piloting.

*Vessel* **ActiveVessel** { **get**; **set**; }

The currently active vessel.

*IList<Vessel>* **Vessels** { **get**; }

A list of all the vessels in the game.

*IDictionary<String, CelestialBody>* **Bodies** { **get**; }

A dictionary of all celestial bodies (planets, moons, etc.) in the game, keyed by the name of the body.

*CelestialBody* **TargetBody** { **get**; **set**; }

The currently targeted celestial body.

*Vessel* **TargetVessel** { **get**; **set**; }

The currently targeted vessel.

*DockingPort* **TargetDockingPort** { **get**; **set**; }

The currently targeted docking port.

void **ClearTarget** ()

Clears the current target.

*IList<String>* **LaunchableVessels** (*String* *craftDirectory*)

Returns a list of vessels from the given *craftDirectory* that can be launched.

#### **Parameters**

- **craftDirectory** – Name of the directory in the current saves “Ships” directory. For example "VAB" or "SPH".

void **LaunchVessel** (*String* *craftDirectory*, *String* *name*, *String* *launchSite*)

Launch a vessel.

#### **Parameters**

- **craftDirectory** – Name of the directory in the current saves “Ships” directory, that contains the craft file. For example "VAB" or "SPH".
- **name** – Name of the vessel to launch. This is the name of the “.craft” file in the save directory, without the “.craft” file extension.
- **launchSite** – Name of the launch site. For example "LaunchPad" or "Runway".

void **LaunchVesselFromVAB** (*String* *name*)

Launch a new vessel from the VAB onto the launchpad.

#### **Parameters**

- **name** – Name of the vessel to launch.

---

**Note:** This is equivalent to calling *SpaceCenter.LaunchVessel* with the craft directory set to “VAB” and the launch site set to “LaunchPad”.

---

void **LaunchVesselFromSPH** (*String name*)  
Launch a new vessel from the SPH onto the runway.

**Parameters**

- **name** – Name of the vessel to launch.

---

**Note:** This is equivalent to calling *SpaceCenter.LaunchVessel* with the craft directory set to “SPH” and the launch site set to “Runway”.

---

void **Save** (*String name*)  
Save the game with a given name. This will create a save file called *name.sfs* in the folder of the current save game.

**Parameters**

void **Load** (*String name*)  
Load the game with the given name. This will create a load a save file called *name.sfs* from the folder of the current save game.

**Parameters**

void **Quicksave** ()  
Save a quicksave.

---

**Note:** This is the same as calling *SpaceCenter.Save* with the name “quicksave”.

---

void **Quickload** ()  
Load a quicksave.

---

**Note:** This is the same as calling *SpaceCenter.Load* with the name “quicksave”.

---

**Boolean UIVisible** { **get**; **set**; }  
Whether the UI is visible.

**Boolean Navball** { **get**; **set**; }  
Whether the navball is visible.

**Double UT** { **get**; }  
The current universal time in seconds.

**Double G** { **get**; }  
The value of the *gravitational constant* *G* in  $N(m/kg)^2$ .

**Single WarpRate** { **get**; }  
The current warp rate. This is the rate at which time is passing for either on-rails or physical time warp. For example, a value of 10 means time is passing 10x faster than normal. Returns 1 if time warp is not active.

**Single WarpFactor** { **get**; }  
The current warp factor. This is the index of the rate at which time is passing for either regular “on-rails” or physical time warp. Returns 0 if time warp is not active. When in on-rails time warp, this is equal to *SpaceCenter.RailsWarpFactor*, and in physics time warp, this is equal to *SpaceCenter.PhysicsWarpFactor*.

**Int32 RailsWarpFactor { get; set; }**

The time warp rate, using regular “on-rails” time warp. A value between 0 and 7 inclusive. 0 means no time warp. Returns 0 if physical time warp is active.

If requested time warp factor cannot be set, it will be set to the next lowest possible value. For example, if the vessel is too close to a planet. See [the KSP wiki](#) for details.

**Int32 PhysicsWarpFactor { get; set; }**

The physical time warp rate. A value between 0 and 3 inclusive. 0 means no time warp. Returns 0 if regular “on-rails” time warp is active.

**Boolean CanRailsWarpAt (Int32 factor = 1)**

Returns `true` if regular “on-rails” time warp can be used, at the specified warp *factor*. The maximum time warp rate is limited by various things, including how close the active vessel is to a planet. See [the KSP wiki](#) for details.

#### Parameters

- **factor** – The warp factor to check.

**Int32 MaximumRailsWarpFactor { get; }**

The current maximum regular “on-rails” warp factor that can be set. A value between 0 and 7 inclusive. See [the KSP wiki](#) for details.

**void WarpTo (Double ut, Single maxRailsRate = 100000.0, Single maxPhysicsRate = 2.0)**

Uses time acceleration to warp forward to a time in the future, specified by universal time *ut*. This call blocks until the desired time is reached. Uses regular “on-rails” or physical time warp as appropriate. For example, physical time warp is used when the active vessel is traveling through an atmosphere. When using regular “on-rails” time warp, the warp rate is limited by *maxRailsRate*, and when using physical time warp, the warp rate is limited by *maxPhysicsRate*.

#### Parameters

- **ut** – The universal time to warp to, in seconds.
- **maxRailsRate** – The maximum warp rate in regular “on-rails” time warp.
- **maxPhysicsRate** – The maximum warp rate in physical time warp.

**Returns** When the time warp is complete.

**Tuple<Double, Double, Double> TransformPosition (Tuple<Double, Double, Double> position, ReferenceFrame from, ReferenceFrame to)**

Converts a position from one reference frame to another.

#### Parameters

- **position** – Position, as a vector, in reference frame *from*.
- **from** – The reference frame that the position is in.
- **to** – The reference frame to convert the position to.

**Returns** The corresponding position, as a vector, in reference frame *to*.

**Tuple<Double, Double, Double> TransformDirection (Tuple<Double, Double, Double> direction, ReferenceFrame from, ReferenceFrame to)**

Converts a direction from one reference frame to another.

#### Parameters

- **direction** – Direction, as a vector, in reference frame *from*.
- **from** – The reference frame that the direction is in.

- **to** – The reference frame to convert the direction to.

**Returns** The corresponding direction, as a vector, in reference frame *to*.

`Tuple<Double, Double, Double, Double> TransformRotation (Tuple<Double, Double, Double, Double> rotation, ReferenceFrame from, ReferenceFrame to)`

Converts a rotation from one reference frame to another.

**Parameters**

- **rotation** – Rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *from*.
- **from** – The reference frame that the rotation is in.
- **to** – The reference frame to convert the rotation to.

**Returns** The corresponding rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *to*.

`Tuple<Double, Double, Double> TransformVelocity (Tuple<Double, Double, Double> position, Tuple<Double, Double, Double> velocity, ReferenceFrame from, ReferenceFrame to)`

Converts a velocity (acting at the specified position) from one reference frame to another. The position is required to take the relative angular velocity of the reference frames into account.

**Parameters**

- **position** – Position, as a vector, in reference frame *from*.
- **velocity** – Velocity, as a vector that points in the direction of travel and whose magnitude is the speed in meters per second, in reference frame *from*.
- **from** – The reference frame that the position and velocity are in.
- **to** – The reference frame to convert the velocity to.

**Returns** The corresponding velocity, as a vector, in reference frame *to*.

`Boolean FARAvailable { get; }`

Whether Ferram Aerospace Research is installed.

`WarpMode WarpMode { get; }`

The current time warp mode. Returns *WarpMode.None* if time warp is not active, *WarpMode.Rails* if regular “on-rails” time warp is active, or *WarpMode.Physics* if physical time warp is active.

`Camera Camera { get; }`

An object that can be used to control the camera.

`WaypointManager WaypointManager { get; }`

The waypoint manager.

`ContractManager ContractManager { get; }`

The contract manager.

**enum WarpMode**

The time warp mode. Returned by *WarpMode*

**Rails**

Time warp is active, and in regular “on-rails” mode.

**Physics**

Time warp is active, and in physical time warp mode.

**None**

Time warp is not active.

### 3.3.2 Vessel

#### class Vessel

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources. Created using *SpaceCenter.ActiveVessel* or *SpaceCenter.Vessels*.

**String Name { get; set; }**

The name of the vessel.

**VesselType Type { get; set; }**

The type of the vessel.

**VesselSituation Situation { get; }**

The situation the vessel is in.

**Boolean Recoverable { get; }**

Whether the vessel is recoverable.

**void Recover ()**

Recover the vessel.

**Double MET { get; }**

The mission elapsed time in seconds.

**String Biome { get; }**

The name of the biome the vessel is currently in.

**Flight Flight (ReferenceFrame referenceFrame = null)**

Returns a *Flight* object that can be used to get flight telemetry for the vessel, in the specified reference frame.

#### Parameters

- **referenceFrame** – Reference frame. Defaults to the vessel's surface reference frame (*Vessel.SurfaceReferenceFrame*).

---

**Note:** When this is called with no arguments, the vessel's surface reference frame is used. This reference frame moves with the vessel, therefore velocities and speeds returned by the flight object will be zero. See the *reference frames tutorial* for examples of getting *the orbital and surface speeds of a vessel*.

---

**Orbit Orbit { get; }**

The current orbit of the vessel.

**Control Control { get; }**

Returns a *Control* object that can be used to manipulate the vessel's control inputs. For example, its pitch/yaw/roll controls, RCS and thrust.

**Comms Comms { get; }**

Returns a *Comms* object that can be used to interact with CommNet for this vessel.

**AutoPilot AutoPilot { get; }**

An *AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

**Resources Resources { get; }**

A *Resources* object, that can be used to get information about resources stored in the vessel.

**Resources ResourcesInDecoupleStage (Int32 stage, Boolean cumulative = True)**

Returns a *Resources* object, that can be used to get information about resources stored in a given *stage*.

#### Parameters

- **stage** – Get resources for parts that are decoupled in this stage.
- **cumulative** – When `false`, returns the resources for parts decoupled in just the given stage. When `true` returns the resources decoupled in the given stage and all subsequent stages combined.

---

**Note:** For details on stage numbering, see the discussion on *Staging*.

---

**Parts** **Parts** { **get**; }

A *Parts* object, that can used to interact with the parts that make up this vessel.

**Single** **Mass** { **get**; }

The total mass of the vessel, including resources, in kg.

**Single** **DryMass** { **get**; }

The total mass of the vessel, excluding resources, in kg.

**Single** **Thrust** { **get**; }

The total thrust currently being produced by the vessel's engines, in Newtons. This is computed by summing *Engine.Thrust* for every engine in the vessel.

**Single** **AvailableThrust** { **get**; }

Gets the total available thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *Engine.AvailableThrust* for every active engine in the vessel.

**Single** **MaxThrust** { **get**; }

The total maximum thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *Engine.MaxThrust* for every active engine.

**Single** **MaxVacuumThrust** { **get**; }

The total maximum thrust that can be produced by the vessel's active engines when the vessel is in a vacuum, in Newtons. This is computed by summing *Engine.MaxVacuumThrust* for every active engine.

**Single** **SpecificImpulse** { **get**; }

The combined specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

**Single** **VacuumSpecificImpulse** { **get**; }

The combined vacuum specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

**Single** **KerbinSeaLevelSpecificImpulse** { **get**; }

The combined specific impulse of all active engines at sea level on Kerbin, in seconds. This is computed using the formula [described here](#).

**Tuple**<**Double**, **Double**, **Double**> **MomentOfInertia** { **get**; }

The moment of inertia of the vessel around its center of mass in  $kg.m^2$ . The inertia values in the returned 3-tuple are around the pitch, roll and yaw directions respectively. This corresponds to the vessels reference frame (*ReferenceFrame*).

**IList**<**Double**> **InertiaTensor** { **get**; }

The inertia tensor of the vessel around its center of mass, in the vessels reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

**Tuple**<**Tuple**<**Double**, **Double**, **Double**>, **Tuple**<**Double**, **Double**, **Double**>> **AvailableTorque** { **get**; }

The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gimbaled engines and aerodynamic control surfaces. Returns the torques in  $N.m$  around each of the coordi-

nate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**`Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> AvailableReactionWheelTorque { get; }`**

The maximum torque that the currently active and powered reaction wheels can generate. Returns the torques in *N.m* around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**`Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> AvailableRCSTorque { get; }`**

The maximum torque that the currently active RCS thrusters can generate. Returns the torques in *N.m* around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**`Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> AvailableEngineTorque { get; }`**

The maximum torque that the currently active and gimbaled engines can generate. Returns the torques in *N.m* around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**`Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> AvailableControlSurfaceTorque { get; }`**

The maximum torque that the aerodynamic control surfaces can generate. Returns the torques in *N.m* around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**`Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> AvailableOtherTorque { get; }`**

The maximum torque that parts (excluding reaction wheels, gimbaled engines, RCS and control surfaces) can generate. Returns the torques in *N.m* around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**`ReferenceFrame ReferenceFrame { get; }`**

The reference frame that is fixed relative to the vessel, and orientated with the vessel.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel.
- The x-axis points out to the right of the vessel.
- The y-axis points in the forward direction of the vessel.
- The z-axis points out of the bottom off the vessel.

**`ReferenceFrame OrbitalReferenceFrame { get; }`**

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

---

**Note:** Be careful not to confuse this with ‘orbit’ mode on the navball.

---

**`ReferenceFrame SurfaceReferenceFrame { get; }`**

The reference frame that is fixed relative to the vessel, and orientated with the surface of the body being orbited.

- The origin is at the center of mass of the vessel.

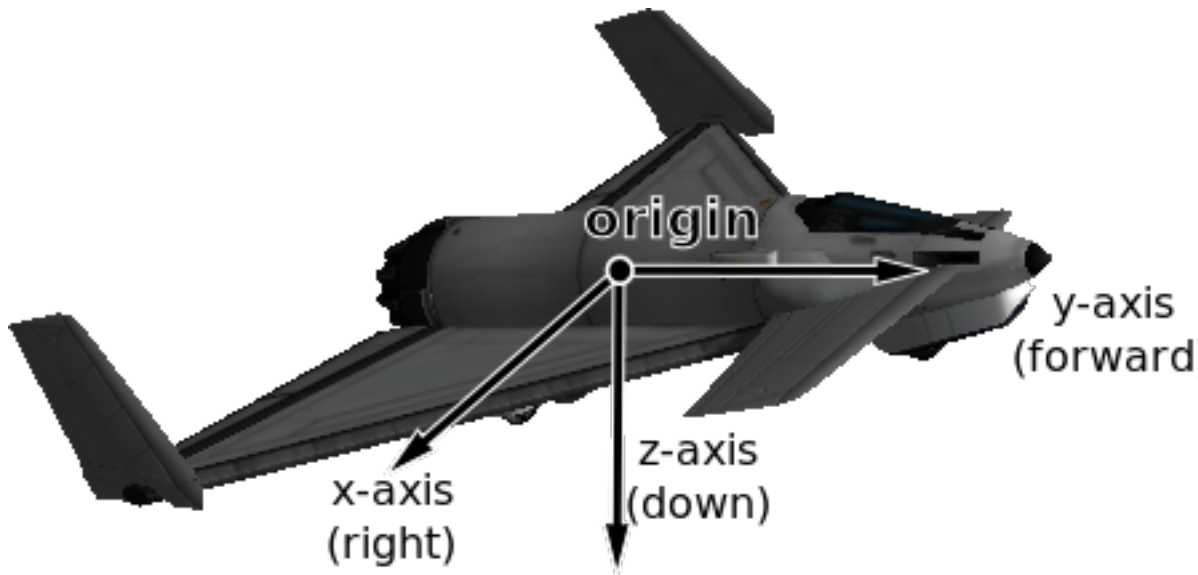


Fig. 3.1: Vessel reference frame origin and axes for the Aegis 3A aircraft

- The axes rotate with the north and up directions on the surface of the body.
- The x-axis points in the [zenith](#) direction (upwards, normal to the body being orbited, from the center of the body towards the center of mass of the vessel).
- The y-axis points northwards towards the [astronomical horizon](#) (north, and tangential to the surface of the body – the direction in which a compass would point when on the surface).
- The z-axis points eastwards towards the [astronomical horizon](#) (east, and tangential to the surface of the body – east on a compass when on the surface).

---

**Note:** Be careful not to confuse this with ‘surface’ mode on the navball.

---

*ReferenceFrame* **SurfaceVelocityReferenceFrame** { **get**; }

The reference frame that is fixed relative to the vessel, and orientated with the velocity vector of the vessel relative to the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel’s velocity vector.
- The y-axis points in the direction of the vessel’s velocity vector, relative to the surface of the body being orbited.
- The z-axis is in the plane of the [astronomical horizon](#).
- The x-axis is orthogonal to the other two axes.

[Tuple](#)<[Double](#), [Double](#), [Double](#)> **Position** (*ReferenceFrame* referenceFrame)

The position of the center of mass of the vessel, in the given reference frame.

#### Parameters

- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.



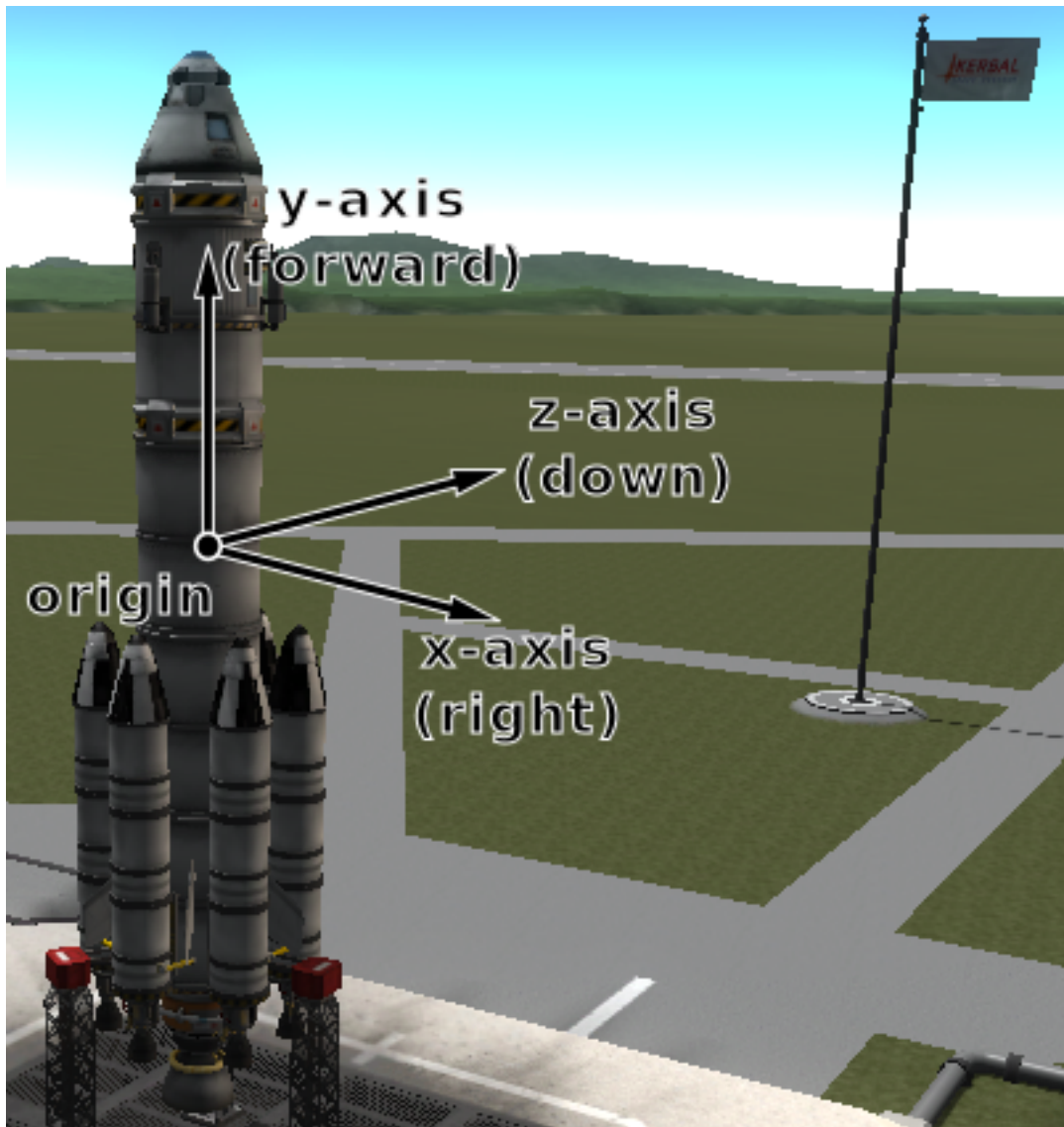


Fig. 3.2: Vessel reference frame origin and axes for the Kerbal-X rocket

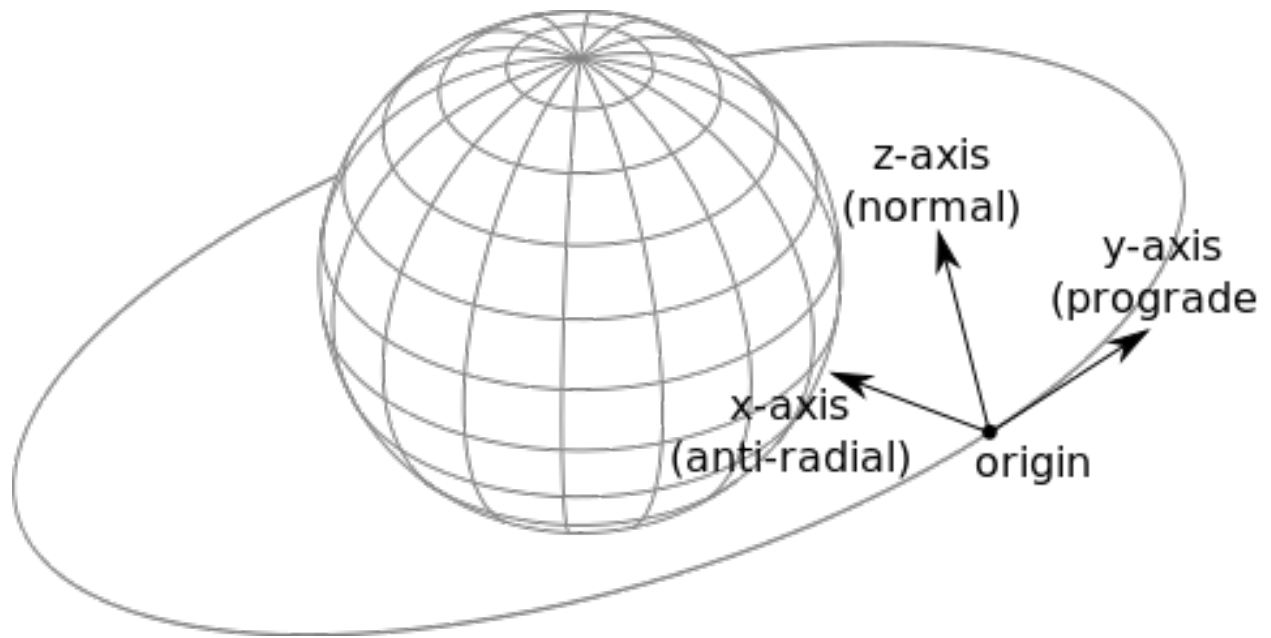


Fig. 3.3: Vessel orbital reference frame origin and axes

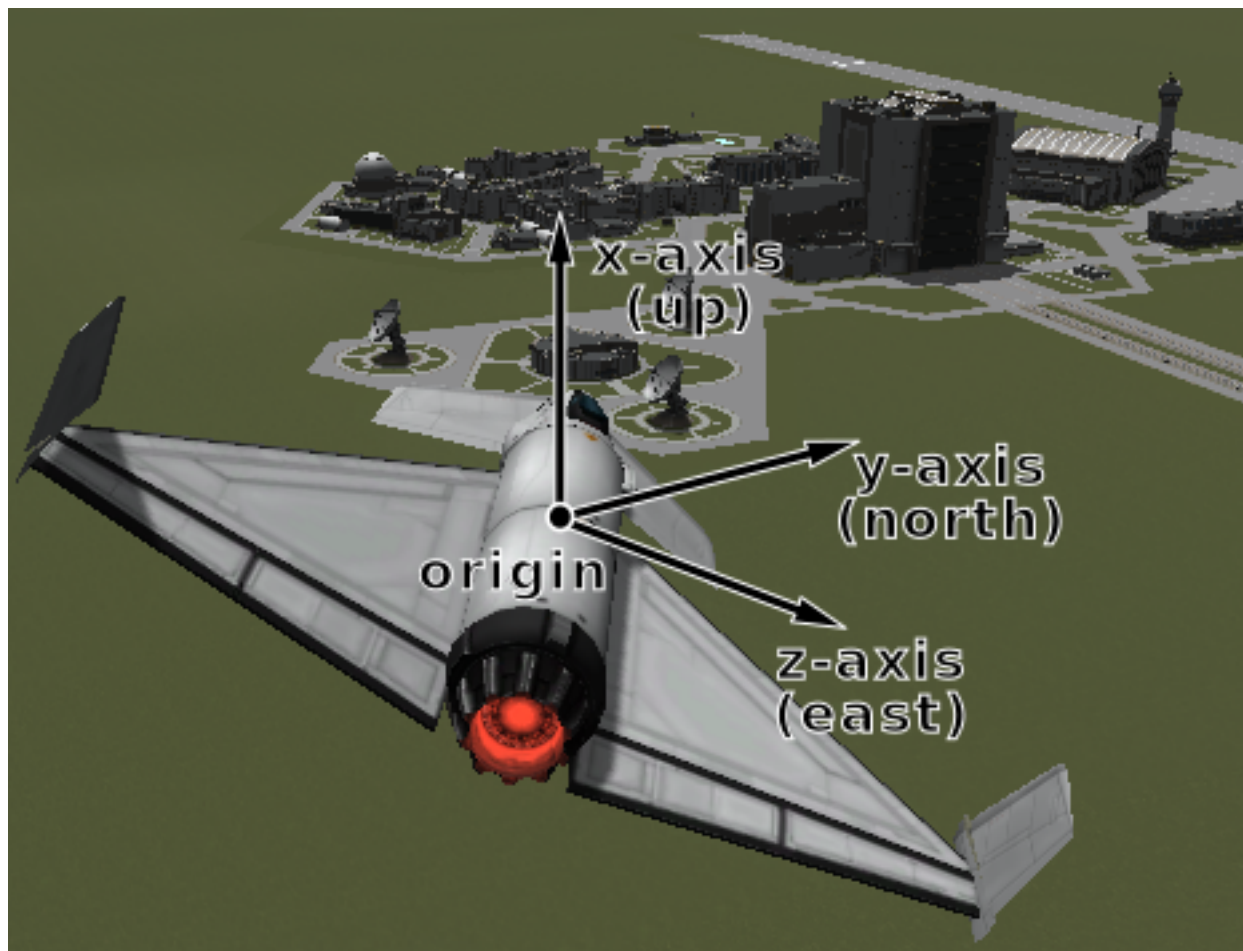


Fig. 3.4: Vessel surface reference frame origin and axes

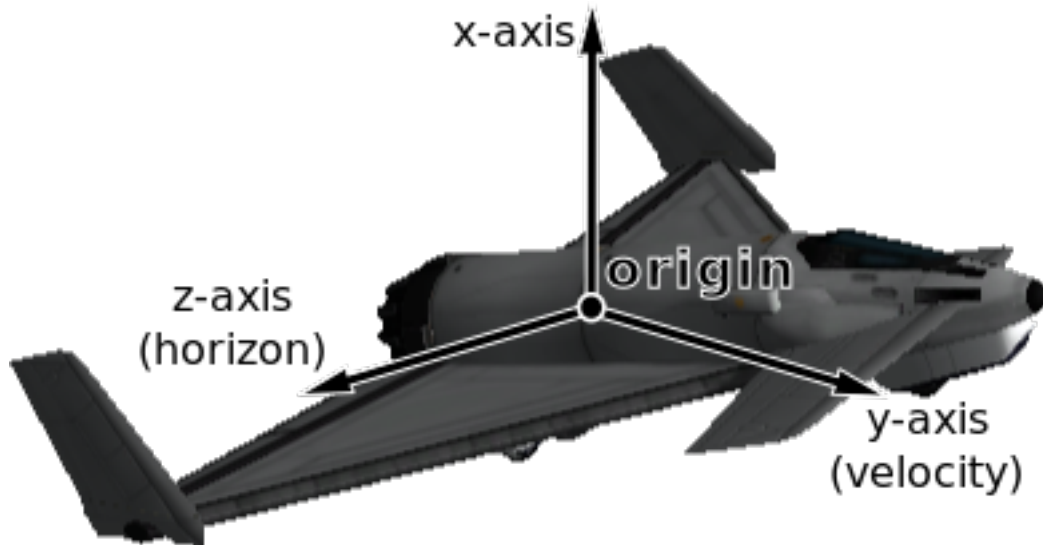


Fig. 3.5: Vessel surface velocity reference frame origin and axes

`Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>>` **BoundingBox** (*ReferenceFrame referenceFrame*)

The axis-aligned bounding box of the vessel in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

`Tuple<Double, Double, Double>` **Velocity** (*ReferenceFrame referenceFrame*)

The velocity of the center of mass of the vessel, in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

`Tuple<Double, Double, Double, Double>` **Rotation** (*ReferenceFrame referenceFrame*)

The rotation of the vessel, in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

`Tuple<Double, Double, Double>` **Direction** (*ReferenceFrame referenceFrame*)

The direction in which the vessel is pointing, in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

`Tuple<Double, Double, Double>` **AngularVelocity** (*ReferenceFrame referenceFrame*)

The angular velocity of the vessel, in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the vessel, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

**enum VesselType**

The type of a vessel. See *Vessel.Type*.

**Base**

Base.

**Debris**

Debris.

**Lander**

Lander.

**Plane**

Plane.

**Probe**

Probe.

**Relay**

Relay.

**Rover**

Rover.

**Ship**

Ship.

**Station**

Station.

**enum VesselSituation**

The situation a vessel is in. See *Vessel.Situation*.

**Docked**

Vessel is docked to another.

**Escaping**

Escaping.

**Flying**

Vessel is flying through an atmosphere.

**Landed**

Vessel is landed on the surface of a body.

**Orbiting**

Vessel is orbiting a body.

**PreLaunch**

Vessel is awaiting launch.

**Splashed**

Vessel has splashed down in an ocean.

**SubOrbital**

Vessel is on a sub-orbital trajectory.

### 3.3.3 CelestialBody

#### class CelestialBody

Represents a celestial body (such as a planet or moon). See *SpaceCenter.Bodies*.

**String Name { get; }**

The name of the body.

**IList<CelestialBody> Satellites { get; }**

A list of celestial bodies that are in orbit around this celestial body.

**Orbit Orbit { get; }**

The orbit of the body.

**Single Mass { get; }**

The mass of the body, in kilograms.

**Single GravitationalParameter { get; }**

The standard gravitational parameter of the body in  $m^3 s^{-2}$ .

**Single SurfaceGravity { get; }**

The acceleration due to gravity at sea level (mean altitude) on the body, in  $m/s^2$ .

**Single RotationalPeriod { get; }**

The sidereal rotational period of the body, in seconds.

**Single RotationalSpeed { get; }**

The rotational speed of the body, in radians per second.

**Double RotationAngle { get; }**

The current rotation angle of the body, in radians. A value between 0 and  $2\pi$

**Double InitialRotation { get; }**

The initial rotation angle of the body (at UT 0), in radians. A value between 0 and  $2\pi$

**Single EquatorialRadius { get; }**

The equatorial radius of the body, in meters.

**Double SurfaceHeight (Double latitude, Double longitude)**

The height of the surface relative to mean sea level, in meters, at the given position. When over water this is equal to 0.

#### Parameters

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.

**Double BedrockHeight (Double latitude, Double longitude)**

The height of the surface relative to mean sea level, in meters, at the given position. When over water, this is the height of the sea-bed and is therefore negative value.

#### Parameters

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.

**Tuple<Double, Double, Double> MSLPosition (Double latitude, Double longitude, ReferenceFrame referenceFrame)**

The position at mean sea level at the given latitude and longitude, in the given reference frame.

#### Parameters

- **latitude** – Latitude in degrees.

- **longitude** – Longitude in degrees.
- **referenceFrame** – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Tuple<Double, Double, Double> SurfacePosition** (*Double latitude, Double longitude, ReferenceFrame referenceFrame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position of the surface of the water.

**Parameters**

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.
- **referenceFrame** – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Tuple<Double, Double, Double> BedrockPosition** (*Double latitude, Double longitude, ReferenceFrame referenceFrame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position at the bottom of the sea-bed.

**Parameters**

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.
- **referenceFrame** – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Tuple<Double, Double, Double> PositionAtAltitude** (*Double latitude, Double longitude, Double altitude, ReferenceFrame referenceFrame*)

The position at the given latitude, longitude and altitude, in the given reference frame.

**Parameters**

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.
- **altitude** – Altitude in meters above sea level.
- **referenceFrame** – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Double AltitudeAtPosition** (*Tuple<Double, Double, Double> position, ReferenceFrame referenceFrame*)

The altitude, in meters, of the given position in the given reference frame.

**Parameters**

- **position** – Position as a vector.
- **referenceFrame** – Reference frame for the position vector.

**Double LatitudeAtPosition** (*Tuple<Double, Double, Double> position, ReferenceFrame referenceFrame*)

The latitude of the given position, in the given reference frame.

**Parameters**

- **position** – Position as a vector.
- **referenceFrame** – Reference frame for the position vector.

**Double LongitudeAtPosition** (*Tuple<Double, Double, Double> position, ReferenceFrame referenceFrame*)

The longitude of the given position, in the given reference frame.

**Parameters**

- **position** – Position as a vector.
- **referenceFrame** – Reference frame for the position vector.

**Single SphereOfInfluence** { **get**; }

The radius of the sphere of influence of the body, in meters.

**Boolean HasAtmosphere** { **get**; }

true if the body has an atmosphere.

**Single AtmosphereDepth** { **get**; }

The depth of the atmosphere, in meters.

**Double AtmosphericDensityAtPosition** (*Tuple<Double, Double, Double> position, ReferenceFrame referenceFrame*)

The atmospheric density at the given position, in  $kg/m^3$ , in the given reference frame.

**Parameters**

- **position** – The position vector at which to measure the density.
- **referenceFrame** – Reference frame that the position vector is in.

**Boolean HasAtmosphericOxygen** { **get**; }

true if there is oxygen in the atmosphere, required for air-breathing engines.

**Double TemperatureAt** (*Tuple<Double, Double, Double> position, ReferenceFrame referenceFrame*)

The temperature on the body at the given position, in the given reference frame.

**Parameters**

- **position** – Position as a vector.
- **referenceFrame** – The reference frame that the position is in.

---

**Note:** This calculation is performed using the bodies current position, which means that the value could be wrong if you want to know the temperature in the far future.

---

**Double DensityAt** (*Double altitude*)

Gets the air density, in  $kg/m^3$ , for the specified altitude above sea level, in meters.

**Parameters**

---

**Note:** This is an approximation, because actual calculations, taking sun exposure into account to compute air temperature, require us to know the exact point on the body where the density is to be computed (knowing the altitude is not enough). However, the difference is small for high altitudes, so it makes very little difference for trajectory prediction.

---

**Double PressureAt** (*Double altitude*)

Gets the air pressure, in Pascals, for the specified altitude above sea level, in meters.

**Parameters**

`ISet<String> Biomes { get; }`

The biomes present on this body.

`String BiomeAt (Double latitude, Double longitude)`

The biome at the given latitude and longitude, in degrees.

**Parameters**

`Single FlyingHighAltitudeThreshold { get; }`

The altitude, in meters, above which a vessel is considered to be flying “high” when doing science.

`Single SpaceHighAltitudeThreshold { get; }`

The altitude, in meters, above which a vessel is considered to be in “high” space when doing science.

`ReferenceFrame ReferenceFrame { get; }`

The reference frame that is fixed relative to the celestial body.

- The origin is at the center of the body.
- The axes rotate with the body.
- The x-axis points from the center of the body towards the intersection of the prime meridian and equator (the position at 0° longitude, 0° latitude).
- The y-axis points from the center of the body towards the north pole.
- The z-axis points from the center of the body towards the equator at 90°E longitude.

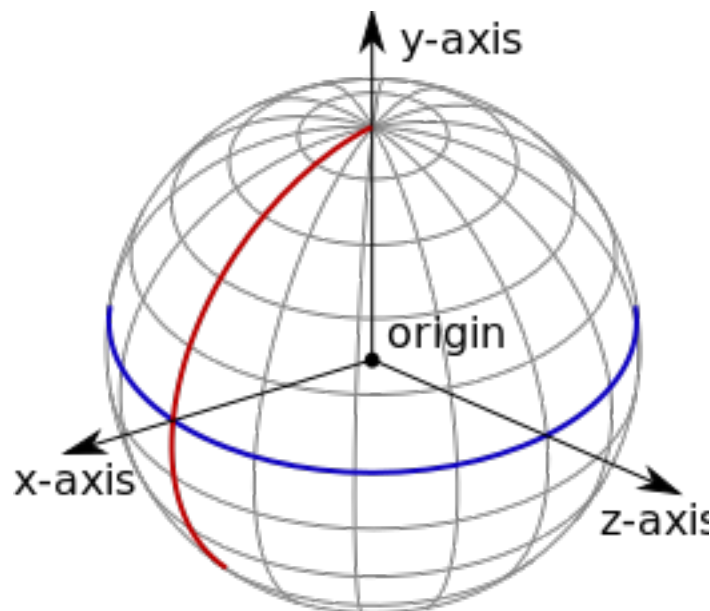


Fig. 3.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

`ReferenceFrame NonRotatingReferenceFrame { get; }`

The reference frame that is fixed relative to this celestial body, and orientated in a fixed direction (it does not rotate with the body).

- The origin is at the center of the body.
- The axes do not rotate.
- The x-axis points in an arbitrary direction through the equator.



- The y-axis points from the center of the body towards the north pole.
- The z-axis points in an arbitrary direction through the equator.

*ReferenceFrame* **OrbitalReferenceFrame** { **get**; }

The reference frame that is fixed relative to this celestial body, but orientated with the body's orbital prograde/normal/radial directions.

- The origin is at the center of the body.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

*Tuple*<Double, Double, Double> **Position** (*ReferenceFrame* *referenceFrame*)

The position of the center of the body, in the specified reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

*Tuple*<Double, Double, Double> **Velocity** (*ReferenceFrame* *referenceFrame*)

The linear velocity of the body, in the specified reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

*Tuple*<Double, Double, Double, Double> **Rotation** (*ReferenceFrame* *referenceFrame*)

The rotation of the body, in the specified reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

*Tuple*<Double, Double, Double> **Direction** (*ReferenceFrame* *referenceFrame*)

The direction in which the north pole of the celestial body is pointing, in the specified reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

*Tuple*<Double, Double, Double> **AngularVelocity** (*ReferenceFrame* *referenceFrame*)

The angular velocity of the body in the specified reference frame.

**Parameters**

- **referenceFrame** – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the body, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

### 3.3.4 Flight

**class Flight**

Used to get flight telemetry for a vessel, by calling *Vessel.Flight*. All of the information returned by this class is given in the reference frame passed to that method. Obtained by calling *Vessel.Flight*.

---

**Note:** To get orbital information, such as the apoapsis or inclination, see *Orbit*.

---

Single **GForce** { **get**; }

The current G force acting on the vessel in  $m/s^2$ .

Double **MeanAltitude** { **get**; }

The altitude above sea level, in meters. Measured from the center of mass of the vessel.

Double **SurfaceAltitude** { **get**; }

The altitude above the surface of the body or sea level, whichever is closer, in meters. Measured from the center of mass of the vessel.

Double **BedrockAltitude** { **get**; }

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor. Measured from the center of mass of the vessel.

Double **Elevation** { **get**; }

The elevation of the terrain under the vessel, in meters. This is the height of the terrain above sea level, and is negative when the vessel is over the sea.

Double **Latitude** { **get**; }

The *latitude* of the vessel for the body being orbited, in degrees.

Double **Longitude** { **get**; }

The *longitude* of the vessel for the body being orbited, in degrees.

Tuple<Double, Double, Double> **Velocity** { **get**; }

The velocity of the vessel, in the reference frame *ReferenceFrame*.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the vessel in meters per second.

Double **Speed** { **get**; }

The speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Double **HorizontalSpeed** { **get**; }

The horizontal speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Double **VerticalSpeed** { **get**; }

The vertical speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Tuple<Double, Double, Double> **CenterOfMass** { **get**; }

The position of the center of mass of the vessel, in the reference frame *ReferenceFrame*

**Returns** The position as a vector.

Tuple<Double, Double, Double, Double> **Rotation** { **get**; }

The rotation of the vessel, in the reference frame *ReferenceFrame*

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

Tuple<Double, Double, Double> **Direction** { **get**; }

The direction that the vessel is pointing in, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

**Single Pitch { get; }**

The pitch of the vessel relative to the horizon, in degrees. A value between  $-90^\circ$  and  $+90^\circ$ .

**Single Heading { get; }**

The heading of the vessel (its angle relative to north), in degrees. A value between  $0^\circ$  and  $360^\circ$ .

**Single Roll { get; }**

The roll of the vessel relative to the horizon, in degrees. A value between  $-180^\circ$  and  $+180^\circ$ .

**Tuple<Double, Double, Double> Prograde { get; }**

The prograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

**Tuple<Double, Double, Double> Retrograde { get; }**

The retrograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

**Tuple<Double, Double, Double> Normal { get; }**

The direction normal to the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

**Tuple<Double, Double, Double> AntiNormal { get; }**

The direction opposite to the normal of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

**Tuple<Double, Double, Double> Radial { get; }**

The radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

**Tuple<Double, Double, Double> AntiRadial { get; }**

The direction opposite to the radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

**Single AtmosphereDensity { get; }**

The current density of the atmosphere around the vessel, in  $kg/m^3$ .

**Single DynamicPressure { get; }**

The dynamic pressure acting on the vessel, in Pascals. This is a measure of the strength of the aerodynamic forces. It is equal to  $\frac{1}{2} \cdot \text{air density} \cdot \text{velocity}^2$ . It is commonly denoted  $Q$ .

**Single StaticPressure { get; }**

The static atmospheric pressure acting on the vessel, in Pascals.

**Single StaticPressureAtMSL { get; }**

The static atmospheric pressure at mean sea level, in Pascals.

**Tuple<Double, Double, Double> AerodynamicForce { get; }**

The total aerodynamic forces acting on the vessel, in reference frame *ReferenceFrame*.

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Tuple<Double, Double, Double> SimulateAerodynamicForceAt** (*CelestialBody* *body*, *Tuple<Double, Double, Double>* *position*, *Tuple<Double, Double, Double>* *velocity*)

Simulate and return the total aerodynamic forces acting on the vessel, if it were to be traveling with the given velocity at the given position in the atmosphere of the given celestial body.

**Parameters**

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Tuple**<Double, Double, Double> **Lift** { **get**; }

The aerodynamic lift currently acting on the vessel.

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Tuple**<Double, Double, Double> **Drag** { **get**; }

The aerodynamic drag currently acting on the vessel.

**Returns** A vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

**Single** **SpeedOfSound** { **get**; }

The speed of sound, in the atmosphere around the vessel, in *m/s*.

**Single** **Mach** { **get**; }

The speed of the vessel, in multiples of the speed of sound.

**Single** **ReynoldsNumber** { **get**; }

The vessels Reynolds number.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**Single** **TrueAirSpeed** { **get**; }

The true air speed of the vessel, in meters per second.

**Single** **EquivalentAirSpeed** { **get**; }

The equivalent air speed of the vessel, in meters per second.

**Single** **TerminalVelocity** { **get**; }

An estimate of the current terminal velocity of the vessel, in meters per second. This is the speed at which the drag forces cancel out the force of gravity.

**Single** **AngleOfAttack** { **get**; }

The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

**Single** **SideslipAngle** { **get**; }

The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

**Single** **TotalAirTemperature** { **get**; }

The total air temperature of the atmosphere around the vessel, in Kelvin. This includes the *Flight*. *StaticAirTemperature* and the vessel's kinetic energy.

**Single** **StaticAirTemperature** { **get**; }

The static (ambient) temperature of the atmosphere around the vessel, in Kelvin.

**Single** **StallFraction** { **get**; }

The current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

---

**Single DragCoefficient { get; }**

The coefficient of drag. This is the amount of drag produced by the vessel. It depends on air speed, air density and wing area.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**Single LiftCoefficient { get; }**

The coefficient of lift. This is the amount of lift produced by the vessel, and depends on air speed, air density and wing area.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**Single BallisticCoefficient { get; }**

The ballistic coefficient.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**Single ThrustSpecificFuelConsumption { get; }**

The thrust specific fuel consumption for the jet engines on the vessel. This is a measure of the efficiency of the engines, with a lower value indicating a more efficient vessel. This value is the number of Newtons of fuel that are burned, per hour, to produce one newton of thrust.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

### 3.3.5 Orbit

#### **class Orbit**

Describes an orbit. For example, the orbit of a vessel, obtained by calling *Vessel.Orbit*, or a celestial body, obtained by calling *CelestialBody.Orbit*.

**CelestialBody Body { get; }**

The celestial body (e.g. planet or moon) around which the object is orbiting.

**Double Apoapsis { get; }**

Gets the apoapsis of the orbit, in meters, from the center of mass of the body being orbited.

---

**Note:** For the apoapsis altitude reported on the in-game map view, use *Orbit.ApoapsisAltitude*.

---

**Double Periapsis { get; }**

The periapsis of the orbit, in meters, from the center of mass of the body being orbited.

---

**Note:** For the periapsis altitude reported on the in-game map view, use *Orbit.PeriapsisAltitude*.

---

**Double ApoapsisAltitude { get; }**

The apoapsis of the orbit, in meters, above the sea level of the body being orbited.

---

**Note:** This is equal to *Orbit.Apoapsis* minus the equatorial radius of the body.

---

**Double PeriapsisAltitude { get; }**

The periapsis of the orbit, in meters, above the sea level of the body being orbited.

---

**Note:** This is equal to *Orbit.Periapsis* minus the equatorial radius of the body.

---

**Double SemiMajorAxis { get; }**

The semi-major axis of the orbit, in meters.

**Double SemiMinorAxis { get; }**

The semi-minor axis of the orbit, in meters.

**Double Radius { get; }**

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

---

**Note:** This value will change over time if the orbit is elliptical.

---

**Double RadiusAt (Double ut)**

The orbital radius at the given time, in meters.

**Parameters**

- **ut** – The universal time to measure the radius at.

**Tuple<Double, Double, Double> PositionAt (Double ut, ReferenceFrame referenceFrame)**

The position at a given time, in the specified reference frame.

**Parameters**

- **ut** – The universal time to measure the position at.
- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Double Speed { get; }**

The current orbital speed of the object in meters per second.

---

**Note:** This value will change over time if the orbit is elliptical.

---

**Double Period { get; }**

The orbital period, in seconds.

**Double TimeToApoapsis { get; }**

The time until the object reaches apoapsis, in seconds.

**Double TimeToPeriapsis { get; }**

The time until the object reaches periapsis, in seconds.

**Double Eccentricity { get; }**

The *eccentricity* of the orbit.

**Double Inclination { get; }**

The *inclination* of the orbit, in radians.

**Double LongitudeOfAscendingNode { get; }**

The longitude of the ascending node, in radians.

**Double ArgumentOfPeriapsis { get; }**

The argument of periapsis, in radians.

**Double MeanAnomalyAtEpoch { get; }**

The mean anomaly at epoch.

**Double Epoch { get; }**

The time since the epoch (the point at which the mean anomaly at epoch was measured, in seconds).

**Double MeanAnomaly { get; }**

The mean anomaly.

**Double MeanAnomalyAtUT (Double ut)**

The mean anomaly at the given time.

#### Parameters

- **ut** – The universal time in seconds.

**Double EccentricAnomaly { get; }**

The eccentric anomaly.

**Double EccentricAnomalyAtUT (Double ut)**

The eccentric anomaly at the given universal time.

#### Parameters

- **ut** – The universal time, in seconds.

**Double TrueAnomaly { get; }**

The true anomaly.

**Double TrueAnomalyAtUT (Double ut)**

The true anomaly at the given time.

#### Parameters

- **ut** – The universal time in seconds.

**Double TrueAnomalyAtRadius (Double radius)**

The true anomaly at the given orbital radius.

#### Parameters

- **radius** – The orbital radius in meters.

**Double UTAtTrueAnomaly (Double trueAnomaly)**

The universal time, in seconds, corresponding to the given true anomaly.

#### Parameters

- **trueAnomaly** – True anomaly.

**Double RadiusAtTrueAnomaly (Double trueAnomaly)**

The orbital radius at the point in the orbit given by the true anomaly.

#### Parameters

- **trueAnomaly** – The true anomaly.

**Double TrueAnomalyAtAN (Vessel target)**

The true anomaly of the ascending node with the given target vessel.

#### Parameters

- **target** – Target vessel.

**Double TrueAnomalyAtDN** (*Vessel target*)

The true anomaly of the descending node with the given target vessel.

**Parameters**

- **target** – Target vessel.

**Double OrbitalSpeed** { **get**; }

The current orbital speed in meters per second.

**Double OrbitalSpeedAt** (*Double time*)

The orbital speed at the given time, in meters per second.

**Parameters**

- **time** – Time from now, in seconds.

**static Tuple<Double, Double, Double> ReferencePlaneNormal** (*ICConnection connection, ReferenceFrame referenceFrame*)

The direction that is normal to the orbits reference plane, in the given reference frame. The reference plane is the plane from which the orbits inclination is measured.

**Parameters**

- **referenceFrame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**static Tuple<Double, Double, Double> ReferencePlaneDirection** (*ICConnection connection, ReferenceFrame referenceFrame*)

The direction from which the orbits longitude of ascending node is measured, in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Double RelativeInclination** (*Vessel target*)

Relative inclination of this orbit and the orbit of the given target vessel, in radians.

**Parameters**

- **target** – Target vessel.

**Double TimeToSOIChange** { **get**; }

The time until the object changes sphere of influence, in seconds. Returns NaN if the object is not going to change sphere of influence.

**Orbit NextOrbit** { **get**; }

If the object is going to change sphere of influence in the future, returns the new orbit after the change. Otherwise returns null.

**Double TimeOfClosestApproach** (*Vessel target*)

Estimates and returns the time at closest approach to a target vessel.

**Parameters**

- **target** – Target vessel.

**Returns** The universal time at closest approach, in seconds.

**Double DistanceAtClosestApproach** (*Vessel target*)

Estimates and returns the distance at closest approach to a target vessel, in meters.



**Parameters**

- **target** – Target vessel.

`IList<IList<Double>>` **ListClosestApproaches** (*Vessel target*, *Int32 orbits*)

Returns the times at closest approach and corresponding distances, to a target vessel.

**Parameters**

- **target** – Target vessel.
- **orbits** – The number of future orbits to search.

**Returns** A list of two lists. The first is a list of times at closest approach, as universal times in seconds. The second is a list of corresponding distances at closest approach, in meters.

### 3.3.6 Control

**class Control**

Used to manipulate the controls of a vessel. This includes adjusting the throttle, enabling/disabling systems such as SAS and RCS, or altering the direction in which the vessel is pointing. Obtained by calling *Vessel.Control*.

---

**Note:** Control inputs (such as pitch, yaw and roll) are zeroed when all clients that have set one or more of these inputs are no longer connected.

---

*ControlSource* **Source** { **get**; }

The source of the vessels control, for example by a kerbal or a probe core.

*ControlState* **State** { **get**; }

The control state of the vessel.

**Boolean SAS** { **get**; **set**; }

The state of SAS.

---

**Note:** Equivalent to *AutoPilot.SAS*

---

*SASMode* **SASMode** { **get**; **set**; }

The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

---

**Note:** Equivalent to *AutoPilot.SASMode*

---

*SpeedMode* **SpeedMode** { **get**; **set**; }

The current *SpeedMode* of the navball. This is the mode displayed next to the speed at the top of the navball.

**Boolean RCS** { **get**; **set**; }

The state of RCS.

**Boolean ReactionWheels** { **get**; **set**; }

Returns whether all reactive wheels on the vessel are active, and sets the active state of all reaction wheels. See *ReactionWheel.Active*.

**Boolean Gear** { **get**; **set**; }

The state of the landing gear/legs.

**Boolean Legs { get; set; }**

Returns whether all landing legs on the vessel are deployed, and sets the deployment state of all landing legs. Does not include wheels (for example landing gear). See *Leg.Deployed*.

**Boolean Wheels { get; set; }**

Returns whether all wheels on the vessel are deployed, and sets the deployment state of all wheels. Does not include landing legs. See *Wheel.Deployed*.

**Boolean Lights { get; set; }**

The state of the lights.

**Boolean Brakes { get; set; }**

The state of the wheel brakes.

**Boolean Antennas { get; set; }**

Returns whether all antennas on the vessel are deployed, and sets the deployment state of all antennas. See *Antenna.Deployed*.

**Boolean CargoBays { get; set; }**

Returns whether any of the cargo bays on the vessel are open, and sets the open state of all cargo bays. See *CargoBay.Open*.

**Boolean Intakes { get; set; }**

Returns whether all of the air intakes on the vessel are open, and sets the open state of all air intakes. See *Intake.Open*.

**Boolean Parachutes { get; set; }**

Returns whether all parachutes on the vessel are deployed, and sets the deployment state of all parachutes. Cannot be set to *false*. See *Parachute.Deployed*.

**Boolean Radiators { get; set; }**

Returns whether all radiators on the vessel are deployed, and sets the deployment state of all radiators. See *Radiator.Deployed*.

**Boolean ResourceHarvesters { get; set; }**

Returns whether all of the resource harvesters on the vessel are deployed, and sets the deployment state of all resource harvesters. See *ResourceHarvester.Deployed*.

**Boolean ResourceHarvestersActive { get; set; }**

Returns whether any of the resource harvesters on the vessel are active, and sets the active state of all resource harvesters. See *ResourceHarvester.Active*.

**Boolean SolarPanels { get; set; }**

Returns whether all solar panels on the vessel are deployed, and sets the deployment state of all solar panels. See *SolarPanel.Deployed*.

**Boolean Abort { get; set; }**

The state of the abort action group.

**Single Throttle { get; set; }**

The state of the throttle. A value between 0 and 1.

**ControlInputMode InputMode { get; set; }**

Sets the behavior of the pitch, yaw, roll and translation control inputs. When set to *additive*, these inputs are added to the vessels current inputs. This mode is the default. When set to *override*, these inputs (if non-zero) override the vessels inputs. This mode prevents keyboard control, or SAS, from interfering with the controls when they are set.

**Single Pitch { get; set; }**

The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

**Single Yaw** { **get**; **set**; }

The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

**Single Roll** { **get**; **set**; }

The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

**Single Forward** { **get**; **set**; }

The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

**Single Up** { **get**; **set**; }

The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

**Single Right** { **get**; **set**; }

The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

**Single WheelThrottle** { **get**; **set**; }

The state of the wheel throttle. A value between -1 and 1. A value of 1 rotates the wheels forwards, a value of -1 rotates the wheels backwards.

**Single WheelSteering** { **get**; **set**; }

The state of the wheel steering. A value between -1 and 1. A value of 1 steers to the left, and a value of -1 steers to the right.

**Int32 CurrentStage** { **get**; }

The current stage of the vessel. Corresponds to the stage number in the in-game UI.

**IList<Vessel> ActivateNextStage** ()

Activates the next stage. Equivalent to pressing the space bar in-game.

**Returns** A list of vessel objects that are jettisoned from the active vessel.

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *SpaceCenter.ActiveVessel* no longer refer to the active vessel.

---

**Boolean GetActionGroup** (UInt32 group)

Returns `true` if the given action group is enabled.

#### Parameters

- **group** – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the *Extended Action Groups mod* is installed.

**void SetActionGroup** (UInt32 group, Boolean state)

Sets the state of the given action group.

#### Parameters

- **group** – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the *Extended Action Groups mod* is installed.

**void ToggleActionGroup** (UInt32 group)

Toggles the state of the given action group.

#### Parameters

- **group** – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the *Extended Action Groups mod* is installed.

**Node AddNode** (Double ut, Single prograde = 0.0, Single normal = 0.0, Single radial = 0.0)

Creates a maneuver node at the given universal time, and returns a *Node* object that can be used to modify

it. Optionally sets the magnitude of the delta-v for the maneuver node in the prograde, normal and radial directions.

#### Parameters

- **ut** – Universal time of the maneuver node.
- **prograde** – Delta-v in the prograde direction.
- **normal** – Delta-v in the normal direction.
- **radial** – Delta-v in the radial direction.

`IList<Node> Nodes { get; }`

Returns a list of all existing maneuver nodes, ordered by time from first to last.

void **RemoveNodes** ()

Remove all maneuver nodes.

#### enum ControlState

The control state of a vessel. See *Control.State*.

##### Full

Full controllable.

##### Partial

Partially controllable.

##### None

Not controllable.

#### enum ControlSource

The control source of a vessel. See *Control.Source*.

##### Kerbal

Vessel is controlled by a Kerbal.

##### Probe

Vessel is controlled by a probe core.

##### None

Vessel is not controlled.

#### enum SASMode

The behavior of the SAS auto-pilot. See *AutoPilot.SASMode*.

##### StabilityAssist

Stability assist mode. Dampen out any rotation.

##### Maneuver

Point in the burn direction of the next maneuver node.

##### Prograde

Point in the prograde direction.

##### Retrograde

Point in the retrograde direction.

##### Normal

Point in the orbit normal direction.

##### AntiNormal

Point in the orbit anti-normal direction.

**Radial**

Point in the orbit radial direction.

**AntiRadial**

Point in the orbit anti-radial direction.

**Target**

Point in the direction of the current target.

**AntiTarget**

Point away from the current target.

**enum SpeedMode**

The mode of the speed reported in the navball. See *Control.SpeedMode*.

**Orbit**

Speed is relative to the vessel's orbit.

**Surface**

Speed is relative to the surface of the body being orbited.

**Target**

Speed is relative to the current target.

**enum ControlInputMode**

See *Control.InputMode*.

**Additive**

Control inputs are added to the vessels current control inputs.

**Override**

Control inputs (when they are non-zero) override the vessels current control inputs.

### 3.3.7 Communications

**class Comms**

Used to interact with CommNet for a given vessel. Obtained by calling *Vessel.Comms*.

**Boolean CanCommunicate { get; }**

Whether the vessel can communicate with KSC.

**Boolean CanTransmitScience { get; }**

Whether the vessel can transmit science data to KSC.

**Double SignalStrength { get; }**

Signal strength to KSC.

**Double SignalDelay { get; }**

Signal delay to KSC in seconds.

**Double Power { get; }**

The combined power of all active antennae on the vessel.

**IList<CommLink> ControlPath { get; }**

The communication path used to control the vessel.

**class CommLink**

Represents a communication node in the network. For example, a vessel or the KSC.

**CommLinkType Type { get; }**

The type of link.

**Double SignalStrength { get; }**

Signal strength of the link.

**CommNode Start { get; }**

Start point of the link.

**CommNode End { get; }**

Start point of the link.

**enum CommLinkType**

The type of a communication link. See *CommLink.Type*.

**Home**

Link is to a base station on Kerbin.

**Control**

Link is to a control source, for example a manned spacecraft.

**Relay**

Link is to a relay satellite.

**class CommNode**

Represents a communication node in the network. For example, a vessel or the KSC.

**String Name { get; }**

Name of the communication node.

**Boolean IsHome { get; }**

Whether the communication node is on Kerbin.

**Boolean IsControlPoint { get; }**

Whether the communication node is a control point, for example a manned vessel.

**Boolean IsVessel { get; }**

Whether the communication node is a vessel.

**Vessel Vessel { get; }**

The vessel for this communication node.

### 3.3.8 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*
  - *Antenna*
  - *Cargo Bay*
  - *Control Surface*
  - *Decoupler*
  - *Docking Port*
  - *Engine*

- *Experiment*
- *Fairing*
- *Intake*
- *Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *RCS*
- *Sensor*
- *Solar Panel*
- *Thruster*
- *Wheel*
- *Trees of Parts*
  - *Traversing the Tree*
  - *Attachment Modes*
- *Fuel Lines*
- *Staging*

## Parts

### class Parts

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling `Vessel.Parts`.

`IList<Part> All { get; }`

A list of all of the vessels parts.

`Part Root { get; }`

The vessels root part.

---

**Note:** See the discussion on *Trees of Parts*.

---

`Part Controlling { get; set; }`

The part from which the vessel is controlled.

`IList<Part> WithName (String name)`

A list of parts whose `Part.Name` is *name*.

### Parameters

**IList<Part> WithTitle** (*String title*)

A list of all parts whose *Part.Title* is *title*.

**Parameters**

**IList<Part> WithTag** (*String tag*)

A list of all parts whose *Part.Tag* is *tag*.

**Parameters**

**IList<Part> WithModule** (*String moduleName*)

A list of all parts that contain a *Module* whose *Module.Name* is *moduleName*.

**Parameters**

**IList<Part> InStage** (*Int32 stage*)

A list of all parts that are activated in the given *stage*.

**Parameters**

---

**Note:** See the discussion on *Staging*.

---

**IList<Part> InDecoupleStage** (*Int32 stage*)

A list of all parts that are decoupled in the given *stage*.

**Parameters**

---

**Note:** See the discussion on *Staging*.

---

**IList<Module> ModulesWithName** (*String moduleName*)

A list of modules (combined across all parts in the vessel) whose *Module.Name* is *moduleName*.

**Parameters**

**IList<Antenna> Antennas** { **get**; }

A list of all antennas in the vessel.

**IList<CargoBay> CargoBays** { **get**; }

A list of all cargo bays in the vessel.

**IList<ControlSurface> ControlSurfaces** { **get**; }

A list of all control surfaces in the vessel.

**IList<Decoupler> Decouplers** { **get**; }

A list of all decouplers in the vessel.

**IList<DockingPort> DockingPorts** { **get**; }

A list of all docking ports in the vessel.

**IList<Engine> Engines** { **get**; }

A list of all engines in the vessel.

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters, jet engines and RCS thrusters.

---

**IList<Experiment> Experiments** { **get**; }

A list of all science experiments in the vessel.



**IList<Fairing> Fairings { get; }**  
A list of all fairings in the vessel.

**IList<Intake> Intakes { get; }**  
A list of all intakes in the vessel.

**IList<Leg> Legs { get; }**  
A list of all landing legs attached to the vessel.

**IList<LaunchClamp> LaunchClamps { get; }**  
A list of all launch clamps attached to the vessel.

**IList<Light> Lights { get; }**  
A list of all lights in the vessel.

**IList<Parachute> Parachutes { get; }**  
A list of all parachutes in the vessel.

**IList<Radiator> Radiators { get; }**  
A list of all radiators in the vessel.

**IList<RCS> RCS { get; }**  
A list of all RCS blocks/thrusters in the vessel.

**IList<ReactionWheel> ReactionWheels { get; }**  
A list of all reaction wheels in the vessel.

**IList<ResourceConverter> ResourceConverters { get; }**  
A list of all resource converters in the vessel.

**IList<ResourceHarvester> ResourceHarvesters { get; }**  
A list of all resource harvesters in the vessel.

**IList<Sensor> Sensors { get; }**  
A list of all sensors in the vessel.

**IList<SolarPanel> SolarPanels { get; }**  
A list of all solar panels in the vessel.

**IList<Wheel> Wheels { get; }**  
A list of all wheels in the vessel.

## Part

### class Part

Represents an individual part. Vessels are made up of multiple parts. Instances of this class can be obtained by several methods in *Parts*.

**String Name { get; }**  
Internal name of the part, as used in *part* *cfg* files. For example “Mark1-2Pod”.

**String Title { get; }**  
Title of the part, as shown when the part is right clicked in-game. For example “Mk1-2 Command Pod”.

**String Tag { get; set; }**  
The name tag for the part. Can be set to a custom string using the in-game user interface.

---

**Note:** This requires either the [NameTag](#) or [kOS](#) mod to be installed.

---

**Boolean Highlighted { get; set; }**

Whether the part is highlighted.

**Tuple<Double, Double, Double> HighlightColor { get; set; }**

The color used to highlight the part, as an RGB triple.

**Double Cost { get; }**

The cost of the part, in units of funds.

**Vessel Vessel { get; }**

The vessel that contains this part.

**Part Parent { get; }**

The parts parent. Returns `null` if the part does not have a parent. This, in combination with `Part.Children`, can be used to traverse the vessels parts tree.

---

**Note:** See the discussion on *Trees of Parts*.

---

**ICollection<Part> Children { get; }**

The parts children. Returns an empty list if the part has no children. This, in combination with `Part.Parent`, can be used to traverse the vessels parts tree.

---

**Note:** See the discussion on *Trees of Parts*.

---

**Boolean AxiallyAttached { get; }**

Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns `false`.

---

**Note:** See the discussion on *Attachment Modes*.

---

**Boolean RadiallyAttached { get; }**

Whether the part is radially attached to its parent, i.e. on the side of its parent. If the part has no parent, returns `false`.

---

**Note:** See the discussion on *Attachment Modes*.

---

**Int32 Stage { get; }**

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

---

**Note:** See the discussion on *Staging*.

---

**Int32 DecoupleStage { get; }**

The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

---

**Note:** See the discussion on *Staging*.

---

**Boolean Massless { get; }**

Whether the part is `massless`.

**Double Mass { get; }**

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

**Double DryMass { get; }**

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

**Boolean Shielded { get; }**

Whether the part is shielded from the exterior of the vessel, for example by a fairing.

**Single DynamicPressure { get; }**

The dynamic pressure acting on the part, in Pascals.

**Double ImpactTolerance { get; }**

The impact tolerance of the part, in meters per second.

**Double Temperature { get; }**

Temperature of the part, in Kelvin.

**Double SkinTemperature { get; }**

Temperature of the skin of the part, in Kelvin.

**Double MaxTemperature { get; }**

Maximum temperature that the part can survive, in Kelvin.

**Double MaxSkinTemperature { get; }**

Maximum temperature that the skin of the part can survive, in Kelvin.

**Single ThermalMass { get; }**

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

**Single ThermalSkinMass { get; }**

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

**Single ThermalResourceMass { get; }**

A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

**Single ThermalConductionFlux { get; }**

The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Single ThermalConvectionFlux { get; }**

The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Single ThermalRadiationFlux { get; }**

The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Single ThermalInternalFlux { get; }**

The rate at which heat energy is being generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Single ThermalSkinToInternalFlux { get; }**

The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy

per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

*Resources* **Resources** { **get**; }

A *Resources* object for the part.

*Boolean* **Crossfeed** { **get**; }

Whether this part is crossfeed capable.

*Boolean* **IsFuelLine** { **get**; }

Whether this part is a fuel line.

*IList*<Part> **FuelLinesFrom** { **get**; }

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

---

**Note:** See the discussion on *Fuel Lines*.

---

*IList*<Part> **FuelLinesTo** { **get**; }

The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

---

**Note:** See the discussion on *Fuel Lines*.

---

*IList*<Module> **Modules** { **get**; }

The modules for this part.

*Antenna* **Antenna** { **get**; }

A *Antenna* if the part is an antenna, otherwise null.

*CargoBay* **CargoBay** { **get**; }

A *CargoBay* if the part is a cargo bay, otherwise null.

*ControlSurface* **ControlSurface** { **get**; }

A *ControlSurface* if the part is an aerodynamic control surface, otherwise null.

*Decoupler* **Decoupler** { **get**; }

A *Decoupler* if the part is a decoupler, otherwise null.

*DockingPort* **DockingPort** { **get**; }

A *DockingPort* if the part is a docking port, otherwise null.

*Engine* **Engine** { **get**; }

An *Engine* if the part is an engine, otherwise null.

*Experiment* **Experiment** { **get**; }

An *Experiment* if the part is a science experiment, otherwise null.

*Fairing* **Fairing** { **get**; }

A *Fairing* if the part is a fairing, otherwise null.

*Intake* **Intake** { **get**; }

An *Intake* if the part is an intake, otherwise null.

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters and jet engines. For RCS thrusters see *RCS*.

---

*Leg* **Leg** { **get**; }

A *Leg* if the part is a landing leg, otherwise null.

*LaunchClamp* **LaunchClamp** { **get**; }

A *LaunchClamp* if the part is a launch clamp, otherwise null.

*Light* **Light** { **get**; }

A *Light* if the part is a light, otherwise null.

*Parachute* **Parachute** { **get**; }

A *Parachute* if the part is a parachute, otherwise null.

*Radiator* **Radiator** { **get**; }

A *Radiator* if the part is a radiator, otherwise null.

*RCS* **RCS** { **get**; }

A *RCS* if the part is an RCS block/thruster, otherwise null.

*ReactionWheel* **ReactionWheel** { **get**; }

A *ReactionWheel* if the part is a reaction wheel, otherwise null.

*ResourceConverter* **ResourceConverter** { **get**; }

A *ResourceConverter* if the part is a resource converter, otherwise null.

*ResourceHarvester* **ResourceHarvester** { **get**; }

A *ResourceHarvester* if the part is a resource harvester, otherwise null.

*Sensor* **Sensor** { **get**; }

A *Sensor* if the part is a sensor, otherwise null.

*SolarPanel* **SolarPanel** { **get**; }

A *SolarPanel* if the part is a solar panel, otherwise null.

*Wheel* **Wheel** { **get**; }

A *Wheel* if the part is a wheel, otherwise null.

**Tuple**<**Double**, **Double**, **Double**> **Position** (*ReferenceFrame* *referenceFrame*)

The position of the part in the given reference frame.

#### Parameters

- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

---

**Note:** This is a fixed position in the part, defined by the parts model. It is not necessarily the same as the parts center of mass. Use *Part.CenterOfMass* to get the parts center of mass.

---

**Tuple**<**Double**, **Double**, **Double**> **CenterOfMass** (*ReferenceFrame* *referenceFrame*)

The position of the parts center of mass in the given reference frame. If the part is physicsless, this is equivalent to *Part.Position*.

#### Parameters

- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Tuple**<**Tuple**<**Double**, **Double**, **Double**>, **Tuple**<**Double**, **Double**, **Double**>> **BoundingBox** (*ReferenceFrame* *referenceFrame*)

The axis-aligned bounding box of the part in the given reference frame.

#### Parameters

- **referenceFrame** – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

---

**Note:** This is computed from the collision mesh of the part. If the part is not collidable, the box has zero volume and is centered on the *Part.Position* of the part.

---

**Tuple<Double, Double, Double> Direction** (*ReferenceFrame referenceFrame*)

The direction the part points in, in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Tuple<Double, Double, Double> Velocity** (*ReferenceFrame referenceFrame*)

The linear velocity of the part in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

**Tuple<Double, Double, Double, Double> Rotation** (*ReferenceFrame referenceFrame*)

The rotation of the part, in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Tuple<Double, Double, Double> MomentOfInertia** { **get**; }

The moment of inertia of the part in  $kg.m^2$  around its center of mass in the parts reference frame (*ReferenceFrame*).

**IList<Double> InertiaTensor** { **get**; }

The inertia tensor of the part in the parts reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

*ReferenceFrame* **ReferenceFrame** { **get**; }

The reference frame that is fixed relative to this part, and centered on a fixed position within the part, defined by the parts model.

- The origin is at the position of the part, as returned by *Part.Position*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

---

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *DockingPort.ReferenceFrame*.

---

*ReferenceFrame* **CenterOfMassReferenceFrame** { **get**; }

The reference frame that is fixed relative to this part, and centered on its center of mass.

- The origin is at the center of mass of the part, as returned by *Part.CenterOfMass*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

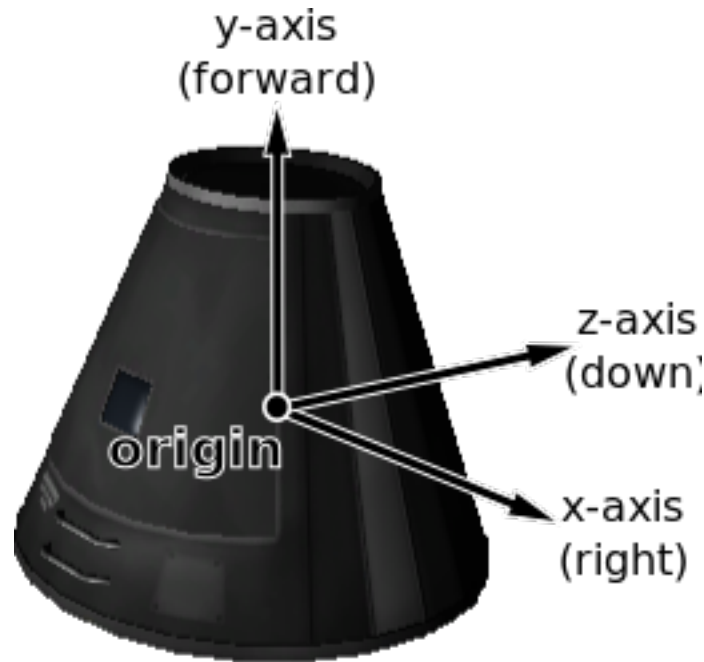


Fig. 3.7: Mk1 Command Pod reference frame origin and axes

---

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *DockingPort.ReferenceFrame*.

---

*Force* **AddForce** (*Tuple*<Double, Double, Double> *force*, *Tuple*<Double, Double, Double> *position*, *ReferenceFrame* *referenceFrame*)

Exert a constant force on the part, acting at the given position.

**Parameters**

- **force** – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** – The position at which the force acts, as a vector.
- **referenceFrame** – The reference frame that the force and position are in.

**Returns** An object that can be used to remove or modify the force.

void **InstantaneousForce** (*Tuple*<Double, Double, Double> *force*, *Tuple*<Double, Double, Double> *position*, *ReferenceFrame* *referenceFrame*)

Exert an instantaneous force on the part, acting at the given position.

**Parameters**

- **force** – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** – The position at which the force acts, as a vector.
- **referenceFrame** – The reference frame that the force and position are in.

---

**Note:** The force is applied instantaneously in a single physics update.

---

**class Force**

Obtained by calling *Part.AddForce*.

*Part* **Part** { **get**; }

The part that this force is applied to.

*Tuple*<*Double*, *Double*, *Double*> **ForceVector** { **get**; **set**; }

The force vector, in Newtons.

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

*Tuple*<*Double*, *Double*, *Double*> **Position** { **get**; **set**; }

The position at which the force acts, in reference frame *ReferenceFrame*.

**Returns** The position as a vector.

*ReferenceFrame* **ReferenceFrame** { **get**; **set**; }

The reference frame of the force vector and position.

void **Remove** ()

Remove the force.

## Module

**class Module**

This can be used to interact with a specific part module. This includes part modules in stock KSP, and those added by mods.

In KSP, each part has zero or more *PartModules* associated with it. Each one contains some of the functionality of the part. For example, an engine has a “ModuleEngines” part module that contains all the functionality of an engine.

*String* **Name** { **get**; }

Name of the PartModule. For example, “ModuleEngines”.

*Part* **Part** { **get**; }

The part that contains this module.

*IDictionary*<*String*, *String*> **Fields** { **get**; }

The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

*Boolean* **HasField** (*String name*)

Returns *true* if the module has a field with the given name.

**Parameters**

- **name** – Name of the field.

*String* **GetField** (*String name*)

Returns the value of a field.

**Parameters**

- **name** – Name of the field.

void **SetFieldInt** (*String name*, *Int32 value*)

Set the value of a field to the given integer number.

**Parameters**

- **name** – Name of the field.



- **value** – Value to set.

void **SetFieldFloat** (*String name*, *Single value*)  
Set the value of a field to the given floating point number.

#### Parameters

- **name** – Name of the field.
- **value** – Value to set.

void **SetFieldString** (*String name*, *String value*)  
Set the value of a field to the given string.

#### Parameters

- **name** – Name of the field.
- **value** – Value to set.

void **ResetField** (*String name*)  
Set the value of a field to its original value.

#### Parameters

- **name** – Name of the field.

*IList<String>* **Events** { *get*; }

A list of the names of all of the modules events. Events are the clickable buttons visible in the right-click menu of the part.

*Boolean* **HasEvent** (*String name*)  
*true* if the module has an event with the given name.

#### Parameters

void **TriggerEvent** (*String name*)  
Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

#### Parameters

*IList<String>* **Actions** { *get*; }

A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

*Boolean* **HasAction** (*String name*)  
*true* if the part has an action with the given name.

#### Parameters

void **SetAction** (*String name*, *Boolean value = True*)  
Set the value of an action with the given name.

#### Parameters

## Specific Types of Part

The following classes provide functionality for specific types of part.

- *Antenna*
- *Cargo Bay*

- *Control Surface*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Experiment*
- *Fairing*
- *Intake*
- *Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *RCS*
- *Sensor*
- *Solar Panel*
- *Thruster*
- *Wheel*

## Antenna

### **class Antenna**

An antenna. Obtained by calling *Part.Antenna*.

*Part* **Part** { **get**; }

The part object for this antenna.

*AntennaState* **State** { **get**; }

The current state of the antenna.

**Boolean Deployable** { **get**; }

Whether the antenna is deployable.

**Boolean Deployed** { **get**; **set**; }

Whether the antenna is deployed.

---

**Note:** Fixed antennas are always deployed. Returns an error if you try to deploy a fixed antenna.

---

**Boolean CanTransmit** { **get**; }

Whether data can be transmitted by this antenna.

```

void Transmit ()
    Transmit data.

void Cancel ()
    Cancel current transmission of data.

Boolean AllowPartial { get; set; }
    Whether partial data transmission is permitted.

Double Power { get; }
    The power of the antenna.

Boolean Combinable { get; }
    Whether the antenna can be combined with other antennae on the vessel to boost the power.

Double CombinableExponent { get; }
    Exponent used to calculate the combined power of multiple antennae on a vessel.

Single PacketInterval { get; }
    Interval between sending packets in seconds.

Single PacketSize { get; }
    Amount of data sent per packet in Mits.

Double PacketResourceCost { get; }
    Units of electric charge consumed per packet sent.

```

```

enum AntennaState
    The state of an antenna. See Antenna.State.

    Deployed
        Antenna is fully deployed.

    Retracted
        Antenna is fully retracted.

    Deploying
        Antenna is being deployed.

    Retracting
        Antenna is being retracted.

    Broken
        Antenna is broken.

```

## Cargo Bay

```

class CargoBay
    A cargo bay. Obtained by calling Part.CargoBay.

    Part Part { get; }
        The part object for this cargo bay.

    CargoBayState State { get; }
        The state of the cargo bay.

    Boolean Open { get; set; }
        Whether the cargo bay is open.

enum CargoBayState
    The state of a cargo bay. See CargoBay.State.

```

**Open**  
Cargo bay is fully open.

**Closed**  
Cargo bay closed and locked.

**Opening**  
Cargo bay is opening.

**Closing**  
Cargo bay is closing.

## Control Surface

### **class ControlSurface**

An aerodynamic control surface. Obtained by calling *Part.ControlSurface*.

**Part Part { get; }**  
The part object for this control surface.

**Boolean PitchEnabled { get; set; }**  
Whether the control surface has pitch control enabled.

**Boolean YawEnabled { get; set; }**  
Whether the control surface has yaw control enabled.

**Boolean RollEnabled { get; set; }**  
Whether the control surface has roll control enabled.

**Single AuthorityLimiter { get; set; }**  
The authority limiter for the control surface, which controls how far the control surface will move.

**Boolean Inverted { get; set; }**  
Whether the control surface movement is inverted.

**Boolean Deployed { get; set; }**  
Whether the control surface has been fully deployed.

**Single SurfaceArea { get; }**  
Surface area of the control surface in  $m^2$ .

**Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> AvailableTorque { get; }**  
The available torque, in Newton meters, that can be produced by this control surface, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.ReferenceFrame*.

## Decoupler

### **class Decoupler**

A decoupler. Obtained by calling *Part.Decoupler*

**Part Part { get; }**  
The part object for this decoupler.

**Vessel Decouple ()**  
Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *SpaceCenter.ActiveVessel* no longer refer to the active vessel.

---

**Boolean Decoupled { get; }**

Whether the decoupler has fired.

**Boolean Staged { get; }**

Whether the decoupler is enabled in the staging sequence.

**Single Impulse { get; }**

The impulse that the decoupler imparts when it is fired, in Newton seconds.

## Docking Port

**class DockingPort**

A docking port. Obtained by calling *Part.DockingPort*

**Part Part { get; }**

The part object for this docking port.

**DockingPortState State { get; }**

The current state of the docking port.

**Part DockedPart { get; }**

The part that this docking port is docked to. Returns *null* if this docking port is not docked to anything.

**Vessel Undock ()**

Undocks the docking port and returns the new *Vessel* that is created. This method can be called for either docking port in a docked pair. Throws an exception if the docking port is not docked to anything.

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *SpaceCenter.ActiveVessel* no longer refer to the active vessel.

---

**Single ReengageDistance { get; }**

The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

**Boolean HasShield { get; }**

Whether the docking port has a shield.

**Boolean Shielded { get; set; }**

The state of the docking ports shield, if it has one.

Returns *true* if the docking port has a shield, and the shield is closed. Otherwise returns *false*. When set to *true*, the shield is closed, and when set to *false* the shield is opened. If the docking port does not have a shield, setting this attribute has no effect.

**Tuple<Double, Double, Double> Position (ReferenceFrame referenceFrame)**

The position of the docking port, in the given reference frame.

### Parameters

- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

`Tuple<Double, Double, Double> Direction (ReferenceFrame referenceFrame)`

The direction that docking port points in, in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

`Tuple<Double, Double, Double, Double> Rotation (ReferenceFrame referenceFrame)`

The rotation of the docking port, in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

`ReferenceFrame ReferenceFrame { get; }`

The reference frame that is fixed relative to this docking port, and oriented with the port.

- The origin is at the position of the docking port.
- The axes rotate with the docking port.
- The x-axis points out to the right side of the docking port.
- The y-axis points in the direction the docking port is facing.
- The z-axis points out of the bottom off the docking port.

---

**Note:** This reference frame is not necessarily equivalent to the reference frame for the part, returned by `Part.ReferenceFrame`.

---

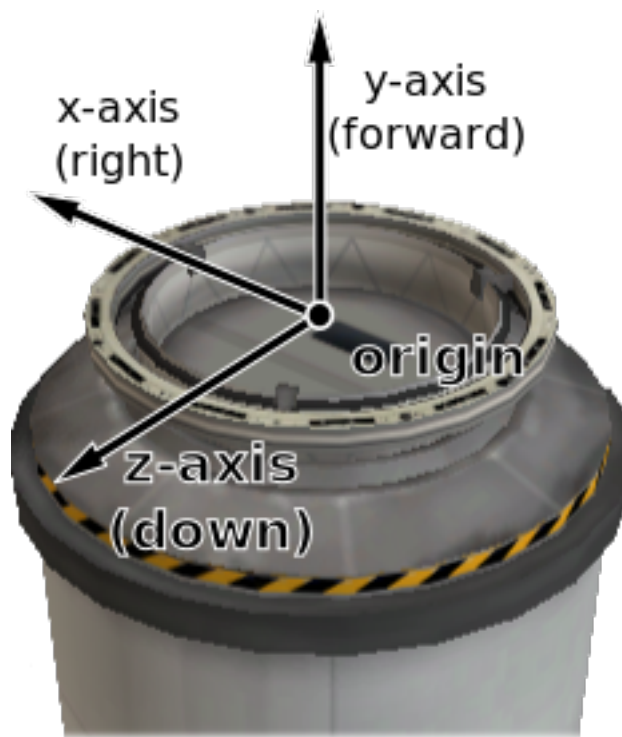


Fig. 3.8: Docking port reference frame origin and axes

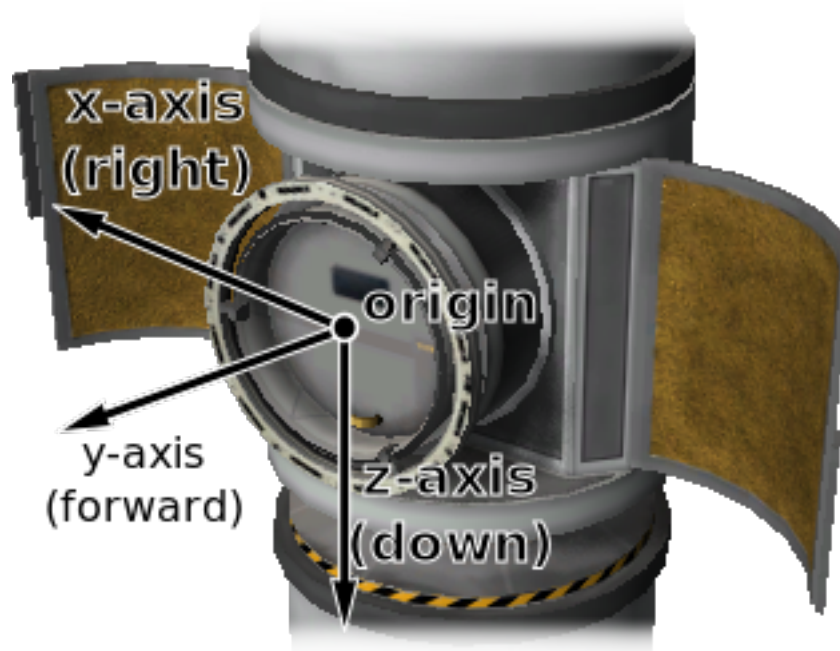


Fig. 3.9: Inline docking port reference frame origin and axes

#### **enum DockingPortState**

The state of a docking port. See *DockingPort.State*.

##### **Ready**

The docking port is ready to dock to another docking port.

##### **Docked**

The docking port is docked to another docking port, or docked to another part (from the VAB/SPH).

##### **Docking**

The docking port is very close to another docking port, but has not docked. It is using magnetic force to acquire a solid dock.

##### **Undocking**

The docking port has just been undocked from another docking port, and is disabled until it moves away by a sufficient distance (*DockingPort.ReengageDistance*).

##### **Shielded**

The docking port has a shield, and the shield is closed.

##### **Moving**

The docking ports shield is currently opening/closing.

## **Engine**

#### **class Engine**

An engine, including ones of various types. For example liquid fuelled gimballed engines, solid rocket boosters and jet engines. Obtained by calling *Part.Engine*.

---

**Note:** For RCS thrusters *Part.RCS*.

---

**Part Part { get; }**

The part object for this engine.

**Boolean Active { get; set; }**

Whether the engine is active. Setting this attribute may have no effect, depending on *Engine.CanShutdown* and *Engine.CanRestart*.

**Single Thrust { get; }**

The current amount of thrust being produced by the engine, in Newtons.

**Single AvailableThrust { get; }**

The amount of thrust, in Newtons, that would be produced by the engine when activated and with its throttle set to 100%. Returns zero if the engine does not have any fuel. Takes the engine's current *Engine.ThrustLimit* and atmospheric conditions into account.

**Single MaxThrust { get; }**

The amount of thrust, in Newtons, that would be produced by the engine when activated and fueled, with its throttle and throttle limiter set to 100%.

**Single MaxVacuumThrust { get; }**

The maximum amount of thrust that can be produced by the engine in a vacuum, in Newtons. This is the amount of thrust produced by the engine when activated, *Engine.ThrustLimit* is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

**Single ThrustLimit { get; set; }**

The thrust limiter of the engine. A value between 0 and 1. Setting this attribute may have no effect, for example the thrust limit for a solid rocket booster cannot be changed in flight.

**IList<Thruster> Thrusters { get; }**

The components of the engine that generate thrust.

---

**Note:** For example, this corresponds to the rocket nozzle on a solid rocket booster, or the individual nozzles on a RAPIER engine. The overall thrust produced by the engine, as reported by *Engine.AvailableThrust*, *Engine.MaxThrust* and others, is the sum of the thrust generated by each thruster.

---

**Single SpecificImpulse { get; }**

The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

**Single VacuumSpecificImpulse { get; }**

The vacuum specific impulse of the engine, in seconds.

**Single KerbinSeaLevelSpecificImpulse { get; }**

The specific impulse of the engine at sea level on Kerbin, in seconds.

**IList<String> PropellantNames { get; }**

The names of the propellants that the engine consumes.

**IDictionary<String, Single> PropellantRatios { get; }**

The ratio of resources that the engine consumes. A dictionary mapping resource names to the ratio at which they are consumed by the engine.

---

**Note:** For example, if the ratios are 0.6 for LiquidFuel and 0.4 for Oxidizer, then for every 0.6 units of LiquidFuel that the engine burns, it will burn 0.4 units of Oxidizer.

---

**IList<Propellant> Propellants { get; }**

The propellants that the engine consumes.



**Boolean HasFuel { get; }**

Whether the engine has any fuel available.

---

**Note:** The engine must be activated for this property to update correctly.

---

**Single Throttle { get; }**

The current throttle setting for the engine. A value between 0 and 1. This is not necessarily the same as the vessel's main throttle setting, as some engines take time to adjust their throttle (such as jet engines).

**Boolean ThrottleLocked { get; }**

Whether the *Control.Throttle* affects the engine. For example, this is `true` for liquid fueled rockets, and `false` for solid rocket boosters.

**Boolean CanRestart { get; }**

Whether the engine can be restarted once shutdown. If the engine cannot be shutdown, returns `false`. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

**Boolean CanShutdown { get; }**

Whether the engine can be shutdown once activated. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

**Boolean HasModes { get; }**

Whether the engine has multiple modes of operation.

**String Mode { get; set; }**

The name of the current engine mode.

**IDictionary<String, Engine> Modes { get; }**

The available modes for the engine. A dictionary mapping mode names to *Engine* objects.

**void ToggleMode ()**

Toggle the current engine mode.

**Boolean AutoModeSwitch { get; set; }**

Whether the engine will automatically switch modes.

**Boolean Gimballled { get; }**

Whether the engine is gimballled.

**Single GimbalRange { get; }**

The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballled.

**Boolean GimbalLocked { get; set; }**

Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballled.

**Single GimbalLimit { get; set; }**

The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

**Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> AvailableTorque { get; }**

The available torque, in Newton meters, that can be produced by this engine, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.ReferenceFrame*. Returns zero if the engine is inactive, or not gimballled.

### **class Propellant**

A propellant for an engine. Obtains by calling *Engine.Propellants*.

**String Name { get; }**

The name of the propellant.

**Double CurrentAmount { get; }**  
The current amount of propellant.

**Double CurrentRequirement { get; }**  
The required amount of propellant.

**Double TotalResourceAvailable { get; }**  
The total amount of the underlying resource currently reachable given resource flow rules.

**Double TotalResourceCapacity { get; }**  
The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

**Boolean IgnoreForIsp { get; }**  
If this propellant should be ignored when calculating required mass flow given specific impulse.

**Boolean IgnoreForThrustCurve { get; }**  
If this propellant should be ignored for thrust curve calculations.

**Boolean DrawStackGauge { get; }**  
If this propellant has a stack gauge or not.

**Boolean IsDeprived { get; }**  
If this propellant is deprived.

**Single Ratio { get; }**  
The propellant ratio.

## Experiment

**class Experiment**  
Obtained by calling *Part.Experiment*.

**Part Part { get; }**  
The part object for this experiment.

**void Run ()**  
Run the experiment.

**void Transmit ()**  
Transmit all experimental data contained by this part.

**void Dump ()**  
Dump the experimental data contained by the experiment.

**void Reset ()**  
Reset the experiment.

**Boolean Deployed { get; }**  
Whether the experiment has been deployed.

**Boolean Rerunnable { get; }**  
Whether the experiment can be re-run.

**Boolean Inoperable { get; }**  
Whether the experiment is inoperable.

**Boolean HasData { get; }**  
Whether the experiment contains data.

**IList<ScienceData> Data { get; }**  
The data contained in this experiment.

**String Biome { get; }**

The name of the biome the experiment is currently in.

**Boolean Available { get; }**

Determines if the experiment is available given the current conditions.

**ScienceSubject ScienceSubject { get; }**

Containing information on the corresponding specific science result for the current conditions. Returns `null` if the experiment is unavailable.

**class ScienceData**

Obtained by calling *Experiment.Data*.

**Single DataAmount { get; }**

Data amount.

**Single ScienceValue { get; }**

Science value.

**Single TransmitValue { get; }**

Transmit value.

**class ScienceSubject**

Obtained by calling *Experiment.ScienceSubject*.

**String Title { get; }**

Title of science subject, displayed in science archives

**Boolean IsComplete { get; }**

Whether the experiment has been completed.

**Single Science { get; }**

Amount of science already earned from this subject, not updated until after transmission/recovery.

**Single ScienceCap { get; }**

Total science allowable for this subject.

**Single DataScale { get; }**

Multiply science value by this to determine data amount in mits.

**Single SubjectValue { get; }**

Multiplier for specific Celestial Body/Experiment Situation combination.

**Single ScientificValue { get; }**

Diminishing value multiplier for decreasing the science value returned from repeated experiments.

## Fairing

**class Fairing**

A fairing. Obtained by calling *Part.Fairing*.

**Part Part { get; }**

The part object for this fairing.

**void Jettison ()**

Jettison the fairing. Has no effect if it has already been jettisoned.

**Boolean Jettisoned { get; }**

Whether the fairing has been jettisoned.

## Intake

### **class Intake**

An air intake. Obtained by calling *Part.Intake*.

*Part* **Part** { **get**; }

The part object for this intake.

**Boolean** **Open** { **get**; **set**; }

Whether the intake is open.

**Single** **Speed** { **get**; }

Speed of the flow into the intake, in *m/s*.

**Single** **Flow** { **get**; }

The rate of flow into the intake, in units of resource per second.

**Single** **Area** { **get**; }

The area of the intake's opening, in square meters.

## Leg

### **class Leg**

A landing leg. Obtained by calling *Part.Leg*.

*Part* **Part** { **get**; }

The part object for this landing leg.

*LegState* **State** { **get**; }

The current state of the landing leg.

**Boolean** **Deployable** { **get**; }

Whether the leg is deployable.

**Boolean** **Deployed** { **get**; **set**; }

Whether the landing leg is deployed.

---

**Note:** Fixed landing legs are always deployed. Returns an error if you try to deploy fixed landing gear.

---

**Boolean** **IsGrounded** { **get**; }

Returns whether the leg is touching the ground.

### **enum LegState**

The state of a landing leg. See *Leg.State*.

#### **Deployed**

Landing leg is fully deployed.

#### **Retracted**

Landing leg is fully retracted.

#### **Deploying**

Landing leg is being deployed.

#### **Retracting**

Landing leg is being retracted.

#### **Broken**

Landing leg is broken.

## Launch Clamp

### class LaunchClamp

A launch clamp. Obtained by calling *Part.LaunchClamp*.

*Part* **Part** { **get**; }

The part object for this launch clamp.

void **Release** ()

Releases the docking clamp. Has no effect if the clamp has already been released.

## Light

### class Light

A light. Obtained by calling *Part.Light*.

*Part* **Part** { **get**; }

The part object for this light.

**Boolean** **Active** { **get**; **set**; }

Whether the light is switched on.

**Tuple**<**Single**, **Single**, **Single**> **Color** { **get**; **set**; }

The color of the light, as an RGB triple.

**Single** **PowerUsage** { **get**; }

The current power usage, in units of charge per second.

## Parachute

### class Parachute

A parachute. Obtained by calling *Part.Parachute*.

*Part* **Part** { **get**; }

The part object for this parachute.

void **Deploy** ()

Deploys the parachute. This has no effect if the parachute has already been deployed.

**Boolean** **Deployed** { **get**; }

Whether the parachute has been deployed.

void **Arm** ()

Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

**Boolean** **Armed** { **get**; }

Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

*ParachuteState* **State** { **get**; }

The current state of the parachute.

**Single** **DeployAltitude** { **get**; **set**; }

The altitude at which the parachute will full deploy, in meters. Only applicable to stock parachutes.

**Single** **DeployMinPressure** { **get**; **set**; }

The minimum pressure at which the parachute will semi-deploy, in atmospheres. Only applicable to stock parachutes.

**enum ParachuteState**

The state of a parachute. See *Parachute.State*.

**Stowed**

The parachute is safely tucked away inside its housing.

**Armed**

The parachute is armed for deployment. (RealChutes only)

**Active**

The parachute is still stowed, but ready to semi-deploy. (Stock parachutes only)

**SemiDeployed**

The parachute has been deployed and is providing some drag, but is not fully deployed yet. (Stock parachutes only)

**Deployed**

The parachute is fully deployed.

**Cut**

The parachute has been cut.

**Radiator****class Radiator**

A radiator. Obtained by calling *Part.Radiator*.

*Part* **Part** { **get**; }

The part object for this radiator.

**Boolean Deployable** { **get**; }

Whether the radiator is deployable.

**Boolean Deployed** { **get**; **set**; }

For a deployable radiator, `true` if the radiator is extended. If the radiator is not deployable, this is always `true`.

*RadiatorState* **State** { **get**; }

The current state of the radiator.

---

**Note:** A fixed radiator is always *RadiatorState.Extended*.

---

**enum RadiatorState**

The state of a radiator. *RadiatorState*

**Extended**

Radiator is fully extended.

**Retracted**

Radiator is fully retracted.

**Extending**

Radiator is being extended.

**Retracting**

Radiator is being retracted.

**Broken**

Radiator is being broken.

## Resource Converter

### class ResourceConverter

A resource converter. Obtained by calling *Part.ResourceConverter*.

*Part* **Part** { **get**; }

The part object for this converter.

*Int32* **Count** { **get**; }

The number of converters in the part.

*String* **Name** (*Int32 index*)

The name of the specified converter.

#### Parameters

- **index** – Index of the converter.

*Boolean* **Active** (*Int32 index*)

True if the specified converter is active.

#### Parameters

- **index** – Index of the converter.

*void* **Start** (*Int32 index*)

Start the specified converter.

#### Parameters

- **index** – Index of the converter.

*void* **Stop** (*Int32 index*)

Stop the specified converter.

#### Parameters

- **index** – Index of the converter.

*ResourceConverterState* **State** (*Int32 index*)

The state of the specified converter.

#### Parameters

- **index** – Index of the converter.

*String* **StatusInfo** (*Int32 index*)

Status information for the specified converter. This is the full status message shown in the in-game UI.

#### Parameters

- **index** – Index of the converter.

*ICollection<String>* **Inputs** (*Int32 index*)

List of the names of resources consumed by the specified converter.

#### Parameters

- **index** – Index of the converter.

*ICollection<String>* **Outputs** (*Int32 index*)

List of the names of resources produced by the specified converter.

#### Parameters

- **index** – Index of the converter.

**enum ResourceConverterState**

The state of a resource converter. See *ResourceConverter.State*.

**Running**

Converter is running.

**Idle**

Converter is idle.

**MissingResource**

Converter is missing a required resource.

**StorageFull**

No available storage for output resource.

**Capacity**

At preset resource capacity.

**Unknown**

Unknown state. Possible with modified resource converters. In this case, check *ResourceConverter.StatusInfo* for more information.

## Resource Harvester

**class ResourceHarvester**

A resource harvester (drill). Obtained by calling *Part.ResourceHarvester*.

*Part* **Part** { **get**; }

The part object for this harvester.

*ResourceHarvesterState* **State** { **get**; }

The state of the harvester.

**Boolean** **Deployed** { **get**; **set**; }

Whether the harvester is deployed.

**Boolean** **Active** { **get**; **set**; }

Whether the harvester is actively drilling.

**Single** **ExtractionRate** { **get**; }

The rate at which the drill is extracting ore, in units per second.

**Single** **ThermalEfficiency** { **get**; }

The thermal efficiency of the drill, as a percentage of its maximum.

**Single** **CoreTemperature** { **get**; }

The core temperature of the drill, in Kelvin.

**Single** **OptimumCoreTemperature** { **get**; }

The core temperature at which the drill will operate with peak efficiency, in Kelvin.

**enum ResourceHarvesterState**

The state of a resource harvester. See *ResourceHarvester.State*.

**Deploying**

The drill is deploying.

**Deployed**

The drill is deployed and ready.

**Retracting**

The drill is retracting.



**Retracted**

The drill is retracted.

**Active**

The drill is running.

**Reaction Wheel****class ReactionWheel**

A reaction wheel. Obtained by calling *Part.ReactionWheel*.

**Part** **Part** { **get**; }

The part object for this reaction wheel.

**Boolean** **Active** { **get**; **set**; }

Whether the reaction wheel is active.

**Boolean** **Broken** { **get**; }

Whether the reaction wheel is broken.

**Tuple**<**Tuple**<**Double**, **Double**, **Double**>, **Tuple**<**Double**, **Double**, **Double**>> **AvailableTorque** { **get**; }

The available torque, in Newton meters, that can be produced by this reaction wheel, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.ReferenceFrame*. Returns zero if the reaction wheel is inactive or broken.

**Tuple**<**Tuple**<**Double**, **Double**, **Double**>, **Tuple**<**Double**, **Double**, **Double**>> **MaxTorque** { **get**; }

The maximum torque, in Newton meters, that can be produced by this reaction wheel, when it is active, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.ReferenceFrame*.

**RCS****class RCS**

An RCS block or thruster. Obtained by calling *Part.RCS*.

**Part** **Part** { **get**; }

The part object for this RCS.

**Boolean** **Active** { **get**; }

Whether the RCS thrusters are active. An RCS thruster is inactive if the RCS action group is disabled (*Control.RCS*), the RCS thruster itself is not enabled (*RCS.Enabled*) or it is covered by a fairing (*Part.Shielded*).

**Boolean** **Enabled** { **get**; **set**; }

Whether the RCS thrusters are enabled.

**Boolean** **PitchEnabled** { **get**; **set**; }

Whether the RCS thruster will fire when pitch control input is given.

**Boolean** **YawEnabled** { **get**; **set**; }

Whether the RCS thruster will fire when yaw control input is given.

**Boolean** **RollEnabled** { **get**; **set**; }

Whether the RCS thruster will fire when roll control input is given.

**Boolean** **ForwardEnabled** { **get**; **set**; }

Whether the RCS thruster will fire when pitch control input is given.

**Boolean UpEnabled { get; set; }**

Whether the RCS thruster will fire when yaw control input is given.

**Boolean RightEnabled { get; set; }**

Whether the RCS thruster will fire when roll control input is given.

**Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> AvailableTorque { get; }**

The available torque, in Newton meters, that can be produced by this RCS, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.ReferenceFrame*. Returns zero if RCS is disable.

**Single MaxThrust { get; }**

The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

**Single MaxVacuumThrust { get; }**

The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

**IList<Thruster> Thrusters { get; }**

A list of thrusters, one of each nozzle in the RCS part.

**Single SpecificImpulse { get; }**

The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

**Single VacuumSpecificImpulse { get; }**

The vacuum specific impulse of the RCS, in seconds.

**Single KerbinSeaLevelSpecificImpulse { get; }**

The specific impulse of the RCS at sea level on Kerbin, in seconds.

**IList<String> Propellants { get; }**

The names of resources that the RCS consumes.

**IDictionary<String, Single> PropellantRatios { get; }**

The ratios of resources that the RCS consumes. A dictionary mapping resource names to the ratios at which they are consumed by the RCS.

**Boolean HasFuel { get; }**

Whether the RCS has fuel available.

---

**Note:** The RCS thruster must be activated for this property to update correctly.

---

## Sensor

### class Sensor

A sensor, such as a thermometer. Obtained by calling *Part.Sensor*.

**Part Part { get; }**

The part object for this sensor.

**Boolean Active { get; set; }**

Whether the sensor is active.

**String Value { get; }**

The current value of the sensor.

## Solar Panel

### class SolarPanel

A solar panel. Obtained by calling *Part.SolarPanel*.

*Part* **Part** { **get**; }

The part object for this solar panel.

**Boolean Deployable** { **get**; }

Whether the solar panel is deployable.

**Boolean Deployed** { **get**; **set**; }

Whether the solar panel is extended.

*SolarPanelState* **State** { **get**; }

The current state of the solar panel.

**Single EnergyFlow** { **get**; }

The current amount of energy being generated by the solar panel, in units of charge per second.

**Single SunExposure** { **get**; }

The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

### enum SolarPanelState

The state of a solar panel. See *SolarPanel.State*.

**Extended**

Solar panel is fully extended.

**Retracted**

Solar panel is fully retracted.

**Extending**

Solar panel is being extended.

**Retracting**

Solar panel is being retracted.

**Broken**

Solar panel is broken.

## Thruster

### class Thruster

The component of an *Engine* or *RCS* part that generates thrust. Can obtained by calling *Engine.Thrusters* or *RCS.Thrusters*.

---

**Note:** Engines can consist of multiple thrusters. For example, the S3 KS-25x4 “Mammoth” has four rocket nozzels, and so consists of four thrusters.

---

*Part* **Part** { **get**; }

The *Part* that contains this thruster.

**Tuple<Double, Double, Double> ThrustPosition** (*ReferenceFrame referenceFrame*)

The position at which the thruster generates thrust, in the given reference frame. For gimballed engines, this takes into account the current rotation of the gimbal.

**Parameters**

- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Tuple<Double, Double, Double> ThrustDirection** (*ReferenceFrame referenceFrame*)

The direction of the force generated by the thruster, in the given reference frame. This is opposite to the direction in which the thruster expels propellant. For gimballed engines, this takes into account the current rotation of the gimbal.

**Parameters**

- **referenceFrame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

*ReferenceFrame* **ThrustReferenceFrame** { **get**; }

A reference frame that is fixed relative to the thruster and orientated with its thrust direction (*Thruster.ThrustDirection*). For gimballed engines, this takes into account the current rotation of the gimbal.

- The origin is at the position of thrust for this thruster (*Thruster.ThrustPosition*).
- The axes rotate with the thrust direction. This is the direction in which the thruster expels propellant, including any gimbaling.
- The y-axis points along the thrust direction.
- The x-axis and z-axis are perpendicular to the thrust direction.

**Boolean Gimbaled** { **get**; }

Whether the thruster is gimballed.

**Tuple<Double, Double, Double> GimbalPosition** (*ReferenceFrame referenceFrame*)

Position around which the gimbal pivots.

**Parameters**

- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Tuple<Double, Double, Double> GimbalAngle** { **get**; }

The current gimbal angle in the pitch, roll and yaw axes, in degrees.

**Tuple<Double, Double, Double> InitialThrustPosition** (*ReferenceFrame referenceFrame*)

The position at which the thruster generates thrust, when the engine is in its initial position (no gimbaling), in the given reference frame.

**Parameters**

- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

---

**Note:** This position can move when the gimbal rotates. This is because the thrust position and gimbal position are not necessarily the same.

---

**Tuple<Double, Double, Double> InitialThrustDirection** (*ReferenceFrame referenceFrame*)

The direction of the force generated by the thruster, when the engine is in its initial position (no gimbaling), in the given reference frame. This is opposite to the direction in which the thruster expels propellant.

**Parameters**

- **referenceFrame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

## Wheel

### class Wheel

A wheel. Includes landing gear and rover wheels. Obtained by calling *Part.Wheel*. Can be used to control the motors, steering and deployment of wheels, among other things.

*Part* **Part** { **get**; }

The part object for this wheel.

*WheelState* **State** { **get**; }

The current state of the wheel.

*Single* **Radius** { **get**; }

Radius of the wheel, in meters.

*Boolean* **Grounded** { **get**; }

Whether the wheel is touching the ground.

*Boolean* **HasBrakes** { **get**; }

Whether the wheel has brakes.

*Single* **Brakes** { **get**; **set**; }

The braking force, as a percentage of maximum, when the brakes are applied.

*Boolean* **AutoFrictionControl** { **get**; **set**; }

Whether automatic friction control is enabled.

*Single* **ManualFrictionControl** { **get**; **set**; }

Manual friction control value. Only has an effect if automatic friction control is disabled. A value between 0 and 5 inclusive.

*Boolean* **Deployable** { **get**; }

Whether the wheel is deployable.

*Boolean* **Deployed** { **get**; **set**; }

Whether the wheel is deployed.

*Boolean* **Powered** { **get**; }

Whether the wheel is powered by a motor.

*Boolean* **MotorEnabled** { **get**; **set**; }

Whether the motor is enabled.

*Boolean* **MotorInverted** { **get**; **set**; }

Whether the direction of the motor is inverted.

*MotorState* **MotorState** { **get**; }

Whether the direction of the motor is inverted.

*Single* **MotorOutput** { **get**; }

The output of the motor. This is the torque currently being generated, in Newton meters.

*Boolean* **TractionControlEnabled** { **get**; **set**; }

Whether automatic traction control is enabled. A wheel only has traction control if it is powered.

*Single* **TractionControl** { **get**; **set**; }

Setting for the traction control. Only takes effect if the wheel has automatic traction control enabled. A value between 0 and 5 inclusive.

**Single DriveLimiter { get; set; }**

Manual setting for the motor limiter. Only takes effect if the wheel has automatic traction control disabled.  
A value between 0 and 100 inclusive.

**Boolean Steerable { get; }**

Whether the wheel has steering.

**Boolean SteeringEnabled { get; set; }**

Whether the wheel steering is enabled.

**Boolean SteeringInverted { get; set; }**

Whether the wheel steering is inverted.

**Boolean HasSuspension { get; }**

Whether the wheel has suspension.

**Single SuspensionSpringStrength { get; }**

Suspension spring strength, as set in the editor.

**Single SuspensionDamperStrength { get; }**

Suspension damper strength, as set in the editor.

**Boolean Broken { get; }**

Whether the wheel is broken.

**Boolean Repairable { get; }**

Whether the wheel is repairable.

**Single Stress { get; }**

Current stress on the wheel.

**Single StressTolerance { get; }**

Stress tolerance of the wheel.

**Single StressPercentage { get; }**

Current stress on the wheel as a percentage of its stress tolerance.

**Single Deflection { get; }**

Current deflection of the wheel.

**Single Slip { get; }**

Current slip of the wheel.

**enum WheelState**

The state of a wheel. See *Wheel.State*.

**Deployed**

Wheel is fully deployed.

**Retracted**

Wheel is fully retracted.

**Deploying**

Wheel is being deployed.

**Retracting**

Wheel is being retracted.

**Broken**

Wheel is broken.

**enum MotorState**

The state of the motor on a powered wheel. See *Wheel.MotorState*.

- Idle
- The motor is idle.
- Running
- The motor is running.
- Disabled
- The motor is disabled.
- Inoperable
- The motor is inoperable.
- NotEnoughResources
- The motor does not have enough resources to run.

Trees of Parts

Vessels in KSP are comprised of a number of parts, connected to one another in a *tree* structure. An example vessel is shown in Figure 1, and the corresponding tree of parts in Figure 2. The craft file for this example can also be downloaded [here](#).

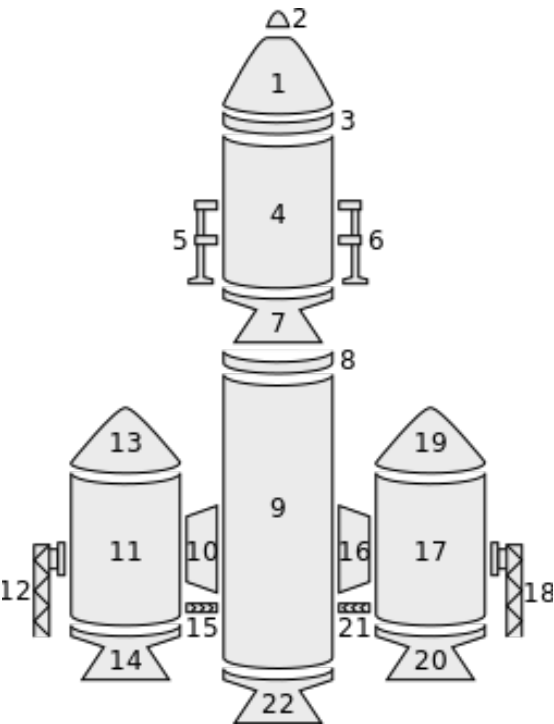


Fig. 3.10: **Figure 1** – Example parts making up a vessel.

Traversing the Tree

The tree of parts can be traversed using the attributes *Parts.Root*, *Part.Parent* and *Part.Children*.

The root of the tree is the same as the vessels *root part* (part number 1 in the example above) and can be obtained by calling *Parts.Root*. A parts children can be obtained by calling *Part.Children*. If the part does not have any children, *Part.Children* returns an empty list. A parts parent can be obtained by calling *Part.Parent*. If the part does not have a parent (as is the case for the root part), *Part.Parent* returns null.

The following C# example uses these attributes to perform a depth-first traversal over all of the parts in a vessel:

```
using System;
using System.Collections.Generic;
using System.Net;
using KRPC.Client;
using KRPC.Client.Sender;

class AttachmentMode
{
    public static void
    {
        using (var connect
            var vess
            = connection.Space
            var root

            var stack = new
            stack.Push (new T
```

```
while (s
    var
    Part
    int
    Cons
    ↪(new String (' ',
    ↪
    ↪(new Tuple<Part, in
        }
    }
}
```

When this code is execute using the craft file for the example vessel pictured above, the following is printed out:

```
Command Pod Mk1
TR-18A Stack Decoupl
FL-T400 Fuel Tank
LV-909 Liquid Fue
TR-18A Stack Dec
FL-T800 Fuel Ta
LV-909 Liquid
TT-70 Radial D
FL-T400 Fuel
TT18-A Launc
FTX-2 Extern
LV-909 Liqui
Aerodynamic
TT-70 Radial D
FL-T400 Fuel
TT18-A Launc
FTX-2 Extern
LV-909 Liqui
Aerodynamic
LT-1 Landing Stru
LT-1 Landing Stru
Mk16 Parachute
```

Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

The root part of a vessel (for example the command pod – part 1) does not have a parent part, so does not have an attachment mode. However, the part is consider to be *axially* attached to nothing.



The following C# example does a depth-first traversal as before, but also prints out the attachment mode used by the part:

```
using System;
using System.Collections.Generic;
using System.Net;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class AttachmentModes
{
    public static void Main ()
    {
        using (var connection = new Connection ()) {
            var vessel_ =
            = connection.SpaceCenter ().ActiveVessel;
            var root = vessel_.Parts.Root;

            var stack = new Stack<Tuple<Part,int>> ();
            stack.Push (new Tuple<Part,int> (root, 0));
            while (stack.Count > 0) {
                var item = stack.Pop ();
                Part part = item.Item1;
                int depth = item.Item2;
                string attachMode =
            (part.AxiallyAttached ? "axial" : "radial");

                Console.WriteLine (new String (' ',
            depth) + part.Title + " - " + attachMode);

                foreach (var child in part.Children)
                    stack.Push_
            (new Tuple<Part,int> (child, depth + 1));
            }
        }
    }
}
```

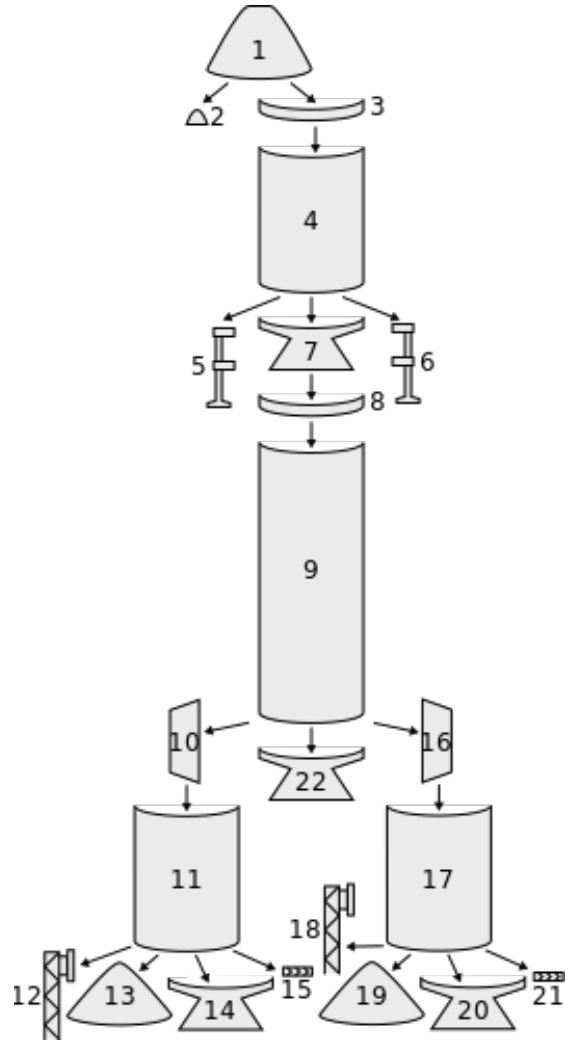


Fig. 3.11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the child part.

When this code is executed using the craft file for the example vessel pictured above, the following is printed out:

```
Command Pod Mk1 - axial
TR-18A Stack Decoupler - axial
FL-T400 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TR-18A Stack Decoupler - axial
FL-T800 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial

TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
TT-70 Radial Decoupler - radial
```

```

FL-T400 Fuel Tank - radial
→ TT18-A Launch Stability Enhancer - radial
  FTX-2 External Fuel Duct - radial
  LV-909 Liquid Fuel Engine - axial
  Aerodynamic Nose Cone - axial
  LT-1 Landing Struts - radial
  LT-1 Landing Struts - radial
  Mk16 Parachute - axial

```

## Fuel Lines

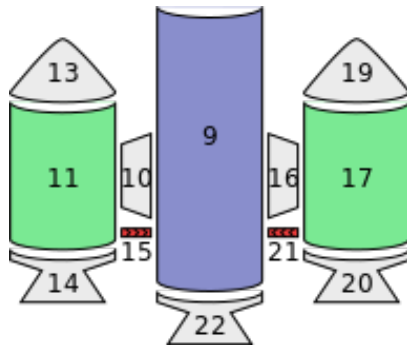


Fig. 3.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.

The attributes *Part.FuelLinesFrom* and *Part.FuelLinesTo* can be used to discover these connections. In the example in Figure 5, when *Part.FuelLinesTo* is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When *Part.FuelLinesFrom* is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).

## Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using *Part.Stage* and *Part.DecoupleStage* respectively. For parts that are not activated by staging, *Part.Stage* returns -1. For parts that are never decoupled, *Part.DecoupleStage* returns a value of -1.

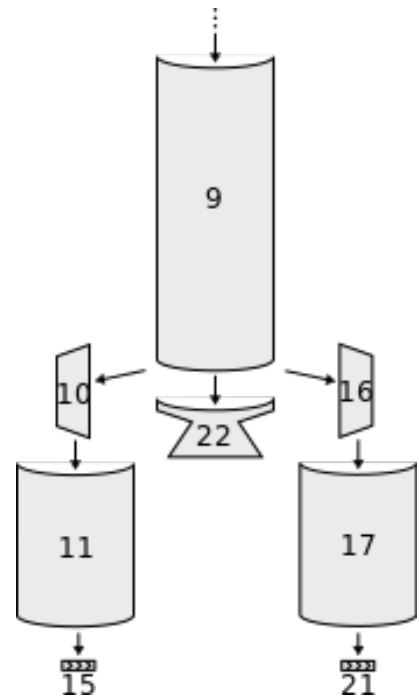


Fig. 3.13: **Figure 4** – A subset of the parts tree from Figure 2 above.

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *activated*. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

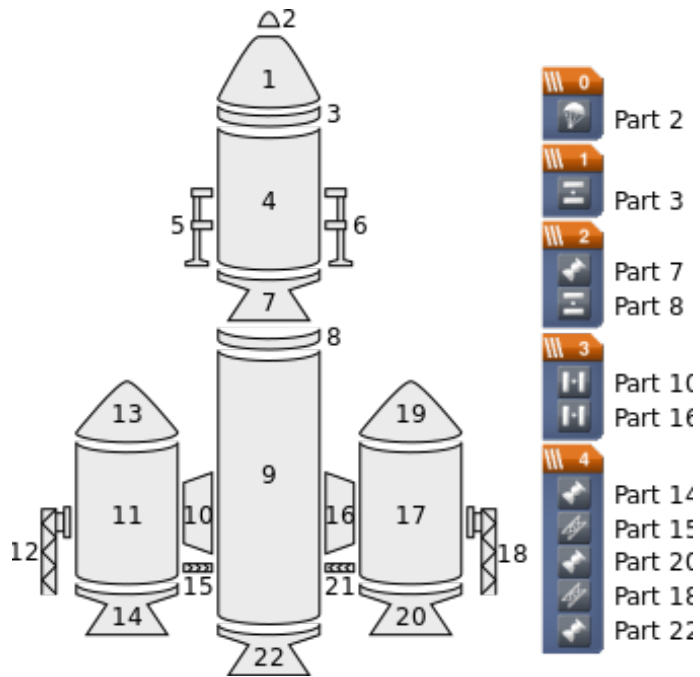


Fig. 3.14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

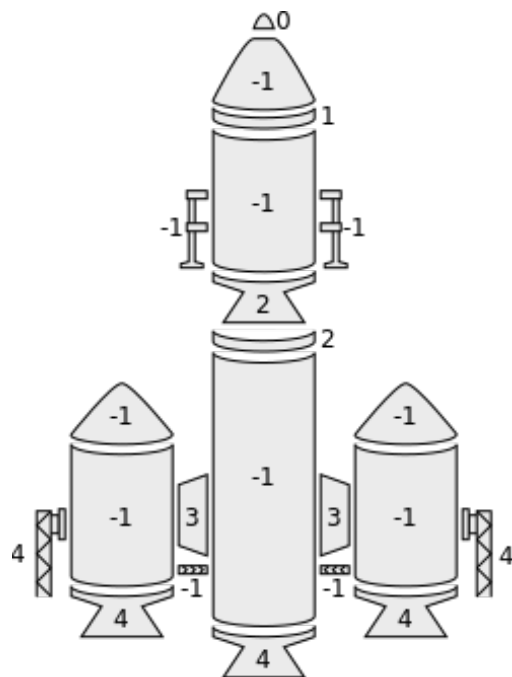


Fig. 3.15: **Figure 7** – The stage in which each part is *activated*.

### 3.3.9 Resources

#### class Resources

Represents the collection of resources stored in a vessel, stage or part. Created by calling *Vessel.Resources*, *Vessel.ResourcesInDecoupleStage* or *Part.Resources*.

**IList<Resource> All { get; }**

All the individual resources that can be stored.

**IList<Resource> WithResource (String name)**

All the individual resources with the given name that can be stored.

#### Parameters

**IList<String> Names { get; }**

A list of resource names that can be stored.

**Boolean HasResource (String name)**

Check whether the named resource can be stored.

#### Parameters

- **name** – The name of the resource.

**Single Amount (String name)**

Returns the amount of a resource that is currently stored.

#### Parameters

- **name** – The name of the resource.

**Single Max (String name)**

Returns the amount of a resource that can be stored.

#### Parameters

- **name** – The name of the resource.

**static Single Density (IConnection connection, String name)**

Returns the density of a resource, in *kg/l*.

#### Parameters

- **name** – The name of the resource.

**static ResourceFlowMode FlowMode (IConnection connection, String name)**

Returns the flow mode of a resource.

#### Parameters

- **name** – The name of the resource.

**Boolean Enabled { get; set; }**

Whether use of all the resources are enabled.

---

**Note:** This is `true` if all of the resources are enabled. If any of the resources are not enabled, this is `false`.

---

#### class Resource

An individual resource stored within a part. Created using methods in the *Resources* class.

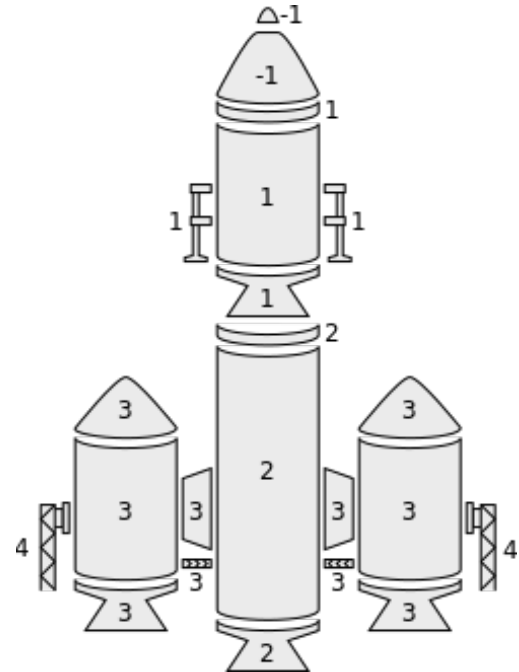


Fig. 3.16: **Figure 8** – The stage in which each part is *decoupled*.

**String Name { get; }**

The name of the resource.

**Part Part { get; }**

The part containing the resource.

**Single Amount { get; }**

The amount of the resource that is currently stored in the part.

**Single Max { get; }**

The total amount of the resource that can be stored in the part.

**Single Density { get; }**

The density of the resource, in *kg/l*.

**ResourceFlowMode FlowMode { get; }**

The flow mode of the resource.

**Boolean Enabled { get; set; }**

Whether use of this resource is enabled.

**class ResourceTransfer**

Transfer resources between parts.

**static ResourceTransfer Start** (*ICConnection connection, Part fromPart, Part toPart, String resource, Single maxAmount*)

Start transferring a resource transfer between a pair of parts. The transfer will move at most *maxAmount* units of the resource, depending on how much of the resource is available in the source part and how much storage is available in the destination part. Use *ResourceTransfer.Complete* to check if the transfer is complete. Use *ResourceTransfer.Amount* to see how much of the resource has been transferred.

#### Parameters

- **fromPart** – The part to transfer to.
- **toPart** – The part to transfer from.
- **resource** – The name of the resource to transfer.
- **maxAmount** – The maximum amount of resource to transfer.

**Single Amount { get; }**

The amount of the resource that has been transferred.

**Boolean Complete { get; }**

Whether the transfer has completed.

**enum ResourceFlowMode**

The way in which a resource flows between parts. See *Resources.FlowMode*.

**Vessel**

The resource flows to any part in the vessel. For example, electric charge.

**Stage**

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

**Adjacent**

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

**None**

The resource does not flow. For example, solid fuel.

### 3.3.10 Node

**class Node**

Represents a maneuver node. Can be created using *Control.AddNode*.

**Double Prograde { get; set; }**

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

**Double Normal { get; set; }**

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

**Double Radial { get; set; }**

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

**Double DeltaV { get; set; }**

The delta-v of the maneuver node, in meters per second.

---

**Note:** Does not change when executing the maneuver node. See *Node.RemainingDeltaV*.

---

**Double RemainingDeltaV { get; }**

Gets the remaining delta-v of the maneuver node, in meters per second. Changes as the node is executed. This is equivalent to the delta-v reported in-game.

**Tuple<Double, Double, Double> BurnVector (ReferenceFrame referenceFrame = null)**

Returns the burn vector for the maneuver node.

**Parameters**

- **referenceFrame** – The reference frame that the returned vector is in. Defaults to *Vessel.OrbitalReferenceFrame*.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

---

**Note:** Does not change when executing the maneuver node. See *Node.RemainingBurnVector*.

---

**Tuple<Double, Double, Double> RemainingBurnVector (ReferenceFrame referenceFrame = null)**

Returns the remaining burn vector for the maneuver node.

**Parameters**

- **referenceFrame** – The reference frame that the returned vector is in. Defaults to *Vessel.OrbitalReferenceFrame*.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

---

**Note:** Changes as the maneuver node is executed. See *Node.BurnVector*.

---

**Double UT { get; set; }**

The universal time at which the maneuver will occur, in seconds.

**Double TimeTo { get; }**

The time until the maneuver node will be encountered, in seconds.

**Orbit Orbit { get; }**

The orbit that results from executing the maneuver node.

**void Remove ()**

Removes the maneuver node.

**ReferenceFrame ReferenceFrame { get; }**

The reference frame that is fixed relative to the maneuver node's burn.

- The origin is at the position of the maneuver node.
- The y-axis points in the direction of the burn.
- The x-axis and z-axis point in arbitrary but fixed directions.

**ReferenceFrame OrbitalReferenceFrame { get; }**

The reference frame that is fixed relative to the maneuver node, and orientated with the orbital prograde/normal/radial directions of the original orbit at the maneuver node's position.

- The origin is at the position of the maneuver node.
- The x-axis points in the orbital anti-radial direction of the original orbit, at the position of the maneuver node.
- The y-axis points in the orbital prograde direction of the original orbit, at the position of the maneuver node.
- The z-axis points in the orbital normal direction of the original orbit, at the position of the maneuver node.

**Tuple<Double, Double, Double> Position (ReferenceFrame referenceFrame)**

The position vector of the maneuver node in the given reference frame.

#### Parameters

- **referenceFrame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Tuple<Double, Double, Double> Direction (ReferenceFrame referenceFrame)**

The direction of the maneuver nodes burn.

#### Parameters

- **referenceFrame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

### 3.3.11 ReferenceFrame

#### **class ReferenceFrame**

Represents a reference frame for positions, rotations and velocities.

Contains:

- The position of the origin.
- The directions of the x, y and z axes.
- The linear velocity of the frame.
- The angular velocity of the frame.

---

**Note:** This class does not contain any properties or methods. It is only used as a parameter to other functions.

---

*static ReferenceFrame* **CreateRelative** (*ICConnection connection, ReferenceFrame referenceFrame, Tuple<Double, Double, Double> position = null, Tuple<Double, Double, Double, Double> rotation = null, Tuple<Double, Double, Double> velocity = null, Tuple<Double, Double, Double> angularVelocity = null*)

Create a relative reference frame. This is a custom reference frame whose components offset the components of a parent reference frame.

#### **Parameters**

- **referenceFrame** – The parent reference frame on which to base this reference frame.
- **position** – The offset of the position of the origin, as a position vector. Defaults to (0, 0, 0)
- **rotation** – The rotation to apply to the parent frames rotation, as a quaternion of the form  $(x, y, z, w)$ . Defaults to (0, 0, 0, 1) (i.e. no rotation)
- **velocity** – The linear velocity to offset the parent frame by, as a vector pointing in the direction of travel, whose magnitude is the speed in meters per second. Defaults to (0, 0, 0).
- **angularVelocity** – The angular velocity to offset the parent frame by, as a vector. This vector points in the direction of the axis of rotation, and its magnitude is the speed of the rotation in radians per second. Defaults to (0, 0, 0).

*static ReferenceFrame* **CreateHybrid** (*ICConnection connection, ReferenceFrame position, ReferenceFrame rotation = null, ReferenceFrame velocity = null, ReferenceFrame angularVelocity = null*)

Create a hybrid reference frame. This is a custom reference frame whose components inherited from other reference frames.

#### **Parameters**

- **position** – The reference frame providing the position of the origin.



- **rotation** – The reference frame providing the rotation of the frame.
- **velocity** – The reference frame providing the linear velocity of the frame.
- **angularVelocity** – The reference frame providing the angular velocity of the frame.

---

**Note:** The *position* reference frame is required but all other reference frames are optional. If omitted, they are set to the *position* reference frame.

---

### 3.3.12 AutoPilot

#### **class AutoPilot**

Provides basic auto-piloting utilities for a vessel. Created by calling *Vessel.AutoPilot*.

---

**Note:** If a client engages the auto-pilot and then closes its connection to the server, the auto-pilot will be disengaged and its target reference frame, direction and roll reset to default.

---

void **Engage** ()  
Engage the auto-pilot.

void **Disengage** ()  
Disengage the auto-pilot.

void **Wait** ()  
Blocks until the vessel is pointing in the target direction and has the target roll (if set). Throws an exception if the auto-pilot has not been engaged.

Single **Error** { **get**; }  
The error, in degrees, between the direction the ship has been asked to point in and the direction it is pointing in. Throws an exception if the auto-pilot has not been engaged and SAS is not enabled or is in stability assist mode.

Single **PitchError** { **get**; }  
The error, in degrees, between the vessels current and target pitch. Throws an exception if the auto-pilot has not been engaged.

Single **HeadingError** { **get**; }  
The error, in degrees, between the vessels current and target heading. Throws an exception if the auto-pilot has not been engaged.

Single **RollError** { **get**; }  
The error, in degrees, between the vessels current and target roll. Throws an exception if the auto-pilot has not been engaged or no target roll is set.

ReferenceFrame **ReferenceFrame** { **get**; **set**; }  
The reference frame for the target direction (*AutoPilot.TargetDirection*).

---

**Note:** An error will be thrown if this property is set to a reference frame that rotates with the vessel being controlled, as it is impossible to rotate the vessel in such a reference frame.

---

**Single TargetPitch { get; set; }**

The target pitch, in degrees, between -90° and +90°.

**Single TargetHeading { get; set; }**

The target heading, in degrees, between 0° and 360°.

**Single TargetRoll { get; set; }**

The target roll, in degrees. NaN if no target roll is set.

**Tuple<Double, Double, Double> TargetDirection { get; set; }**

Direction vector corresponding to the target pitch and heading. This is in the reference frame specified by *ReferenceFrame*.

**void TargetPitchAndHeading (Single pitch, Single heading)**

Set target pitch and heading angles.

#### Parameters

- **pitch** – Target pitch angle, in degrees between -90° and +90°.
- **heading** – Target heading angle, in degrees between 0° and 360°.

**Boolean SAS { get; set; }**

The state of SAS.

---

**Note:** Equivalent to *Control.SAS*

---

**SASMode SASMode { get; set; }**

The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

---

**Note:** Equivalent to *Control.SASMode*

---

**Double RollThreshold { get; set; }**

The threshold at which the autopilot will try to match the target roll angle, if any. Defaults to 5 degrees.

**Tuple<Double, Double, Double> StoppingTime { get; set; }**

The maximum amount of time that the vessel should need to come to a complete stop. This determines the maximum angular velocity of the vessel. A vector of three stopping times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 0.5 seconds for each axis.

**Tuple<Double, Double, Double> DecelerationTime { get; set; }**

The time the vessel should take to come to a stop pointing in the target direction. This determines the angular acceleration used to decelerate the vessel. A vector of three times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 5 seconds for each axis.

**Tuple<Double, Double, Double> AttenuationAngle { get; set; }**

The angle at which the autopilot considers the vessel to be pointing close to the target. This determines the midpoint of the target velocity attenuation function. A vector of three angles, in degrees, one for each of the pitch, roll and yaw axes. Defaults to 1° for each axis.

**Boolean AutoTune { get; set; }**

Whether the rotation rate controllers PID parameters should be automatically tuned using the vessels moment of inertia and available torque. Defaults to true. See *AutoPilot.TimeToPeak* and *AutoPilot.Overshoot*.

**Tuple<Double, Double, Double> TimeToPeak { get; set; }**

The target time to peak used to autotune the PID controllers. A vector of three times, in seconds, for each of the pitch, roll and yaw axes. Defaults to 3 seconds for each axis.

**Tuple<Double, Double, Double> Overshoot { get; set; }**

The target overshoot percentage used to autotune the PID controllers. A vector of three values, between 0 and 1, for each of the pitch, roll and yaw axes. Defaults to 0.01 for each axis.

**Tuple<Double, Double, Double> PitchPIDGains { get; set; }**

Gains for the pitch PID controller.

---

**Note:** When *AutoPilot.AutoTune* is true, these values are updated automatically, which will overwrite any manual changes.

---

**Tuple<Double, Double, Double> RollPIDGains { get; set; }**

Gains for the roll PID controller.

---

**Note:** When *AutoPilot.AutoTune* is true, these values are updated automatically, which will overwrite any manual changes.

---

**Tuple<Double, Double, Double> YawPIDGains { get; set; }**

Gains for the yaw PID controller.

---

**Note:** When *AutoPilot.AutoTune* is true, these values are updated automatically, which will overwrite any manual changes.

---

### 3.3.13 Camera

**class Camera**

Controls the game's camera. Obtained by calling *SpaceCenter.Camera*.

**CameraMode Mode { get; set; }**

The current mode of the camera.

**Single Pitch { get; set; }**

The pitch of the camera, in degrees. A value between *Camera.MinPitch* and *Camera.MaxPitch*

**Single Heading { get; set; }**

The heading of the camera, in degrees.

**Single Distance { get; set; }**

The distance from the camera to the subject, in meters. A value between *Camera.MinDistance* and *Camera.MaxDistance*.

**Single MinPitch { get; }**

The minimum pitch of the camera.

**Single MaxPitch { get; }**

The maximum pitch of the camera.

**Single MinDistance { get; }**

Minimum distance from the camera to the subject, in meters.

**Single MaxDistance { get; }**

Maximum distance from the camera to the subject, in meters.

**Single DefaultDistance { get; }**

Default distance from the camera to the subject, in meters.

**CelestialBody FocussedBody { get; set; }**

In map mode, the celestial body that the camera is focussed on. Returns *null* if the camera is not focussed on a celestial body. Returns an error if the camera is not in map mode.

**Vessel FocussedVessel { get; set; }**

In map mode, the vessel that the camera is focussed on. Returns *null* if the camera is not focussed on a vessel. Returns an error if the camera is not in map mode.

**Node FocussedNode { get; set; }**

In map mode, the maneuver node that the camera is focussed on. Returns *null* if the camera is not focussed on a maneuver node. Returns an error if the camera is not in map mode.

**enum CameraMode**

See *Camera.Mode*.

**Automatic**

The camera is showing the active vessel, in “auto” mode.

**Free**

The camera is showing the active vessel, in “free” mode.

**Chase**

The camera is showing the active vessel, in “chase” mode.

**Locked**

The camera is showing the active vessel, in “locked” mode.

**Orbital**

The camera is showing the active vessel, in “orbital” mode.

**IVA**

The Intra-Vehicular Activity view is being shown.

**Map**

The map view is being shown.

### 3.3.14 Waypoints

#### class WaypointManager

Waypoints are the location markers you can see on the map view showing you where contracts are targeted for. With this structure, you can obtain coordinate data for the locations of these waypoints.

Obtained by calling *SpaceCenter.WaypointManager*.

*ICollection<Waypoint>* **Waypoints** { **get**; }

A list of all existing waypoints.

*Waypoint* **AddWaypoint** (*Double* latitude, *Double* longitude, *CelestialBody* body, *String* name)

Creates a waypoint at the given position at ground level, and returns a *Waypoint* object that can be used to modify it.

#### Parameters

- **latitude** – Latitude of the waypoint.
- **longitude** – Longitude of the waypoint.
- **body** – Celestial body the waypoint is attached to.
- **name** – Name of the waypoint.

*Waypoint* **AddWaypointAtAltitude** (*Double* latitude, *Double* longitude, *Double* altitude, *CelestialBody* body, *String* name)

Creates a waypoint at the given position and altitude, and returns a *Waypoint* object that can be used to modify it.

#### Parameters

- **latitude** – Latitude of the waypoint.
- **longitude** – Longitude of the waypoint.
- **altitude** – Altitude (above sea level) of the waypoint.
- **body** – Celestial body the waypoint is attached to.
- **name** – Name of the waypoint.

*IDictionary<String, Int32>* **Colors** { **get**; }

An example map of known color - seed pairs. Any other integers may be used as seed.

*ICollection<String>* **Icons** { **get**; }

Returns all available icons (from "Game-Data/Squad/Contracts/Icons").

#### class Waypoint

Represents a waypoint. Can be created using *WaypointManager.AddWaypoint*.

*CelestialBody* **Body** { **get**; **set**; }

The celestial body the waypoint is attached to.

*String* **Name** { **get**; **set**; }

The name of the waypoint as it appears on the map and the contract.

*Int32* **Color** { **get**; **set**; }

The seed of the icon color. See *WaypointManager.Colors* for example colors.

**String Icon { get; set; }**

The icon of the waypoint.

**Double Latitude { get; set; }**

The latitude of the waypoint.

**Double Longitude { get; set; }**

The longitude of the waypoint.

**Double MeanAltitude { get; set; }**

The altitude of the waypoint above sea level, in meters.

**Double SurfaceAltitude { get; set; }**

The altitude of the waypoint above the surface of the body or sea level, whichever is closer, in meters.

**Double BedrockAltitude { get; set; }**

The altitude of the waypoint above the surface of the body, in meters. When over water, this is the altitude above the sea floor.

**Boolean NearSurface { get; }**

true if the waypoint is near to the surface of a body.

**Boolean Grounded { get; }**

true if the waypoint is attached to the ground.

**Int32 Index { get; }**

The integer index of this waypoint within its cluster of sibling waypoints. In other words, when you have a cluster of waypoints called “Somewhere Alpha”, “Somewhere Beta” and “Somewhere Gamma”, the alpha site has index 0, the beta site has index 1 and the gamma site has index 2. When *Waypoint.Clustered* is false, this is zero.

**Boolean Clustered { get; }**

true if this waypoint is part of a set of clustered waypoints with greek letter names appended (Alpha, Beta, Gamma, etc). If true, there is a one-to-one correspondence with the greek letter name and the *Waypoint.Index*.

**Boolean HasContract { get; }**

Whether the waypoint belongs to a contract.

**Contract Contract { get; }**

The associated contract.

**void Remove ()**

Removes the waypoint.

### 3.3.15 Contracts

**class ContractManager**

Contracts manager. Obtained by calling *SpaceCenter.WaypointManager*.

**ISet<String> Types { get; }**

A list of all contract types.

**IList<Contract> AllContracts { get; }**

A list of all contracts.

`IList<Contract> ActiveContracts { get; }`

A list of all active contracts.

`IList<Contract> OfferedContracts { get; }`

A list of all offered, but unaccepted, contracts.

`IList<Contract> CompletedContracts { get; }`

A list of all completed contracts.

`IList<Contract> FailedContracts { get; }`

A list of all failed contracts.

**class Contract**

A contract. Can be accessed using *SpaceCenter.ContractManager*.

`String Type { get; }`

Type of the contract.

`String Title { get; }`

Title of the contract.

`String Description { get; }`

Description of the contract.

`String Notes { get; }`

Notes for the contract.

`String Synopsis { get; }`

Synopsis for the contract.

`IList<String> Keywords { get; }`

Keywords for the contract.

`ContractState State { get; }`

State of the contract.

`Boolean Seen { get; }`

Whether the contract has been seen.

`Boolean Read { get; }`

Whether the contract has been read.

`Boolean Active { get; }`

Whether the contract is active.

`Boolean Failed { get; }`

Whether the contract has been failed.

`Boolean CanBeCanceled { get; }`

Whether the contract can be canceled.

`Boolean CanBeDeclined { get; }`

Whether the contract can be declined.

`Boolean CanBeFailed { get; }`

Whether the contract can be failed.

`void Accept ()`

Accept an offered contract.

`void Cancel ()`

Cancel an active contract.

**void Decline ()**  
Decline an offered contract.

**Double FundsAdvance { get; }**  
Funds received when accepting the contract.

**Double FundsCompletion { get; }**  
Funds received on completion of the contract.

**Double FundsFailure { get; }**  
Funds lost if the contract is failed.

**Double ReputationCompletion { get; }**  
Reputation gained on completion of the contract.

**Double ReputationFailure { get; }**  
Reputation lost if the contract is failed.

**Double ScienceCompletion { get; }**  
Science gained on completion of the contract.

**IList<ContractParameter> Parameters { get; }**  
Parameters for the contract.

**enum ContractState**  
The state of a contract. See *Contract.State*.

**Active**  
The contract is active.

**Canceled**  
The contract has been canceled.

**Completed**  
The contract has been completed.

**DeadlineExpired**  
The deadline for the contract has expired.

**Declined**  
The contract has been declined.

**Failed**  
The contract has been failed.

**Generated**  
The contract has been generated.

**Offered**  
The contract has been offered to the player.

**OfferExpired**  
The contract was offered to the player, but the offer expired.

**Withdrawn**  
The contract has been withdrawn.

**class ContractParameter**  
A contract parameter. See *Contract.Parameters*.

**String Title { get; }**  
Title of the parameter.



**String Notes { get; }**  
Notes for the parameter.

**IList<ContractParameter> Children { get; }**  
Child contract parameters.

**Boolean Completed { get; }**  
Whether the parameter has been completed.

**Boolean Failed { get; }**  
Whether the parameter has been failed.

**Boolean Optional { get; }**  
Whether the contract parameter is optional.

**Double FundsCompletion { get; }**  
Funds received on completion of the contract parameter.

**Double FundsFailure { get; }**  
Funds lost if the contract parameter is failed.

**Double ReputationCompletion { get; }**  
Reputation gained on completion of the contract parameter.

**Double ReputationFailure { get; }**  
Reputation lost if the contract parameter is failed.

**Double ScienceCompletion { get; }**  
Science gained on completion of the contract parameter.

### 3.3.16 Geometry Types

#### Vectors

3-dimensional vectors are represented as a 3-tuple. For example:

```
using System;
using System.Net;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class VectorExample
{
    public static void Main ()
    {
        using (var connection = new Connection ()) {
            var
            vessel = connection.SpaceCenter ().ActiveVessel;
            Tuple<double,
            double, double> v = vessel.Flight ().Prograde;
            Console.WriteLine
            (v.Item1 + ", " + v.Item2 + ", " + v.Item3);
        }
    }
}
```

## Quaternions

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

```
using System;
using System.Net;
using KRPC.Client;
using KRPC.Client.Services.SpaceCenter;

class QuaternionExample
{
    public static void Main ()
    {
        using (var connection = new Connection ()) {
            var spaceCenter = connection.SpaceCenter ();
            var vessel = spaceCenter.ActiveVessel;
            Tuple<double, double, double, double> q = vessel.Flight ().Rotation;
            Console.WriteLine (q.Item1 + ", " + q.Item2 + ", " + q.Item3 + ", " + q.Item4);
        }
    }
}
```

## 3.4 Drawing API

### 3.4.1 Drawing

#### **class Drawing**

Provides functionality for drawing objects in the flight scene.

**Line AddLine** (*Tuple<Double, Double, Double> start, Tuple<Double, Double, Double> end, SpaceCenter.ReferenceFrame referenceFrame, Boolean visible = True*)  
Draw a line in the scene.

#### **Parameters**

- **start** – Position of the start of the line.
- **end** – Position of the end of the line.
- **referenceFrame** – Reference frame that the positions are in.
- **visible** – Whether the line is visible.

**Line AddDirection** (*Tuple<Double, Double, Double> direction, SpaceCenter.ReferenceFrame referenceFrame, Single length = 10.0, Boolean visible = True*)  
Draw a direction vector in the scene, from the center of mass of the active vessel.

#### **Parameters**

- **direction** – Direction to draw the line in.
- **referenceFrame** – Reference frame that the direction is in.

- **length** – The length of the line.
- **visible** – Whether the line is visible.

*Polygon* **AddPolygon** (*ICollection<Tuple<Double, Double, Double>> vertices, SpaceCenter.ReferenceFrame referenceFrame, Boolean visible = True*)

Draw a polygon in the scene, defined by a list of vertices.

#### Parameters

- **vertices** – Vertices of the polygon.
- **referenceFrame** – Reference frame that the vertices are in.
- **visible** – Whether the polygon is visible.

*Text* **AddText** (*String text, SpaceCenter.ReferenceFrame referenceFrame, Tuple<Double, Double, Double> position, Tuple<Double, Double, Double, Double> rotation, Boolean visible = True*)

Draw text in the scene.

#### Parameters

- **text** – The string to draw.
- **referenceFrame** – Reference frame that the text position is in.
- **position** – Position of the text.
- **rotation** – Rotation of the text, as a quaternion.
- **visible** – Whether the text is visible.

*void* **Clear** (*Boolean clientOnly = False*)

Remove all objects being drawn.

#### Parameters

- **clientOnly** – If true, only remove objects created by the calling client.

### 3.4.2 Line

#### **class Line**

A line. Created using *Drawing.AddLine*.

*Tuple<Double, Double, Double>* **Start** { *get; set;*  }

Start position of the line.

*Tuple<Double, Double, Double>* **End** { *get; set;*  }

End position of the line.

*SpaceCenter.ReferenceFrame* **ReferenceFrame** { *get; set;*  }

Reference frame for the positions of the object.

*Boolean* **Visible** { *get; set;*  }

Whether the object is visible.

*Tuple<Double, Double, Double>* **Color** { *get; set;*  }

Set the color

*String* **Material** { *get; set;*  }

Material used to render the object. Creates the material from a shader with the given name.

**Single Thickness** { **get**; **set**; }  
Set the thickness

**void Remove** ()  
Remove the object.

### 3.4.3 Polygon

**class Polygon**

A polygon. Created using *Drawing.AddPolygon*.

**ICollection<Tuple<Double, Double, Double>> Vertices** { **get**; **set**; }  
Vertices for the polygon.

*SpaceCenter.ReferenceFrame* **ReferenceFrame** { **get**; **set**; }  
Reference frame for the positions of the object.

**Boolean Visible** { **get**; **set**; }  
Whether the object is visible.

**void Remove** ()  
Remove the object.

**Tuple<Double, Double, Double> Color** { **get**; **set**; }  
Set the color

**String Material** { **get**; **set**; }  
Material used to render the object. Creates the material from a shader with the given name.

**Single Thickness** { **get**; **set**; }  
Set the thickness

### 3.4.4 Text

**class Text**

Text. Created using *Drawing.AddText*.

**Tuple<Double, Double, Double> Position** { **get**; **set**; }  
Position of the text.

**Tuple<Double, Double, Double, Double> Rotation** { **get**; **set**; }  
Rotation of the text as a quaternion.

*SpaceCenter.ReferenceFrame* **ReferenceFrame** { **get**; **set**; }  
Reference frame for the positions of the object.

**Boolean Visible** { **get**; **set**; }  
Whether the object is visible.

**void Remove** ()  
Remove the object.

**String Content** { **get**; **set**; }  
The text string

**String Font** { **get**; **set**; }  
Name of the font

`static IList<String> AvailableFonts (IConnection connection)`  
 A list of all available fonts.

`Int32 Size { get; set; }`  
 Font size.

`Single CharacterSize { get; set; }`  
 Character size.

`UI.FontStyle Style { get; set; }`  
 Font style.

`Tuple<Double, Double, Double> Color { get; set; }`  
 Set the color

`String Material { get; set; }`  
 Material used to render the object. Creates the material from a shader with the given name.

`UI.TextAlignment Alignment { get; set; }`  
 Alignment.

`Single LineSpacing { get; set; }`  
 Line spacing.

`UI.TextAnchor Anchor { get; set; }`  
 Anchor.

## 3.5 InfernalRobotics API

Provides RPCs to interact with the `InfernalRobotics` mod. Provides the following classes:

### 3.5.1 InfernalRobotics

**class InfernalRobotics**  
 This service provides functionality to interact with `InfernalRobotics`.

`Boolean Available { get; }`  
 Whether Infernal Robotics is installed.

`IList<ServoGroup> ServoGroups (SpaceCenter.Vessel vessel)`  
 A list of all the servo groups in the given *vessel*.

#### Parameters

`ServoGroup ServoGroupWithName (SpaceCenter.Vessel vessel, String name)`  
 Returns the servo group in the given *vessel* with the given *name*, or `null` if none exists. If multiple servo groups have the same name, only one of them is returned.

#### Parameters

- **vessel** – Vessel to check.
- **name** – Name of servo group to find.

Servo **ServoWithName** (*SpaceCenter.Vessel vessel*, *String name*)

Returns the servo in the given *vessel* with the given *name* or `null` if none exists. If multiple servos have the same name, only one of them is returned.

#### Parameters

- **vessel** – Vessel to check.
- **name** – Name of the servo to find.

### 3.5.2 ServoGroup

#### class ServoGroup

A group of servos, obtained by calling *InfernalRobotics.ServoGroups* or *InfernalRobotics.ServoGroupWithName*. Represents the “Servo Groups” in the InfernalRobotics UI.

*String* **Name** { **get**; **set**; }

The name of the group.

*String* **ForwardKey** { **get**; **set**; }

The key assigned to be the “forward” key for the group.

*String* **ReverseKey** { **get**; **set**; }

The key assigned to be the “reverse” key for the group.

*Single* **Speed** { **get**; **set**; }

The speed multiplier for the group.

*Boolean* **Expanded** { **get**; **set**; }

Whether the group is expanded in the InfernalRobotics UI.

*ICollection<Servo>* **Servos** { **get**; }

The servos that are in the group.

Servo **ServoWithName** (*String name*)

Returns the servo with the given *name* from this group, or `null` if none exists.

#### Parameters

- **name** – Name of servo to find.

*ICollection<SpaceCenter.Part>* **Parts** { **get**; }

The parts containing the servos in the group.

void **MoveRight** ()

Moves all of the servos in the group to the right.

void **MoveLeft** ()

Moves all of the servos in the group to the left.

void **MoveCenter** ()

Moves all of the servos in the group to the center.

void **MoveNextPreset** ()

Moves all of the servos in the group to the next preset.

void **MovePrevPreset** ()

Moves all of the servos in the group to the previous preset.

void **Stop** ()  
 Stops the servos in the group.

### 3.5.3 Servo

**class Servo**  
 Represents a servo. Obtained using *ServoGroup.Servos*, *ServoGroup.ServoWithName* or *InfernalRobotics.ServoWithName*.

**String Name { get; set; }**  
 The name of the servo.

**SpaceCenter.Part Part { get; }**  
 The part containing the servo.

**Boolean Highlight { set; }**  
 Whether the servo should be highlighted in-game.

**Single Position { get; }**  
 The position of the servo.

**Single MinConfigPosition { get; }**  
 The minimum position of the servo, specified by the part configuration.

**Single MaxConfigPosition { get; }**  
 The maximum position of the servo, specified by the part configuration.

**Single MinPosition { get; set; }**  
 The minimum position of the servo, specified by the in-game tweak menu.

**Single MaxPosition { get; set; }**  
 The maximum position of the servo, specified by the in-game tweak menu.

**Single ConfigSpeed { get; }**  
 The speed multiplier of the servo, specified by the part configuration.

**Single Speed { get; set; }**  
 The speed multiplier of the servo, specified by the in-game tweak menu.

**Single CurrentSpeed { get; set; }**  
 The current speed at which the servo is moving.

**Single Acceleration { get; set; }**  
 The current speed multiplier set in the UI.

**Boolean IsMoving { get; }**  
 Whether the servo is moving.

**Boolean IsFreeMoving { get; }**  
 Whether the servo is freely moving.

**Boolean IsLocked { get; set; }**  
 Whether the servo is locked.

**Boolean IsAxisInverted** { **get**; **set**; }

Whether the servos axis is inverted.

**void MoveRight** ()

Moves the servo to the right.

**void MoveLeft** ()

Moves the servo to the left.

**void MoveCenter** ()

Moves the servo to the center.

**void MoveNextPreset** ()

Moves the servo to the next preset.

**void MovePrevPreset** ()

Moves the servo to the previous preset.

**void MoveTo** (*Single position*, *Single speed*)

Moves the servo to *position* and sets the speed multiplier to *speed*.

#### Parameters

- **position** – The position to move the servo to.
- **speed** – Speed multiplier for the movement.

**void Stop** ()

Stops the servo.

### 3.5.4 Example

The following example gets the control group named “MyGroup”, prints out the names and positions of all of the servos in the group, then moves all of the servos to the right for 1 second.

```
using System;
using System.Net;
using System.Threading;
using KRPC.Client;
using KRPC.Client.Services.InfernalRobotics;
using KRPC.Client.Services.SpaceCenter;

class InfernalRoboticsExample
{
    public static void Main ()
    {
        using (var connection = new Connection (
            name: "InfernalRobotics Example")) {
            var
↪ vessel = connection.SpaceCenter ().ActiveVessel;
            ↪ var ir = connection.InfernalRobotics ();

            var group
↪ = ir.ServoGroupWithName (vessel, "MyGroup");
            if (group == null) {
                ↪ Console.WriteLine ("Group not found");
                return;
            }
        }
    }
}
```



```

    }

    foreach (var servo in group.Servos)
        Console.
↪WriteLine (servo.Name + " " + servo.Position);

    group.MoveRight ();
    Thread.Sleep (1000);
    group.Stop ();
}
}
}

```

## 3.6 Kerbal Alarm Clock API

Provides RPCs to interact with the [Kerbal Alarm Clock](#) mod. Provides the following classes:

### 3.6.1 KerbalAlarmClock

#### **class KerbalAlarmClock**

This service provides functionality to interact with [Kerbal Alarm Clock](#).

#### **Boolean Available { get; }**

Whether Kerbal Alarm Clock is available.

#### **ICollection<Alarm> Alarms { get; }**

A list of all the alarms.

#### **Alarm AlarmWithName (String name)**

Get the alarm with the given *name*, or null if no alarms have that name. If more than one alarm has the name, only returns one of them.

#### **Parameters**

- **name** – Name of the alarm to search for.

#### **ICollection<Alarm> AlarmsWithType (AlarmType type)**

Get a list of alarms of the specified *type*.

#### **Parameters**

- **type** – Type of alarm to return.

#### **Alarm CreateAlarm (AlarmType type, String name, Double ut)**

Create a new alarm and return it.

#### **Parameters**

- **type** – Type of the new alarm.
- **name** – Name of the new alarm.
- **ut** – Time at which the new alarm should trigger.

### 3.6.2 Alarm

**class Alarm**

Represents an alarm. Obtained by calling *KerbalAlarmClock.Alarms*, *KerbalAlarmClock.AlarmWithName* or *KerbalAlarmClock.AlarmsWithType*.

*AlarmAction* **Action** { **get**; **set**; }

The action that the alarm triggers.

**Double** **Margin** { **get**; **set**; }

The number of seconds before the event that the alarm will fire.

**Double** **Time** { **get**; **set**; }

The time at which the alarm will fire.

*AlarmType* **Type** { **get**; }

The type of the alarm.

**String** **ID** { **get**; }

The unique identifier for the alarm.

**String** **Name** { **get**; **set**; }

The short name of the alarm.

**String** **Notes** { **get**; **set**; }

The long description of the alarm.

**Double** **Remaining** { **get**; }

The number of seconds until the alarm will fire.

**Boolean** **Repeat** { **get**; **set**; }

Whether the alarm will be repeated after it has fired.

**Double** **RepeatPeriod** { **get**; **set**; }

The time delay to automatically create an alarm after it has fired.

*SpaceCenter.Vessel* **Vessel** { **get**; **set**; }

The vessel that the alarm is attached to.

*SpaceCenter.CelestialBody* **XferOriginBody** { **get**; **set**; }

The celestial body the vessel is departing from.

*SpaceCenter.CelestialBody* **XferTargetBody** { **get**; **set**; }

The celestial body the vessel is arriving at.

**void** **Remove** ()

Removes the alarm.

### 3.6.3 AlarmType

**enum AlarmType**

The type of an alarm.

**Raw**

An alarm for a specific date/time or a specific period in the future.

**Maneuver**

An alarm based on the next maneuver node on the current ships flight path. This node will be stored and can be restored when you come back to the ship.

**ManeuverAuto**

See *AlarmType.Maneuver*.

**Apoapsis**

An alarm for furthest part of the orbit from the planet.

**Periapsis**

An alarm for nearest part of the orbit from the planet.

**AscendingNode**

Ascending node for the targeted object, or equatorial ascending node.

**DescendingNode**

Descending node for the targeted object, or equatorial descending node.

**Closest**

An alarm based on the closest approach of this vessel to the targeted vessel, some number of orbits into the future.

**Contract**

An alarm based on the expiry or deadline of contracts in career modes.

**ContractAuto**

See *AlarmType.Contract*.

**Crew**

An alarm that is attached to a crew member.

**Distance**

An alarm that is triggered when a selected target comes within a chosen distance.

**EarthTime**

An alarm based on the time in the “Earth” alternative Universe (aka the Real World).

**LaunchRendezvous**

An alarm that fires as your landed craft passes under the orbit of your target.

**SOIChange**

An alarm manually based on when the next SOI point is on the flight path or set to continually monitor the active flight path and add alarms as it detects SOI changes.

**SOIChangeAuto**

See *AlarmType.SOIChange*.

**Transfer**

An alarm based on Interplanetary Transfer Phase Angles, i.e. when should I launch to planet X? Based on Kosmo Not’s post and used in Olex’s Calculator.

**TransferModelled**

See *AlarmType.Transfer*.

### 3.6.4 AlarmAction

**enum AlarmAction**

The action performed by an alarm when it fires.

**DoNothing**

Don't do anything at all...

**DoNothingDeleteWhenPassed**

Don't do anything, and delete the alarm.

**KillWarp**

Drop out of time warp.

**KillWarpOnly**

Drop out of time warp.

**MessageOnly**

Display a message.

**PauseGame**

Pause the game.

### 3.6.5 Example

The following example creates a new alarm for the active vessel. The alarm is set to trigger after 10 seconds have passed, and display a message.

```
using System;
using System.Net;
using KRPC.Client;
using KRPC.Client.Services.KerbalAlarmClock;
using KRPC.Client.Services.SpaceCenter;

class KerbalAlarmClockExample
{
    public static void Main ()
    {
        using (var connection = new Connection_
↵ (name: "Kerbal Alarm Clock Example")) {
            ↵
            ↵ var kac = connection.KerbalAlarmClock ();
            ↵ var alarm = kac.CreateAlarm (
                AlarmType.Raw, "My_
↵ New Alarm", connection.SpaceCenter ().UT + 10);
            ↵ alarm.Notes = "10 seconds_
↵ have now passed since the alarm was created.";
            ↵
            ↵ alarm.Action = AlarmAction.MessageOnly;
            }
        }
    }
}
```

## 3.7 RemoteTech API

Provides RPCs to interact with the [RemoteTech](#) mod. Provides the following classes:

### 3.7.1 RemoteTech

#### **class RemoteTech**

This service provides functionality to interact with [RemoteTech](#).

**Boolean Available { get; }**

Whether RemoteTech is installed.

**IList<String> GroundStations { get; }**

The names of the ground stations.

**Antenna Antenna (SpaceCenter.Part part)**

Get the antenna object for a particular part.

#### **Parameters**

**Comms Comms (SpaceCenter.Vessel vessel)**

Get a communications object, representing the communication capability of a particular vessel.

#### **Parameters**

### 3.7.2 Comms

#### **class Comms**

Communications for a vessel.

**SpaceCenter.Vessel Vessel { get; }**

Get the vessel.

**Boolean HasLocalControl { get; }**

Whether the vessel can be controlled locally.

**Boolean HasFlightComputer { get; }**

Whether the vessel has a flight computer on board.

**Boolean HasConnection { get; }**

Whether the vessel has any connection.

**Boolean HasConnectionToGroundStation { get; }**

Whether the vessel has a connection to a ground station.

**Double SignalDelay { get; }**

The shortest signal delay to the vessel, in seconds.

**Double SignalDelayToGroundStation { get; }**

The signal delay between the vessel and the closest ground station, in seconds.

**Double SignalDelayToVessel (SpaceCenter.Vessel other)**

The signal delay between the this vessel and another vessel, in seconds.

#### **Parameters**

`ICollection<Antenna> Antennas { get; }`  
The antennas for this vessel.

### 3.7.3 Antenna

#### **class Antenna**

A RemoteTech antenna. Obtained by calling `Comms.Antennas` or `RemoteTech.Antenna`.

`SpaceCenter.Part Part { get; }`  
Get the part containing this antenna.

`Boolean HasConnection { get; }`  
Whether the antenna has a connection.

`Target Target { get; set; }`  
The object that the antenna is targetting. This property can be used to set the target to `Target.None` or `Target.ActiveVessel`. To set the target to a celestial body, ground station or vessel see `Antenna.TargetBody`, `Antenna.TargetGroundStation` and `Antenna.TargetVessel`.

`SpaceCenter.CelestialBody TargetBody { get; set; }`  
The celestial body the antenna is targetting.

`String TargetGroundStation { get; set; }`  
The ground station the antenna is targetting.

`SpaceCenter.Vessel TargetVessel { get; set; }`  
The vessel the antenna is targetting.

**enum Target**  
The type of object an antenna is targetting. See `Antenna.Target`.

**ActiveVessel**  
The active vessel.

**CelestialBody**  
A celestial body.

**GroundStation**  
A ground station.

**Vessel**  
A specific vessel.

**None**  
No target.

### 3.7.4 Example

The following example sets the target of a dish on the active vessel then prints out the signal delay to the active vessel.

```
using System;
using KRPC.Client;
using KRPC.Client.Services.RemoteTech;
```

```

using KRPC.Client.Services.SpaceCenter;

class RemoteTechExample
{
    public static void Main ()
    {
        using (var connection_
↪ = new Connection ("RemoteTech Example")) {
            var sc = connection.SpaceCenter ();
            var rt = connection.RemoteTech ();
            var vessel = sc.ActiveVessel;

            // Set a dish target
            var part =_
↪vessel.Parts.WithTitle ("Reflectron KR-7") [0];
            var antenna = rt.Antenna (part);

            _
↪ antenna.TargetBody = sc.Bodies ["Jool"];

            _
↪ // Get info about the vessels communications
            var comms = rt.Comms (vessel);
            Console.WriteLine_
↪("Signal delay = " + comms.SignalDelay);
        }
    }
}

```

## 3.8 User Interface API

### 3.8.1 UI

#### class UI

Provides functionality for drawing and interacting with in-game user interface elements.

*Canvas* **StockCanvas** { **get**; }

The stock UI canvas.

*Canvas* **AddCanvas** ()

Add a new canvas.

---

**Note:** If you want to add UI elements to KSP's stock UI canvas, use *UI.StockCanvas*.

---

void **Message** (*String* content, *Single* duration = 1.0, *MessagePosition* position = 1)

Display a message on the screen.

#### Parameters

- **content** – Message content.
- **duration** – Duration before the message disappears, in seconds.
- **position** – Position to display the message.

---

**Note:** The message appears just like a stock message, for example quicksave or quickload messages.

---

void **Clear** (*Boolean clientOnly = False*)

Remove all user interface elements.

**Parameters**

- **clientOnly** – If true, only remove objects created by the calling client.

enum **MessagePosition**

Message position.

**TopLeft**

Top left.

**TopCenter**

Top center.

**TopRight**

Top right.

**BottomCenter**

Bottom center.

## 3.8.2 Canvas

class **Canvas**

A canvas for user interface elements. See *UI.StockCanvas* and *UI.AddCanvas*.

*RectTransform* **RectTransform** { **get**; }

The rect transform for the canvas.

*Boolean* **Visible** { **get**; **set**; }

Whether the UI object is visible.

*Panel* **AddPanel** (*Boolean visible = True*)

Create a new container for user interface elements.

**Parameters**

- **visible** – Whether the panel is visible.

*Text* **AddText** (*String content, Boolean visible = True*)

Add text to the canvas.

**Parameters**

- **content** – The text.
- **visible** – Whether the text is visible.

*InputField* **AddInputField** (*Boolean visible = True*)

Add an input field to the canvas.

**Parameters**

- **visible** – Whether the input field is visible.



*Button* **AddButton** (*String content*, *Boolean visible = True*)  
Add a button to the canvas.

#### Parameters

- **content** – The label for the button.
- **visible** – Whether the button is visible.

void **Remove** ()  
Remove the UI object.

### 3.8.3 Panel

**class Panel**  
A container for user interface elements. See *Canvas.AddPanel*.

*RectTransform* **RectTransform** { *get*; }  
The rect transform for the panel.

*Boolean* **Visible** { *get*; *set*; }  
Whether the UI object is visible.

*Panel* **AddPanel** (*Boolean visible = True*)  
Create a panel within this panel.

#### Parameters

- **visible** – Whether the new panel is visible.

*Text* **AddText** (*String content*, *Boolean visible = True*)  
Add text to the panel.

#### Parameters

- **content** – The text.
- **visible** – Whether the text is visible.

*InputField* **AddInputField** (*Boolean visible = True*)  
Add an input field to the panel.

#### Parameters

- **visible** – Whether the input field is visible.

*Button* **AddButton** (*String content*, *Boolean visible = True*)  
Add a button to the panel.

#### Parameters

- **content** – The label for the button.
- **visible** – Whether the button is visible.

void **Remove** ()  
Remove the UI object.

### 3.8.4 Text

**class Text**  
A text label. See *Panel.AddText*.

*RectTransform* **RectTransform** { **get**; }  
The rect transform for the text.

**Boolean Visible** { **get**; **set**; }  
Whether the UI object is visible.

**String Content** { **get**; **set**; }  
The text string

**String Font** { **get**; **set**; }  
Name of the font

**IList<String> AvailableFonts** { **get**; }  
A list of all available fonts.

**Int32 Size** { **get**; **set**; }  
Font size.

*FontStyle* **Style** { **get**; **set**; }  
Font style.

**Tuple<Double, Double, Double> Color** { **get**; **set**; }  
Set the color

*TextAnchor* **Alignment** { **get**; **set**; }  
Alignment.

**Single LineSpacing** { **get**; **set**; }  
Line spacing.

**void Remove** ()  
Remove the UI object.

**enum FontStyle**  
Font style.

**Normal**  
Normal.

**Bold**  
Bold.

**Italic**  
Italic.

**BoldAndItalic**  
Bold and italic.

**enum TextAlignment**  
Text alignment.

**Left**  
Left aligned.

**Right**  
Right aligned.

**Center**  
Center aligned.

**enum TextAnchor**  
Text alignment.

**LowerCenter**

Lower center.

**LowerLeft**

Lower left.

**LowerRight**

Lower right.

**MiddleCenter**

Middle center.

**MiddleLeft**

Middle left.

**MiddleRight**

Middle right.

**UpperCenter**

Upper center.

**UpperLeft**

Upper left.

**UpperRight**

Upper right.

### 3.8.5 Button

**class Button**A text label. See *Panel.AddButton*.*RectTransform* **RectTransform** { **get**; }

The rect transform for the text.

**Boolean Visible** { **get**; **set**; }

Whether the UI object is visible.

*Text* **Text** { **get**; }

The text for the button.

**Boolean Clicked** { **get**; **set**; }

Whether the button has been clicked.

---

**Note:** This property is set to true when the user clicks the button. A client script should reset the property to false in order to detect subsequent button presses.

---

void **Remove** ()

Remove the UI object.

### 3.8.6 InputField

**class InputField**An input field. See *Panel.AddInputField*.*RectTransform* **RectTransform** { **get**; }

The rect transform for the input field.

**Boolean Visible { get; set; }**  
Whether the UI object is visible.

**String Value { get; set; }**  
The value of the input field.

**Text Text { get; }**  
The text component of the input field.

---

**Note:** Use *InputField.Value* to get and set the value in the field. This object can be used to alter the style of the input field's text.

---

**Boolean Changed { get; set; }**  
Whether the input field has been changed.

---

**Note:** This property is set to true when the user modifies the value of the input field. A client script should reset the property to false in order to detect subsequent changes.

---

**void Remove ()**  
Remove the UI object.

### 3.8.7 Rect Transform

**class RectTransform**  
A Unity engine Rect Transform for a UI object. See the [Unity manual](#) for more details.

**Tuple<Double, Double> Position { get; set; }**  
Position of the rectangles pivot point relative to the anchors.

**Tuple<Double, Double, Double> LocalPosition { get; set; }**  
Position of the rectangles pivot point relative to the anchors.

**Tuple<Double, Double> Size { get; set; }**  
Width and height of the rectangle.

**Tuple<Double, Double> UpperRight { get; set; }**  
Position of the rectangles upper right corner relative to the anchors.

**Tuple<Double, Double> LowerLeft { get; set; }**  
Position of the rectangles lower left corner relative to the anchors.

**Tuple<Double, Double> Anchor { set; }**  
Set the minimum and maximum anchor points as a fraction of the size of the parent rectangle.

**Tuple<Double, Double> AnchorMax { get; set; }**  
The anchor point for the lower left corner of the rectangle defined as a fraction of the size of the parent rectangle.

**Tuple<Double, Double> AnchorMin { get; set; }**  
The anchor point for the upper right corner of the rectangle defined as a fraction of the size of the parent rectangle.

`Tuple<Double, Double> Pivot { get; set; }`

Location of the pivot point around which the rectangle rotates, defined as a fraction of the size of the rectangle itself.

`Tuple<Double, Double, Double, Double> Rotation { get; set; }`

Rotation, as a quaternion, of the object around its pivot point.

`Tuple<Double, Double, Double> Scale { get; set; }`

Scale factor applied to the object in the x, y and z dimensions.



## 4.1 C++ Client

This client provides a C++ API for interacting with a kRPC server.

### 4.1.1 Installing the Library

#### Dependencies

First you need to install kRPC's dependencies: [ASIO](#) which is used for network communication and [protobuf](#) which is used to serialize messages.

ASIO is a headers-only library. The boost version is not required, installing the non-Boost variant is sufficient. On Ubuntu, this can be done using apt:

```
sudo apt-get install libasio-dev
```

Alternatively it can be downloaded [from the ASIO website](#).

Protobuf version 3 is also required, and can be [downloaded from GitHub](#). Installation instructions [can be found here](#).

#### Using the configure script

Once the dependencies have been installed, you can install the kRPC client library and headers using the configure script provided with the source. [Download the source archive](#), extract it and then execute the following:

```
./configure
make
sudo make install
sudo ldconfig
```

#### Using CMake

Alternatively, you can install the client library and headers using CMake. [Download the source archive](#), extract it and execute the following:

```
cmake .
make
sudo make install
sudo ldconfig
```

## Manual installation

The library is fairly simple to build manually if you can't use the configure script or CMake. The headers are in the `include` directory and the source files are in `src`.

### 4.1.2 Using the Library

A kRPC program needs to be compiled with C++11 support enabled, and linked against `libkrpc` and `libprotobuf`. The following example program connects to the server, queries it for its version and prints it out:

```
#include <iostream>
#include <krpc.hpp>
#include <krpc/services/krpc.hpp>

int main() {
    auto conn = krpc::connect();
    krpc::services::KRPC krpc(&conn);
    std::cout << "Connected to kRPC server version " << krpc.get_status().version() << "\n";
    std::endl;
}
```

To compile this program using GCC, save the source as `main.cpp` and run the following:

```
g++ main.cpp -std=c++11 -lkrpc -lprotobuf
```

---

**Note:** If you get linker errors claiming that there are undefined references to `google::protobuf::...` you probably have an older version of protobuf installed on your system. In this case, replace `-lprotobuf` with `-l:libprotobuf.so.10` in the above command so that GCC uses the correct version of the library.

---

### 4.1.3 Connecting to the Server

The `krpc::connect()` function is used to open a connection to a server. It returns a client object (of type `krpc::Client`) through which you can interact with the server. When called without any arguments, it will connect to the local machine on the default port numbers. You can specify different connection settings, and also a descriptive name for the connection, as follows:

```
#include <iostream>
#include <krpc.hpp>
#include <krpc/services/krpc.hpp>

int main() {
    auto conn = krpc::connect("My Example Program", "192.168.1.10", 1000, 1001);
    krpc::services::KRPC krpc(&conn);
    std::cout << krpc.get_status().version() << std::endl;
}
```

### 4.1.4 Calling Remote Procedures

The kRPC server provides *procedures* that a client can run. These procedures are arranged in groups called *services* to keep things organized. The functionality for the services are defined in the header files in `krpc/services/...`



For example, all of the functionality provided by the SpaceCenter service is contained in the header file `krpc/services/space_center.hpp`.

To interact with a service, you must include its header file and create an instance of the service, passing a `krpc::Client` object to its constructor.

The following example demonstrates how to invoke remote procedures using the C++ client. It instantiates the SpaceCenter service and calls `krpc::services::SpaceCenter::active_vessel()` to get an object representing the active vessel (of type `krpc::services::SpaceCenter::Vessel`). It sets the name of the vessel and then prints out its altitude:

```
#include <iostream>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    auto conn = krpc::connect();
    krpc::services::SpaceCenter sc(&conn);
    auto vessel = sc.active_vessel();
    vessel.set_name("My Vessel");
    auto flight_info = vessel.flight();
    std::cout << flight_info.mean_altitude() << std::endl;
}
```

#### 4.1.5 Streaming Data from the Server

A common use case for kRPC is to continuously extract data from the game. The naive approach to do this would be to repeatedly call a remote procedure, such as in the following which repeatedly prints the position of the active vessel:

```
#include <iostream>
#include <iomanip>
#include <tuple>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    auto conn = krpc::connect();
    krpc::services::SpaceCenter sc(&conn);
    auto vessel = sc.active_vessel();
    auto ref_frame = vessel.orbit().body().reference_frame();
    while (true) {
        auto pos = vessel.position(ref_frame);
        std::cout << std::fixed << std::setprecision(1);
        std::cout << std::get<0>(pos) << ", "
                  << std::get<1>(pos) << ", "
                  << std::get<2>(pos) << std::endl;
    }
}
```

This approach requires significant communication overhead as request/response messages are repeatedly sent between the client and server. kRPC provides a more efficient mechanism to achieve this, called *streams*.

A stream repeatedly executes a procedure on the server (with a fixed set of argument values) and sends the result to the client. It only requires a single message to be sent to the server to establish the stream, which will then continuously send data to the client until the stream is closed.

The following example does the same thing as above using streams:

```

#include <iostream>
#include <iomanip>
#include <tuple>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    auto conn = krpc::connect();
    krpc::services::SpaceCenter sc(&conn);
    auto vessel = sc.active_vessel();
    auto ref_frame = vessel.orbit().body().reference_frame();
    auto pos_stream = vessel.position_stream(ref_frame);
    while (true) {
        auto pos = pos_stream();
        std::cout << std::fixed << std::setprecision(1);
        std::cout << std::get<0>(pos) << ", "
                  << std::get<1>(pos) << ", "
                  << std::get<2>(pos) << std::endl;
    }
}

```

It calls `position_stream` once at the start of the program to create the stream, and then repeatedly prints the position returned by the stream. The stream is automatically closed when the client disconnects.

A stream can be created for any function call (except property setters) by adding `_stream` to the end of the functions name. This returns a stream object of type `template <typename T> krpc::Stream`, where `T` is the return type of the original function. The most recent value of the stream can be obtained by calling `krpc::Stream::operator()()`. A stream can be stopped and removed from the server by calling `krpc::Stream::remove()` on the stream object. All of a clients streams are automatically stopped when it disconnects.

Updates to streams can be paused by calling `krpc::Client::freeze_streams()`. After this call, all streams will have their values frozen to values from the same physics tick. Updates can be resumed by calling `krpc::Client::thaw_streams()`. This is useful if you need to perform some computation using stream values and require all of the stream values to be from the same physics tick.

## 4.1.6 Synchronizing with Stream Updates

A common use case for kRPC is to wait until the value returned by a method or attribute changes, and then take some action. kRPC provides two mechanisms to do this efficiently: *condition variables* and *callbacks*.

### Condition Variables

Each stream has a condition variable associated with it, that is notified whenever the value of the stream changes. The condition variables are instances of `std::condition_variable`. These can be used to block the current thread of execution until the value of the stream changes.

The following example waits until the abort button is pressed in game, by waiting for the value of `krpc::services::SpaceCenter::Control::abort()` to change to true:

```

#include <iostream>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {

```

```

auto conn = krpc::connect();
krpc::services::SpaceCenter sc(&conn);
auto control = sc.active_vessel().control();
auto abort = control.abort_stream();
abort.acquire();
while (!abort())
    abort.wait();
abort.release();
}

```

This code creates a stream, acquires a lock on the streams condition variable (by calling `acquire`) and then repeatedly checks the value of `abort`. It leaves the loop when it changes to true.

The body of the loop calls `wait` on the stream, which causes the program to block until the value changes. This prevents the loop from ‘spinning’ and so it does not consume processing resources whilst waiting.

**Note:** The stream does not start receiving updates until the first call to `wait`. This means that the example code will not miss any updates to the streams value, as it will have already locked the condition variable before the first stream update is received.

## Callbacks

Streams allow you to register callback functions that are called whenever the value of the stream changes. Callback functions should take a single argument, which is the new value of the stream, and should return nothing.

For example the following program registers two callbacks that are invoked when the value of `krpc::services::SpaceCenter::Control::abort()` changes:

```

#include <iostream>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

void check_abort1(bool x) {
    std::cout << "Abort 1 called with a value of " << x << std::endl;
}

void check_abort2(bool x) {
    std::cout << "Abort 2 called with a value of " << x << std::endl;
}

int main() {
    auto conn = krpc::connect();
    krpc::services::SpaceCenter sc(&conn);
    auto control = sc.active_vessel().control();
    auto abort = control.abort_stream();

    abort.add_callback(check_abort1);
    abort.add_callback(check_abort2);
    abort.start();

    // Keep the program running...
    while (true) {
    }
}

```

---

**Note:** When a stream is created it does not start receiving updates until `start` is called. This is implicitly called when accessing the value of a stream, but as this example does not do this an explicit call to `start` is required.

---

---

**Note:** The callbacks are registered before the call to `start` so that stream updates are not missed.

---

---

**Note:** The callback function may be called from a different thread to that which created the stream. Any changes to shared state must therefore be protected with appropriate synchronization.

---

### 4.1.7 Custom Events

Some procedures return event objects of type `krpc::Event`. These allow you to wait until an event occurs, by calling `krpc::Event::wait()`. Under the hood, these are implemented using streams and condition variables.

Custom events can also be created. An expression API allows you to create code that runs on the server and these can be used to build a custom event. For example, the following creates the expression `mean_altitude > 1000` and then creates an event that will be triggered when the expression returns true:

```
#include <iostream>
#include <krpc.hpp>
#include <krpc/services/krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    auto conn = krpc::connect();
    krpc::services::KRPC krpc(&conn);
    krpc::services::SpaceCenter sc(&conn);
    auto flight = sc.active_vessel().flight();

    // Get the remote procedure call as a message object,
    // so it can be passed to the server
    auto mean_altitude = flight.mean_altitude_call();

    // Create an expression on the server
    typedef krpc::services::KRPC::Expression Expr;
    auto expr = Expr::greater_than(conn,
        Expr::call(conn, mean_altitude),
        Expr::constant_double(conn, 1000));

    auto event = krpc.add_event(expr);
    event.acquire();
    event.wait();
    std::cout << "Altitude reached 1000m" << std::endl;
    event.release();
}
```

### 4.1.8 Client API Reference

*Client* **connect** (**const** std::string &name = "", **const** std::string &address = "127.0.0.1", unsigned int *rpc\_port* = 50000, unsigned int *stream\_port* = 50001)

This function creates a connection to a kRPC server. It returns a `krpc::Client` object, through which the

server can be communicated with.

### Parameters

- **name** (*std::string*) – A descriptive name for the connection. This is passed to the server and appears in the in-game server window.
- **address** (*std::string*) – The address of the server to connect to. Can either be a hostname or an IP address in dotted decimal notation. Defaults to '127.0.0.1'.
- **rpc\_port** (*unsigned int*) – The port number of the RPC Server. Defaults to 50000. This should match the RPC port number of the server you want to connect to.
- **stream\_port** (*unsigned int*) – The port number of the Stream Server. Defaults to 50001. This should match the stream port number of the server you want to connect to.

### class Client

This class provides the interface for communicating with the server. It is used by service class instances to invoke remote procedure calls. Instances of this class can be obtained by calling *krpc::connect()*.

void **close** ()

Closes the connection to the server.

void **freeze\_streams** ()

Pause stream updates, after the next stream update message is received. This function blocks until the streams have been frozen.

void **thaw\_streams** ()

Resume stream updates. Before this function returns, the last received update message is applied to the streams.

template<typename T>

### class Stream

This class represents a stream. See *Streaming Data from the Server*.

Streams are created by calling a remove procedure with `_stream` appended to its name.

Stream objects are copy constructible and assignable. A stream is removed from the server when all stream objects that refer to it are destroyed.

void **start** (bool *wait* = true)

Starts the stream. When a stream is created it does not start sending updates to the client until this method is called.

If *wait* is true, this method will block until at least one update has been received from the server.

If *wait* is false, the method starts the stream and returns immediately. Subsequent calls to `operator()` may throw a `krpc::StreamError` exception.

T **operator** () ()

Get the most recently received value from the stream.

std::condition\_variable &**get\_condition** () const

A condition variable that is notified whenever the value of the stream changes.

std::unique\_lock<std::mutex> &**get\_condition\_lock** () const

The lock for the condition variable.

void **acquire** ()

Acquires a lock on the mutex for the condition variable.

void **release** ()

Releases the lock on the mutex for the condition variable.

void **wait** (double *timeout* = -1)

This method blocks until the value of the stream changes or the operation times out.

The streams condition variable must be locked (by calling `acquire`) before calling this method.

If *timeout* is specified and is greater than or equal to 0, it is the timeout in seconds for the operation.

If the stream has not been started this method calls `start (false)` to start the stream (without waiting for at least one update to be received).

void **add\_callback** (const std::function<void> *T*

> &*callback*) Adds a callback function that is invoked whenever the value of the stream changes. The callback function should take one argument, which is passed the new value of the stream.

---

**Note:** The callback function may be called from a different thread to that which created the stream. Any changes to shared state must therefore be protected with appropriate synchronization.

---

void **remove** ()

Removes the stream from the server.

bool **operator==** (const *Stream*<*T*> &*rhs*)

Returns true if the two stream objects are bound to the same stream.

bool **operator!=** (const *Stream*<*T*> &*rhs*)

Returns true if the two stream objects are bound to different streams.

**operator bool** ()

Returns whether the stream object is bound to a stream.

#### **class Event**

This class represents an event. See *Custom Events*. It is wrapper around a `Stream<bool>` that indicates when the event occurs.

Event objects are copy constructible and assignable. An event is removed from the server when all event objects that refer to it are destroyed.

void **start** ()

Starts the event. When an event is created, it will not receive updates from the server until this method is called.

std::condition\_variable &**get\_condition** () const

The condition variable that is notified whenever the event occurs.

std::unique\_lock<std::mutex> &**get\_condition\_lock** () const

The lock for the condition variable.

void **acquire** ()

Acquires a lock on the mutex for the condition variable.

void **release** ()

Releases the lock on the mutex for the condition variable.

void **wait** (double *timeout* = -1)

This method blocks until the event occurs or the operation times out.

The events condition variable must be locked before calling this method.

If *timeout* is specified and is greater than or equal to 0, it is the timeout in seconds for the operation.

If the event has not been started this method calls `start ()` to start the underlying stream.

```

void add_callback (const std::function<void>
    > &callback) Adds a callback function that is invoked whenever the event occurs. The callback function
    should be a function that takes zero arguments.

void remove ()
    Removes the event from the server.

Stream<bool> stream ()
    Returns the underlying stream for the event.

bool operator== (const Event &rhs)
    Returns true if the two event objects are bound to the same underlying stream.

bool operator!= (const Event &rhs)
    Returns true if the two event objects are bound to different underlying streams.

operator bool ()
    Returns whether the event object is bound to a stream.

```

## 4.2 KRPC API

### 4.2.1 KRPC

None None None None

```

class KRPC : public krpc::Service
    Main kRPC service, used by clients to interact with basic server functionality.

    KRPC (krpc::Client *client)
        Construct an instance of this service.

    std::string get_client_id ()
        Returns the identifier for the current client.

    std::string get_client_name ()
        Returns the name of the current client. This is an empty string if the client has no name.

    std::vector<std::tuple<std::string, std::string, std::string>> clients ()
        A list of RPC clients that are currently connected to the server. Each entry in the list is a clients identifier,
        name and address.

    krpc::schema::Status get_status ()
        Returns some information about the server, such as the version.

    krpc::schema::Services get_services ()
        Returns information on all services, procedures, classes, properties etc. provided by the server. Can be
        used by client libraries to automatically create functionality such as stubs.

    GameScene current_game_scene ()
        Get the current game scene.

    bool paused ()

    void set_paused (bool value)
        Whether the game is paused.

enum struct GameScene
    The game scene. See current_game_scene ().

```

**enumerator space\_center**

The game scene showing the Kerbal Space Center buildings.

**enumerator flight**

The game scene showing a vessel in flight (or on the launchpad/runway).

**enumerator tracking\_station**

The tracking station.

**enumerator editor\_vab**

The Vehicle Assembly Building.

**enumerator editor\_sph**

The Space Plane Hangar.

**class InvalidOperationException**

A method call was made to a method that is invalid given the current state of the object.

**class ArgumentException**

A method was invoked where at least one of the passed arguments does not meet the parameter specification of the method.

**class ArgumentNullException**

A null reference was passed to a method that does not accept it as a valid argument.

**class ArgumentOutOfRangeException**

The value of an argument is outside the allowable range of values as defined by the invoked method.

## 4.2.2 Expressions

**class Expression**

A server side expression.

**static Expression constant\_double** (*Client &connection*, double *value*)

A constant value of type double.

**Parameters**

**static Expression constant\_float** (*Client &connection*, float *value*)

A constant value of type float.

**Parameters**

**static Expression constant\_int** (*Client &connection*, int32\_t *value*)

A constant value of type int.

**Parameters**

**static Expression constant\_string** (*Client &connection*, std::string *value*)

A constant value of type string.

**Parameters**

**static Expression call** (*Client &connection*, krpc::schema::ProcedureCall *call*)

An RPC call.

**Parameters**

**static Expression equal** (*Client &connection*, Expression *arg0*, Expression *arg1*)

Equality comparison.

**Parameters**



**static Expression not\_equal** (*Client &connection, Expression arg0, Expression arg1*)  
Inequality comparison.

**Parameters**

**static Expression greater\_than** (*Client &connection, Expression arg0, Expression arg1*)  
Greater than numerical comparison.

**Parameters**

**static Expression greater\_than\_or\_equal** (*Client &connection, Expression arg0, Expression arg1*)  
Greater than or equal numerical comparison.

**Parameters**

**static Expression less\_than** (*Client &connection, Expression arg0, Expression arg1*)  
Less than numerical comparison.

**Parameters**

**static Expression less\_than\_or\_equal** (*Client &connection, Expression arg0, Expression arg1*)  
Less than or equal numerical comparison.

**Parameters**

**static Expression and\_\_** (*Client &connection, Expression arg0, Expression arg1*)  
Boolean and operator.

**Parameters**

**static Expression or\_\_** (*Client &connection, Expression arg0, Expression arg1*)  
Boolean or operator.

**Parameters**

**static Expression exclusive\_or** (*Client &connection, Expression arg0, Expression arg1*)  
Boolean exclusive-or operator.

**Parameters**

**static Expression not\_\_** (*Client &connection, Expression arg*)  
Boolean negation operator.

**Parameters**

**static Expression add** (*Client &connection, Expression arg0, Expression arg1*)  
Numerical addition.

**Parameters**

**static Expression subtract** (*Client &connection, Expression arg0, Expression arg1*)  
Numerical subtraction.

**Parameters**

**static Expression multiply** (*Client &connection, Expression arg0, Expression arg1*)  
Numerical multiplication.

**Parameters**

**static Expression divide** (*Client &connection, Expression arg0, Expression arg1*)  
Numerical division.

**Parameters**

**static Expression modulo** (*Client &connection, Expression arg0, Expression arg1*)  
Numerical modulo operator.

**Parameters**

**Returns** The remainder of arg0 divided by arg1

**static Expression power** (*Client &connection, Expression arg0, Expression arg1*)  
Numerical power operator.

**Parameters**

**Returns** arg0 raised to the power of arg1

**static Expression left\_shift** (*Client &connection, Expression arg0, Expression arg1*)  
Bitwise left shift.

**Parameters**

**static Expression right\_shift** (*Client &connection, Expression arg0, Expression arg1*)  
Bitwise right shift.

**Parameters**

**static Expression to\_double** (*Client &connection, Expression arg*)  
Convert to a double type.

**Parameters**

**static Expression to\_float** (*Client &connection, Expression arg*)  
Convert to a float type.

**Parameters**

**static Expression to\_int** (*Client &connection, Expression arg*)  
Convert to an int type.

**Parameters**

## 4.3 SpaceCenter API

### 4.3.1 SpaceCenter

**class SpaceCenter** : public *krpc::Service*

Provides functionality to interact with Kerbal Space Program. This includes controlling the active vessel, managing its resources, planning maneuver nodes and auto-piloting.

**SpaceCenter** (*krpc::Client \*client*)  
Construct an instance of this service.

*Vessel* **active\_vessel** ()

void **set\_active\_vessel** (*Vessel value*)  
The currently active vessel.

std::vector<*Vessel*> **vessels** ()  
A list of all the vessels in the game.

std::map<std::string, *CelestialBody*> **bodies** ()  
A dictionary of all celestial bodies (planets, moons, etc.) in the game, keyed by the name of the body.

*CelestialBody* **target\_body** ()

void **set\_target\_body** (*CelestialBody value*)  
The currently targeted celestial body.

*Vessel* **target\_vessel** ()

void **set\_target\_vessel** (*Vessel value*)  
The currently targeted vessel.

*DockingPort* **target\_docking\_port** ()

void **set\_target\_docking\_port** (*DockingPort value*)  
The currently targeted docking port.

void **clear\_target** ()  
Clears the current target.

std::vector<std::string> **launchable\_vessels** (std::string *craft\_directory*)  
Returns a list of vessels from the given *craft\_directory* that can be launched.

#### Parameters

- **craft\_directory** – Name of the directory in the current saves “Ships” directory. For example "VAB" or "SPH".

void **launch\_vessel** (std::string *craft\_directory*, std::string *name*, std::string *launch\_site*)  
Launch a vessel.

#### Parameters

- **craft\_directory** – Name of the directory in the current saves “Ships” directory, that contains the craft file. For example "VAB" or "SPH".
- **name** – Name of the vessel to launch. This is the name of the “.craft” file in the save directory, without the “.craft” file extension.
- **launch\_site** – Name of the launch site. For example "LaunchPad" or "Runway".

void **launch\_vessel\_from\_vab** (std::string *name*)  
Launch a new vessel from the VAB onto the launchpad.

#### Parameters

- **name** – Name of the vessel to launch.

---

**Note:** This is equivalent to calling *launch\_vessel()* with the craft directory set to “VAB” and the launch site set to “LaunchPad”.

---

void **launch\_vessel\_from\_sph** (std::string *name*)  
Launch a new vessel from the SPH onto the runway.

#### Parameters

- **name** – Name of the vessel to launch.

---

**Note:** This is equivalent to calling *launch\_vessel()* with the craft directory set to “SPH” and the launch site set to “Runway”.

---

void **save** (std::string *name*)  
Save the game with a given name. This will create a save file called *name.sfs* in the folder of the current save game.

**Parameters**

void **load** (std::string *name*)

Load the game with the given name. This will create a load a save file called `name.sfs` from the folder of the current save game.

**Parameters**

void **quicksave** ()

Save a quicksave.

---

**Note:** This is the same as calling `save()` with the name “quicksave”.

---

void **quickload** ()

Load a quicksave.

---

**Note:** This is the same as calling `load()` with the name “quicksave”.

---

bool **ui\_visible** ()

void **set\_ui\_visible** (bool *value*)

Whether the UI is visible.

bool **navball** ()

void **set\_navball** (bool *value*)

Whether the navball is visible.

double **ut** ()

The current universal time in seconds.

double **g** ()

The value of the [gravitational constant](#)  $G$  in  $N(m/kg)^2$ .

float **warp\_rate** ()

The current warp rate. This is the rate at which time is passing for either on-rails or physical time warp. For example, a value of 10 means time is passing 10x faster than normal. Returns 1 if time warp is not active.

float **warp\_factor** ()

The current warp factor. This is the index of the rate at which time is passing for either regular “on-rails” or physical time warp. Returns 0 if time warp is not active. When in on-rails time warp, this is equal to `rails_warp_factor()`, and in physics time warp, this is equal to `physics_warp_factor()`.

int32\_t **rails\_warp\_factor** ()

void **set\_rails\_warp\_factor** (int32\_t *value*)

The time warp rate, using regular “on-rails” time warp. A value between 0 and 7 inclusive. 0 means no time warp. Returns 0 if physical time warp is active.

If requested time warp factor cannot be set, it will be set to the next lowest possible value. For example, if the vessel is too close to a planet. See [the KSP wiki](#) for details.

int32\_t **physics\_warp\_factor** ()

void **set\_physics\_warp\_factor** (int32\_t *value*)

The physical time warp rate. A value between 0 and 3 inclusive. 0 means no time warp. Returns 0 if regular “on-rails” time warp is active.

bool **can\_rails\_warp\_at** (int32\_t *factor* = 1)

Returns `true` if regular “on-rails” time warp can be used, at the specified warp *factor*. The maximum time warp rate is limited by various things, including how close the active vessel is to a planet. See [the KSP wiki](#) for details.

#### Parameters

- **factor** – The warp factor to check.

int32\_t **maximum\_rails\_warp\_factor** ()

The current maximum regular “on-rails” warp factor that can be set. A value between 0 and 7 inclusive. See [the KSP wiki](#) for details.

void **warp\_to** (double *ut*, float *max\_rails\_rate* = 100000.0, float *max\_physics\_rate* = 2.0)

Uses time acceleration to warp forward to a time in the future, specified by universal time *ut*. This call blocks until the desired time is reached. Uses regular “on-rails” or physical time warp as appropriate. For example, physical time warp is used when the active vessel is traveling through an atmosphere. When using regular “on-rails” time warp, the warp rate is limited by *max\_rails\_rate*, and when using physical time warp, the warp rate is limited by *max\_physics\_rate*.

#### Parameters

- **ut** – The universal time to warp to, in seconds.
- **max\_rails\_rate** – The maximum warp rate in regular “on-rails” time warp.
- **max\_physics\_rate** – The maximum warp rate in physical time warp.

**Returns** When the time warp is complete.

std::tuple<double, double, double> **transform\_position** (std::tuple<double, double, double> *position*, *ReferenceFrame from*, *ReferenceFrame to*)

Converts a position from one reference frame to another.

#### Parameters

- **position** – Position, as a vector, in reference frame *from*.
- **from** – The reference frame that the position is in.
- **to** – The reference frame to convert the position to.

**Returns** The corresponding position, as a vector, in reference frame *to*.

std::tuple<double, double, double> **transform\_direction** (std::tuple<double, double, double> *direction*, *ReferenceFrame from*, *ReferenceFrame to*)

Converts a direction from one reference frame to another.

#### Parameters

- **direction** – Direction, as a vector, in reference frame *from*.
- **from** – The reference frame that the direction is in.
- **to** – The reference frame to convert the direction to.

**Returns** The corresponding direction, as a vector, in reference frame *to*.

std::tuple<double, double, double, double> **transform\_rotation** (std::tuple<double, double, double, double> *rotation*, *ReferenceFrame from*, *ReferenceFrame to*)

Converts a rotation from one reference frame to another.

#### Parameters

- **rotation** – Rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *from*.
- **from** – The reference frame that the rotation is in.
- **to** – The reference frame to convert the rotation to.

**Returns** The corresponding rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *to*.

`std::tuple<double, double, double> transform_velocity (std::tuple<double, double, double> position, std::tuple<double, double, double> velocity, ReferenceFrame from, ReferenceFrame to)`

Converts a velocity (acting at the specified position) from one reference frame to another. The position is required to take the relative angular velocity of the reference frames into account.

**Parameters**

- **position** – Position, as a vector, in reference frame *from*.
- **velocity** – Velocity, as a vector that points in the direction of travel and whose magnitude is the speed in meters per second, in reference frame *from*.
- **from** – The reference frame that the position and velocity are in.
- **to** – The reference frame to convert the velocity to.

**Returns** The corresponding velocity, as a vector, in reference frame *to*.

`bool far_available ()`

Whether [Ferram Aerospace Research](#) is installed.

`WarpMode warp_mode ()`

The current time warp mode. Returns `WarpMode::none` if time warp is not active, `WarpMode::rails` if regular “on-rails” time warp is active, or `WarpMode::physics` if physical time warp is active.

`Camera camera ()`

An object that can be used to control the camera.

`WaypointManager waypoint_manager ()`

The waypoint manager.

`ContractManager contract_manager ()`

The contract manager.

**enum struct WarpMode**

The time warp mode. Returned by `WarpMode`

**enumerator rails**

Time warp is active, and in regular “on-rails” mode.

**enumerator physics**

Time warp is active, and in physical time warp mode.

**enumerator none**

Time warp is not active.

## 4.3.2 Vessel

**class Vessel**

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources. Created using `active_vessel()` or `vessels()`.

`std::string name ()`  
 The name of the vessel.

`VesselType type ()`  
 The type of the vessel.

`VesselSituation situation ()`  
 The situation the vessel is in.

`bool recoverable ()`  
 Whether the vessel is recoverable.

`void recover ()`  
 Recover the vessel.

`double met ()`  
 The mission elapsed time in seconds.

`std::string biome ()`  
 The name of the biome the vessel is currently in.

`Flight flight (ReferenceFrame reference_frame = ReferenceFrame())`  
 Returns a *Flight* object that can be used to get flight telemetry for the vessel, in the specified reference frame.

#### Parameters

- **reference\_frame** – Reference frame. Defaults to the vessel's surface reference frame (`Vessel::surface_reference_frame ()`).

---

**Note:** When this is called with no arguments, the vessel's surface reference frame is used. This reference frame moves with the vessel, therefore velocities and speeds returned by the flight object will be zero. See the *reference frames tutorial* for examples of getting *the orbital and surface speeds of a vessel*.

---

`Orbit orbit ()`  
 The current orbit of the vessel.

`Control control ()`  
 Returns a *Control* object that can be used to manipulate the vessel's control inputs. For example, its pitch/yaw/roll controls, RCS and thrust.

`Comms comms ()`  
 Returns a *Comms* object that can be used to interact with CommNet for this vessel.

`AutoPilot auto_pilot ()`  
 An *AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

`Resources resources ()`  
 A *Resources* object, that can be used to get information about resources stored in the vessel.

`Resources resources_in_decouple_stage (int32_t stage, bool cumulative = true)`  
 Returns a *Resources* object, that can be used to get information about resources stored in a given *stage*.

#### Parameters

- **stage** – Get resources for parts that are decoupled in this stage.

- **cumulative** – When `false`, returns the resources for parts decoupled in just the given stage. When `true` returns the resources decoupled in the given stage and all subsequent stages combined.

---

**Note:** For details on stage numbering, see the discussion on *Staging*.

---

*Parts* **parts** ()

A *Parts* object, that can used to interact with the parts that make up this vessel.

float **mass** ()

The total mass of the vessel, including resources, in kg.

float **dry\_mass** ()

The total mass of the vessel, excluding resources, in kg.

float **thrust** ()

The total thrust currently being produced by the vessel's engines, in Newtons. This is computed by summing *Engine::thrust()* for every engine in the vessel.

float **available\_thrust** ()

Gets the total available thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *Engine::available\_thrust()* for every active engine in the vessel.

float **max\_thrust** ()

The total maximum thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *Engine::max\_thrust()* for every active engine.

float **max\_vacuum\_thrust** ()

The total maximum thrust that can be produced by the vessel's active engines when the vessel is in a vacuum, in Newtons. This is computed by summing *Engine::max\_vacuum\_thrust()* for every active engine.

float **specific\_impulse** ()

The combined specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

float **vacuum\_specific\_impulse** ()

The combined vacuum specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

float **kerbin\_sea\_level\_specific\_impulse** ()

The combined specific impulse of all active engines at sea level on Kerbin, in seconds. This is computed using the formula [described here](#).

std::tuple<double, double, double> **moment\_of\_inertia** ()

The moment of inertia of the vessel around its center of mass in  $kg.m^2$ . The inertia values in the returned 3-tuple are around the pitch, roll and yaw directions respectively. This corresponds to the vessels reference frame (*ReferenceFrame*).

std::vector<double> **inertia\_tensor** ()

The inertia tensor of the vessel around its center of mass, in the vessels reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **available\_torque** ()

The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gim-balled engines and aerodynamic control surfaces. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.



`std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> available_reaction_wheel_torque()`

The maximum torque that the currently active and powered reaction wheels can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

`std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> available_rcs_torque()`

The maximum torque that the currently active RCS thrusters can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

`std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> available_engine_torque()`

The maximum torque that the currently active and gimbaled engines can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

`std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> available_control_surface_torque()`

The maximum torque that the aerodynamic control surfaces can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

`std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> available_other_torque()`

The maximum torque that parts (excluding reaction wheels, gimbaled engines, RCS and control surfaces) can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

*ReferenceFrame* **reference\_frame**()

The reference frame that is fixed relative to the vessel, and orientated with the vessel.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel.
- The x-axis points out to the right of the vessel.
- The y-axis points in the forward direction of the vessel.
- The z-axis points out of the bottom off the vessel.

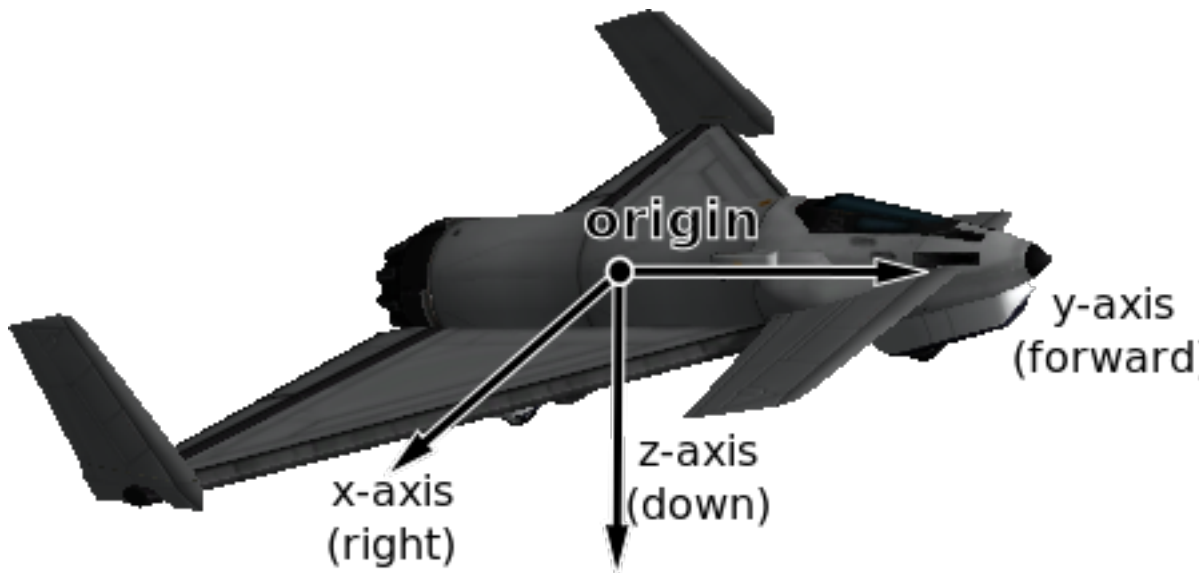


Fig. 4.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

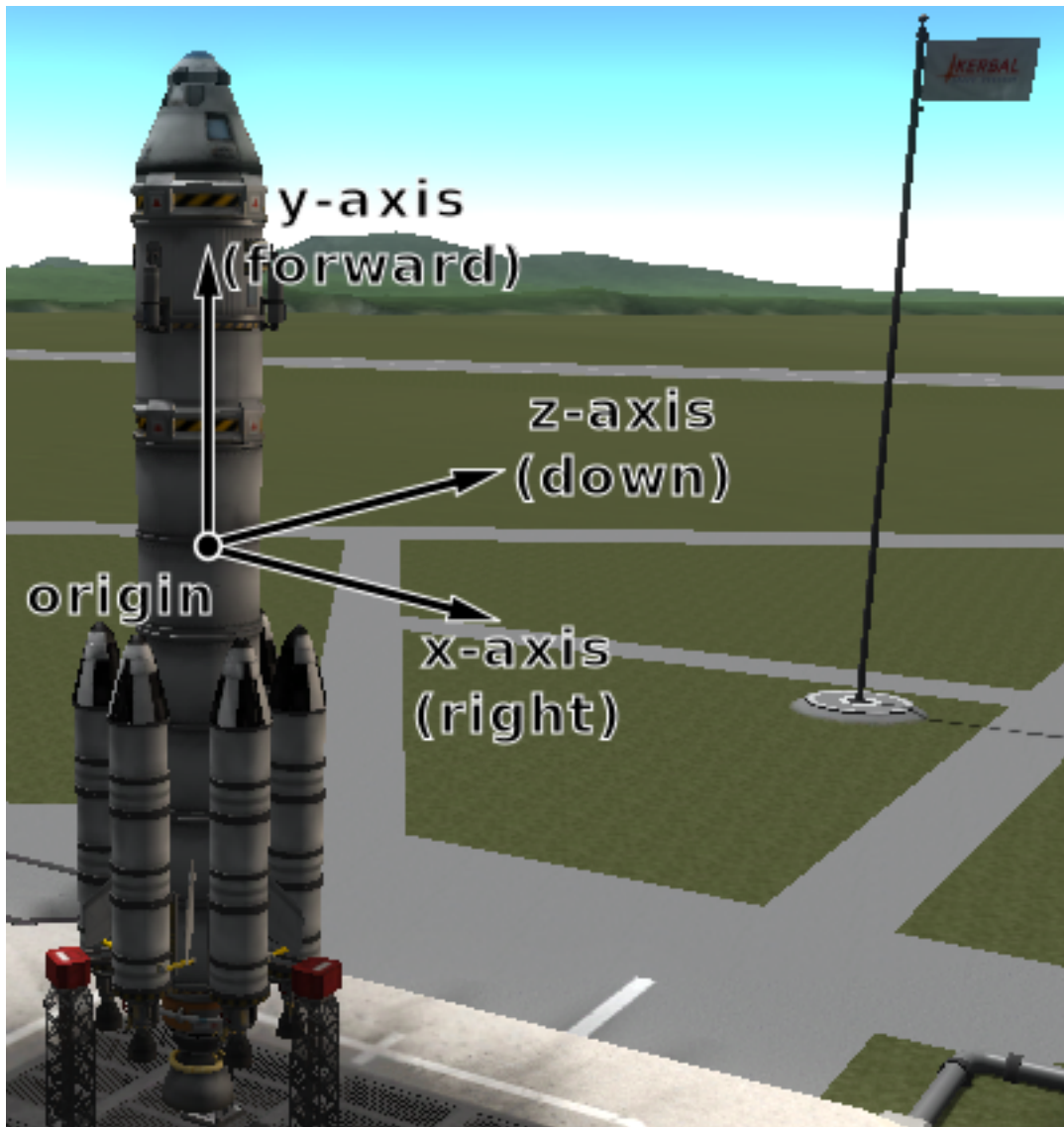


Fig. 4.2: Vessel reference frame origin and axes for the Kerbal-X rocket

*ReferenceFrame* **orbital\_reference\_frame** ()

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

---

**Note:** Be careful not to confuse this with ‘orbit’ mode on the navball.

---

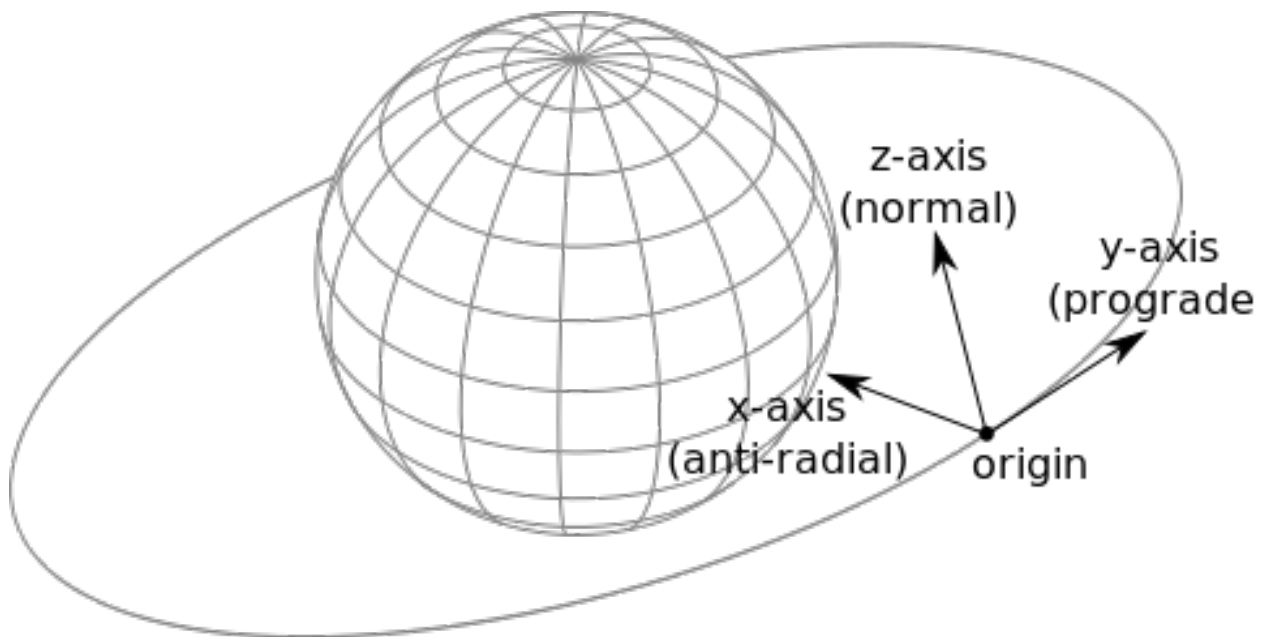


Fig. 4.3: Vessel orbital reference frame origin and axes

*ReferenceFrame* **surface\_reference\_frame** ()

The reference frame that is fixed relative to the vessel, and orientated with the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the north and up directions on the surface of the body.
- The x-axis points in the [zenith](#) direction (upwards, normal to the body being orbited, from the center of the body towards the center of mass of the vessel).
- The y-axis points northwards towards the [astronomical horizon](#) (north, and tangential to the surface of the body – the direction in which a compass would point when on the surface).
- The z-axis points eastwards towards the [astronomical horizon](#) (east, and tangential to the surface of the body – east on a compass when on the surface).

---

**Note:** Be careful not to confuse this with ‘surface’ mode on the navball.

---

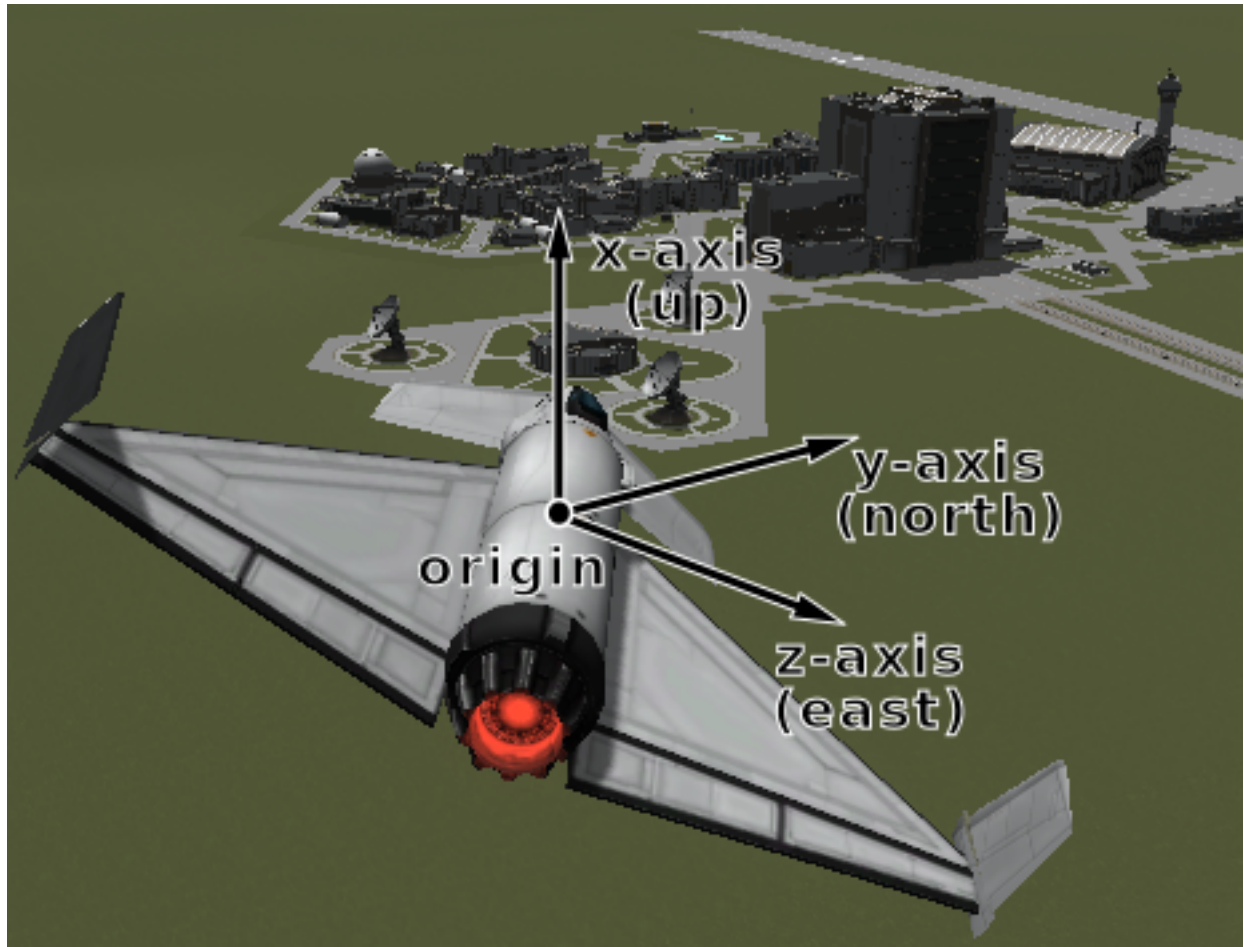


Fig. 4.4: Vessel surface reference frame origin and axes

*ReferenceFrame* **surface\_velocity\_reference\_frame** ()

The reference frame that is fixed relative to the vessel, and orientated with the velocity vector of the vessel relative to the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel's velocity vector.
- The y-axis points in the direction of the vessel's velocity vector, relative to the surface of the body being orbited.
- The z-axis is in the plane of the [astronomical horizon](#).
- The x-axis is orthogonal to the other two axes.

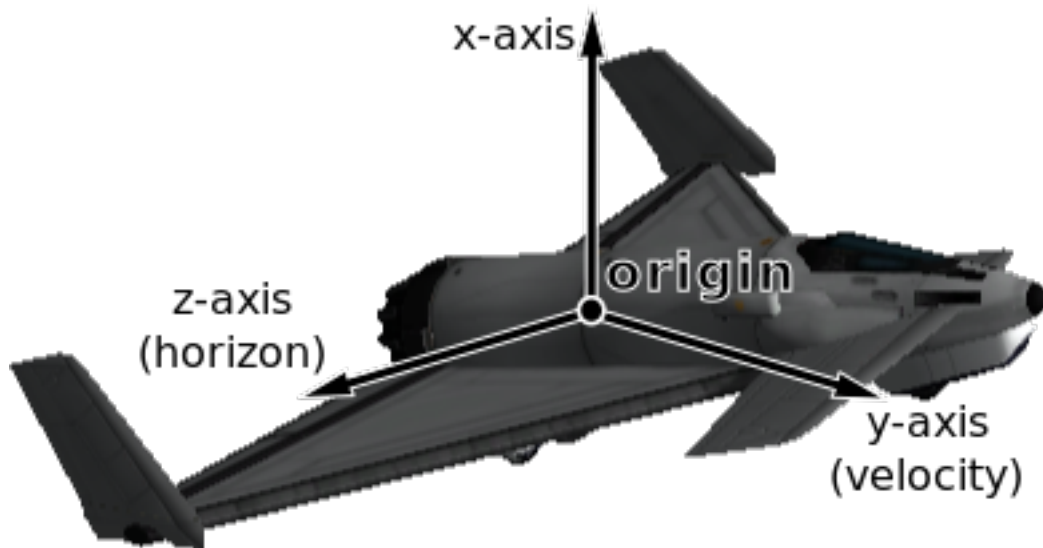


Fig. 4.5: Vessel surface velocity reference frame origin and axes

`std::tuple<double, double, double> position (ReferenceFrame reference_frame)`

The position of the center of mass of the vessel, in the given reference frame.

**Parameters**

- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

`std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> bounding_box (ReferenceFrame reference_frame)`

The axis-aligned bounding box of the vessel in the given reference frame.

**Parameters**

- **reference\_frame** – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

`std::tuple<double, double, double> velocity (ReferenceFrame reference_frame)`

The velocity of the center of mass of the vessel, in the given reference frame.

**Parameters**

- **reference\_frame** – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

`std::tuple<double, double, double, double> rotation (ReferenceFrame reference_frame)`

The rotation of the vessel, in the given reference frame.

**Parameters**

- **reference\_frame** – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

`std::tuple<double, double, double> direction (ReferenceFrame reference_frame)`

The direction in which the vessel is pointing, in the given reference frame.

**Parameters**

- **reference\_frame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

`std::tuple<double, double, double> angular_velocity (ReferenceFrame reference_frame)`

The angular velocity of the vessel, in the given reference frame.

**Parameters**

- **reference\_frame** – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the vessel, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

**enum struct VesselType**

The type of a vessel. See `Vessel::type()`.

**enumerator base**

Base.

**enumerator debris**

Debris.

**enumerator lander**

Lander.

**enumerator plane**

Plane.

**enumerator probe**

Probe.

**enumerator relay**

Relay.

**enumerator rover**

Rover.

**enumerator ship**

Ship.

**enumerator station**

Station.

**enum struct VesselSituation**

The situation a vessel is in. See `Vessel::situation()`.

**enumerator docked**  
Vessel is docked to another.

**enumerator escaping**  
Escaping.

**enumerator flying**  
Vessel is flying through an atmosphere.

**enumerator landed**  
Vessel is landed on the surface of a body.

**enumerator orbiting**  
Vessel is orbiting a body.

**enumerator pre\_launch**  
Vessel is awaiting launch.

**enumerator splashed**  
Vessel has splashed down in an ocean.

**enumerator sub\_orbital**  
Vessel is on a sub-orbital trajectory.

### 4.3.3 CelestialBody

#### **class CelestialBody**

Represents a celestial body (such as a planet or moon). See *bodies()*.

std::string **name**()  
The name of the body.

std::vector<CelestialBody> **satellites**()  
A list of celestial bodies that are in orbit around this celestial body.

Orbit **orbit**()  
The orbit of the body.

float **mass**()  
The mass of the body, in kilograms.

float **gravitational\_parameter**()  
The [standard gravitational parameter](#) of the body in  $m^3s^{-2}$ .

float **surface\_gravity**()  
The acceleration due to gravity at sea level (mean altitude) on the body, in  $m/s^2$ .

float **rotational\_period**()  
The sidereal rotational period of the body, in seconds.

float **rotational\_speed**()  
The rotational speed of the body, in radians per second.

double **rotation\_angle**()  
The current rotation angle of the body, in radians. A value between 0 and  $2\pi$

double **initial\_rotation**()  
The initial rotation angle of the body (at UT 0), in radians. A value between 0 and  $2\pi$

float **equatorial\_radius**()  
The equatorial radius of the body, in meters.

double **surface\_height** (double *latitude*, double *longitude*)

The height of the surface relative to mean sea level, in meters, at the given position. When over water this is equal to 0.

**Parameters**

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.

double **bedrock\_height** (double *latitude*, double *longitude*)

The height of the surface relative to mean sea level, in meters, at the given position. When over water, this is the height of the sea-bed and is therefore negative value.

**Parameters**

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.

std::tuple<double, double, double> **msl\_position** (double *latitude*, double *longitude*, *ReferenceFrame* *reference\_frame*)

The position at mean sea level at the given latitude and longitude, in the given reference frame.

**Parameters**

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.
- **reference\_frame** – Reference frame for the returned position vector.

**Returns** Position as a vector.

std::tuple<double, double, double> **surface\_position** (double *latitude*, double *longitude*, *ReferenceFrame* *reference\_frame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position of the surface of the water.

**Parameters**

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.
- **reference\_frame** – Reference frame for the returned position vector.

**Returns** Position as a vector.

std::tuple<double, double, double> **bedrock\_position** (double *latitude*, double *longitude*, *ReferenceFrame* *reference\_frame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position at the bottom of the sea-bed.

**Parameters**

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.
- **reference\_frame** – Reference frame for the returned position vector.

**Returns** Position as a vector.

std::tuple<double, double, double> **position\_at\_altitude** (double *latitude*, double *longitude*, double *altitude*, *ReferenceFrame* *reference\_frame*)

The position at the given latitude, longitude and altitude, in the given reference frame.



**Parameters**

- **latitude** – Latitude in degrees.
- **longitude** – Longitude in degrees.
- **altitude** – Altitude in meters above sea level.
- **reference\_frame** – Reference frame for the returned position vector.

**Returns** Position as a vector.

double **altitude\_at\_position** (std::tuple<double, double, double> *position*, *ReferenceFrame reference\_frame*)

The altitude, in meters, of the given position in the given reference frame.

**Parameters**

- **position** – Position as a vector.
- **reference\_frame** – Reference frame for the position vector.

double **latitude\_at\_position** (std::tuple<double, double, double> *position*, *ReferenceFrame reference\_frame*)

The latitude of the given position, in the given reference frame.

**Parameters**

- **position** – Position as a vector.
- **reference\_frame** – Reference frame for the position vector.

double **longitude\_at\_position** (std::tuple<double, double, double> *position*, *ReferenceFrame reference\_frame*)

The longitude of the given position, in the given reference frame.

**Parameters**

- **position** – Position as a vector.
- **reference\_frame** – Reference frame for the position vector.

float **sphere\_of\_influence** ()

The radius of the sphere of influence of the body, in meters.

bool **has\_atmosphere** ()

true if the body has an atmosphere.

float **atmosphere\_depth** ()

The depth of the atmosphere, in meters.

double **atmospheric\_density\_at\_position** (std::tuple<double, double, double> *position*, *ReferenceFrame reference\_frame*)

The atmospheric density at the given position, in  $kg/m^3$ , in the given reference frame.

**Parameters**

- **position** – The position vector at which to measure the density.
- **reference\_frame** – Reference frame that the position vector is in.

bool **has\_atmospheric\_oxygen** ()

true if there is oxygen in the atmosphere, required for air-breathing engines.

double **temperature\_at** (std::tuple<double, double, double> *position*, *ReferenceFrame reference\_frame*)

The temperature on the body at the given position, in the given reference frame.

**Parameters**

- **position** – Position as a vector.
- **reference\_frame** – The reference frame that the position is in.

---

**Note:** This calculation is performed using the bodies current position, which means that the value could be wrong if you want to know the temperature in the far future.

---

double **density\_at** (double *altitude*)

Gets the air density, in  $kg/m^3$ , for the specified altitude above sea level, in meters.

**Parameters**

---

**Note:** This is an approximation, because actual calculations, taking sun exposure into account to compute air temperature, require us to know the exact point on the body where the density is to be computed (knowing the altitude is not enough). However, the difference is small for high altitudes, so it makes very little difference for trajectory prediction.

---

double **pressure\_at** (double *altitude*)

Gets the air pressure, in Pascals, for the specified altitude above sea level, in meters.

**Parameters**

std::set<std::string> **biomes** ()

The biomes present on this body.

std::string **biome\_at** (double *latitude*, double *longitude*)

The biome at the given latitude and longitude, in degrees.

**Parameters**

float **flying\_high\_altitude\_threshold** ()

The altitude, in meters, above which a vessel is considered to be flying “high” when doing science.

float **space\_high\_altitude\_threshold** ()

The altitude, in meters, above which a vessel is considered to be in “high” space when doing science.

*ReferenceFrame* **reference\_frame** ()

The reference frame that is fixed relative to the celestial body.

- The origin is at the center of the body.
- The axes rotate with the body.
- The x-axis points from the center of the body towards the intersection of the prime meridian and equator (the position at 0° longitude, 0° latitude).
- The y-axis points from the center of the body towards the north pole.
- The z-axis points from the center of the body towards the equator at 90°E longitude.

*ReferenceFrame* **non\_rotating\_reference\_frame** ()

The reference frame that is fixed relative to this celestial body, and orientated in a fixed direction (it does not rotate with the body).

- The origin is at the center of the body.
- The axes do not rotate.
- The x-axis points in an arbitrary direction through the equator.

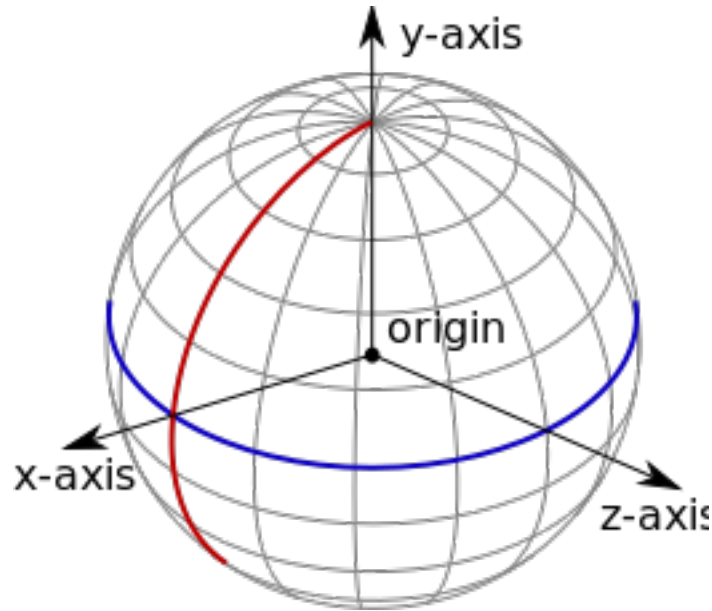


Fig. 4.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

- The y-axis points from the center of the body towards the north pole.
- The z-axis points in an arbitrary direction through the equator.

*ReferenceFrame* **orbital\_reference\_frame** ()

The reference frame that is fixed relative to this celestial body, but orientated with the body's orbital prograde/normal/radial directions.

- The origin is at the center of the body.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

`std::tuple<double, double, double>` **position** (*ReferenceFrame* *reference\_frame*)

The position of the center of the body, in the specified reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

`std::tuple<double, double, double>` **velocity** (*ReferenceFrame* *reference\_frame*)

The linear velocity of the body, in the specified reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

`std::tuple<double, double, double, double>` **rotation** (*ReferenceFrame* *reference\_frame*)

The rotation of the body, in the specified reference frame.

**Parameters**

- **reference\_frame** – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

`std::tuple<double, double, double> direction (ReferenceFrame reference_frame)`

The direction in which the north pole of the celestial body is pointing, in the specified reference frame.

**Parameters**

- **reference\_frame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

`std::tuple<double, double, double> angular_velocity (ReferenceFrame reference_frame)`

The angular velocity of the body in the specified reference frame.

**Parameters**

- **reference\_frame** – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the body, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

## 4.3.4 Flight

**class Flight**

Used to get flight telemetry for a vessel, by calling `Vessel::flight()`. All of the information returned by this class is given in the reference frame passed to that method. Obtained by calling `Vessel::flight()`.

---

**Note:** To get orbital information, such as the apoapsis or inclination, see *Orbit*.

---

`float g_force ()`

The current G force acting on the vessel in  $m/s^2$ .

`double mean_altitude ()`

The altitude above sea level, in meters. Measured from the center of mass of the vessel.

`double surface_altitude ()`

The altitude above the surface of the body or sea level, whichever is closer, in meters. Measured from the center of mass of the vessel.

`double bedrock_altitude ()`

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor. Measured from the center of mass of the vessel.

`double elevation ()`

The elevation of the terrain under the vessel, in meters. This is the height of the terrain above sea level, and is negative when the vessel is over the sea.

`double latitude ()`

The *latitude* of the vessel for the body being orbited, in degrees.

`double longitude ()`

The *longitude* of the vessel for the body being orbited, in degrees.

`std::tuple<double, double, double> velocity ()`

The velocity of the vessel, in the reference frame *ReferenceFrame*.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the vessel in meters per second.

double **speed** ()

The speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

double **horizontal\_speed** ()

The horizontal speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

double **vertical\_speed** ()

The vertical speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

std::tuple<double, double, double> **center\_of\_mass** ()

The position of the center of mass of the vessel, in the reference frame *ReferenceFrame*

**Returns** The position as a vector.

std::tuple<double, double, double, double> **rotation** ()

The rotation of the vessel, in the reference frame *ReferenceFrame*

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

std::tuple<double, double, double> **direction** ()

The direction that the vessel is pointing in, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

float **pitch** ()

The pitch of the vessel relative to the horizon, in degrees. A value between  $-90^\circ$  and  $+90^\circ$ .

float **heading** ()

The heading of the vessel (its angle relative to north), in degrees. A value between  $0^\circ$  and  $360^\circ$ .

float **roll** ()

The roll of the vessel relative to the horizon, in degrees. A value between  $-180^\circ$  and  $+180^\circ$ .

std::tuple<double, double, double> **prograde** ()

The prograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

std::tuple<double, double, double> **retrograde** ()

The retrograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

std::tuple<double, double, double> **normal** ()

The direction normal to the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

std::tuple<double, double, double> **anti\_normal** ()

The direction opposite to the normal of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

std::tuple<double, double, double> **radial** ()

The radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

std::tuple<double, double, double> **anti\_radial** ()

The direction opposite to the radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

float **atmosphere\_density** ()

The current density of the atmosphere around the vessel, in  $kg/m^3$ .

float **dynamic\_pressure** ()

The dynamic pressure acting on the vessel, in Pascals. This is a measure of the strength of the aerodynamic forces. It is equal to  $\frac{1}{2} \cdot \text{air density} \cdot \text{velocity}^2$ . It is commonly denoted  $Q$ .

float **static\_pressure** ()

The static atmospheric pressure acting on the vessel, in Pascals.

float **static\_pressure\_at\_msl** ()

The static atmospheric pressure at mean sea level, in Pascals.

std::tuple<double, double, double> **aerodynamic\_force** ()

The total aerodynamic forces acting on the vessel, in reference frame *ReferenceFrame*.

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

std::tuple<double, double, double> **simulate\_aerodynamic\_force\_at** (*CelestialBody* *body*,  
std::tuple<double, double, double> *position*,  
std::tuple<double, double, double> *velocity*)

Simulate and return the total aerodynamic forces acting on the vessel, if it were to be traveling with the given velocity at the given position in the atmosphere of the given celestial body.

#### Parameters

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

std::tuple<double, double, double> **lift** ()

The [aerodynamic lift](#) currently acting on the vessel.

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

std::tuple<double, double, double> **drag** ()

The [aerodynamic drag](#) currently acting on the vessel.

**Returns** A vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

float **speed\_of\_sound** ()

The speed of sound, in the atmosphere around the vessel, in  $m/s$ .

float **mach** ()

The speed of the vessel, in multiples of the speed of sound.

float **reynolds\_number** ()

The vessels Reynolds number.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

float **true\_air\_speed** ()

The [true air speed](#) of the vessel, in meters per second.

float **equivalent\_air\_speed** ()

The [equivalent air speed](#) of the vessel, in meters per second.

float **terminal\_velocity** ()

An estimate of the current terminal velocity of the vessel, in meters per second. This is the speed at which the drag forces cancel out the force of gravity.

float **angle\_of\_attack** ()

The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

float **sideslip\_angle** ()

The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

float **total\_air\_temperature** ()

The [total air temperature](#) of the atmosphere around the vessel, in Kelvin. This includes the *Flight::static\_air\_temperature()* and the vessel's kinetic energy.

float **static\_air\_temperature** ()

The [static \(ambient\) temperature](#) of the atmosphere around the vessel, in Kelvin.

float **stall\_fraction** ()

The current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

float **drag\_coefficient** ()

The coefficient of drag. This is the amount of drag produced by the vessel. It depends on air speed, air density and wing area.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

float **lift\_coefficient** ()

The coefficient of lift. This is the amount of lift produced by the vessel, and depends on air speed, air density and wing area.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

float **ballistic\_coefficient** ()

The [ballistic coefficient](#).

---

**Note:** Requires [Ferram Aerospace Research](#).

---

float **thrust\_specific\_fuel\_consumption** ()

The thrust specific fuel consumption for the jet engines on the vessel. This is a measure of the efficiency of the engines, with a lower value indicating a more efficient vessel. This value is the number of Newtons of fuel that are burned, per hour, to produce one newton of thrust.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

### 4.3.5 Orbit

**class Orbit**

Describes an orbit. For example, the orbit of a vessel, obtained by calling `Vessel::orbit()`, or a celestial body, obtained by calling `CelestialBody::orbit()`.

*CelestialBody* **body** ()

The celestial body (e.g. planet or moon) around which the object is orbiting.

double **apoapsis** ()

Gets the apoapsis of the orbit, in meters, from the center of mass of the body being orbited.

---

**Note:** For the apoapsis altitude reported on the in-game map view, use `Orbit::apoapsis_altitude()`.

---

double **periapsis** ()

The periapsis of the orbit, in meters, from the center of mass of the body being orbited.

---

**Note:** For the periapsis altitude reported on the in-game map view, use `Orbit::periapsis_altitude()`.

---

double **apoapsis\_altitude** ()

The apoapsis of the orbit, in meters, above the sea level of the body being orbited.

---

**Note:** This is equal to `Orbit::apoapsis()` minus the equatorial radius of the body.

---

double **periapsis\_altitude** ()

The periapsis of the orbit, in meters, above the sea level of the body being orbited.

---

**Note:** This is equal to `Orbit::periapsis()` minus the equatorial radius of the body.

---

double **semi\_major\_axis** ()

The semi-major axis of the orbit, in meters.

double **semi\_minor\_axis** ()

The semi-minor axis of the orbit, in meters.

double **radius** ()

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

---

**Note:** This value will change over time if the orbit is elliptical.

---

double **radius\_at** (double *ut*)

The orbital radius at the given time, in meters.

**Parameters**

- **ut** – The universal time to measure the radius at.

std::tuple<double, double, double> **position\_at** (double *ut*, *ReferenceFrame* *reference\_frame*)

The position at a given time, in the specified reference frame.



**Parameters**

- **ut** – The universal time to measure the position at.
- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

double **speed** ()

The current orbital speed of the object in meters per second.

---

**Note:** This value will change over time if the orbit is elliptical.

---

double **period** ()

The orbital period, in seconds.

double **time\_to\_apoapsis** ()

The time until the object reaches apoapsis, in seconds.

double **time\_to\_periapsis** ()

The time until the object reaches periapsis, in seconds.

double **eccentricity** ()

The *eccentricity* of the orbit.

double **inclination** ()

The *inclination* of the orbit, in radians.

double **longitude\_of\_ascending\_node** ()

The *longitude of the ascending node*, in radians.

double **argument\_of\_periapsis** ()

The *argument of periapsis*, in radians.

double **mean\_anomaly\_at\_epoch** ()

The *mean anomaly at epoch*.

double **epoch** ()

The time since the epoch (the point at which the *mean anomaly at epoch* was measured, in seconds.

double **mean\_anomaly** ()

The *mean anomaly*.

double **mean\_anomaly\_at\_ut** (double *ut*)

The mean anomaly at the given time.

**Parameters**

- **ut** – The universal time in seconds.

double **eccentric\_anomaly** ()

The *eccentric anomaly*.

double **eccentric\_anomaly\_at\_ut** (double *ut*)

The eccentric anomaly at the given universal time.

**Parameters**

- **ut** – The universal time, in seconds.

double **true\_anomaly** ()

The *true anomaly*.

double **true\_anomaly\_at\_ut** (double *ut*)

The true anomaly at the given time.

**Parameters**

- **ut** – The universal time in seconds.

double **true\_anomaly\_at\_radius** (double *radius*)

The true anomaly at the given orbital radius.

**Parameters**

- **radius** – The orbital radius in meters.

double **ut\_at\_true\_anomaly** (double *true\_anomaly*)

The universal time, in seconds, corresponding to the given true anomaly.

**Parameters**

- **true\_anomaly** – True anomaly.

double **radius\_at\_true\_anomaly** (double *true\_anomaly*)

The orbital radius at the point in the orbit given by the true anomaly.

**Parameters**

- **true\_anomaly** – The true anomaly.

double **true\_anomaly\_at\_an** (*Vessel target*)

The true anomaly of the ascending node with the given target vessel.

**Parameters**

- **target** – Target vessel.

double **true\_anomaly\_at\_dn** (*Vessel target*)

The true anomaly of the descending node with the given target vessel.

**Parameters**

- **target** – Target vessel.

double **orbital\_speed** ()

The current orbital speed in meters per second.

double **orbital\_speed\_at** (double *time*)

The orbital speed at the given time, in meters per second.

**Parameters**

- **time** – Time from now, in seconds.

**static** std::tuple<double, double, double> **reference\_plane\_normal** (*Client* &*connection*,  
*ReferenceFrame* *reference\_frame*)

The direction that is normal to the orbits reference plane, in the given reference frame. The reference plane is the plane from which the orbits inclination is measured.

**Parameters**

- **reference\_frame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**static** std::tuple<double, double, double> **reference\_plane\_direction** (*Client &connection,*  
*ReferenceFrame ref-*  
*erence\_frame*)

The direction from which the orbits longitude of ascending node is measured, in the given reference frame.

**Parameters**

- **reference\_frame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

double **relative\_inclination** (*Vessel target*)

Relative inclination of this orbit and the orbit of the given target vessel, in radians.

**Parameters**

- **target** – Target vessel.

double **time\_to\_soi\_change** ()

The time until the object changes sphere of influence, in seconds. Returns NaN if the object is not going to change sphere of influence.

*Orbit* **next\_orbit** ()

If the object is going to change sphere of influence in the future, returns the new orbit after the change. Otherwise returns NULL.

double **time\_of\_closest\_approach** (*Vessel target*)

Estimates and returns the time at closest approach to a target vessel.

**Parameters**

- **target** – Target vessel.

**Returns** The universal time at closest approach, in seconds.

double **distance\_at\_closest\_approach** (*Vessel target*)

Estimates and returns the distance at closest approach to a target vessel, in meters.

**Parameters**

- **target** – Target vessel.

std::vector<std::vector<double>>> **list\_closest\_approaches** (*Vessel target, int32\_t orbits*)

Returns the times at closest approach and corresponding distances, to a target vessel.

**Parameters**

- **target** – Target vessel.
- **orbits** – The number of future orbits to search.

**Returns** A list of two lists. The first is a list of times at closest approach, as universal times in seconds. The second is a list of corresponding distances at closest approach, in meters.

## 4.3.6 Control

### **class Control**

Used to manipulate the controls of a vessel. This includes adjusting the throttle, enabling/disabling systems such as SAS and RCS, or altering the direction in which the vessel is pointing. Obtained by calling *Vessel::control()*.

---

**Note:** Control inputs (such as pitch, yaw and roll) are zeroed when all clients that have set one or more of these inputs are no longer connected.

---

*ControlSource* **source** ()

The source of the vessels control, for example by a kerbal or a probe core.

*ControlState* **state** ()

The control state of the vessel.

bool **sas** ()

void **set\_sas** (bool *value*)

The state of SAS.

---

**Note:** Equivalent to *AutoPilot::sas()*

---

*SASMode* **sas\_mode** ()

void **set\_sas\_mode** (*SASMode value*)

The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

---

**Note:** Equivalent to *AutoPilot::sas\_mode()*

---

*SpeedMode* **speed\_mode** ()

void **set\_speed\_mode** (*SpeedMode value*)

The current *SpeedMode* of the navball. This is the mode displayed next to the speed at the top of the navball.

bool **rcs** ()

void **set\_rcs** (bool *value*)

The state of RCS.

bool **reaction\_wheels** ()

void **set\_reaction\_wheels** (bool *value*)

Returns whether all reactive wheels on the vessel are active, and sets the active state of all reaction wheels. See *ReactionWheel::active()*.

bool **gear** ()

void **set\_gear** (bool *value*)

The state of the landing gear/legs.

bool **legs** ()

void **set\_legs** (bool *value*)

Returns whether all landing legs on the vessel are deployed, and sets the deployment state of all landing legs. Does not include wheels (for example landing gear). See *Leg::deployed()*.

bool **wheels** ()

void **set\_wheels** (bool *value*)

Returns whether all wheels on the vessel are deployed, and sets the deployment state of all wheels. Does not include landing legs. See *Wheel::deployed()*.

bool **lights** ()

void **set\_lights** (bool *value*)  
The state of the lights.

bool **brakes** ()

void **set\_brakes** (bool *value*)  
The state of the wheel brakes.

bool **antennas** ()

void **set\_antennas** (bool *value*)  
Returns whether all antennas on the vessel are deployed, and sets the deployment state of all antennas. See *Antenna::deployed()*.

bool **cargo\_bays** ()

void **set\_cargo\_bays** (bool *value*)  
Returns whether any of the cargo bays on the vessel are open, and sets the open state of all cargo bays. See *CargoBay::open()*.

bool **intakes** ()

void **set\_intakes** (bool *value*)  
Returns whether all of the air intakes on the vessel are open, and sets the open state of all air intakes. See *Intake::open()*.

bool **parachutes** ()

void **set\_parachutes** (bool *value*)  
Returns whether all parachutes on the vessel are deployed, and sets the deployment state of all parachutes. Cannot be set to false. See *Parachute::deployed()*.

bool **radiators** ()

void **set\_radiators** (bool *value*)  
Returns whether all radiators on the vessel are deployed, and sets the deployment state of all radiators. See *Radiator::deployed()*.

bool **resource\_harvesters** ()

void **set\_resource\_harvesters** (bool *value*)  
Returns whether all of the resource harvesters on the vessel are deployed, and sets the deployment state of all resource harvesters. See *ResourceHarvester::deployed()*.

bool **resource\_harvesters\_active** ()

void **set\_resource\_harvesters\_active** (bool *value*)  
Returns whether any of the resource harvesters on the vessel are active, and sets the active state of all resource harvesters. See *ResourceHarvester::active()*.

bool **solar\_panels** ()

void **set\_solar\_panels** (bool *value*)  
Returns whether all solar panels on the vessel are deployed, and sets the deployment state of all solar panels. See *SolarPanel::deployed()*.

bool **abort** ()

void **set\_abort** (bool *value*)  
The state of the abort action group.

float **throttle** ()

void **set\_throttle** (float *value*)

The state of the throttle. A value between 0 and 1.

*ControlInputMode* **input\_mode** ()

void **set\_input\_mode** (*ControlInputMode value*)

Sets the behavior of the pitch, yaw, roll and translation control inputs. When set to additive, these inputs are added to the vessels current inputs. This mode is the default. When set to override, these inputs (if non-zero) override the vessels inputs. This mode prevents keyboard control, or SAS, from interfering with the controls when they are set.

float **pitch** ()

void **set\_pitch** (float *value*)

The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

float **yaw** ()

void **set\_yaw** (float *value*)

The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

float **roll** ()

void **set\_roll** (float *value*)

The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

float **forward** ()

void **set\_forward** (float *value*)

The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

float **up** ()

void **set\_up** (float *value*)

The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

float **right** ()

void **set\_right** (float *value*)

The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

float **wheel\_throttle** ()

void **set\_wheel\_throttle** (float *value*)

The state of the wheel throttle. A value between -1 and 1. A value of 1 rotates the wheels forwards, a value of -1 rotates the wheels backwards.

float **wheel\_steering** ()

void **set\_wheel\_steering** (float *value*)

The state of the wheel steering. A value between -1 and 1. A value of 1 steers to the left, and a value of -1 steers to the right.

int32\_t **current\_stage** ()

The current stage of the vessel. Corresponds to the stage number in the in-game UI.

std::vector<*Vessel*> **activate\_next\_stage** ()

Activates the next stage. Equivalent to pressing the space bar in-game.

**Returns** A list of vessel objects that are jettisoned from the active vessel.

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *active\_vessel* () no longer refer to the active vessel.

---

bool **get\_action\_group** (uint32\_t *group*)  
Returns `true` if the given action group is enabled.

**Parameters**

- **group** – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.

void **set\_action\_group** (uint32\_t *group*, bool *state*)  
Sets the state of the given action group.

**Parameters**

- **group** – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.

void **toggle\_action\_group** (uint32\_t *group*)  
Toggles the state of the given action group.

**Parameters**

- **group** – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.

*Node* **add\_node** (double *ut*, float *prograde* = 0.0, float *normal* = 0.0, float *radial* = 0.0)

Creates a maneuver node at the given universal time, and returns a *Node* object that can be used to modify it. Optionally sets the magnitude of the delta-v for the maneuver node in the prograde, normal and radial directions.

**Parameters**

- **ut** – Universal time of the maneuver node.
- **prograde** – Delta-v in the prograde direction.
- **normal** – Delta-v in the normal direction.
- **radial** – Delta-v in the radial direction.

std::vector<*Node*> **nodes** ()  
Returns a list of all existing maneuver nodes, ordered by time from first to last.

void **remove\_nodes** ()  
Remove all maneuver nodes.

**enum struct ControlState**

The control state of a vessel. See *Control::state()*.

**enumerator full**  
Full controllable.

**enumerator partial**  
Partially controllable.

**enumerator none**  
Not controllable.

**enum struct ControlSource**

The control source of a vessel. See *Control::source()*.

**enumerator kerbal**  
Vessel is controlled by a Kerbal.

**enumerator probe**  
Vessel is controlled by a probe core.

**enumerator none**

Vessel is not controlled.

**enum struct SASMode**

The behavior of the SAS auto-pilot. See *AutoPilot::sas\_mode()*.

**enumerator stability\_assist**

Stability assist mode. Dampen out any rotation.

**enumerator maneuver**

Point in the burn direction of the next maneuver node.

**enumerator prograde**

Point in the prograde direction.

**enumerator retrograde**

Point in the retrograde direction.

**enumerator normal**

Point in the orbit normal direction.

**enumerator anti\_normal**

Point in the orbit anti-normal direction.

**enumerator radial**

Point in the orbit radial direction.

**enumerator anti\_radial**

Point in the orbit anti-radial direction.

**enumerator target**

Point in the direction of the current target.

**enumerator anti\_target**

Point away from the current target.

**enum struct SpeedMode**

The mode of the speed reported in the navball. See *Control::speed\_mode()*.

**enumerator orbit**

Speed is relative to the vessel's orbit.

**enumerator surface**

Speed is relative to the surface of the body being orbited.

**enumerator target**

Speed is relative to the current target.

**enum struct ControlInputMode**

See *Control::input\_mode()*.

**enumerator additive**

Control inputs are added to the vessels current control inputs.

**enumerator override**

Control inputs (when they are non-zero) override the vessels current control inputs.

## 4.3.7 Communications

**class Comms**

Used to interact with CommNet for a given vessel. Obtained by calling *Vessel::comms()*.



```

bool can_communicate ()
    Whether the vessel can communicate with KSC.

bool can_transmit_science ()
    Whether the vessel can transmit science data to KSC.

double signal_strength ()
    Signal strength to KSC.

double signal_delay ()
    Signal delay to KSC in seconds.

double power ()
    The combined power of all active antennae on the vessel.

std::vector<CommLink> control_path ()
    The communication path used to control the vessel.

class CommLink
    Represents a communication node in the network. For example, a vessel or the KSC.

    CommLinkType type ()
        The type of link.

    double signal_strength ()
        Signal strength of the link.

    CommNode start ()
        Start point of the link.

    CommNode end ()
        Start point of the link.

enum struct CommLinkType
    The type of a communication link. See CommLink::type().

    enumerator home
        Link is to a base station on Kerbin.

    enumerator control
        Link is to a control source, for example a manned spacecraft.

    enumerator relay
        Link is to a relay satellite.

class CommNode
    Represents a communication node in the network. For example, a vessel or the KSC.

    std::string name ()
        Name of the communication node.

    bool is_home ()
        Whether the communication node is on Kerbin.

    bool is_control_point ()
        Whether the communication node is a control point, for example a manned vessel.

    bool is_vessel ()
        Whether the communication node is a vessel.

    Vessel vessel ()
        The vessel for this communication node.

```

### 4.3.8 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*
  - *Antenna*
  - *Cargo Bay*
  - *Control Surface*
  - *Decoupler*
  - *Docking Port*
  - *Engine*
  - *Experiment*
  - *Fairing*
  - *Intake*
  - *Leg*
  - *Launch Clamp*
  - *Light*
  - *Parachute*
  - *Radiator*
  - *Resource Converter*
  - *Resource Harvester*
  - *Reaction Wheel*
  - *RCS*
  - *Sensor*
  - *Solar Panel*
  - *Thruster*
  - *Wheel*
- *Trees of Parts*
  - *Traversing the Tree*
  - *Attachment Modes*
- *Fuel Lines*
- *Staging*

## Parts

### class **Parts**

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling `Vessel::parts()`.

`std::vector<Part> all ()`

A list of all of the vessels parts.

`Part root ()`

The vessels root part.

---

**Note:** See the discussion on *Trees of Parts*.

---

`Part controlling ()`

void `set_controlling (Part value)`

The part from which the vessel is controlled.

`std::vector<Part> with_name (std::string name)`

A list of parts whose `Part::name()` is *name*.

#### Parameters

`std::vector<Part> with_title (std::string title)`

A list of all parts whose `Part::title()` is *title*.

#### Parameters

`std::vector<Part> with_tag (std::string tag)`

A list of all parts whose `Part::tag()` is *tag*.

#### Parameters

`std::vector<Part> with_module (std::string module_name)`

A list of all parts that contain a *Module* whose `Module::name()` is *module\_name*.

#### Parameters

`std::vector<Part> in_stage (int32_t stage)`

A list of all parts that are activated in the given *stage*.

#### Parameters

---

**Note:** See the discussion on *Staging*.

---

`std::vector<Part> in_decouple_stage (int32_t stage)`

A list of all parts that are decoupled in the given *stage*.

#### Parameters

---

**Note:** See the discussion on *Staging*.

---

`std::vector<Module> modules_with_name (std::string module_name)`

A list of modules (combined across all parts in the vessel) whose `Module::name()` is *module\_name*.

#### Parameters

`std::vector<Antenna> antennas ()`  
A list of all antennas in the vessel.

`std::vector<CargoBay> cargo_bays ()`  
A list of all cargo bays in the vessel.

`std::vector<ControlSurface> control_surfaces ()`  
A list of all control surfaces in the vessel.

`std::vector<Decoupler> decouplers ()`  
A list of all decouplers in the vessel.

`std::vector<DockingPort> docking_ports ()`  
A list of all docking ports in the vessel.

`std::vector<Engine> engines ()`  
A list of all engines in the vessel.

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters, jet engines and RCS thrusters.

---

`std::vector<Experiment> experiments ()`  
A list of all science experiments in the vessel.

`std::vector<Fairing> fairings ()`  
A list of all fairings in the vessel.

`std::vector<Intake> intakes ()`  
A list of all intakes in the vessel.

`std::vector<Leg> legs ()`  
A list of all landing legs attached to the vessel.

`std::vector<LaunchClamp> launch_clamps ()`  
A list of all launch clamps attached to the vessel.

`std::vector<Light> lights ()`  
A list of all lights in the vessel.

`std::vector<Parachute> parachutes ()`  
A list of all parachutes in the vessel.

`std::vector<Radiator> radiators ()`  
A list of all radiators in the vessel.

`std::vector<RCS> rcs ()`  
A list of all RCS blocks/thrusters in the vessel.

`std::vector<ReactionWheel> reaction_wheels ()`  
A list of all reaction wheels in the vessel.

`std::vector<ResourceConverter> resource_converters ()`  
A list of all resource converters in the vessel.

`std::vector<ResourceHarvester> resource_harvesters ()`  
A list of all resource harvesters in the vessel.

`std::vector<Sensor> sensors ()`  
A list of all sensors in the vessel.

`std::vector<SolarPanel> solar_panels ()`  
A list of all solar panels in the vessel.

`std::vector<Wheel> wheels ()`  
 A list of all wheels in the vessel.

## Part

### class Part

Represents an individual part. Vessels are made up of multiple parts. Instances of this class can be obtained by several methods in *Parts*.

`std::string name ()`  
 Internal name of the part, as used in [part cfg files](#). For example “Mark1-2Pod”.

`std::string title ()`  
 Title of the part, as shown when the part is right clicked in-game. For example “Mk1-2 Command Pod”.

`std::string tag ()`

`void set_tag (std::string value)`  
 The name tag for the part. Can be set to a custom string using the in-game user interface.

---

**Note:** This requires either the [NameTag](#) or [kOS](#) mod to be installed.

---

`bool highlighted ()`

`void set_highlighted (bool value)`  
 Whether the part is highlighted.

`std::tuple<double, double, double> highlight_color ()`

`void set_highlight_color (std::tuple<double, double, double> value)`  
 The color used to highlight the part, as an RGB triple.

`double cost ()`  
 The cost of the part, in units of funds.

`Vessel vessel ()`  
 The vessel that contains this part.

`Part parent ()`  
 The parts parent. Returns NULL if the part does not have a parent. This, in combination with `Part::children()`, can be used to traverse the vessels parts tree.

---

**Note:** See the discussion on *Trees of Parts*.

---

`std::vector<Part> children ()`  
 The parts children. Returns an empty list if the part has no children. This, in combination with `Part::parent()`, can be used to traverse the vessels parts tree.

---

**Note:** See the discussion on *Trees of Parts*.

---

`bool axially_attached ()`  
 Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns `false`.

---

**Note:** See the discussion on *Attachment Modes*.

---

bool **radially\_attached** ()

Whether the part is radially attached to its parent, i.e. on the side of its parent. If the part has no parent, returns `false`.

---

**Note:** See the discussion on *Attachment Modes*.

---

int32\_t **stage** ()

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

---

**Note:** See the discussion on *Staging*.

---

int32\_t **decouple\_stage** ()

The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

---

**Note:** See the discussion on *Staging*.

---

bool **massless** ()

Whether the part is `massless`.

double **mass** ()

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

double **dry\_mass** ()

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

bool **shielded** ()

Whether the part is shielded from the exterior of the vessel, for example by a fairing.

float **dynamic\_pressure** ()

The dynamic pressure acting on the part, in Pascals.

double **impact\_tolerance** ()

The impact tolerance of the part, in meters per second.

double **temperature** ()

Temperature of the part, in Kelvin.

double **skin\_temperature** ()

Temperature of the skin of the part, in Kelvin.

double **max\_temperature** ()

Maximum temperature that the part can survive, in Kelvin.

double **max\_skin\_temperature** ()

Maximum temperature that the skin of the part can survive, in Kelvin.

float **thermal\_mass** ()

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

float **thermal\_skin\_mass** ()

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

float **thermal\_resource\_mass** ()

A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

float **thermal\_conduction\_flux** ()

The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **thermal\_convection\_flux** ()

The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **thermal\_radiation\_flux** ()

The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **thermal\_internal\_flux** ()

The rate at which heat energy is being generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **thermal\_skin\_to\_internal\_flux** ()

The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

*Resources* **resources** ()

A *Resources* object for the part.

bool **crossfeed** ()

Whether this part is crossfeed capable.

bool **is\_fuel\_line** ()

Whether this part is a fuel line.

std::vector<Part> **fuel\_lines\_from** ()

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

---

**Note:** See the discussion on *Fuel Lines*.

---

std::vector<Part> **fuel\_lines\_to** ()

The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

---

**Note:** See the discussion on *Fuel Lines*.

---

std::vector<Module> **modules** ()

The modules for this part.

*Antenna* **antenna** ()

A *Antenna* if the part is an antenna, otherwise NULL.

*CargoBay* **cargo\_bay** ()

A *CargoBay* if the part is a cargo bay, otherwise NULL.

*ControlSurface* **control\_surface** ()

A *ControlSurface* if the part is an aerodynamic control surface, otherwise NULL.

*Decoupler* **decoupler** ()

A *Decoupler* if the part is a decoupler, otherwise NULL.

*DockingPort* **docking\_port** ()

A *DockingPort* if the part is a docking port, otherwise NULL.

*Engine* **engine** ()

An *Engine* if the part is an engine, otherwise NULL.

*Experiment* **experiment** ()

An *Experiment* if the part is a science experiment, otherwise NULL.

*Fairing* **fairing** ()

A *Fairing* if the part is a fairing, otherwise NULL.

*Intake* **intake** ()

An *Intake* if the part is an intake, otherwise NULL.

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters and jet engines. For RCS thrusters see *RCS*.

---

*Leg* **leg** ()

A *Leg* if the part is a landing leg, otherwise NULL.

*LaunchClamp* **launch\_clamp** ()

A *LaunchClamp* if the part is a launch clamp, otherwise NULL.

*Light* **light** ()

A *Light* if the part is a light, otherwise NULL.

*Parachute* **parachute** ()

A *Parachute* if the part is a parachute, otherwise NULL.

*Radiator* **radiator** ()

A *Radiator* if the part is a radiator, otherwise NULL.

*RCS* **rcs** ()

A *RCS* if the part is an RCS block/thruster, otherwise NULL.

*ReactionWheel* **reaction\_wheel** ()

A *ReactionWheel* if the part is a reaction wheel, otherwise NULL.

*ResourceConverter* **resource\_converter** ()

A *ResourceConverter* if the part is a resource converter, otherwise NULL.

*ResourceHarvester* **resource\_harvester** ()

A *ResourceHarvester* if the part is a resource harvester, otherwise NULL.

*Sensor* **sensor** ()

A *Sensor* if the part is a sensor, otherwise NULL.

*SolarPanel* **solar\_panel** ()

A *SolarPanel* if the part is a solar panel, otherwise NULL.

*Wheel* **wheel** ()

A *Wheel* if the part is a wheel, otherwise NULL.



`std::tuple<double, double, double> position (ReferenceFrame reference_frame)`

The position of the part in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

---

**Note:** This is a fixed position in the part, defined by the parts model. It is not necessarily the same as the parts center of mass. Use `Part::center_of_mass()` to get the parts center of mass.

---

`std::tuple<double, double, double> center_of_mass (ReferenceFrame reference_frame)`

The position of the parts center of mass in the given reference frame. If the part is physicsless, this is equivalent to `Part::position()`.

#### Parameters

- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

`std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> bounding_box (ReferenceFrame ref-  
er-  
ence_frame)`

The axis-aligned bounding box of the part in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

---

**Note:** This is computed from the collision mesh of the part. If the part is not collidable, the box has zero volume and is centered on the `Part::position()` of the part.

---

`std::tuple<double, double, double> direction (ReferenceFrame reference_frame)`

The direction the part points in, in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

`std::tuple<double, double, double> velocity (ReferenceFrame reference_frame)`

The linear velocity of the part in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

`std::tuple<double, double, double, double> rotation (ReferenceFrame reference_frame)`

The rotation of the part, in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

`std::tuple<double, double, double> moment_of_inertia()`

The moment of inertia of the part in  $kg.m^2$  around its center of mass in the parts reference frame (*ReferenceFrame*).

`std::vector<double> inertia_tensor()`

The inertia tensor of the part in the parts reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

*ReferenceFrame* **reference\_frame()**

The reference frame that is fixed relative to this part, and centered on a fixed position within the part, defined by the parts model.

- The origin is at the position of the part, as returned by *Part::position()*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

---

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *DockingPort::reference\_frame()*.

---

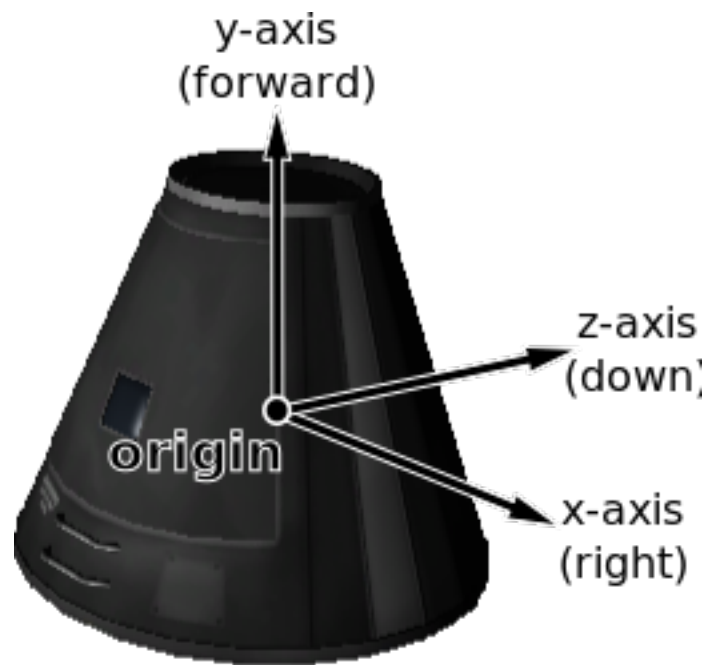


Fig. 4.7: Mk1 Command Pod reference frame origin and axes

*ReferenceFrame* **center\_of\_mass\_reference\_frame()**

The reference frame that is fixed relative to this part, and centered on its center of mass.

- The origin is at the center of mass of the part, as returned by *Part::center\_of\_mass()*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

---

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by `DockingPort::reference_frame()`.

---

Force **add\_force** (std::tuple<double, double, double> *force*, std::tuple<double, double, double> *position*, *ReferenceFrame* *reference\_frame*)

Exert a constant force on the part, acting at the given position.

**Parameters**

- **force** – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** – The position at which the force acts, as a vector.
- **reference\_frame** – The reference frame that the force and position are in.

**Returns** An object that can be used to remove or modify the force.

void **instantaneous\_force** (std::tuple<double, double, double> *force*, std::tuple<double, double, double> *position*, *ReferenceFrame* *reference\_frame*)

Exert an instantaneous force on the part, acting at the given position.

**Parameters**

- **force** – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** – The position at which the force acts, as a vector.
- **reference\_frame** – The reference frame that the force and position are in.

---

**Note:** The force is applied instantaneously in a single physics update.

---

**class Force**

Obtained by calling `Part::add_force()`.

*Part* **part** ()

The part that this force is applied to.

std::tuple<double, double, double> **force\_vector** ()

void **set\_force\_vector** (std::tuple<double, double, double> *value*)

The force vector, in Newtons.

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

std::tuple<double, double, double> **position** ()

void **set\_position** (std::tuple<double, double, double> *value*)

The position at which the force acts, in reference frame *ReferenceFrame*.

**Returns** The position as a vector.

*ReferenceFrame* **reference\_frame** ()

void **set\_reference\_frame** (*ReferenceFrame* *value*)

The reference frame of the force vector and position.

void **remove** ()

Remove the force.

## Module

### class Module

This can be used to interact with a specific part module. This includes part modules in stock KSP, and those added by mods.

In KSP, each part has zero or more [PartModules](#) associated with it. Each one contains some of the functionality of the part. For example, an engine has a “ModuleEngines” part module that contains all the functionality of an engine.

`std::string name ()`

Name of the PartModule. For example, “ModuleEngines”.

`Part part ()`

The part that contains this module.

`std::map<std::string, std::string> fields ()`

The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

`bool has_field (std::string name)`

Returns `true` if the module has a field with the given name.

#### Parameters

- **name** – Name of the field.

`std::string get_field (std::string name)`

Returns the value of a field.

#### Parameters

- **name** – Name of the field.

`void set_field_int (std::string name, int32_t value)`

Set the value of a field to the given integer number.

#### Parameters

- **name** – Name of the field.
- **value** – Value to set.

`void set_field_float (std::string name, float value)`

Set the value of a field to the given floating point number.

#### Parameters

- **name** – Name of the field.
- **value** – Value to set.

`void set_field_string (std::string name, std::string value)`

Set the value of a field to the given string.

#### Parameters

- **name** – Name of the field.
- **value** – Value to set.

`void reset_field (std::string name)`

Set the value of a field to its original value.

#### Parameters

- **name** – Name of the field.

`std::vector<std::string> events ()`

A list of the names of all of the modules events. Events are the clickable buttons visible in the right-click menu of the part.

`bool has_event (std::string name)`

`true` if the module has an event with the given name.

#### Parameters

`void trigger_event (std::string name)`

Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

#### Parameters

`std::vector<std::string> actions ()`

A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

`bool has_action (std::string name)`

`true` if the part has an action with the given name.

#### Parameters

`void set_action (std::string name, bool value = true)`

Set the value of an action with the given name.

#### Parameters

## Specific Types of Part

The following classes provide functionality for specific types of part.

- *Antenna*
- *Cargo Bay*
- *Control Surface*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Experiment*
- *Fairing*
- *Intake*
- *Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*

- *Resource Harvester*
- *Reaction Wheel*
- *RCS*
- *Sensor*
- *Solar Panel*
- *Thruster*
- *Wheel*

## Antenna

### **class Antenna**

An antenna. Obtained by calling *Part::antenna()*.

*Part* **part** ()

The part object for this antenna.

*AntennaState* **state** ()

The current state of the antenna.

bool **deployable** ()

Whether the antenna is deployable.

bool **deployed** ()

void **set\_deployed** (bool *value*)

Whether the antenna is deployed.

---

**Note:** Fixed antennas are always deployed. Returns an error if you try to deploy a fixed antenna.

---

bool **can\_transmit** ()

Whether data can be transmitted by this antenna.

void **transmit** ()

Transmit data.

void **cancel** ()

Cancel current transmission of data.

bool **allow\_partial** ()

void **set\_allow\_partial** (bool *value*)

Whether partial data transmission is permitted.

double **power** ()

The power of the antenna.

bool **combinable** ()

Whether the antenna can be combined with other antennae on the vessel to boost the power.

double **combinable\_exponent** ()

Exponent used to calculate the combined power of multiple antennae on a vessel.

float **packet\_interval** ()

Interval between sending packets in seconds.

float **packet\_size** ()  
 Amount of data sent per packet in Mits.

double **packet\_resource\_cost** ()  
 Units of electric charge consumed per packet sent.

**enum struct AntennaState**

The state of an antenna. See *Antenna::state()*.

**enumerator deployed**  
 Antenna is fully deployed.

**enumerator retracted**  
 Antenna is fully retracted.

**enumerator deploying**  
 Antenna is being deployed.

**enumerator retracting**  
 Antenna is being retracted.

**enumerator broken**  
 Antenna is broken.

## Cargo Bay

**class CargoBay**

A cargo bay. Obtained by calling *Part::cargo\_bay()*.

*Part* **part** ()  
 The part object for this cargo bay.

*CargoBayState* **state** ()  
 The state of the cargo bay.

bool **open** ()

void **set\_open** (bool *value*)  
 Whether the cargo bay is open.

**enum struct CargoBayState**

The state of a cargo bay. See *CargoBay::state()*.

**enumerator open**  
 Cargo bay is fully open.

**enumerator closed**  
 Cargo bay closed and locked.

**enumerator opening**  
 Cargo bay is opening.

**enumerator closing**  
 Cargo bay is closing.

## Control Surface

**class ControlSurface**

An aerodynamic control surface. Obtained by calling *Part::control\_surface()*.

*Part* **part** ()

The part object for this control surface.

bool **pitch\_enabled** ()

void **set\_pitch\_enabled** (bool *value*)

Whether the control surface has pitch control enabled.

bool **yaw\_enabled** ()

void **set\_yaw\_enabled** (bool *value*)

Whether the control surface has yaw control enabled.

bool **roll\_enabled** ()

void **set\_roll\_enabled** (bool *value*)

Whether the control surface has roll control enabled.

float **authority\_limiter** ()

void **set\_authority\_limiter** (float *value*)

The authority limiter for the control surface, which controls how far the control surface will move.

bool **inverted** ()

void **set\_inverted** (bool *value*)

Whether the control surface movement is inverted.

bool **deployed** ()

void **set\_deployed** (bool *value*)

Whether the control surface has been fully deployed.

float **surface\_area** ()

Surface area of the control surface in  $m^2$ .

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **available\_torque** ()

The available torque, in Newton meters, that can be produced by this control surface, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel::reference\_frame* ().

## Decoupler

**class Decoupler**

A decoupler. Obtained by calling *Part::decoupler* ()

*Part* **part** ()

The part object for this decoupler.

*Vessel* **decouple** ()

Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *active\_vessel* () no longer refer to the active vessel.

---

bool **decoupled** ()

Whether the decoupler has fired.



bool **staged** ()  
Whether the decoupler is enabled in the staging sequence.

float **impulse** ()  
The impulse that the decoupler imparts when it is fired, in Newton seconds.

## Docking Port

### class DockingPort

A docking port. Obtained by calling *Part::docking\_port()*

Part **part** ()  
The part object for this docking port.

DockingPortState **state** ()  
The current state of the docking port.

Part **docked\_part** ()  
The part that this docking port is docked to. Returns NULL if this docking port is not docked to anything.

Vessel **undock** ()  
Undocks the docking port and returns the new *Vessel* that is created. This method can be called for either docking port in a docked pair. Throws an exception if the docking port is not docked to anything.

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *active\_vessel()* no longer refer to the active vessel.

---

float **reengage\_distance** ()  
The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

bool **has\_shield** ()  
Whether the docking port has a shield.

bool **shielded** ()

void **set\_shielded** (bool *value*)  
The state of the docking ports shield, if it has one.

Returns `true` if the docking port has a shield, and the shield is closed. Otherwise returns `false`. When set to `true`, the shield is closed, and when set to `false` the shield is opened. If the docking port does not have a shield, setting this attribute has no effect.

std::tuple<double, double, double> **position** (ReferenceFrame *reference\_frame*)  
The position of the docking port, in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

std::tuple<double, double, double> **direction** (ReferenceFrame *reference\_frame*)  
The direction that docking port points in, in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

`std::tuple<double, double, double, double> rotation (ReferenceFrame reference_frame)`

The rotation of the docking port, in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

*ReferenceFrame* **reference\_frame** ()

The reference frame that is fixed relative to this docking port, and oriented with the port.

- The origin is at the position of the docking port.
- The axes rotate with the docking port.
- The x-axis points out to the right side of the docking port.
- The y-axis points in the direction the docking port is facing.
- The z-axis points out of the bottom off the docking port.

---

**Note:** This reference frame is not necessarily equivalent to the reference frame for the part, returned by `Part::reference_frame()`.

---



Fig. 4.8: Docking port reference frame origin and axes

**enum struct DockingPortState**

The state of a docking port. See `DockingPort::state()`.

**enumerator ready**

The docking port is ready to dock to another docking port.

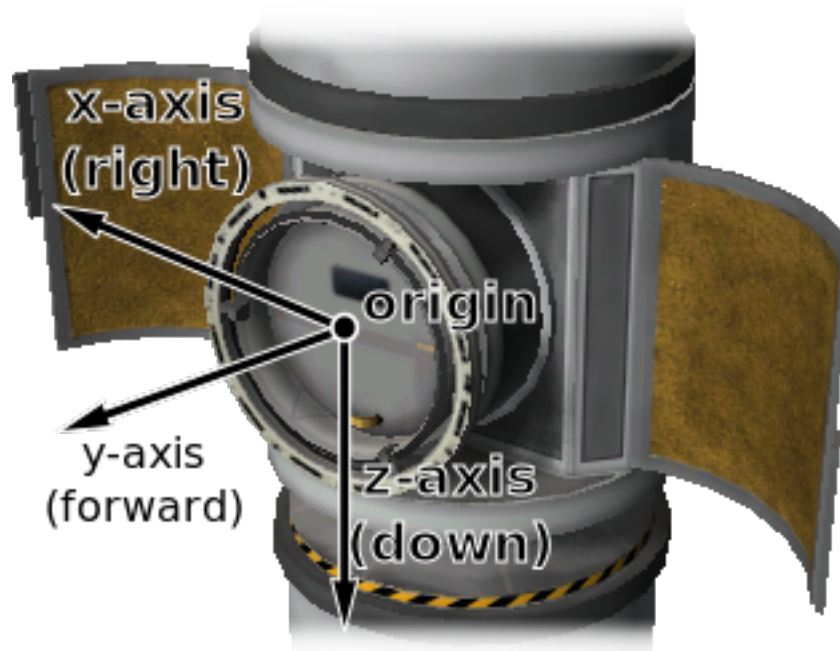


Fig. 4.9: Inline docking port reference frame origin and axes

**enumerator docked**

The docking port is docked to another docking port, or docked to another part (from the VAB/SPH).

**enumerator docking**

The docking port is very close to another docking port, but has not docked. It is using magnetic force to acquire a solid dock.

**enumerator undocking**

The docking port has just been undocked from another docking port, and is disabled until it moves away by a sufficient distance (*DockingPort::reengage\_distance()*).

**enumerator shielded**

The docking port has a shield, and the shield is closed.

**enumerator moving**

The docking ports shield is currently opening/closing.

## Engine

**class Engine**

An engine, including ones of various types. For example liquid fuelled gimballed engines, solid rocket boosters and jet engines. Obtained by calling *Part::engine()*.

---

**Note:** For RCS thrusters *Part::rcs()*.

---

*Part* **part** ()

The part object for this engine.

bool **active** ()

void **set\_active** (bool *value*)

Whether the engine is active. Setting this attribute may have no effect, depending on *Engine::can\_shutdown()* and *Engine::can\_restart()*.

float **thrust** ()

The current amount of thrust being produced by the engine, in Newtons.

float **available\_thrust** ()

The amount of thrust, in Newtons, that would be produced by the engine when activated and with its throttle set to 100%. Returns zero if the engine does not have any fuel. Takes the engine's current *Engine::thrust\_limit()* and atmospheric conditions into account.

float **max\_thrust** ()

The amount of thrust, in Newtons, that would be produced by the engine when activated and fueled, with its throttle and throttle limiter set to 100%.

float **max\_vacuum\_thrust** ()

The maximum amount of thrust that can be produced by the engine in a vacuum, in Newtons. This is the amount of thrust produced by the engine when activated, *Engine::thrust\_limit()* is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

float **thrust\_limit** ()

void **set\_thrust\_limit** (float *value*)

The thrust limiter of the engine. A value between 0 and 1. Setting this attribute may have no effect, for example the thrust limit for a solid rocket booster cannot be changed in flight.

std::vector<Thruster> **thrusters** ()

The components of the engine that generate thrust.

---

**Note:** For example, this corresponds to the rocket nozzle on a solid rocket booster, or the individual nozzles on a RAPIER engine. The overall thrust produced by the engine, as reported by *Engine::available\_thrust()*, *Engine::max\_thrust()* and others, is the sum of the thrust generated by each thruster.

---

float **specific\_impulse** ()

The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

float **vacuum\_specific\_impulse** ()

The vacuum specific impulse of the engine, in seconds.

float **kerbin\_sea\_level\_specific\_impulse** ()

The specific impulse of the engine at sea level on Kerbin, in seconds.

std::vector<std::string> **propellant\_names** ()

The names of the propellants that the engine consumes.

std::map<std::string, float> **propellant\_ratios** ()

The ratio of resources that the engine consumes. A dictionary mapping resource names to the ratio at which they are consumed by the engine.

---

**Note:** For example, if the ratios are 0.6 for LiquidFuel and 0.4 for Oxidizer, then for every 0.6 units of LiquidFuel that the engine burns, it will burn 0.4 units of Oxidizer.

---

std::vector<Propellant> **propellants** ()

The propellants that the engine consumes.

bool **has\_fuel** ()

Whether the engine has any fuel available.

---

**Note:** The engine must be activated for this property to update correctly.

---

float **throttle** ()

The current throttle setting for the engine. A value between 0 and 1. This is not necessarily the same as the vessel's main throttle setting, as some engines take time to adjust their throttle (such as jet engines).

bool **throttle\_locked** ()

Whether the `Control::throttle()` affects the engine. For example, this is `true` for liquid fueled rockets, and `false` for solid rocket boosters.

bool **can\_restart** ()

Whether the engine can be restarted once shutdown. If the engine cannot be shutdown, returns `false`. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

bool **can\_shutdown** ()

Whether the engine can be shutdown once activated. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

bool **has\_modes** ()

Whether the engine has multiple modes of operation.

std::string **mode** ()

void **set\_mode** (std::string *value*)

The name of the current engine mode.

std::map<std::string, *Engine*> **modes** ()

The available modes for the engine. A dictionary mapping mode names to *Engine* objects.

void **toggle\_mode** ()

Toggle the current engine mode.

bool **auto\_mode\_switch** ()

void **set\_auto\_mode\_switch** (bool *value*)

Whether the engine will automatically switch modes.

bool **gimballed** ()

Whether the engine is gimballed.

float **gimbal\_range** ()

The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

bool **gimbal\_locked** ()

void **set\_gimbal\_locked** (bool *value*)

Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

float **gimbal\_limit** ()

void **set\_gimbal\_limit** (float *value*)

The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **available\_torque** ()

The available torque, in Newton meters, that can be produced by this engine, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the `Vessel::reference_frame()`. Returns zero if the engine is inactive, or not gimballed.

**class Propellant**

A propellant for an engine. Obtains by calling *Engine::propellants()*.

std::string **name** ()

The name of the propellant.

double **current\_amount** ()

The current amount of propellant.

double **current\_requirement** ()

The required amount of propellant.

double **total\_resource\_available** ()

The total amount of the underlying resource currently reachable given resource flow rules.

double **total\_resource\_capacity** ()

The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

bool **ignore\_for\_isp** ()

If this propellant should be ignored when calculating required mass flow given specific impulse.

bool **ignore\_for\_thrust\_curve** ()

If this propellant should be ignored for thrust curve calculations.

bool **draw\_stack\_gauge** ()

If this propellant has a stack gauge or not.

bool **is\_deprived** ()

If this propellant is deprived.

float **ratio** ()

The propellant ratio.

## Experiment

**class Experiment**

Obtained by calling *Part::experiment()*.

*Part* **part** ()

The part object for this experiment.

void **run** ()

Run the experiment.

void **transmit** ()

Transmit all experimental data contained by this part.

void **dump** ()

Dump the experimental data contained by the experiment.

void **reset** ()

Reset the experiment.

bool **deployed** ()

Whether the experiment has been deployed.

bool **rerunnable** ()

Whether the experiment can be re-run.

bool **inoperable** ()

Whether the experiment is inoperable.

bool **has\_data** ()  
Whether the experiment contains data.

std::vector<ScienceData> **data** ()  
The data contained in this experiment.

std::string **biome** ()  
The name of the biome the experiment is currently in.

bool **available** ()  
Determines if the experiment is available given the current conditions.

ScienceSubject **science\_subject** ()  
Containing information on the corresponding specific science result for the current conditions. Returns NULL if the experiment is unavailable.

#### class ScienceData

Obtained by calling *Experiment::data()*.

float **data\_amount** ()  
Data amount.

float **science\_value** ()  
Science value.

float **transmit\_value** ()  
Transmit value.

#### class ScienceSubject

Obtained by calling *Experiment::science\_subject()*.

std::string **title** ()  
Title of science subject, displayed in science archives

bool **is\_complete** ()  
Whether the experiment has been completed.

float **science** ()  
Amount of science already earned from this subject, not updated until after transmission/recovery.

float **science\_cap** ()  
Total science allowable for this subject.

float **data\_scale** ()  
Multiply science value by this to determine data amount in mits.

float **subject\_value** ()  
Multiplier for specific Celestial Body/Experiment Situation combination.

float **scientific\_value** ()  
Diminishing value multiplier for decreasing the science value returned from repeated experiments.

## Fairing

#### class Fairing

A fairing. Obtained by calling *Part::fairing()*.

Part **part** ()  
The part object for this fairing.

void **jettison** ()  
Jettison the fairing. Has no effect if it has already been jettisoned.

bool **jettisoned**()  
Whether the fairing has been jettisoned.

## Intake

**class Intake**  
An air intake. Obtained by calling *Part::intake()*.

*Part* **part**()  
The part object for this intake.

bool **open**()

void **set\_open**(bool *value*)  
Whether the intake is open.

float **speed**()  
Speed of the flow into the intake, in *m/s*.

float **flow**()  
The rate of flow into the intake, in units of resource per second.

float **area**()  
The area of the intake's opening, in square meters.

## Leg

**class Leg**  
A landing leg. Obtained by calling *Part::leg()*.

*Part* **part**()  
The part object for this landing leg.

*LegState* **state**()  
The current state of the landing leg.

bool **deployable**()  
Whether the leg is deployable.

bool **deployed**()

void **set\_deployed**(bool *value*)  
Whether the landing leg is deployed.

---

**Note:** Fixed landing legs are always deployed. Returns an error if you try to deploy fixed landing gear.

---

bool **is\_grounded**()  
Returns whether the leg is touching the ground.

**enum struct LegState**  
The state of a landing leg. See *Leg::state()*.

**enumerator deployed**  
Landing leg is fully deployed.

**enumerator retracted**  
Landing leg is fully retracted.



**enumerator deploying**

Landing leg is being deployed.

**enumerator retracting**

Landing leg is being retracted.

**enumerator broken**

Landing leg is broken.

## Launch Clamp

**class LaunchClamp**A launch clamp. Obtained by calling *Part::launch\_clamp()*.*Part* **part** ()

The part object for this launch clamp.

void **release** ()

Releases the docking clamp. Has no effect if the clamp has already been released.

## Light

**class Light**A light. Obtained by calling *Part::light()*.*Part* **part** ()

The part object for this light.

bool **active** ()void **set\_active** (bool *value*)

Whether the light is switched on.

std::tuple<float, float, float> **color** ()void **set\_color** (std::tuple<float, float, float> *value*)

The color of the light, as an RGB triple.

float **power\_usage** ()

The current power usage, in units of charge per second.

## Parachute

**class Parachute**A parachute. Obtained by calling *Part::parachute()*.*Part* **part** ()

The part object for this parachute.

void **deploy** ()

Deploys the parachute. This has no effect if the parachute has already been deployed.

bool **deployed** ()

Whether the parachute has been deployed.

void **arm** ()

Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

bool **armed** ()

Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

*ParachuteState* **state** ()

The current state of the parachute.

float **deploy\_altitude** ()

void **set\_deploy\_altitude** (float *value*)

The altitude at which the parachute will full deploy, in meters. Only applicable to stock parachutes.

float **deploy\_min\_pressure** ()

void **set\_deploy\_min\_pressure** (float *value*)

The minimum pressure at which the parachute will semi-deploy, in atmospheres. Only applicable to stock parachutes.

**enum struct ParachuteState**

The state of a parachute. See *Parachute::state* ().

**enumerator stowed**

The parachute is safely tucked away inside its housing.

**enumerator armed**

The parachute is armed for deployment. (RealChutes only)

**enumerator active**

The parachute is still stowed, but ready to semi-deploy. (Stock parachutes only)

**enumerator semi\_deployed**

The parachute has been deployed and is providing some drag, but is not fully deployed yet. (Stock parachutes only)

**enumerator deployed**

The parachute is fully deployed.

**enumerator cut**

The parachute has been cut.

## Radiator

**class Radiator**

A radiator. Obtained by calling *Part::radiator* ().

*Part* **part** ()

The part object for this radiator.

bool **deployable** ()

Whether the radiator is deployable.

bool **deployed** ()

void **set\_deployed** (bool *value*)

For a deployable radiator, `true` if the radiator is extended. If the radiator is not deployable, this is always `true`.

*RadiatorState* **state** ()

The current state of the radiator.

---

**Note:** A fixed radiator is always *RadiatorState::extended*.

---

**enum struct RadiatorState**

The state of a radiator. *RadiatorState*

**enumerator extended**

Radiator is fully extended.

**enumerator retracted**

Radiator is fully retracted.

**enumerator extending**

Radiator is being extended.

**enumerator retracting**

Radiator is being retracted.

**enumerator broken**

Radiator is being broken.

**Resource Converter****class ResourceConverter**

A resource converter. Obtained by calling *Part::resource\_converter()*.

**Part part ()**

The part object for this converter.

**int32\_t count ()**

The number of converters in the part.

**std::string name (int32\_t index)**

The name of the specified converter.

**Parameters**

- **index** – Index of the converter.

**bool active (int32\_t index)**

True if the specified converter is active.

**Parameters**

- **index** – Index of the converter.

**void start (int32\_t index)**

Start the specified converter.

**Parameters**

- **index** – Index of the converter.

**void stop (int32\_t index)**

Stop the specified converter.

**Parameters**

- **index** – Index of the converter.

**ResourceConverterState state (int32\_t index)**

The state of the specified converter.

**Parameters**

- **index** – Index of the converter.

std::string **status\_info** (int32\_t *index*)

Status information for the specified converter. This is the full status message shown in the in-game UI.

**Parameters**

- **index** – Index of the converter.

std::vector<std::string> **inputs** (int32\_t *index*)

List of the names of resources consumed by the specified converter.

**Parameters**

- **index** – Index of the converter.

std::vector<std::string> **outputs** (int32\_t *index*)

List of the names of resources produced by the specified converter.

**Parameters**

- **index** – Index of the converter.

**enum struct ResourceConverterState**

The state of a resource converter. See *ResourceConverter::state()*.

**enumerator running**

Converter is running.

**enumerator idle**

Converter is idle.

**enumerator missing\_resource**

Converter is missing a required resource.

**enumerator storage\_full**

No available storage for output resource.

**enumerator capacity**

At preset resource capacity.

**enumerator unknown**

Unknown state. Possible with modified resource converters. In this case, check *ResourceConverter::status\_info()* for more information.

## Resource Harvester

**class ResourceHarvester**

A resource harvester (drill). Obtained by calling *Part::resource\_harvester()*.

*Part* **part** ()

The part object for this harvester.

*ResourceHarvesterState* **state** ()

The state of the harvester.

bool **deployed** ()

void **set\_deployed** (bool *value*)

Whether the harvester is deployed.

bool **active** ()

void **set\_active** (bool *value*)

Whether the harvester is actively drilling.

float **extraction\_rate** ()  
 The rate at which the drill is extracting ore, in units per second.

float **thermal\_efficiency** ()  
 The thermal efficiency of the drill, as a percentage of its maximum.

float **core\_temperature** ()  
 The core temperature of the drill, in Kelvin.

float **optimum\_core\_temperature** ()  
 The core temperature at which the drill will operate with peak efficiency, in Kelvin.

#### enum struct ResourceHarvesterState

The state of a resource harvester. See *ResourceHarvester::state()*.

enumerator **deploying**  
 The drill is deploying.

enumerator **deployed**  
 The drill is deployed and ready.

enumerator **retracting**  
 The drill is retracting.

enumerator **retracted**  
 The drill is retracted.

enumerator **active**  
 The drill is running.

## Reaction Wheel

### class ReactionWheel

A reaction wheel. Obtained by calling *Part::reaction\_wheel()*.

Part **part** ()  
 The part object for this reaction wheel.

bool **active** ()

void **set\_active** (bool *value*)  
 Whether the reaction wheel is active.

bool **broken** ()  
 Whether the reaction wheel is broken.

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **available\_torque** ()  
 The available torque, in Newton meters, that can be produced by this reaction wheel, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel::reference\_frame()*. Returns zero if the reaction wheel is inactive or broken.

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **max\_torque** ()  
 The maximum torque, in Newton meters, that can be produced by this reaction wheel, when it is active, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel::reference\_frame()*.

## RCS

### class RCS

An RCS block or thruster. Obtained by calling *Part::rcs()*.

*Part* **part** ()

The part object for this RCS.

bool **active** ()

Whether the RCS thrusters are active. An RCS thruster is inactive if the RCS action group is disabled (*Control::rcs()*), the RCS thruster itself is not enabled (*RCS::enabled()*) or it is covered by a fairing (*Part::shielded()*).

bool **enabled** ()

void **set\_enabled** (bool *value*)

Whether the RCS thrusters are enabled.

bool **pitch\_enabled** ()

void **set\_pitch\_enabled** (bool *value*)

Whether the RCS thruster will fire when pitch control input is given.

bool **yaw\_enabled** ()

void **set\_yaw\_enabled** (bool *value*)

Whether the RCS thruster will fire when yaw control input is given.

bool **roll\_enabled** ()

void **set\_roll\_enabled** (bool *value*)

Whether the RCS thruster will fire when roll control input is given.

bool **forward\_enabled** ()

void **set\_forward\_enabled** (bool *value*)

Whether the RCS thruster will fire when pitch control input is given.

bool **up\_enabled** ()

void **set\_up\_enabled** (bool *value*)

Whether the RCS thruster will fire when yaw control input is given.

bool **right\_enabled** ()

void **set\_right\_enabled** (bool *value*)

Whether the RCS thruster will fire when roll control input is given.

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **available\_torque** ()

The available torque, in Newton meters, that can be produced by this RCS, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel::reference\_frame()*. Returns zero if RCS is disable.

float **max\_thrust** ()

The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

float **max\_vacuum\_thrust** ()

The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

std::vector<*Thruster*> **thrusters** ()

A list of thrusters, one of each nozzle in the RCS part.

float **specific\_impulse** ()

The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

float **vacuum\_specific\_impulse** ()

The vacuum specific impulse of the RCS, in seconds.

float **kerbin\_sea\_level\_specific\_impulse** ()  
 The specific impulse of the RCS at sea level on Kerbin, in seconds.

std::vector<std::string> **propellants** ()  
 The names of resources that the RCS consumes.

std::map<std::string, float> **propellant\_ratios** ()  
 The ratios of resources that the RCS consumes. A dictionary mapping resource names to the ratios at which they are consumed by the RCS.

bool **has\_fuel** ()  
 Whether the RCS has fuel available.

---

**Note:** The RCS thruster must be activated for this property to update correctly.

---

## Sensor

### class Sensor

A sensor, such as a thermometer. Obtained by calling *Part::sensor()*.

*Part* **part** ()  
 The part object for this sensor.

bool **active** ()

void **set\_active** (bool *value*)  
 Whether the sensor is active.

std::string **value** ()  
 The current value of the sensor.

## Solar Panel

### class SolarPanel

A solar panel. Obtained by calling *Part::solar\_panel()*.

*Part* **part** ()  
 The part object for this solar panel.

bool **deployable** ()  
 Whether the solar panel is deployable.

bool **deployed** ()

void **set\_deployed** (bool *value*)  
 Whether the solar panel is extended.

*SolarPanelState* **state** ()  
 The current state of the solar panel.

float **energy\_flow** ()  
 The current amount of energy being generated by the solar panel, in units of charge per second.

float **sun\_exposure** ()  
 The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

**enum struct SolarPanelState**

The state of a solar panel. See *SolarPanel::state()*.

**enumerator extended**

Solar panel is fully extended.

**enumerator retracted**

Solar panel is fully retracted.

**enumerator extending**

Solar panel is being extended.

**enumerator retracting**

Solar panel is being retracted.

**enumerator broken**

Solar panel is broken.

## Thruster

**class Thruster**

The component of an *Engine* or *RCS* part that generates thrust. Can be obtained by calling *Engine::thrusters()* or *RCS::thrusters()*.

---

**Note:** Engines can consist of multiple thrusters. For example, the S3 KS-25x4 “Mammoth” has four rocket nozzles, and so consists of four thrusters.

---

**Part part()**

The *Part* that contains this thruster.

`std::tuple<double, double, double> thrust_position (ReferenceFrame reference_frame)`

The position at which the thruster generates thrust, in the given reference frame. For gimbaled engines, this takes into account the current rotation of the gimbal.

**Parameters**

- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

`std::tuple<double, double, double> thrust_direction (ReferenceFrame reference_frame)`

The direction of the force generated by the thruster, in the given reference frame. This is opposite to the direction in which the thruster expels propellant. For gimbaled engines, this takes into account the current rotation of the gimbal.

**Parameters**

- **reference\_frame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**ReferenceFrame thrust\_reference\_frame()**

A reference frame that is fixed relative to the thruster and orientated with its thrust direction (*Thruster::thrust\_direction()*). For gimbaled engines, this takes into account the current rotation of the gimbal.

- The origin is at the position of thrust for this thruster (*Thruster::thrust\_position()*).
- The axes rotate with the thrust direction. This is the direction in which the thruster expels propellant, including any gimbaling.



- The y-axis points along the thrust direction.
- The x-axis and z-axis are perpendicular to the thrust direction.

bool **gimballed** ()

Whether the thruster is gimballed.

std::tuple<double, double, double> **gimbal\_position** (*ReferenceFrame reference\_frame*)

Position around which the gimbal pivots.

#### Parameters

- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

std::tuple<double, double, double> **gimbal\_angle** ()

The current gimbal angle in the pitch, roll and yaw axes, in degrees.

std::tuple<double, double, double> **initial\_thrust\_position** (*ReferenceFrame reference\_frame*)

The position at which the thruster generates thrust, when the engine is in its initial position (no gimballing), in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

---

**Note:** This position can move when the gimbal rotates. This is because the thrust position and gimbal position are not necessarily the same.

---

std::tuple<double, double, double> **initial\_thrust\_direction** (*ReferenceFrame reference\_frame*)

The direction of the force generated by the thruster, when the engine is in its initial position (no gimballing), in the given reference frame. This is opposite to the direction in which the thruster expels propellant.

#### Parameters

- **reference\_frame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

## Wheel

### class Wheel

A wheel. Includes landing gear and rover wheels. Obtained by calling *Part::wheel()*. Can be used to control the motors, steering and deployment of wheels, among other things.

*Part* **part** ()

The part object for this wheel.

*WheelState* **state** ()

The current state of the wheel.

float **radius** ()

Radius of the wheel, in meters.

bool **grounded** ()  
Whether the wheel is touching the ground.

bool **has\_brakes** ()  
Whether the wheel has brakes.

float **brakes** ()

void **set\_brakes** (float *value*)  
The braking force, as a percentage of maximum, when the brakes are applied.

bool **auto\_friction\_control** ()

void **set\_auto\_friction\_control** (bool *value*)  
Whether automatic friction control is enabled.

float **manual\_friction\_control** ()

void **set\_manual\_friction\_control** (float *value*)  
Manual friction control value. Only has an effect if automatic friction control is disabled. A value between 0 and 5 inclusive.

bool **deployable** ()  
Whether the wheel is deployable.

bool **deployed** ()

void **set\_deployed** (bool *value*)  
Whether the wheel is deployed.

bool **powered** ()  
Whether the wheel is powered by a motor.

bool **motor\_enabled** ()

void **set\_motor\_enabled** (bool *value*)  
Whether the motor is enabled.

bool **motor\_inverted** ()

void **set\_motor\_inverted** (bool *value*)  
Whether the direction of the motor is inverted.

*MotorState* **motor\_state** ()  
Whether the direction of the motor is inverted.

float **motor\_output** ()  
The output of the motor. This is the torque currently being generated, in Newton meters.

bool **traction\_control\_enabled** ()

void **set\_traction\_control\_enabled** (bool *value*)  
Whether automatic traction control is enabled. A wheel only has traction control if it is powered.

float **traction\_control** ()

void **set\_traction\_control** (float *value*)  
Setting for the traction control. Only takes effect if the wheel has automatic traction control enabled. A value between 0 and 5 inclusive.

float **drive\_limiter** ()

void **set\_drive\_limiter** (float *value*)  
Manual setting for the motor limiter. Only takes effect if the wheel has automatic traction control disabled. A value between 0 and 100 inclusive.

```

bool steerable ()
    Whether the wheel has steering.

bool steering_enabled ()

void set_steering_enabled (bool value)
    Whether the wheel steering is enabled.

bool steering_inverted ()

void set_steering_inverted (bool value)
    Whether the wheel steering is inverted.

bool has_suspension ()
    Whether the wheel has suspension.

float suspension_spring_strength ()
    Suspension spring strength, as set in the editor.

float suspension_damper_strength ()
    Suspension damper strength, as set in the editor.

bool broken ()
    Whether the wheel is broken.

bool repairable ()
    Whether the wheel is repairable.

float stress ()
    Current stress on the wheel.

float stress_tolerance ()
    Stress tolerance of the wheel.

float stress_percentage ()
    Current stress on the wheel as a percentage of its stress tolerance.

float deflection ()
    Current deflection of the wheel.

float slip ()
    Current slip of the wheel.

enum struct WheelState
    The state of a wheel. See Wheel::state().

    enumerator deployed
        Wheel is fully deployed.

    enumerator retracted
        Wheel is fully retracted.

    enumerator deploying
        Wheel is being deployed.

    enumerator retracting
        Wheel is being retracted.

    enumerator broken
        Wheel is broken.

enum struct MotorState
    The state of the motor on a powered wheel. See Wheel::motor_state().

```

**enumerator idle**

The motor is idle.

**enumerator running**

The motor is running.

**enumerator disabled**

The motor is disabled.

**enumerator inoperable**

The motor is inoperable.

**enumerator not\_enough\_resources**

The motor does not have enough resources to run.

## Trees of Parts

Vessels in KSP are comprised of a number of parts, connected to one another in a *tree* structure. An example vessel is shown in Figure 1, and the corresponding tree of parts in Figure 2. The craft file for this example can also be downloaded [here](#).

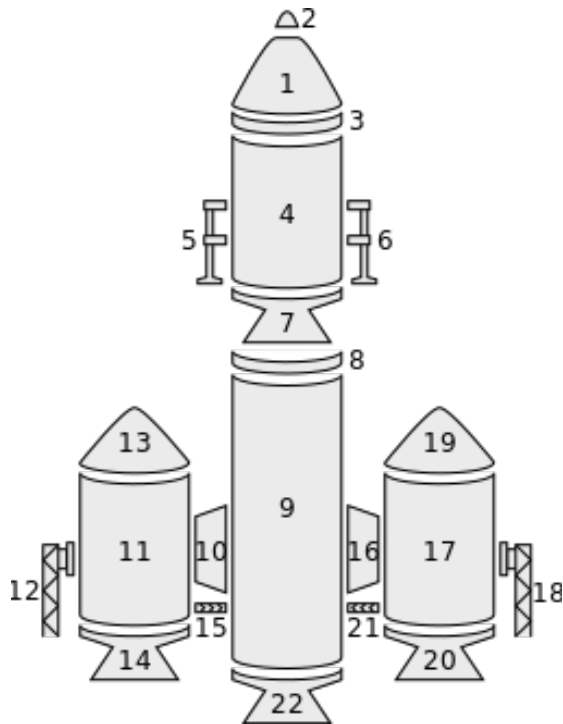


Fig. 4.10: **Figure 1** – Example parts making up a vessel.

## Traversing the Tree

The tree of parts can be traversed using the attributes `Parts::root()`, `Part::parent()` and `Part::children()`.

The root of the tree is the same as the vessels *root part* (part number 1 in the example above) and can be obtained by calling `Parts::root()`. A parts children can be obtained by calling `Part::children()`. If the part does not have any children, `Part::children()` returns an empty list. A parts parent can be obtained by calling `Part::parent()`. If the part does not have a parent (as is the case for the root part), `Part::parent()` returns `NULL`.

The following C++ example uses these attributes to perform a depth-first traversal over all of the parts in a vessel:

```
#include <iostream>
#include <stack>
#include <string>
#include <utility>
#include <krpc.hpp>
#include <krpc/service/

using
↳SpaceCenter = krpc

int main() {
    krpc::Client conn
    SpaceCenter sc(&co
    auto vessel = sc.a

    auto root = vessel
    std::stack
    ↳<std::pair<SpaceCe
```

```
stack.push(std::pa
↪<SpaceCenter::Part
while (!stack.empty
    auto part = stac
    auto depth = sta
    stack.pop();
    std::cout << std
↪ ' ') << part.titl
    for (auto child
        stack.push(std
↪<SpaceCenter::Part
    }
}
```

When this code is execute using the craft file for the example vessel pictured above, the following is printed out:

```
Command Pod Mk1
TR-18A Stack Decoupl
FL-T400 Fuel Tank
LV-909 Liquid Fue
TR-18A Stack Dec
FL-T800 Fuel Ta
LV-909 Liquid
TT-70 Radial D
FL-T400 Fuel
TT18-A Launc
FTX-2 Extern
LV-909 Liqui
Aerodynamic
TT-70 Radial D
FL-T400 Fuel
TT18-A Launc
FTX-2 Extern
LV-909 Liqui
Aerodynamic
LT-1 Landing Stru
LT-1 Landing Stru
Mk16 Parachute
```

Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

The root part of a vessel (for example the command pod – part 1) does not have a parent part, so does not have an attachment mode. However, the part is consider to be *axially* attached to

nothing.

The following C++ example does a depth-first traversal as before, but also prints out the attachment mode used by the part:

```
#include <iostream>
#include <stack>
#include <string>
#include <utility>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

using_
↳ SpaceCenter = krpc::services::SpaceCenter;

int main() {
    auto conn = krpc::connect();
    SpaceCenter sc(&conn);
    auto vessel = sc.active_vessel();

    auto root = vessel.parts().root();
    std::stack
↳ <std::pair<SpaceCenter::Part, int> > stack;
    stack.push(std::pair
↳ <SpaceCenter::Part, int>(root, 0));
    while (!stack.empty()) {
        auto part = stack.top().first;
        auto depth = stack.top().second;
        stack.pop();
        std::string attach_mode = part.
↳ axially_attached() ? "axial" : "radial";
        std::cout_
↳ << std::string(depth, ' ') << part.title()_
↳ << " - " << attach_mode << std::endl;
        auto children = part.children();
        for (auto child : children) {
            stack.push(std::pair
↳ <SpaceCenter::Part, int>(child, depth+1);
        }
    }
}
```

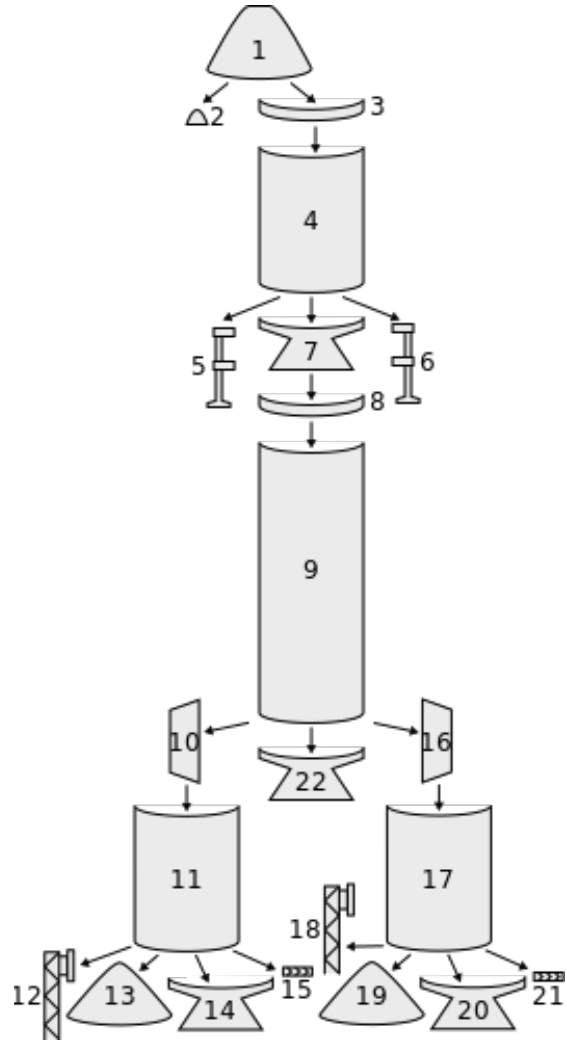


Fig. 4.11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the child part.

When this code is executed using the craft file for the example vessel pictured above, the following is printed out:

```
Command Pod Mk1 - axial
TR-18A Stack Decoupler - axial
FL-T400 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TR-18A Stack Decoupler - axial
FL-T800 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
↳
↳ TT18-A Launch Stability Enhancer - radial
   FTX-2 External Fuel Duct - radial
   LV-909 Liquid Fuel Engine - axial
   Aerodynamic Nose Cone - axial
   TT-70 Radial Decoupler - radial
```

```

FL-T400 Fuel Tank - radial
→ TT18-A Launch Stability Enhancer - radial
  FTX-2 External Fuel Duct - radial
  LV-909 Liquid Fuel Engine - axial
  Aerodynamic Nose Cone - axial
  LT-1 Landing Struts - radial
  LT-1 Landing Struts - radial
  Mk16 Parachute - axial

```

## Fuel Lines

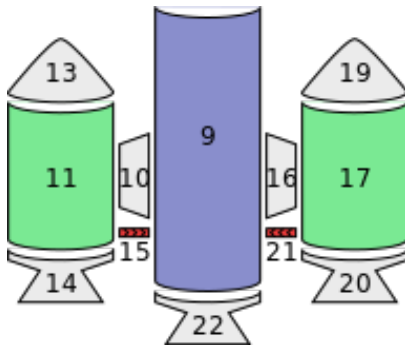


Fig. 4.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.

The attributes `Part::fuel_lines_from()` and `Part::fuel_lines_to()` can be used to discover these connections. In the example in Figure 5, when `Part::fuel_lines_to()` is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When `Part::fuel_lines_from()` is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).

## Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using `Part::stage()` and `Part::decouple_stage()`

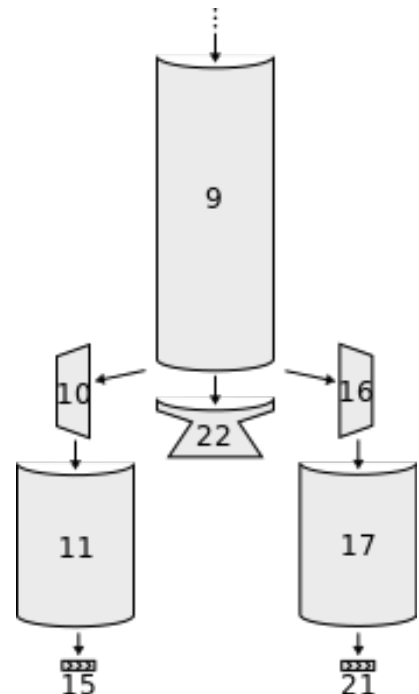


Fig. 4.13: **Figure 4** – A subset of the parts tree from Figure 2 above.

respectively. For parts that are not activated by staging, `Part::stage()` returns -1. For parts that are never decoupled, `Part::decouple_stage()` returns a value of -1.

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *activated*. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

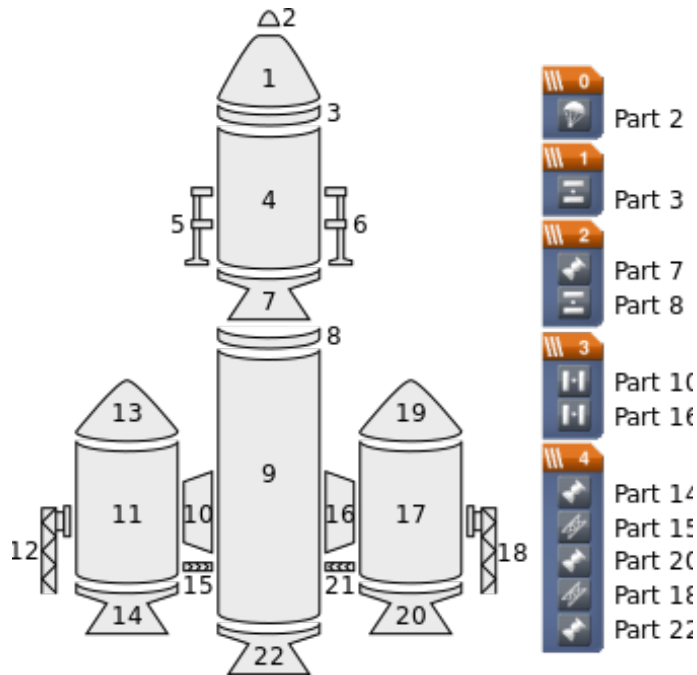


Fig. 4.14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.



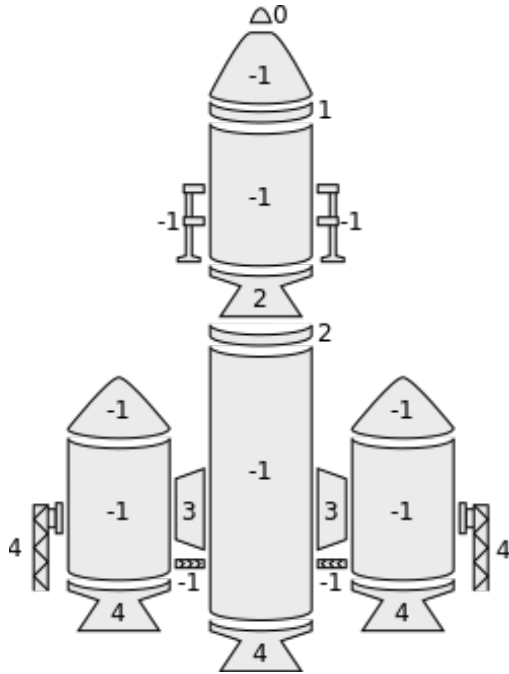


Fig. 4.15: **Figure 7** – The stage in which each part is *activated*.

### 4.3.9 Resources

#### class Resources

Represents the collection of resources stored in a vessel, stage or part. Created by calling `Vessel::resources()`, `Vessel::resources_in_decouple_stage()` or `Part::resources()`.

`std::vector<Resource> all()`

All the individual resources that can be stored.

`std::vector<Resource> with_resource(std::string name)`

All the individual resources with the given name that can be stored.

#### Parameters

`std::vector<std::string> names()`

A list of resource names that can be stored.

`bool has_resource(std::string name)`

Check whether the named resource can be stored.

#### Parameters

- **name** – The name of the resource.

`float amount(std::string name)`

Returns the amount of a resource that is currently stored.

#### Parameters

- **name** – The name of the resource.

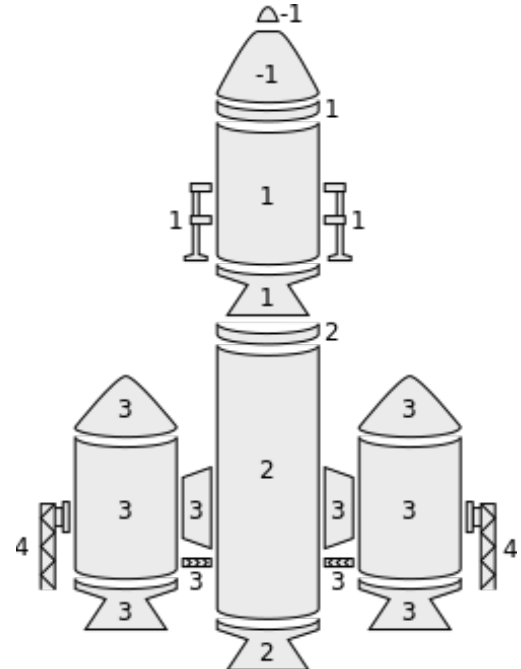


Fig. 4.16: **Figure 8** – The stage in which each part is *decoupled*.

float **max** (std::string *name*)

Returns the amount of a resource that can be stored.

**Parameters**

- **name** – The name of the resource.

static float **density** (*Client &connection*, std::string *name*)

Returns the density of a resource, in *kg/l*.

**Parameters**

- **name** – The name of the resource.

static *ResourceFlowMode* **flow\_mode** (*Client &connection*, std::string *name*)

Returns the flow mode of a resource.

**Parameters**

- **name** – The name of the resource.

bool **enabled** ()

void **set\_enabled** (bool *value*)

Whether use of all the resources are enabled.

---

**Note:** This is `true` if all of the resources are enabled. If any of the resources are not enabled, this is `false`.

---

**class Resource**

An individual resource stored within a part. Created using methods in the *Resources* class.

std::string **name** ()

The name of the resource.

*Part* **part** ()

The part containing the resource.

float **amount** ()

The amount of the resource that is currently stored in the part.

float **max** ()

The total amount of the resource that can be stored in the part.

float **density** ()

The density of the resource, in *kg/l*.

*ResourceFlowMode* **flow\_mode** ()

The flow mode of the resource.

bool **enabled** ()

void **set\_enabled** (bool *value*)

Whether use of this resource is enabled.

**class ResourceTransfer**

Transfer resources between parts.

**static ResourceTransfer start** (*Client &connection, Part from\_part, Part to\_part, std::string resource, float max\_amount*)

Start transferring a resource transfer between a pair of parts. The transfer will move at most *max\_amount* units of the resource, depending on how much of the resource is available in the source part and how much storage is available in the destination part. Use *ResourceTransfer::complete()* to check if the transfer is complete. Use *ResourceTransfer::amount()* to see how much of the resource has been transferred.

#### Parameters

- **from\_part** – The part to transfer to.
- **to\_part** – The part to transfer from.
- **resource** – The name of the resource to transfer.
- **max\_amount** – The maximum amount of resource to transfer.

float **amount** ()

The amount of the resource that has been transferred.

bool **complete** ()

Whether the transfer has completed.

**enum struct ResourceFlowMode**

The way in which a resource flows between parts. See *Resources::flow\_mode()*.

**enumerator vessel**

The resource flows to any part in the vessel. For example, electric charge.

**enumerator stage**

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

**enumerator adjacent**

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

**enumerator none**

The resource does not flow. For example, solid fuel.

### 4.3.10 Node

**class Node**

Represents a maneuver node. Can be created using *Control::add\_node()*.

double **prograde** ()

void **set\_prograde** (double *value*)

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

double **normal** ()

void **set\_normal** (double *value*)

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

double **radial** ()

void **set\_radial** (double *value*)

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

double **delta\_v** ()

void **set\_delta\_v** (double *value*)

The delta-v of the maneuver node, in meters per second.

---

**Note:** Does not change when executing the maneuver node. See `Node::remaining_delta_v()`.

---

double **remaining\_delta\_v** ()

Gets the remaining delta-v of the maneuver node, in meters per second. Changes as the node is executed. This is equivalent to the delta-v reported in-game.

std::tuple<double, double, double> **burn\_vector** (*ReferenceFrame reference\_frame* = ReferenceFrame())

Returns the burn vector for the maneuver node.

#### Parameters

- **reference\_frame** – The reference frame that the returned vector is in. Defaults to `Vessel::orbital_reference_frame()`.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

---

**Note:** Does not change when executing the maneuver node. See `Node::remaining_burn_vector()`.

---

std::tuple<double, double, double> **remaining\_burn\_vector** (*ReferenceFrame reference\_frame* = ReferenceFrame())

Returns the remaining burn vector for the maneuver node.

#### Parameters

- **reference\_frame** – The reference frame that the returned vector is in. Defaults to `Vessel::orbital_reference_frame()`.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

---

**Note:** Changes as the maneuver node is executed. See `Node::burn_vector()`.

---

double **ut** ()

void **set\_ut** (double *value*)

The universal time at which the maneuver will occur, in seconds.

double **time\_to** ()

The time until the maneuver node will be encountered, in seconds.

Orbit **orbit** ()

The orbit that results from executing the maneuver node.

void **remove** ()

Removes the maneuver node.

ReferenceFrame **reference\_frame** ()

The reference frame that is fixed relative to the maneuver node's burn.

- The origin is at the position of the maneuver node.
- The y-axis points in the direction of the burn.
- The x-axis and z-axis point in arbitrary but fixed directions.

ReferenceFrame **orbital\_reference\_frame** ()

The reference frame that is fixed relative to the maneuver node, and orientated with the orbital prograde/normal/radial directions of the original orbit at the maneuver node's position.

- The origin is at the position of the maneuver node.
- The x-axis points in the orbital anti-radial direction of the original orbit, at the position of the maneuver node.
- The y-axis points in the orbital prograde direction of the original orbit, at the position of the maneuver node.
- The z-axis points in the orbital normal direction of the original orbit, at the position of the maneuver node.

std::tuple<double, double, double> **position** (ReferenceFrame *reference\_frame*)

The position vector of the maneuver node in the given reference frame.

#### Parameters

- **reference\_frame** – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

std::tuple<double, double, double> **direction** (ReferenceFrame *reference\_frame*)

The direction of the maneuver nodes burn.

#### Parameters

- **reference\_frame** – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

### 4.3.11 ReferenceFrame

**class ReferenceFrame**

Represents a reference frame for positions, rotations and velocities.

Contains:

- The position of the origin.
- The directions of the x, y and z axes.
- The linear velocity of the frame.
- The angular velocity of the frame.

---

**Note:** This class does not contain any properties or methods. It is only used as a parameter to other functions.

---

```
static ReferenceFrame create_relative (Client &connection, ReferenceFrame reference_frame,  
                                         std::tuple<double, double, double> position = (0.0, 0.0,  
                                         0.0), std::tuple<double, double, double, double> rota-  
                                         tion = (0.0, 0.0, 0.0, 1.0), std::tuple<double, double,  
                                         double> velocity = (0.0, 0.0, 0.0), std::tuple<double,  
                                         double, double> angular_velocity = (0.0, 0.0, 0.0))
```

Create a relative reference frame. This is a custom reference frame whose components offset the components of a parent reference frame.

#### Parameters

- **reference\_frame** – The parent reference frame on which to base this reference frame.
- **position** – The offset of the position of the origin, as a position vector. Defaults to (0, 0, 0)
- **rotation** – The rotation to apply to the parent frames rotation, as a quaternion of the form  $(x, y, z, w)$ . Defaults to (0, 0, 0, 1) (i.e. no rotation)
- **velocity** – The linear velocity to offset the parent frame by, as a vector pointing in the direction of travel, whose magnitude is the speed in meters per second. Defaults to (0, 0, 0).
- **angular\_velocity** – The angular velocity to offset the parent frame by, as a vector. This vector points in the direction of the axis of rotation, and its magnitude is the speed of the rotation in radians per second. Defaults to (0, 0, 0).

```
static ReferenceFrame create_hybrid (Client &connection, ReferenceFrame position, Refer-  
                                         enceFrame rotation = ReferenceFrame(), ReferenceFrame  
                                         velocity = ReferenceFrame(), ReferenceFrame angu-  
                                         lar_velocity = ReferenceFrame())
```

Create a hybrid reference frame. This is a custom reference frame whose components inherited from other reference frames.

#### Parameters

- **position** – The reference frame providing the position of the origin.
- **rotation** – The reference frame providing the rotation of the frame.
- **velocity** – The reference frame providing the linear velocity of the frame.

- **angular\_velocity** – The reference frame providing the angular velocity of the frame.

---

**Note:** The *position* reference frame is required but all other reference frames are optional. If omitted, they are set to the *position* reference frame.

---

### 4.3.12 AutoPilot

#### **class AutoPilot**

Provides basic auto-piloting utilities for a vessel. Created by calling `Vessel::auto_pilot()`.

---

**Note:** If a client engages the auto-pilot and then closes its connection to the server, the auto-pilot will be disengaged and its target reference frame, direction and roll reset to default.

---

**void engage()**  
Engage the auto-pilot.

**void disengage()**  
Disengage the auto-pilot.

**void wait()**  
Blocks until the vessel is pointing in the target direction and has the target roll (if set). Throws an exception if the auto-pilot has not been engaged.

**float error()**  
The error, in degrees, between the direction the ship has been asked to point in and the direction it is pointing in. Throws an exception if the auto-pilot has not been engaged and SAS is not enabled or is in stability assist mode.

**float pitch\_error()**  
The error, in degrees, between the vessels current and target pitch. Throws an exception if the auto-pilot has not been engaged.

**float heading\_error()**  
The error, in degrees, between the vessels current and target heading. Throws an exception if the auto-pilot has not been engaged.

**float roll\_error()**  
The error, in degrees, between the vessels current and target roll. Throws an exception if the auto-pilot has not been engaged or no target roll is set.

**ReferenceFrame reference\_frame()**

**void set\_reference\_frame** (*ReferenceFrame value*)  
The reference frame for the target direction (`AutoPilot::target_direction()`).

---

**Note:** An error will be thrown if this property is set to a reference frame that rotates with the vessel being controlled, as it is impossible to rotate the vessel in such a reference frame.

---

float **target\_pitch** ()

void **set\_target\_pitch** (float *value*)

The target pitch, in degrees, between -90° and +90°.

float **target\_heading** ()

void **set\_target\_heading** (float *value*)

The target heading, in degrees, between 0° and 360°.

float **target\_roll** ()

void **set\_target\_roll** (float *value*)

The target roll, in degrees. NaN if no target roll is set.

std::tuple<double, double, double> **target\_direction** ()

void **set\_target\_direction** (std::tuple<double, double, double> *value*)

Direction vector corresponding to the target pitch and heading. This is in the reference frame specified by *ReferenceFrame*.

void **target\_pitch\_and\_heading** (float *pitch*, float *heading*)

Set target pitch and heading angles.

#### Parameters

- **pitch** – Target pitch angle, in degrees between -90° and +90°.
- **heading** – Target heading angle, in degrees between 0° and 360°.

bool **sas** ()

void **set\_sas** (bool *value*)

The state of SAS.

---

**Note:** Equivalent to *Control::sas* ()

---

SASMode **sas\_mode** ()

void **set\_sas\_mode** (SASMode *value*)

The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

---

**Note:** Equivalent to *Control::sas\_mode* ()

---

double **roll\_threshold** ()



void **set\_roll\_threshold** (double *value*)

The threshold at which the autopilot will try to match the target roll angle, if any. Defaults to 5 degrees.

std::tuple<double, double, double> **stopping\_time** ()

void **set\_stopping\_time** (std::tuple<double, double, double> *value*)

The maximum amount of time that the vessel should need to come to a complete stop. This determines the maximum angular velocity of the vessel. A vector of three stopping times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 0.5 seconds for each axis.

std::tuple<double, double, double> **deceleration\_time** ()

void **set\_deceleration\_time** (std::tuple<double, double, double> *value*)

The time the vessel should take to come to a stop pointing in the target direction. This determines the angular acceleration used to decelerate the vessel. A vector of three times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 5 seconds for each axis.

std::tuple<double, double, double> **attenuation\_angle** ()

void **set\_attenuation\_angle** (std::tuple<double, double, double> *value*)

The angle at which the autopilot considers the vessel to be pointing close to the target. This determines the midpoint of the target velocity attenuation function. A vector of three angles, in degrees, one for each of the pitch, roll and yaw axes. Defaults to 1° for each axis.

bool **auto\_tune** ()

void **set\_auto\_tune** (bool *value*)

Whether the rotation rate controllers PID parameters should be automatically tuned using the vessels moment of inertia and available torque. Defaults to true. See `AutoPilot::time_to_peak()` and `AutoPilot::overshoot()`.

std::tuple<double, double, double> **time\_to\_peak** ()

void **set\_time\_to\_peak** (std::tuple<double, double, double> *value*)

The target time to peak used to autotune the PID controllers. A vector of three times, in seconds, for each of the pitch, roll and yaw axes. Defaults to 3 seconds for each axis.

std::tuple<double, double, double> **overshoot** ()

void **set\_overshoot** (std::tuple<double, double, double> *value*)

The target overshoot percentage used to autotune the PID controllers. A vector of three values, between 0 and 1, for each of the pitch, roll and yaw axes. Defaults to 0.01 for each axis.

`std::tuple<double, double, double> pitch_pid_gains ()`

void **set\_pitch\_pid\_gains** (std::tuple<double, double, double> *value*)  
Gains for the pitch PID controller.

---

**Note:** When *AutoPilot::auto\_tune()* is true, these values are updated automatically, which will overwrite any manual changes.

---

`std::tuple<double, double, double> roll_pid_gains ()`

void **set\_roll\_pid\_gains** (std::tuple<double, double, double> *value*)  
Gains for the roll PID controller.

---

**Note:** When *AutoPilot::auto\_tune()* is true, these values are updated automatically, which will overwrite any manual changes.

---

`std::tuple<double, double, double> yaw_pid_gains ()`

void **set\_yaw\_pid\_gains** (std::tuple<double, double, double> *value*)  
Gains for the yaw PID controller.

---

**Note:** When *AutoPilot::auto\_tune()* is true, these values are updated automatically, which will overwrite any manual changes.

---

### 4.3.13 Camera

#### **class Camera**

Controls the game's camera. Obtained by calling *camera()*.

*CameraMode* **mode** ()

void **set\_mode** (*CameraMode value*)  
The current mode of the camera.

float **pitch** ()

void **set\_pitch** (float *value*)  
The pitch of the camera, in degrees. A value between *Camera::min\_pitch()* and *Camera::max\_pitch()*

float **heading** ()

void **set\_heading** (float *value*)  
The heading of the camera, in degrees.

float **distance**()

void **set\_distance**(float *value*)

The distance from the camera to the subject, in meters. A value between *Camera::min\_distance()* and *Camera::max\_distance()*.

float **min\_pitch**()

The minimum pitch of the camera.

float **max\_pitch**()

The maximum pitch of the camera.

float **min\_distance**()

Minimum distance from the camera to the subject, in meters.

float **max\_distance**()

Maximum distance from the camera to the subject, in meters.

float **default\_distance**()

Default distance from the camera to the subject, in meters.

*CelestialBody* **focussed\_body**()

void **set\_focussed\_body**(*CelestialBody value*)

In map mode, the celestial body that the camera is focussed on. Returns NULL if the camera is not focussed on a celestial body. Returns an error if the camera is not in map mode.

*Vessel* **focussed\_vessel**()

void **set\_focussed\_vessel**(*Vessel value*)

In map mode, the vessel that the camera is focussed on. Returns NULL if the camera is not focussed on a vessel. Returns an error if the camera is not in map mode.

*Node* **focussed\_node**()

void **set\_focussed\_node**(*Node value*)

In map mode, the maneuver node that the camera is focussed on. Returns NULL if the camera is not focussed on a maneuver node. Returns an error if the camera is not in map mode.

**enum struct CameraMode**

See *Camera::mode()*.

**enumerator automatic**

The camera is showing the active vessel, in “auto” mode.

**enumerator free**

The camera is showing the active vessel, in “free” mode.

**enumerator chase**

The camera is showing the active vessel, in “chase” mode.

**enumerator locked**

The camera is showing the active vessel, in “locked” mode.

**enumerator orbital**

The camera is showing the active vessel, in “orbital” mode.

**enumerator iva**

The Intra-Vehicular Activity view is being shown.

**enumerator map**

The map view is being shown.

## 4.3.14 Waypoints

**class WaypointManager**

Waypoints are the location markers you can see on the map view showing you where contracts are targeted for. With this structure, you can obtain coordinate data for the locations of these waypoints. Obtained by calling `waypoint_manager()`.

`std::vector<Waypoint> waypoints()`

A list of all existing waypoints.

`Waypoint add_waypoint` (double *latitude*, double *longitude*, *CelestialBody* *body*, std::string *name*)

Creates a waypoint at the given position at ground level, and returns a *Waypoint* object that can be used to modify it.

**Parameters**

- **latitude** – Latitude of the waypoint.
- **longitude** – Longitude of the waypoint.
- **body** – Celestial body the waypoint is attached to.
- **name** – Name of the waypoint.

`Waypoint add_waypoint_at_altitude` (double *latitude*, double *longitude*, double *altitude*, *CelestialBody* *body*, std::string *name*)

Creates a waypoint at the given position and altitude, and returns a *Waypoint* object that can be used to modify it.

**Parameters**

- **latitude** – Latitude of the waypoint.
- **longitude** – Longitude of the waypoint.
- **altitude** – Altitude (above sea level) of the waypoint.
- **body** – Celestial body the waypoint is attached to.
- **name** – Name of the waypoint.

`std::map<std::string, int32_t> colors()`

An example map of known color - seed pairs. Any other integers may be used as seed.

`std::vector<std::string> icons()`

Returns all available icons (from “Game-Data/Squad/Contracts/Icons”).

**class Waypoint**

Represents a waypoint. Can be created using `WaypointManager::add_waypoint()`.

*CelestialBody* **body** ()

void **set\_body** (*CelestialBody value*)

The celestial body the waypoint is attached to.

std::string **name** ()

void **set\_name** (std::string *value*)

The name of the waypoint as it appears on the map and the contract.

int32\_t **color** ()

void **set\_color** (int32\_t *value*)

The seed of the icon color. See *WaypointManager::colors()* for example colors.

std::string **icon** ()

void **set\_icon** (std::string *value*)

The icon of the waypoint.

double **latitude** ()

void **set\_latitude** (double *value*)

The latitude of the waypoint.

double **longitude** ()

void **set\_longitude** (double *value*)

The longitude of the waypoint.

double **mean\_altitude** ()

void **set\_mean\_altitude** (double *value*)

The altitude of the waypoint above sea level, in meters.

double **surface\_altitude** ()

void **set\_surface\_altitude** (double *value*)

The altitude of the waypoint above the surface of the body or sea level, whichever is closer, in meters.

double **bedrock\_altitude** ()

void **set\_bedrock\_altitude** (double *value*)

The altitude of the waypoint above the surface of the body, in meters. When over water, this is the altitude above the sea floor.

bool **near\_surface** ()

true if the waypoint is near to the surface of a body.

bool **grounded** ()

true if the waypoint is attached to the ground.

`int32_t index ()`

The integer index of this waypoint within its cluster of sibling waypoints. In other words, when you have a cluster of waypoints called “Somewhere Alpha”, “Somewhere Beta” and “Somewhere Gamma”, the alpha site has index 0, the beta site has index 1 and the gamma site has index 2. When `Waypoint::clustered()` is `false`, this is zero.

`bool clustered ()`

`true` if this waypoint is part of a set of clustered waypoints with greek letter names appended (Alpha, Beta, Gamma, etc). If `true`, there is a one-to-one correspondence with the greek letter name and the `Waypoint::index()`.

`bool has_contract ()`

Whether the waypoint belongs to a contract.

`Contract contract ()`

The associated contract.

`void remove ()`

Removes the waypoint.

### 4.3.15 Contracts

**class ContractManager**

Contracts manager. Obtained by calling `waypoint_manager()`.

`std::set<std::string> types ()`

A list of all contract types.

`std::vector<Contract> all_contracts ()`

A list of all contracts.

`std::vector<Contract> active_contracts ()`

A list of all active contracts.

`std::vector<Contract> offered_contracts ()`

A list of all offered, but unaccepted, contracts.

`std::vector<Contract> completed_contracts ()`

A list of all completed contracts.

`std::vector<Contract> failed_contracts ()`

A list of all failed contracts.

**class Contract**

A contract. Can be accessed using `contract_manager()`.

`std::string type ()`

Type of the contract.

`std::string title ()`

Title of the contract.

`std::string description ()`

Description of the contract.

`std::string notes ()`

Notes for the contract.

---

```

std::string synopsis ()
    Synopsis for the contract.

std::vector<std::string> keywords ()
    Keywords for the contract.

ContractState state ()
    State of the contract.

bool seen ()
    Whether the contract has been seen.

bool read ()
    Whether the contract has been read.

bool active ()
    Whether the contract is active.

bool failed ()
    Whether the contract has been failed.

bool can_be_canceled ()
    Whether the contract can be canceled.

bool can_be_declined ()
    Whether the contract can be declined.

bool can_be_failed ()
    Whether the contract can be failed.

void accept ()
    Accept an offered contract.

void cancel ()
    Cancel an active contract.

void decline ()
    Decline an offered contract.

double funds_advance ()
    Funds received when accepting the contract.

double funds_completion ()
    Funds received on completion of the contract.

double funds_failure ()
    Funds lost if the contract is failed.

double reputation_completion ()
    Reputation gained on completion of the contract.

double reputation_failure ()
    Reputation lost if the contract is failed.

double science_completion ()
    Science gained on completion of the contract.

std::vector<ContractParameter> parameters ()
    Parameters for the contract.

enum struct ContractState
    The state of a contract. See Contract::state().

```

**enumerator active**

The contract is active.

**enumerator canceled**

The contract has been canceled.

**enumerator completed**

The contract has been completed.

**enumerator deadline\_expired**

The deadline for the contract has expired.

**enumerator declined**

The contract has been declined.

**enumerator failed**

The contract has been failed.

**enumerator generated**

The contract has been generated.

**enumerator offered**

The contract has been offered to the player.

**enumerator offer\_expired**

The contract was offered to the player, but the offer expired.

**enumerator withdrawn**

The contract has been withdrawn.

**class ContractParameter**

A contract parameter. See *Contract::parameters()*.

**std::string title()**

Title of the parameter.

**std::string notes()**

Notes for the parameter.

**std::vector<ContractParameter> children()**

Child contract parameters.

**bool completed()**

Whether the parameter has been completed.

**bool failed()**

Whether the parameter has been failed.

**bool optional()**

Whether the contract parameter is optional.

**double funds\_completion()**

Funds received on completion of the contract parameter.

**double funds\_failure()**

Funds lost if the contract parameter is failed.

**double reputation\_completion()**

Reputation gained on completion of the contract parameter.

**double reputation\_failure()**

Reputation lost if the contract parameter is failed.



double **science\_completion()**  
 Science gained on completion of the contract parameter.

## 4.3.16 Geometry Types

### Vectors

3-dimensional vectors are represented as a 3-tuple. For example:

```
#include <iostream>
#include <tuple>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client conn = krpc::connect();
    krpc::services::SpaceCenter sc(&conn);
    std::tuple<double, double, double>
    ↪ v = sc.active_vessel().flight().prograde();
    std::cout << std::get<0>(v) << " "
               << std::get<1>(v) << " "
               << std::get<2>(v) << std::endl;
}
```

### Quaternions

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

```
#include <iostream>
#include <tuple>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>

int main() {
    krpc::Client conn = krpc::connect();
    krpc::services::SpaceCenter sc(&conn);
    std::tuple<double, double, double, double>
    ↪ q = sc.active_vessel().flight().rotation();
    std::cout << std::get<0>(q) << " "
               << std::get<1>(q) << " "
               << std::get<2>(q) << " "
               << std::get<3>(q) << std::endl;
}
```

## 4.4 Drawing API

### 4.4.1 Drawing

**class Drawing** : public krpc::Service  
 Provides functionality for drawing objects in the flight scene.

**Drawing** (*krpc::Client \*client*)

Construct an instance of this service.

Line **add\_line** (*std::tuple<double, double, double> start, std::tuple<double, double, double> end, SpaceCenter::ReferenceFrame reference\_frame, bool visible = true*)  
Draw a line in the scene.

**Parameters**

- **start** – Position of the start of the line.
- **end** – Position of the end of the line.
- **reference\_frame** – Reference frame that the positions are in.
- **visible** – Whether the line is visible.

Line **add\_direction** (*std::tuple<double, double, double> direction, SpaceCenter::ReferenceFrame reference\_frame, float length = 10.0, bool visible = true*)  
Draw a direction vector in the scene, from the center of mass of the active vessel.

**Parameters**

- **direction** – Direction to draw the line in.
- **reference\_frame** – Reference frame that the direction is in.
- **length** – The length of the line.
- **visible** – Whether the line is visible.

Polygon **add\_polygon** (*std::vector<std::tuple<double, double, double>> vertices, SpaceCenter::ReferenceFrame reference\_frame, bool visible = true*)  
Draw a polygon in the scene, defined by a list of vertices.

**Parameters**

- **vertices** – Vertices of the polygon.
- **reference\_frame** – Reference frame that the vertices are in.
- **visible** – Whether the polygon is visible.

Text **add\_text** (*std::string text, SpaceCenter::ReferenceFrame reference\_frame, std::tuple<double, double, double> position, std::tuple<double, double, double, double> rotation, bool visible = true*)  
Draw text in the scene.

**Parameters**

- **text** – The string to draw.
- **reference\_frame** – Reference frame that the text position is in.
- **position** – Position of the text.
- **rotation** – Rotation of the text, as a quaternion.
- **visible** – Whether the text is visible.

void **clear** (*bool client\_only = false*)  
Remove all objects being drawn.

**Parameters**

- **client\_only** – If true, only remove objects created by the calling client.

### 4.4.2 Line

#### **class Line**

A line. Created using *add\_line()*.

std::tuple<double, double, double> **start** ()

void **set\_start** (std::tuple<double, double, double> *value*)  
Start position of the line.

std::tuple<double, double, double> **end** ()

void **set\_end** (std::tuple<double, double, double> *value*)  
End position of the line.

*SpaceCenter::ReferenceFrame* **reference\_frame** ()

void **set\_reference\_frame** (*SpaceCenter::ReferenceFrame value*)  
Reference frame for the positions of the object.

bool **visible** ()

void **set\_visible** (bool *value*)  
Whether the object is visible.

std::tuple<double, double, double> **color** ()

void **set\_color** (std::tuple<double, double, double> *value*)  
Set the color

std::string **material** ()

void **set\_material** (std::string *value*)  
Material used to render the object. Creates the material from a  
shader with the given name.

float **thickness** ()

void **set\_thickness** (float *value*)  
Set the thickness

void **remove** ()  
Remove the object.

### 4.4.3 Polygon

#### **class Polygon**

A polygon. Created using *add\_polygon()*.

std::vector<std::tuple<double, double, double>> **vertices** ()

void **set\_vertices** (std::vector<std::tuple<double, double, double>> *value*)  
Vertices for the polygon.

*SpaceCenter::ReferenceFrame* **reference\_frame** ()

void **set\_reference\_frame** (*SpaceCenter::ReferenceFrame value*)  
Reference frame for the positions of the object.

bool **visible** ()

void **set\_visible** (bool *value*)  
Whether the object is visible.

void **remove** ()  
Remove the object.

std::tuple<double, double, double> **color** ()

void **set\_color** (std::tuple<double, double, double> *value*)  
Set the color

std::string **material** ()

void **set\_material** (std::string *value*)  
Material used to render the object. Creates the material from a shader with the given name.

float **thickness** ()

void **set\_thickness** (float *value*)  
Set the thickness

#### 4.4.4 Text

**class Text**  
Text. Created using *add\_text* ().

std::tuple<double, double, double> **position** ()

void **set\_position** (std::tuple<double, double, double> *value*)  
Position of the text.

std::tuple<double, double, double, double> **rotation** ()

void **set\_rotation** (std::tuple<double, double, double, double> *value*)  
Rotation of the text as a quaternion.

*SpaceCenter::ReferenceFrame* **reference\_frame** ()

void **set\_reference\_frame** (*SpaceCenter::ReferenceFrame value*)  
Reference frame for the positions of the object.

bool **visible** ()

void **set\_visible** (bool *value*)  
Whether the object is visible.

```

void remove ()
    Remove the object.

std::string content ()

void set_content (std::string value)
    The text string

std::string font ()

void set_font (std::string value)
    Name of the font

static std::vector<std::string> available_fonts (Client &connection)
    A list of all available fonts.

int32_t size ()

void set_size (int32_t value)
    Font size.

float character_size ()

void set_character_size (float value)
    Character size.

UI::FontStyle style ()

void set_style (UI::FontStyle value)
    Font style.

std::tuple<double, double, double> color ()

void set_color (std::tuple<double, double, double> value)
    Set the color

std::string material ()

void set_material (std::string value)
    Material used to render the object.  Creates the material from a
    shader with the given name.

UI::TextAlignment alignment ()

void set_alignment (UI::TextAlignment value)
    Alignment.

float line_spacing ()

void set_line_spacing (float value)
    Line spacing.

UI::TextAnchor anchor ()

```

void **set\_anchor** (*UI::TextAnchor value*)  
Anchor.

## 4.5 InfernalRobotics API

Provides RPCs to interact with the [InfernalRobotics](#) mod. Provides the following classes:

### 4.5.1 InfernalRobotics

**class InfernalRobotics** : public `krpc::Service`  
This service provides functionality to interact with [InfernalRobotics](#).

**InfernalRobotics** (`krpc::Client *client`)  
Construct an instance of this service.

bool **available** ()  
Whether Infernal Robotics is installed.

std::vector<*ServoGroup*> **servo\_groups** (*SpaceCenter::Vessel vessel*)  
A list of all the servo groups in the given *vessel*.

#### Parameters

*ServoGroup* **servo\_group\_with\_name** (*SpaceCenter::Vessel vessel*, std::string *name*)  
Returns the servo group in the given *vessel* with the given *name*, or NULL if none exists. If multiple servo groups have the same name, only one of them is returned.

#### Parameters

- **vessel** – Vessel to check.
- **name** – Name of servo group to find.

*Servo* **servo\_with\_name** (*SpaceCenter::Vessel vessel*, std::string *name*)  
Returns the servo in the given *vessel* with the given *name* or NULL if none exists. If multiple servos have the same name, only one of them is returned.

#### Parameters

- **vessel** – Vessel to check.
- **name** – Name of the servo to find.

### 4.5.2 ServoGroup

**class ServoGroup**  
A group of servos, obtained by calling *servo\_groups()* or *servo\_group\_with\_name()*. Represents the “Servo Groups” in the InfernalRobotics UI.

std::string **name** ()

void **set\_name** (std::string *value*)  
 The name of the group.

std::string **forward\_key** ()

void **set\_forward\_key** (std::string *value*)  
 The key assigned to be the “forward” key for the group.

std::string **reverse\_key** ()

void **set\_reverse\_key** (std::string *value*)  
 The key assigned to be the “reverse” key for the group.

float **speed** ()

void **set\_speed** (float *value*)  
 The speed multiplier for the group.

bool **expanded** ()

void **set\_expanded** (bool *value*)  
 Whether the group is expanded in the InfernalRobotics UI.

std::vector<Servo> **servos** ()  
 The servos that are in the group.

Servo **servo\_with\_name** (std::string *name*)  
 Returns the servo with the given *name* from this group, or NULL if none exists.

#### Parameters

- **name** – Name of servo to find.

std::vector<SpaceCenter::Part> **parts** ()  
 The parts containing the servos in the group.

void **move\_right** ()  
 Moves all of the servos in the group to the right.

void **move\_left** ()  
 Moves all of the servos in the group to the left.

void **move\_center** ()  
 Moves all of the servos in the group to the center.

void **move\_next\_preset** ()  
 Moves all of the servos in the group to the next preset.

void **move\_prev\_preset** ()  
 Moves all of the servos in the group to the previous preset.

void **stop** ()  
 Stops the servos in the group.

### 4.5.3 Servo

**class Servo**  
 Represents a servo. Obtained using *ServoGroup::servos()*,

*ServoGroup::servo\_with\_name()* or  
*servo\_with\_name()*.

std::string **name**()

void **set\_name**(std::string *value*)

The name of the servo.

*SpaceCenter::Part* **part**()

The part containing the servo.

void **set\_highlight**(bool *value*)

Whether the servo should be highlighted in-game.

float **position**()

The position of the servo.

float **min\_config\_position**()

The minimum position of the servo, specified by the part configuration.

float **max\_config\_position**()

The maximum position of the servo, specified by the part configuration.

float **min\_position**()

void **set\_min\_position**(float *value*)

The minimum position of the servo, specified by the in-game tweak menu.

float **max\_position**()

void **set\_max\_position**(float *value*)

The maximum position of the servo, specified by the in-game tweak menu.

float **config\_speed**()

The speed multiplier of the servo, specified by the part configuration.

float **speed**()

void **set\_speed**(float *value*)

The speed multiplier of the servo, specified by the in-game tweak menu.

float **current\_speed**()

void **set\_current\_speed**(float *value*)

The current speed at which the servo is moving.

float **acceleration**()

void **set\_acceleration**(float *value*)

The current speed multiplier set in the UI.



bool **is\_moving**()  
Whether the servo is moving.

bool **is\_free\_moving**()  
Whether the servo is freely moving.

bool **is\_locked**()

void **set\_is\_locked**(bool *value*)  
Whether the servo is locked.

bool **is\_axis\_inverted**()

void **set\_is\_axis\_inverted**(bool *value*)  
Whether the servos axis is inverted.

void **move\_right**()  
Moves the servo to the right.

void **move\_left**()  
Moves the servo to the left.

void **move\_center**()  
Moves the servo to the center.

void **move\_next\_preset**()  
Moves the servo to the next preset.

void **move\_prev\_preset**()  
Moves the servo to the previous preset.

void **move\_to**(float *position*, float *speed*)  
Moves the servo to *position* and sets the speed multiplier to *speed*.

#### Parameters

- **position** – The position to move the servo to.
- **speed** – Speed multiplier for the movement.

void **stop**()  
Stops the servo.

### 4.5.4 Example

The following example gets the control group named “MyGroup”, prints out the names and positions of all of the servos in the group, then moves all of the servos to the right for 1 second.

```
#include <iostream>
#include <vector>
#include <thread>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>
#include <krpc/services/infernal_robotics.hpp>

using SpaceCenter = krpc::services::SpaceCenter;
using InfernalRobotics_
↳ krpc::services::InfernalRobotics;
```

```
int main() {
    auto conn_
    ↪= krpc::connect("InfernalRobotics Example");
    SpaceCenter space_center(&conn);
    InfernalRobotics infernal_robotics(&conn);

    InfernalRobotics::ServoGroup_
    ↪group = infernal_robotics.servo_group_with_
    ↪name(space_center.active_vessel(), "MyGroup");
    if (group == InfernalRobotics::ServoGroup())
        std::cout << "Group not found" << std::endl;

    std::vector<InfernalRobotics::Servo>
    ↪ servos = group.servos();
    for (auto servo : servos)
        std::cout << servo.
    ↪name() << " " << servo.position() << std::endl;

    group.move_right();
    std::this_
    ↪thread::sleep_for(std::chrono::seconds(1));
    group.stop();
}
```

## 4.6 Kerbal Alarm Clock API

Provides RPCs to interact with the [Kerbal Alarm Clock](#) mod. Provides the following classes:

### 4.6.1 KerbalAlarmClock

**class KerbalAlarmClock** : public `krpc::Service`  
This service provides functionality to interact with [Kerbal Alarm Clock](#).

**KerbalAlarmClock** (`krpc::Client *client`)  
Construct an instance of this service.

bool **available** ()  
Whether Kerbal Alarm Clock is available.

std::vector<*Alarm*> **alarms** ()  
A list of all the alarms.

*Alarm* **alarm\_with\_name** (std::string *name*)  
Get the alarm with the given *name*, or NULL if no alarms have that name. If more than one alarm has the name, only returns one of them.

#### Parameters

- **name** – Name of the alarm to search for.

std::vector<*Alarm*> **alarms\_with\_type** (*AlarmType* *type*)  
Get a list of alarms of the specified *type*.

**Parameters**

- **type** – Type of alarm to return.

*Alarm* **create\_alarm** (*AlarmType* type, std::string name, double ut)  
Create a new alarm and return it.

**Parameters**

- **type** – Type of the new alarm.
- **name** – Name of the new alarm.
- **ut** – Time at which the new alarm should trigger.

## 4.6.2 Alarm

**class Alarm**

Represents an alarm. Obtained by calling *alarms()*, *alarm\_with\_name()* or *alarms\_with\_type()*.

*AlarmAction* **action** ()

void **set\_action** (*AlarmAction* value)  
The action that the alarm triggers.

double **margin** ()

void **set\_margin** (double value)  
The number of seconds before the event that the alarm will fire.

double **time** ()

void **set\_time** (double value)  
The time at which the alarm will fire.

*AlarmType* **type** ()  
The type of the alarm.

std::string **id** ()  
The unique identifier for the alarm.

std::string **name** ()

void **set\_name** (std::string value)  
The short name of the alarm.

std::string **notes** ()

void **set\_notes** (std::string value)  
The long description of the alarm.

double **remaining** ()  
The number of seconds until the alarm will fire.

bool **repeat** ()

void **set\_repeat** (bool *value*)  
Whether the alarm will be repeated after it has fired.

double **repeat\_period** ()

void **set\_repeat\_period** (double *value*)  
The time delay to automatically create an alarm after it has fired.

*SpaceCenter::Vessel* **vessel** ()

void **set\_vessel** (*SpaceCenter::Vessel value*)  
The vessel that the alarm is attached to.

*SpaceCenter::CelestialBody* **xfer\_origin\_body** ()

void **set\_xfer\_origin\_body** (*SpaceCenter::CelestialBody value*)  
The celestial body the vessel is departing from.

*SpaceCenter::CelestialBody* **xfer\_target\_body** ()

void **set\_xfer\_target\_body** (*SpaceCenter::CelestialBody value*)  
The celestial body the vessel is arriving at.

void **remove** ()  
Removes the alarm.

### 4.6.3 AlarmType

**enum struct AlarmType**  
The type of an alarm.

**enumerator raw**  
An alarm for a specific date/time or a specific period in the future.

**enumerator maneuver**  
An alarm based on the next maneuver node on the current ships flight path. This node will be stored and can be restored when you come back to the ship.

**enumerator maneuver\_auto**  
See *AlarmType::maneuver*.

**enumerator apoapsis**  
An alarm for furthest part of the orbit from the planet.

**enumerator periapsis**  
An alarm for nearest part of the orbit from the planet.

**enumerator ascending\_node**  
Ascending node for the targeted object, or equatorial ascending node.

**enumerator descending\_node**  
Descending node for the targeted object, or equatorial descending node.

**enumerator closest**

An alarm based on the closest approach of this vessel to the targeted vessel, some number of orbits into the future.

**enumerator contract**

An alarm based on the expiry or deadline of contracts in career modes.

**enumerator contract\_auto**

See *AlarmType::contract*.

**enumerator crew**

An alarm that is attached to a crew member.

**enumerator distance**

An alarm that is triggered when a selected target comes within a chosen distance.

**enumerator earth\_time**

An alarm based on the time in the “Earth” alternative Universe (aka the Real World).

**enumerator launch\_rendevous**

An alarm that fires as your landed craft passes under the orbit of your target.

**enumerator soi\_change**

An alarm manually based on when the next SOI point is on the flight path or set to continually monitor the active flight path and add alarms as it detects SOI changes.

**enumerator soi\_change\_auto**

See *AlarmType::soi\_change*.

**enumerator transfer**

An alarm based on Interplanetary Transfer Phase Angles, i.e. when should I launch to planet X? Based on Kosmo Not’s post and used in Olex’s Calculator.

**enumerator transfer\_modelled**

See *AlarmType::transfer*.

## 4.6.4 AlarmAction

**enum struct AlarmAction**

The action performed by an alarm when it fires.

**enumerator do\_nothing**

Don’t do anything at all...

**enumerator do\_nothing\_delete\_when\_passed**

Don’t do anything, and delete the alarm.

**enumerator kill\_warp**

Drop out of time warp.

**enumerator kill\_warp\_only**

Drop out of time warp.

**enumerator message\_only**

Display a message.

**enumerator pause\_game**

Pause the game.

## 4.6.5 Example

The following example creates a new alarm for the active vessel. The alarm is set to trigger after 10 seconds have passed, and display a message.

```
#include <iostream>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>
#include <krpc/services/kerbal_alarm_clock.hpp>

using KerbalAlarmClock_
↳= krpc::services::KerbalAlarmClock;

int main() {
    krpc::Client conn_
↳= krpc::connect("Kerbal Alarm Clock Example");
    krpc::services::SpaceCenter sc(&conn);
    KerbalAlarmClock kac(&conn);

    auto alarm = kac.
↳create_alarm(KerbalAlarmClock::AlarmType::raw,
               "My New Alarm",
               sc.ut()+10);

    alarm.set_notes("10 seconds_
↳have now passed since the alarm was created.");
    alarm.set_
↳action(KerbalAlarmClock::AlarmAction::message_
↳only);
}
```

## 4.7 RemoteTech API

Provides RPCs to interact with the [RemoteTech](#) mod. Provides the following classes:

### 4.7.1 RemoteTech

**class RemoteTech** : public krpc::Service

This service provides functionality to interact with [RemoteTech](#).

**RemoteTech** (krpc::Client \*client)

Construct an instance of this service.

bool **available** ()

Whether RemoteTech is installed.

std::vector<std::string> **ground\_stations** ()

The names of the ground stations.

*Antenna* **antenna** (*SpaceCenter::Part part*)

Get the antenna object for a particular part.

#### Parameters

*Comms* **comms** (*SpaceCenter::Vessel vessel*)

Get a communications object, representing the communication capability of a particular vessel.

#### Parameters

### 4.7.2 Comms

#### **class Comms**

Communications for a vessel.

*SpaceCenter::Vessel* **vessel** ()

Get the vessel.

bool **has\_local\_control** ()

Whether the vessel can be controlled locally.

bool **has\_flight\_computer** ()

Whether the vessel has a flight computer on board.

bool **has\_connection** ()

Whether the vessel has any connection.

bool **has\_connection\_to\_ground\_station** ()

Whether the vessel has a connection to a ground station.

double **signal\_delay** ()

The shortest signal delay to the vessel, in seconds.

double **signal\_delay\_to\_ground\_station** ()

The signal delay between the vessel and the closest ground station, in seconds.

double **signal\_delay\_to\_vessel** (*SpaceCenter::Vessel other*)

The signal delay between the this vessel and another vessel, in seconds.

#### Parameters

std::vector<*Antenna*> **antennas** ()

The antennas for this vessel.

### 4.7.3 Antenna

#### **class Antenna**

A RemoteTech antenna. Obtained by calling *Comms::antennas* () or *antenna* ().

*SpaceCenter::Part* **part** ()

Get the part containing this antenna.

bool **has\_connection** ()

Whether the antenna has a connection.

*Target* **target** ()

void **set\_target** (*Target value*)

The object that the antenna is targetting. This property can be used to set the target to *Target::none* or *Target::active\_vessel*. To set the target to a celestial body, ground station or vessel see *Antenna::target\_body()*, *Antenna::target\_ground\_station()* and *Antenna::target\_vessel()*.

*SpaceCenter::CelestialBody* **target\_body** ()

void **set\_target\_body** (*SpaceCenter::CelestialBody value*)

The celestial body the antenna is targetting.

std::string **target\_ground\_station** ()

void **set\_target\_ground\_station** (std::string *value*)

The ground station the antenna is targetting.

*SpaceCenter::Vessel* **target\_vessel** ()

void **set\_target\_vessel** (*SpaceCenter::Vessel value*)

The vessel the antenna is targetting.

**enum struct Target**

The type of object an antenna is targetting. See *Antenna::target()*.

**enumerator active\_vessel**

The active vessel.

**enumerator celestial\_body**

A celestial body.

**enumerator ground\_station**

A ground station.

**enumerator vessel**

A specific vessel.

**enumerator none**

No target.

## 4.7.4 Example

The following example sets the target of a dish on the active vessel then prints out the signal delay to the active vessel.

```
#include <iostream>
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>
#include <krpc/services/remote_tech.hpp>

int main() {
    krpc::Client_
    ↪ conn = krpc::connect("RemoteTech Example");
```



```

krpc::services::SpaceCenter space_center(&conn);
krpc::services::RemoteTech remote_tech(&conn);
auto vessel = space_center.active_vessel();

// Set a dish target
auto part = vessel.
↳parts().with_title("Reflectron KR-7").front();
auto antenna = remote_tech.antenna(part);
antenna.
↳set_target_body(space_center.bodies()["Jool"]);

// Get info about the vessels communications
auto comms = remote_tech.comms(vessel);
std::cout << "Signal_
↳delay = " << comms.signal_delay() << std::endl;
}

```

## 4.8 User Interface API

### 4.8.1 UI

**class UI** : public krpc::Service

Provides functionality for drawing and interacting with in-game user interface elements.

**UI** (krpc::Client \*client)

Construct an instance of this service.

*Canvas* **stock\_canvas** ()

The stock UI canvas.

*Canvas* **add\_canvas** ()

Add a new canvas.

---

**Note:** If you want to add UI elements to KSPs stock UI canvas, use `stock_canvas()`.

---

**void message** (std::string content, float duration = 1.0, MessagePosition position = static\_cast<MessagePosition>(1))  
Display a message on the screen.

#### Parameters

- **content** – Message content.
- **duration** – Duration before the message disappears, in seconds.
- **position** – Position to display the message.

---

**Note:** The message appears just like a stock message, for example quicksave or quickload messages.

---

**void clear** (bool client\_only = false)

Remove all user interface elements.

**Parameters**

- **client\_only** – If true, only remove objects created by the calling client.

**enum struct MessagePosition**

Message position.

**enumerator top\_left**

Top left.

**enumerator top\_center**

Top center.

**enumerator top\_right**

Top right.

**enumerator bottom\_center**

Bottom center.

## 4.8.2 Canvas

**class Canvas**

A canvas for user interface elements. See *stock\_canvas()* and *add\_canvas()*.

*RectTransform* **rect\_transform()**

The rect transform for the canvas.

bool **visible()**

void **set\_visible**(bool *value*)

Whether the UI object is visible.

*Panel* **add\_panel**(bool *visible* = true)

Create a new container for user interface elements.

**Parameters**

- **visible** – Whether the panel is visible.

*Text* **add\_text**(std::string *content*, bool *visible* = true)

Add text to the canvas.

**Parameters**

- **content** – The text.
- **visible** – Whether the text is visible.

*InputField* **add\_input\_field**(bool *visible* = true)

Add an input field to the canvas.

**Parameters**

- **visible** – Whether the input field is visible.

*Button* **add\_button**(std::string *content*, bool *visible* = true)

Add a button to the canvas.

**Parameters**

- **content** – The label for the button.

- **visible** – Whether the button is visible.

void **remove** ()  
Remove the UI object.

### 4.8.3 Panel

**class Panel**  
A container for user interface elements. See  
*Canvas::add\_panel()*.

*RectTransform* **rect\_transform** ()  
The rect transform for the panel.

bool **visible** ()

void **set\_visible** (bool *value*)  
Whether the UI object is visible.

*Panel* **add\_panel** (bool *visible* = true)  
Create a panel within this panel.

#### Parameters

- **visible** – Whether the new panel is visible.

*Text* **add\_text** (std::string *content*, bool *visible* = true)  
Add text to the panel.

#### Parameters

- **content** – The text.
- **visible** – Whether the text is visible.

*InputField* **add\_input\_field** (bool *visible* = true)  
Add an input field to the panel.

#### Parameters

- **visible** – Whether the input field is visible.

*Button* **add\_button** (std::string *content*, bool *visible* = true)  
Add a button to the panel.

#### Parameters

- **content** – The label for the button.
- **visible** – Whether the button is visible.

void **remove** ()  
Remove the UI object.

### 4.8.4 Text

**class Text**  
A text label. See *Panel::add\_text()*.

*RectTransform* **rect\_transform** ()  
The rect transform for the text.

**bool visible()**

void **set\_visible**(bool *value*)  
Whether the UI object is visible.

std::string **content()**

void **set\_content**(std::string *value*)  
The text string

std::string **font()**

void **set\_font**(std::string *value*)  
Name of the font

std::vector<std::string> **available\_fonts()**  
A list of all available fonts.

int32\_t **size()**

void **set\_size**(int32\_t *value*)  
Font size.

*FontStyle* **style()**

void **set\_style**(*FontStyle* *value*)  
Font style.

std::tuple<double, double, double> **color()**

void **set\_color**(std::tuple<double, double, double> *value*)  
Set the color

*TextAnchor* **alignment()**

void **set\_alignment**(*TextAnchor* *value*)  
Alignment.

float **line\_spacing()**

void **set\_line\_spacing**(float *value*)  
Line spacing.

void **remove()**  
Remove the UI object.

**enum struct FontStyle**  
Font style.

**enumerator normal**  
Normal.

**enumerator bold**  
Bold.

**enumerator italic**  
Italic.

**enumerator bold\_and\_italic**  
Bold and italic.

**enum struct TextAlignment**  
Text alignment.

**enumerator left**  
Left aligned.

**enumerator right**  
Right aligned.

**enumerator center**  
Center aligned.

**enum struct TextAnchor**  
Text alignment.

**enumerator lower\_center**  
Lower center.

**enumerator lower\_left**  
Lower left.

**enumerator lower\_right**  
Lower right.

**enumerator middle\_center**  
Middle center.

**enumerator middle\_left**  
Middle left.

**enumerator middle\_right**  
Middle right.

**enumerator upper\_center**  
Upper center.

**enumerator upper\_left**  
Upper left.

**enumerator upper\_right**  
Upper right.

### 4.8.5 Button

**class Button**  
A text label. See *Panel::add\_button()*.

*RectTransform* **rect\_transform()**  
The rect transform for the text.

bool **visible()**

void **set\_visible**(bool *value*)  
Whether the UI object is visible.

*Text* **text** ()

The text for the button.

bool **clicked** ()

void **set\_clicked** (bool *value*)

Whether the button has been clicked.

---

**Note:** This property is set to true when the user clicks the button. A client script should reset the property to false in order to detect subsequent button presses.

---

void **remove** ()

Remove the UI object.

## 4.8.6 InputField

**class InputField**

An input field. See *Panel::add\_input\_field()*.

*RectTransform* **rect\_transform** ()

The rect transform for the input field.

bool **visible** ()

void **set\_visible** (bool *value*)

Whether the UI object is visible.

std::string **value** ()

void **set\_value** (std::string *value*)

The value of the input field.

*Text* **text** ()

The text component of the input field.

---

**Note:** Use *InputField::value()* to get and set the value in the field. This object can be used to alter the style of the input field's text.

---

bool **changed** ()

void **set\_changed** (bool *value*)

Whether the input field has been changed.

---

**Note:** This property is set to true when the user modifies the value of the input field. A client script should reset the property to false in order to detect subsequent changes.

---

void **remove** ()

Remove the UI object.

### 4.8.7 Rect Transform

#### **class RectTransform**

A Unity engine Rect Transform for a UI object. See the [Unity manual](#) for more details.

`std::tuple<double, double> position ()`

`void set_position (std::tuple<double, double> value)`

Position of the rectangles pivot point relative to the anchors.

`std::tuple<double, double, double> local_position ()`

`void set_local_position (std::tuple<double, double, double> value)`

Position of the rectangles pivot point relative to the anchors.

`std::tuple<double, double> size ()`

`void set_size (std::tuple<double, double> value)`

Width and height of the rectangle.

`std::tuple<double, double> upper_right ()`

`void set_upper_right (std::tuple<double, double> value)`

Position of the rectangles upper right corner relative to the anchors.

`std::tuple<double, double> lower_left ()`

`void set_lower_left (std::tuple<double, double> value)`

Position of the rectangles lower left corner relative to the anchors.

`void set_anchor (std::tuple<double, double> value)`

Set the minimum and maximum anchor points as a fraction of the size of the parent rectangle.

`std::tuple<double, double> anchor_max ()`

`void set_anchor_max (std::tuple<double, double> value)`

The anchor point for the lower left corner of the rectangle defined as a fraction of the size of the parent rectangle.

`std::tuple<double, double> anchor_min ()`

`void set_anchor_min (std::tuple<double, double> value)`

The anchor point for the upper right corner of the rectangle defined as a fraction of the size of the parent rectangle.

`std::tuple<double, double> pivot ()`

`void set_pivot (std::tuple<double, double> value)`

Location of the pivot point around which the rectangle rotates, defined as a fraction of the size of the rectangle itself.

`std::tuple<double, double, double, double> rotation ()`

void **set\_rotation** (std::tuple<double, double, double, double> *value*)

Rotation, as a quaternion, of the object around its pivot point.

std::tuple<double, double, double> **scale** ( )

void **set\_scale** (std::tuple<double, double, double> *value*)

Scale factor applied to the object in the x, y and z dimensions.



## 5.1 Java Client

This client provides a Java API for interacting with a kRPC server. A jar containing the `krpc.client` package can be [downloaded from GitHub](#). It requires Java version 1.8.

### 5.1.1 Using the Library

The kRPC client library depends on the [protobuf](#) and [javatuples](#) libraries. A prebuilt jar for protobuf is available via [Maven](#). Note that you need protobuf version 3. Version 2 is not compatible with kRPC.

The following example program connects to the server, queries it for its version and prints it out:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KRPC;

import java.io.IOException;

public class Basic {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        KRPC krpc = KRPC.newInstance(connection);
        System.out.println("Connected to kRPC version " + krpc.getStatus().getVersion());
        connection.close();
    }
}
```

To compile this program using `javac` on the command line, save the source as `Example.java` and run the following:

```
javac -cp krpc-java-0.4.0.jar:protobuf-java-3.4.0.jar:javatuples-1.2.jar Example.java
```

You may need to change the paths to the JAR files.

### 5.1.2 Connecting to the Server

To connect to a server, call `Connection.newInstance()` which returns a connection object. All interaction with the server is done via this object. When constructed without any arguments, it will connect to the local machine on the default port numbers. You can specify different connection settings, and also a descriptive name for the connection, as follows:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KRPC;

import java.io.IOException;

public class Connecting {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("Remote example", "my.domain.name",
↪ 1000, 1001);
        System.out.println(KRPC.newInstance(connection).getStatus().getVersion());
        connection.close();
    }
}
```

### 5.1.3 Calling Remote Procedures

The kRPC server provides *procedures* that a client can run. These procedures are arranged in groups called *services* to keep things organized. The functionality for the services are defined in the package `krpc.client.services`. For example, all of the functionality provided by the `SpaceCenter` service is contained in the class `krpc.client.services.SpaceCenter`.

To interact with a service, you must first instantiate it. You can then call its methods and properties to invoke remote procedures. The following example demonstrates how to do this. It instantiates the `SpaceCenter` service and calls `krpc.client.services.SpaceCenter.SpaceCenter.getActiveVessel()` to get an object representing the active vessel (of type `krpc.client.services.SpaceCenter.Vessel`). It sets the name of the vessel and then prints out its altitude:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;

import java.io.IOException;

public class RemoteProcedures {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("Vessel Name");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        System.out.println(vessel.getName());
        connection.close();
    }
}
```

### 5.1.4 Streaming Data from the Server

A common use case for kRPC is to continuously extract data from the game. The naive approach to do this would be to repeatedly call a remote procedure, such as in the following which repeatedly prints the position of the active vessel:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KRPC;
```

```

import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.ReferenceFrame;
import krpc.client.services.SpaceCenter.Vessel;

import java.io.IOException;

public class Streaming1 {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        ReferenceFrame refframe = vessel.getOrbit().getBody().getReferenceFrame();
        while (true) {
            System.out.println(vessel.position(refframe));
        }
    }
}

```

This approach requires significant communication overhead as request/response messages are repeatedly sent between the client and server. kRPC provides a more efficient mechanism to achieve this, called *streams*.

A stream repeatedly executes a procedure on the server (with a fixed set of argument values) and sends the result to the client. It only requires a single message to be sent to the server to establish the stream, which will then continuously send data to the client until the stream is closed.

The following example does the same thing as above using streams:

```

import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.Stream;
import krpc.client.StreamException;
import krpc.client.services.KRPC;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.ReferenceFrame;
import krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Triplet;

import java.io.IOException;

public class Streaming2 {
    public static void main(String[] args) throws IOException, RPCException,
↳StreamException {
        Connection connection = Connection.newInstance();
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        ReferenceFrame refframe = vessel.getOrbit().getBody().getReferenceFrame();
        Stream<Triplet<Double,Double,Double>> vesselStream = connection.addStream(vessel,
↳"position", refframe);
        while (true) {
            System.out.println(vesselStream.get());
        }
    }
}

```

It calls *Connection.addStream* once at the start of the program to create the stream, and then repeatedly prints the position returned by the stream. The stream is automatically closed when the client disconnects.

A stream can be created for any method call by calling *Connection.addStream* and passing it information about

which method to stream. The example above passes a remote object, the name of the method to call, followed by the arguments to pass to the method (if any). `Connection.addStream` returns a stream object of type `Stream`. The most recent value of the stream can be obtained by calling `Stream.get`. A stream can be stopped and removed from the server by calling `Stream.remove` on the stream object. All of a clients streams are automatically stopped when it disconnects.

### 5.1.5 Synchronizing with Stream Updates

A common use case for kRPC is to wait until the value returned by a method or attribute changes, and then take some action. kRPC provides two mechanisms to do this efficiently: *condition variables* and *callbacks*.

#### Condition Variables

Each stream has a condition variable associated with it, that is notified whenever the value of the stream changes. These can be used to block the current thread of execution until the value of the stream changes.

The following example waits until the abort button is pressed in game, by waiting for the value of `krpc.client.services.SpaceCenter.Control.getAbort()` to change to true:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.Stream;
import krpc.client.StreamException;
import krpc.client.services.KRPC;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Control;

import java.io.IOException;

public class ConditionVariables {
    public static void main(String[] args) throws IOException, RPCException,
↳StreamException {
        Connection connection = Connection.newInstance();
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Control control = spaceCenter.getActiveVessel().getControl();
        Stream<Boolean> abort = connection.addStream(control, "getAbort");
        synchronized (abort.getCondition()) {
            while (!abort.get()) {
                abort.waitForUpdate();
            }
        }
    }
}
```

This code creates a stream, acquires a lock on the streams condition variable (by using a `synchronized` block) and then repeatedly checks the value of `getAbort`. It leaves the loop when it changes to true.

The body of the loop calls `waitForUpdate` on the stream, which causes the program to block until the value changes. This prevents the loop from ‘spinning’ and so it does not consume processing resources whilst waiting.

---

**Note:** The stream does not start receiving updates until the first call to `waitForUpdate`. This means that the example code will not miss any updates to the streams value, as it will have already locked the condition variable before the first stream update is received.

---

## Callbacks

Streams allow you to register callback functions that are called whenever the value of the stream changes. Callback functions should take a single argument, which is the new value of the stream, and should return nothing.

For example the following program registers two callbacks that are invoked when the value of `krpc.client.services.SpaceCenter.Control.getAbort()` changes:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.Stream;
import krpc.client.StreamException;
import krpc.client.services.KRPC;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Control;

import java.io.IOException;

public class Callbacks {
    public static void main(String[] args) throws IOException, RPCException,
↳StreamException {
        Connection connection = Connection.newInstance();
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Control control = spaceCenter.getActiveVessel().getControl();
        Stream<Boolean> abort = connection.addStream(control, "getAbort");
        abort.addCallback(
            (Boolean x) -> {
                System.out.println("Abort 1 called with a value of " + x);
            });
        abort.addCallback(
            (Boolean x) -> {
                System.out.println("Abort 2 called with a value of " + x);
            });
        abort.start();

        // Keep the program running...
        while (true) {
        }
    }
}
```

---

**Note:** When a stream is created it does not start receiving updates until `start` is called. This is implicitly called when accessing the value of a stream, but as this example does not do this an explicit call to `start` is required.

---



---

**Note:** The callbacks are registered before the call to `start` so that stream updates are not missed.

---



---

**Note:** The callback function may be called from a different thread to that which created the stream. Any changes to shared state must therefore be protected with appropriate synchronization.

---

## 5.1.6 Custom Events

Some procedures return event objects of type *Event*. These allow you to wait until an event occurs, by calling *Event.waitFor*. Under the hood, these are implemented using streams and condition variables.

Custom events can also be created. An expression API allows you to create code that runs on the server and these can be used to build a custom event. For example, the following creates the expression `MeanAltitude > 1000` and then creates an event that will be triggered when the expression returns true:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.Event;
import krpc.client.StreamException;
import krpc.client.services.KRPC;
import krpc.client.services.KRPC.Expression;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Flight;
import krpc.schema.KRPC.ProcedureCall;

import java.io.IOException;

public class CustomEvent {
    public static void main(String[] args) throws IOException, RPCException,
↳StreamException {
        Connection connection = Connection.newInstance();
        KRPC krpc = KRPC.newInstance(connection);
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Flight flight = spaceCenter.getActiveVessel().flight(null);

        // Get the remote procedure call as a message object,
        // so it can be passed to the server
        ProcedureCall meanAltitude = connection.getCall(flight, "getMeanAltitude");

        // Create an expression on the server
        Expression expr = Expression.greaterThan(connection,
            Expression.call(connection, meanAltitude),
            Expression.constantDouble(connection, 1000));

        Event event = krpc.addEvent(expr);
        synchronized (event.getCondition()) {
            event.waitFor();
            System.out.println("Altitude reached 1000m");
        }
    }
}
```

## 5.1.7 Client API Reference

### class **Connection**

A connection to the kRPC server. All interaction with kRPC is performed via an instance of this class.

static *Connection* **newInstance** ()

static *Connection* **newInstance** (String name)

static *Connection* **newInstance** (String name, String address)

```
static Connection newInstance (String name, String address, int rpcPort, int streamPort)
static Connection newInstance (String name, java.net.InetAddress address)
static Connection newInstance (String name, java.net.InetAddress address, int rpcPort, int
                                streamPort)
```

Create a connection to the server, using the given connection details.

**Parameters**

- **name** (*String*) – A descriptive name for the connection. This is passed to the server and appears in the in-game server window.
- **address** (*String*) – The address of the server to connect to. Can either be a hostname, an IP address as a string or a `java.net.InetAddress` object. Defaults to 127.0.0.1.
- **rpc\_port** (*int*) – The port number of the RPC Server. Defaults to 50000. This should match the RPC port number of the server you want to connect to.
- **stream\_port** (*int*) – The port number of the Stream Server. Defaults to 50001. This should match the stream port number of the server you want to connect to.

```
Stream<T> addStream (Class<?> clazz, String method, Object... args)
```

Create a stream for a static method call to the given class.

```
Stream<T> addStream (RemoteObject instance, String method, Object... args)
```

Create a stream for a method call to the given remote object.

```
krpc.schema.KRPC.ProcedureCall getCall (Class<?> clazz, String method, Object... args)
```

Returns a procedure call message for the given static method call. This allows descriptions of procedure calls to be passed to the server, for example when constructing custom events. See *Custom Events*.

```
krpc.schema.KRPC.ProcedureCall getCall (RemoteObject instance, String method, Object... args)
```

Returns a procedure call message for the given method call. This allows descriptions of procedure calls to be passed to the server, for example when constructing custom events. See *Custom Events*.

```
void close ()
```

Close the connection.

```
class Stream<T>
```

This class represents a stream. See *Streaming Data from the Server*.

Stream objects implement `hashCode` and `equals` such that two stream objects are equal if they are bound to the same stream on the server.

```
void start ()
```

```
void startAndWait ()
```

```
T get ()
```

Returns the most recent value for the stream. If executing the remote procedure for the stream throws an exception, calling this method will rethrow the exception. Raises a `StreamException` if no update has been received from the server.

If the stream has not been started this method calls `startAndWait()` to start the stream and wait until at least one update has been received.

```
Object getCondition ()
```

A condition variable that is notified (using `notifyAll`) whenever the value of the stream changes.

```
void waitForUpdate ()
```

```
void waitForUpdateWithTimeout (double timeout)
```

These methods block until the value of the stream changes or the operation times out.

The streams condition variable must be locked before calling this method.

If *timeout* is specified it is the timeout in seconds for the operation.

If the stream has not been started this method calls `start` to start the stream (without waiting for at least one update to be received).

void **addCallback** (java.util.function.Consumer<T> *callback*)

Adds a callback function that is invoked whenever the value of the stream changes. The callback function should take one argument, which is passed the new value of the stream.

---

**Note:** The callback function may be called from a different thread to that which created the stream. Any changes to shared state must therefore be protected with appropriate synchronization.

---

void **remove** ()

Remove the stream from the server.

class **Event**

This class represents an event. See *Custom Events*. It is wrapper around a `Stream` that indicates when the event occurs.

Event objects implement `hashCode` and `equals` such that two event objects are equal if they are bound to the same underlying stream on the server.

void **start** ()

Starts the event. When an event is created, it will not receive updates from the server until this method is called.

Object **getCondition** ()

The condition variable that is notified (using `notifyAll`) whenever the event occurs.

void **waitFor** ()

void **waitForWithTimeout** (double *timeout*)

These methods block until the event occurs or the operation times out.

The events condition variable must be locked before calling this method.

If *timeout* is specified it is the timeout in seconds for the operation.

If the event has not been started this method calls `start ()` to start the underlying stream.

void **addCallback** (java.lang.Callable *callback*)

Adds a callback function that is invoked whenever the event occurs. The callback function should be a function that takes zero arguments.

void **remove** ()

Removes the event from the server.

Stream<Boolean> **getStream** ()

Returns the underlying stream for the event.

abstract class **RemoteObject**

The abstract base class for all remote objects.

## 5.2 KRPC API

### 5.2.1 KRPC

None None None None



public class **KRPC**

Main kRPC service, used by clients to interact with basic server functionality.

byte[] **getClientID** ()

Returns the identifier for the current client.

String **getClientName** ()

Returns the name of the current client. This is an empty string if the client has no name.

java.util.List<org.javatuples.Triplet<Byte[], String, String>> **getClients** ()

A list of RPC clients that are currently connected to the server. Each entry in the list is a clients identifier, name and address.

krpc.schema.KRPC.Status **getStatus** ()

Returns some information about the server, such as the version.

krpc.schema.KRPC.Services **getServices** ()

Returns information on all services, procedures, classes, properties etc. provided by the server. Can be used by client libraries to automatically create functionality such as stubs.

GameScene **getCurrentGameScene** ()

Get the current game scene.

boolean **getPaused** ()

void **setPaused** (boolean *value*)

Whether the game is paused.

public enum **GameScene**

The game scene. See *getCurrentGameScene* ().

public GameScene **SPACE\_CENTER**

The game scene showing the Kerbal Space Center buildings.

public GameScene **FLIGHT**

The game scene showing a vessel in flight (or on the launchpad/runway).

public GameScene **TRACKING\_STATION**

The tracking station.

public GameScene **EDITOR\_VAB**

The Vehicle Assembly Building.

public GameScene **EDITOR\_SPH**

The Space Plane Hangar.

public class **InvalidOperationException**

A method call was made to a method that is invalid given the current state of the object.

public class **ArgumentException**

A method was invoked where at least one of the passed arguments does not meet the parameter specification of the method.

public class **ArgumentNullException**

A null reference was passed to a method that does not accept it as a valid argument.

public class **ArgumentOutOfRangeException**

The value of an argument is outside the allowable range of values as defined by the invoked method.

## 5.2.2 Expressions

public class **Expression**

A server side expression.

static *Expression* **constantDouble** (*Connection connection*, double *value*)

A constant value of type double.

**Parameters**

- **value** (*double*) –

static *Expression* **constantFloat** (*Connection connection*, float *value*)

A constant value of type float.

**Parameters**

- **value** (*float*) –

static *Expression* **constantInt** (*Connection connection*, int *value*)

A constant value of type int.

**Parameters**

- **value** (*int*) –

static *Expression* **constantString** (*Connection connection*, *String* *value*)

A constant value of type string.

**Parameters**

- **value** (*String*) –

static *Expression* **call** (*Connection connection*, *krpc.schema.KRPC.ProcedureCall call*)

An RPC call.

**Parameters**

- **call** (*krpc.schema.KRPC.ProcedureCall*) –

static *Expression* **equal** (*Connection connection*, *Expression arg0*, *Expression arg1*)

Equality comparison.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

static *Expression* **notEqual** (*Connection connection*, *Expression arg0*, *Expression arg1*)

Inequality comparison.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

static *Expression* **greaterThan** (*Connection connection*, *Expression arg0*, *Expression arg1*)

Greater than numerical comparison.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

static Expression **greaterThanOrEqualTo** (*Connection connection, Expression arg0, Expression arg1*)  
Greater than or equal numerical comparison.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

static Expression **lessThan** (*Connection connection, Expression arg0, Expression arg1*)  
Less than numerical comparison.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

static Expression **lessThanOrEqualTo** (*Connection connection, Expression arg0, Expression arg1*)  
Less than or equal numerical comparison.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

static Expression **and** (*Connection connection, Expression arg0, Expression arg1*)  
Boolean and operator.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

static Expression **or** (*Connection connection, Expression arg0, Expression arg1*)  
Boolean or operator.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

static Expression **exclusiveOr** (*Connection connection, Expression arg0, Expression arg1*)  
Boolean exclusive-or operator.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

static Expression **not** (*Connection connection, Expression arg*)  
Boolean negation operator.

**Parameters**

- **arg** (Expression) –

static Expression **add** (*Connection connection, Expression arg0, Expression arg1*)  
Numerical addition.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **subtract** (*Connection connection*, *Expression arg0*, *Expression arg1*)  
Numerical subtraction.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

static *Expression* **multiply** (*Connection connection*, *Expression arg0*, *Expression arg1*)  
Numerical multiplication.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

static *Expression* **divide** (*Connection connection*, *Expression arg0*, *Expression arg1*)  
Numerical division.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

static *Expression* **modulo** (*Connection connection*, *Expression arg0*, *Expression arg1*)  
Numerical modulo operator.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

**Returns** The remainder of arg0 divided by arg1

static *Expression* **power** (*Connection connection*, *Expression arg0*, *Expression arg1*)  
Numerical power operator.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

**Returns** arg0 raised to the power of arg1

static *Expression* **leftShift** (*Connection connection*, *Expression arg0*, *Expression arg1*)  
Bitwise left shift.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

static *Expression* **rightShift** (*Connection connection*, *Expression arg0*, *Expression arg1*)  
Bitwise right shift.

**Parameters**

- **arg0** (*Expression*) –
- **arg1** (*Expression*) –

static *Expression* **toDouble** (*Connection connection*, *Expression arg*)  
Convert to a double type.

**Parameters**

- **arg** (Expression) –

static *Expression* **toFloat** (*Connection connection*, *Expression arg*)  
Convert to a float type.

**Parameters**

- **arg** (Expression) –

static *Expression* **toInt** (*Connection connection*, *Expression arg*)  
Convert to an int type.

**Parameters**

- **arg** (Expression) –

## 5.3 SpaceCenter API

### 5.3.1 SpaceCenter

public class **SpaceCenter**

Provides functionality to interact with Kerbal Space Program. This includes controlling the active vessel, managing its resources, planning maneuver nodes and auto-piloting.

*Vessel* **getActiveVessel** ()

void **setActiveVessel** (*Vessel value*)  
The currently active vessel.

java.util.List<*Vessel*> **getVessels** ()  
A list of all the vessels in the game.

java.util.Map<*String*, *CelestialBody*> **getBodies** ()  
A dictionary of all celestial bodies (planets, moons, etc.) in the game, keyed by the name of the body.

*CelestialBody* **getTargetBody** ()

void **setTargetBody** (*CelestialBody value*)  
The currently targeted celestial body.

*Vessel* **getTargetVessel** ()

void **setTargetVessel** (*Vessel value*)  
The currently targeted vessel.

*DockingPort* **getTargetDockingPort** ()

void **setTargetDockingPort** (*DockingPort value*)  
The currently targeted docking port.

void **clearTarget** ()  
Clears the current target.

java.util.List<*String*> **launchableVessels** (*String craftDirectory*)  
Returns a list of vessels from the given *craftDirectory* that can be launched.

**Parameters**

- **craftDirectory** (*String*) – Name of the directory in the current saves “Ships” directory. For example “VAB” or “SPH”.

void **launchVessel** (*String* craftDirectory, *String* name, *String* launchSite)

Launch a vessel.

**Parameters**

- **craftDirectory** (*String*) – Name of the directory in the current saves “Ships” directory, that contains the craft file. For example "VAB" or "SPH".
- **name** (*String*) – Name of the vessel to launch. This is the name of the “.craft” file in the save directory, without the “.craft” file extension.
- **launchSite** (*String*) – Name of the launch site. For example "LaunchPad" or "Runway".

void **launchVesselFromVAB** (*String* name)

Launch a new vessel from the VAB onto the launchpad.

**Parameters**

- **name** (*String*) – Name of the vessel to launch.

---

**Note:** This is equivalent to calling *launchVessel(String, String, String)* with the craft directory set to “VAB” and the launch site set to “LaunchPad”.

---

void **launchVesselFromSPH** (*String* name)

Launch a new vessel from the SPH onto the runway.

**Parameters**

- **name** (*String*) – Name of the vessel to launch.

---

**Note:** This is equivalent to calling *launchVessel(String, String, String)* with the craft directory set to “SPH” and the launch site set to “Runway”.

---

void **save** (*String* name)

Save the game with a given name. This will create a save file called *name.sfs* in the folder of the current save game.

**Parameters**

- **name** (*String*) –

void **load** (*String* name)

Load the game with the given name. This will create a load a save file called *name.sfs* from the folder of the current save game.

**Parameters**

- **name** (*String*) –

void **quicksave** ()

Save a quicksave.

---

**Note:** This is the same as calling *save(String)* with the name “quicksave”.

---

void **quickload** ()

Load a quicksave.

---

**Note:** This is the same as calling `load(String)` with the name “quicksave”.

---

boolean **getUIVisible** ()

void **setUIVisible** (boolean *value*)

Whether the UI is visible.

boolean **getNavball** ()

void **setNavball** (boolean *value*)

Whether the navball is visible.

double **getUT** ()

The current universal time in seconds.

double **getG** ()

The value of the [gravitational constant](#)  $G$  in  $N(m/kg)^2$ .

float **getWarpRate** ()

The current warp rate. This is the rate at which time is passing for either on-rails or physical time warp. For example, a value of 10 means time is passing 10x faster than normal. Returns 1 if time warp is not active.

float **getWarpFactor** ()

The current warp factor. This is the index of the rate at which time is passing for either regular “on-rails” or physical time warp. Returns 0 if time warp is not active. When in on-rails time warp, this is equal to `getRailsWarpFactor()`, and in physics time warp, this is equal to `getPhysicsWarpFactor()`.

int **getRailsWarpFactor** ()

void **setRailsWarpFactor** (int *value*)

The time warp rate, using regular “on-rails” time warp. A value between 0 and 7 inclusive. 0 means no time warp. Returns 0 if physical time warp is active.

If requested time warp factor cannot be set, it will be set to the next lowest possible value. For example, if the vessel is too close to a planet. See [the KSP wiki](#) for details.

int **getPhysicsWarpFactor** ()

void **setPhysicsWarpFactor** (int *value*)

The physical time warp rate. A value between 0 and 3 inclusive. 0 means no time warp. Returns 0 if regular “on-rails” time warp is active.

boolean **canRailsWarpAt** (int *factor*)

Returns `true` if regular “on-rails” time warp can be used, at the specified warp *factor*. The maximum time warp rate is limited by various things, including how close the active vessel is to a planet. See [the KSP wiki](#) for details.

#### Parameters

- **factor** (*int*) – The warp factor to check.

int **getMaximumRailsWarpFactor** ()

The current maximum regular “on-rails” warp factor that can be set. A value between 0 and 7 inclusive. See [the KSP wiki](#) for details.

void **warpTo** (double *ut*, float *maxRailsRate*, float *maxPhysicsRate*)

Uses time acceleration to warp forward to a time in the future, specified by universal time *ut*. This call blocks until the desired time is reached. Uses regular “on-rails” or physical time warp as appropriate. For example, physical time warp is used when the active vessel is traveling through an atmosphere. When

using regular “on-rails” time warp, the warp rate is limited by *maxRailsRate*, and when using physical time warp, the warp rate is limited by *maxPhysicsRate*.

#### Parameters

- **ut** (*double*) – The universal time to warp to, in seconds.
- **maxRailsRate** (*float*) – The maximum warp rate in regular “on-rails” time warp.
- **maxPhysicsRate** (*float*) – The maximum warp rate in physical time warp.

**Returns** When the time warp is complete.

org.javatuples.Triplet<Double, Double, Double> **transformPosition** (org.javatuples.Triplet<Double, Double, Double> *position*, ReferenceFrame *from*, ReferenceFrame *to*)

Converts a position from one reference frame to another.

#### Parameters

- **position** (*org.javatuples.Triplet<Double, Double, Double>*) – Position, as a vector, in reference frame *from*.
- **from** (ReferenceFrame) – The reference frame that the position is in.
- **to** (ReferenceFrame) – The reference frame to convert the position to.

**Returns** The corresponding position, as a vector, in reference frame *to*.

org.javatuples.Triplet<Double, Double, Double> **transformDirection** (org.javatuples.Triplet<Double, Double, Double> *direction*, ReferenceFrame *from*, ReferenceFrame *to*)

Converts a direction from one reference frame to another.

#### Parameters

- **direction** (*org.javatuples.Triplet<Double, Double, Double>*) – Direction, as a vector, in reference frame *from*.
- **from** (ReferenceFrame) – The reference frame that the direction is in.
- **to** (ReferenceFrame) – The reference frame to convert the direction to.

**Returns** The corresponding direction, as a vector, in reference frame *to*.

org.javatuples.Quartet<Double, Double, Double, Double> **transformRotation** (org.javatuples.Quartet<Double, Double, Double, Double> *rotation*, ReferenceFrame *from*, ReferenceFrame *to*)

Converts a rotation from one reference frame to another.

#### Parameters

- **rotation** (*org.javatuples.Quartet<Double, Double, Double, Double>*) – Rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *from*.
- **from** (ReferenceFrame) – The reference frame that the rotation is in.
- **to** (ReferenceFrame) – The reference frame to convert the rotation to.



**Returns** The corresponding rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *to*.

`org.javatuples.Triplet<Double, Double, Double> transformVelocity` (`org.javatuples.Triplet<Double, Double, Double> position`, `org.javatuples.Triplet<Double, Double, Double> velocity`, `ReferenceFrame from`, `ReferenceFrame to`)

Converts a velocity (acting at the specified position) from one reference frame to another. The position is required to take the relative angular velocity of the reference frames into account.

#### Parameters

- **position** (`org.javatuples.Triplet<Double, Double, Double>`) – Position, as a vector, in reference frame *from*.
- **velocity** (`org.javatuples.Triplet<Double, Double, Double>`) – Velocity, as a vector that points in the direction of travel and whose magnitude is the speed in meters per second, in reference frame *from*.
- **from** (`ReferenceFrame`) – The reference frame that the position and velocity are in.
- **to** (`ReferenceFrame`) – The reference frame to convert the velocity to.

**Returns** The corresponding velocity, as a vector, in reference frame *to*.

`boolean getFARAvailable ()`

Whether [Ferram Aerospace Research](#) is installed.

`WarpMode getWarpMode ()`

The current time warp mode. Returns `WarpMode.NONE` if time warp is not active, `WarpMode.RAILS` if regular “on-rails” time warp is active, or `WarpMode.PHYSICS` if physical time warp is active.

`Camera getCamera ()`

An object that can be used to control the camera.

`WaypointManager getWaypointManager ()`

The waypoint manager.

`ContractManager getContractManager ()`

The contract manager.

`public enum WarpMode`

The time warp mode. Returned by `WarpMode`

`public WarpMode RAILS`

Time warp is active, and in regular “on-rails” mode.

`public WarpMode PHYSICS`

Time warp is active, and in physical time warp mode.

`public WarpMode NONE`

Time warp is not active.

### 5.3.2 Vessel

`public class Vessel`

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources. Created using `getActiveVessel ()` or `getVessels ()`.

`String getName ()`

void **setName** (*String value*)

The name of the vessel.

*VesselType* **getType** ()

void **setType** (*VesselType value*)

The type of the vessel.

*VesselSituation* **getSituation** ()

The situation the vessel is in.

boolean **getRecoverable** ()

Whether the vessel is recoverable.

void **recover** ()

Recover the vessel.

double **getMET** ()

The mission elapsed time in seconds.

*String* **getBiome** ()

The name of the biome the vessel is currently in.

*Flight* **flight** (*ReferenceFrame referenceFrame*)

Returns a *Flight* object that can be used to get flight telemetry for the vessel, in the specified reference frame.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – Reference frame. Defaults to the vessel's surface reference frame (*Vessel.getSurfaceReferenceFrame()*).

---

**Note:** When this is called with no arguments, the vessel's surface reference frame is used. This reference frame moves with the vessel, therefore velocities and speeds returned by the flight object will be zero. See the *reference frames tutorial* for examples of getting the orbital and surface speeds of a vessel.

---

*Orbit* **getOrbit** ()

The current orbit of the vessel.

*Control* **getControl** ()

Returns a *Control* object that can be used to manipulate the vessel's control inputs. For example, its pitch/yaw/roll controls, RCS and thrust.

*Comms* **getComms** ()

Returns a *Comms* object that can be used to interact with CommNet for this vessel.

*AutoPilot* **getAutoPilot** ()

An *AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

*Resources* **getResources** ()

A *Resources* object, that can be used to get information about resources stored in the vessel.

*Resources* **resourcesInDecoupleStage** (*int stage*, *boolean cumulative*)

Returns a *Resources* object, that can be used to get information about resources stored in a given *stage*.

#### Parameters

- **stage** (*int*) – Get resources for parts that are decoupled in this stage.
- **cumulative** (*boolean*) – When *false*, returns the resources for parts decoupled in just the given stage. When *true* returns the resources decoupled in the given stage and all subsequent stages combined.

---

**Note:** For details on stage numbering, see the discussion on *Staging*.

---

**Parts** `getParts ()`

A *Parts* object, that can used to interact with the parts that make up this vessel.

`float getMass ()`

The total mass of the vessel, including resources, in kg.

`float getDryMass ()`

The total mass of the vessel, excluding resources, in kg.

`float getThrust ()`

The total thrust currently being produced by the vessel's engines, in Newtons. This is computed by summing *Engine.getThrust ()* for every engine in the vessel.

`float getAvailableThrust ()`

Gets the total available thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *Engine.getAvailableThrust ()* for every active engine in the vessel.

`float getMaxThrust ()`

The total maximum thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *Engine.getMaxThrust ()* for every active engine.

`float getMaxVacuumThrust ()`

The total maximum thrust that can be produced by the vessel's active engines when the vessel is in a vacuum, in Newtons. This is computed by summing *Engine.getMaxVacuumThrust ()* for every active engine.

`float getSpecificImpulse ()`

The combined specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

`float getVacuumSpecificImpulse ()`

The combined vacuum specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

`float getKerbinSeaLevelSpecificImpulse ()`

The combined specific impulse of all active engines at sea level on Kerbin, in seconds. This is computed using the formula [described here](#).

`org.javatuples.Triplet<Double, Double, Double> getMomentOfInertia ()`

The moment of inertia of the vessel around its center of mass in  $kg.m^2$ . The inertia values in the returned 3-tuple are around the pitch, roll and yaw directions respectively. This corresponds to the vessels reference frame (*ReferenceFrame*).

`java.util.List<Double> getInertiaTensor ()`

The inertia tensor of the vessel around its center of mass, in the vessels reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

`org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> getAvailableTorque ()`

The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gimbaled engines and aerodynamic control surfaces. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

`org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> getAvailableReactionWheelTorque ()`

The maximum torque that the currently active and powered reaction wheels can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

`org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> getAv`

The maximum torque that the currently active RCS thrusters can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

`org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> getAv`

The maximum torque that the currently active and gimbaled engines can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

`org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> getAv`

The maximum torque that the aerodynamic control surfaces can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

`org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> getAv`

The maximum torque that parts (excluding reaction wheels, gimbaled engines, RCS and control surfaces) can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

*ReferenceFrame* **getReferenceFrame** ()

The reference frame that is fixed relative to the vessel, and orientated with the vessel.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel.
- The x-axis points out to the right of the vessel.
- The y-axis points in the forward direction of the vessel.
- The z-axis points out of the bottom off the vessel.

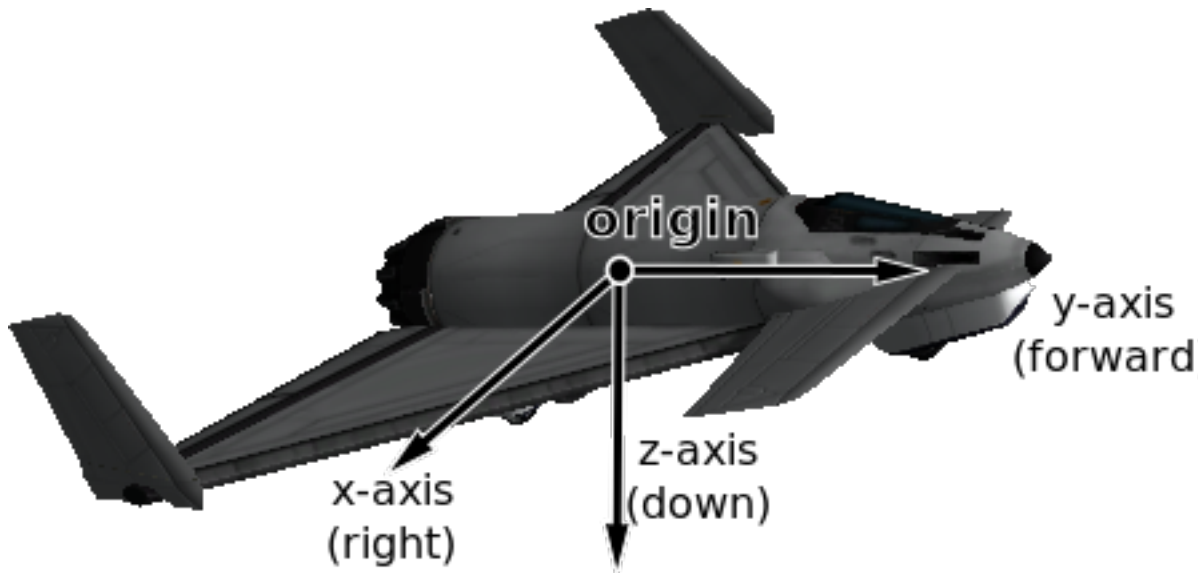


Fig. 5.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

*ReferenceFrame* **getOrbitalReferenceFrame** ()

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel.

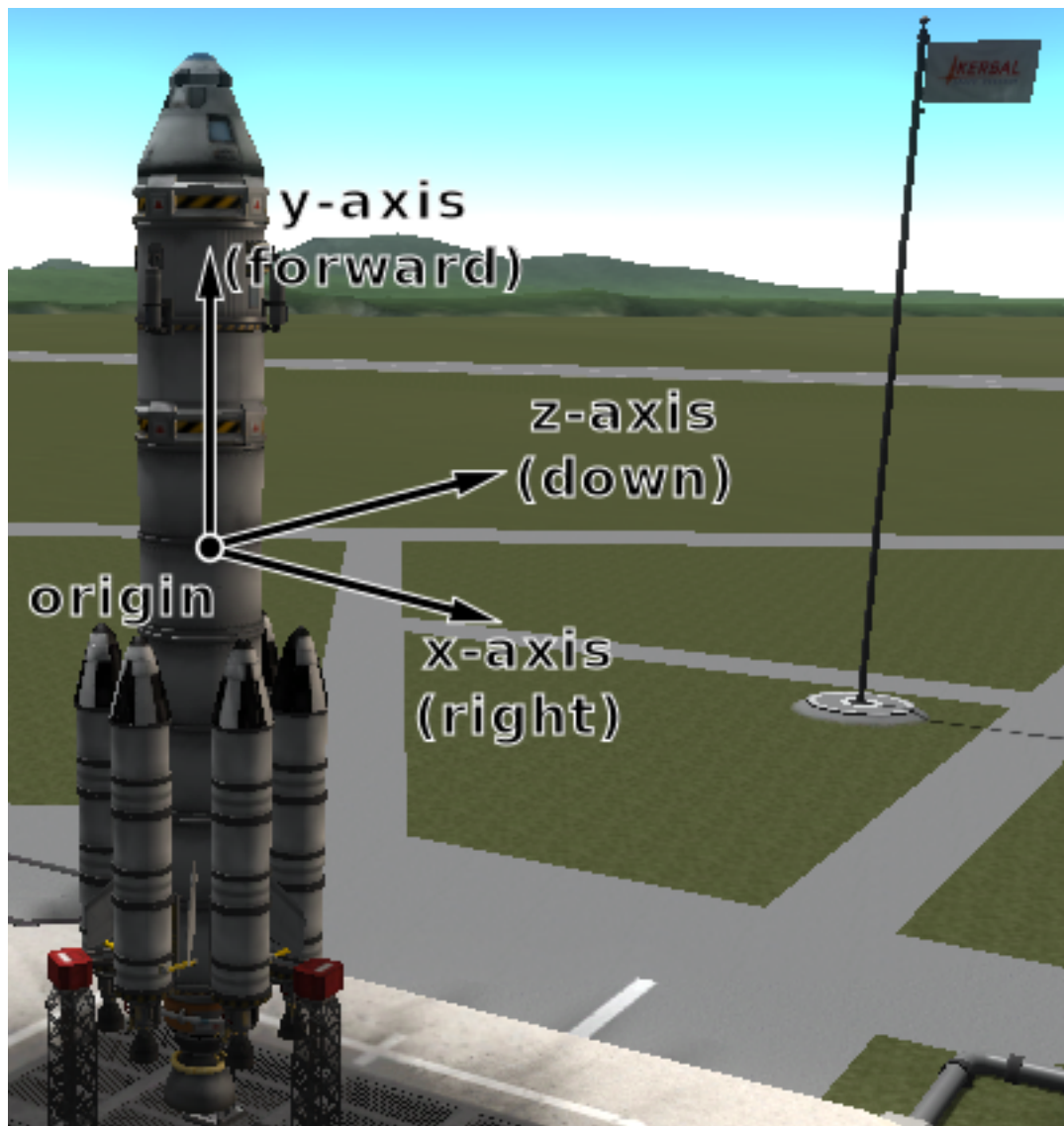


Fig. 5.2: Vessel reference frame origin and axes for the Kerbal-X rocket

- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

---

**Note:** Be careful not to confuse this with ‘orbit’ mode on the navball.

---

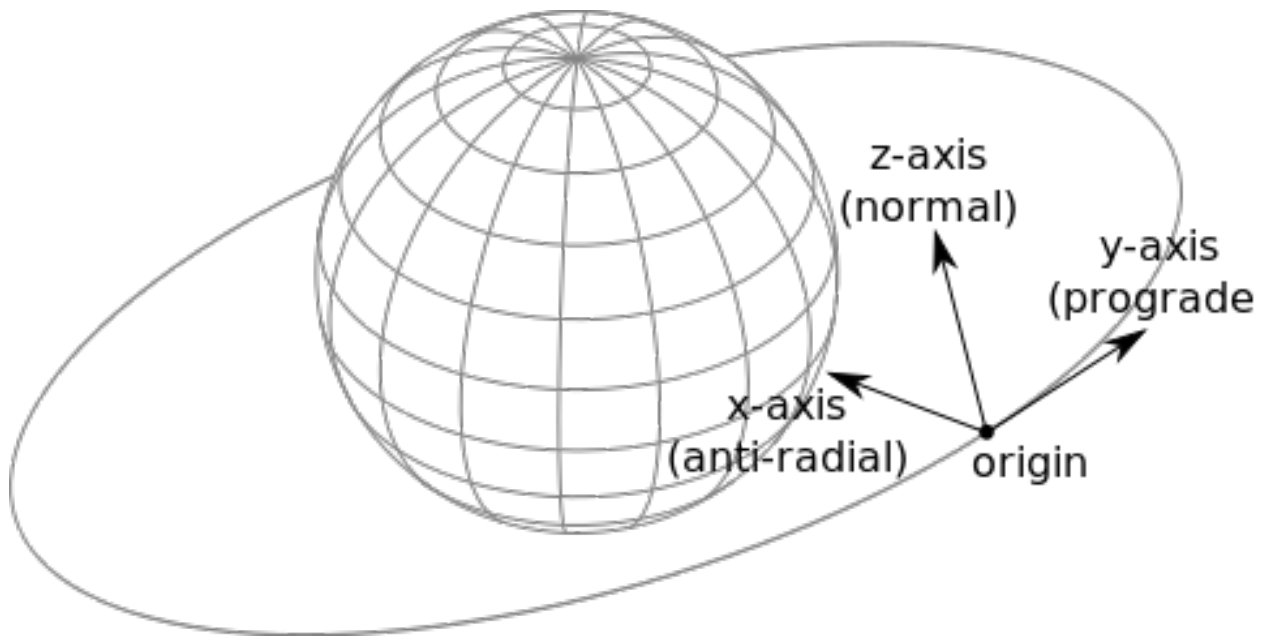


Fig. 5.3: Vessel orbital reference frame origin and axes

*ReferenceFrame* **getSurfaceReferenceFrame** ()

The reference frame that is fixed relative to the vessel, and orientated with the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the north and up directions on the surface of the body.
- The x-axis points in the [zenith](#) direction (upwards, normal to the body being orbited, from the center of the body towards the center of mass of the vessel).
- The y-axis points northwards towards the [astronomical horizon](#) (north, and tangential to the surface of the body – the direction in which a compass would point when on the surface).
- The z-axis points eastwards towards the [astronomical horizon](#) (east, and tangential to the surface of the body – east on a compass when on the surface).

---

**Note:** Be careful not to confuse this with ‘surface’ mode on the navball.

---

*ReferenceFrame* **getSurfaceVelocityReferenceFrame** ()

The reference frame that is fixed relative to the vessel, and orientated with the velocity vector of the vessel relative to the surface of the body being orbited.

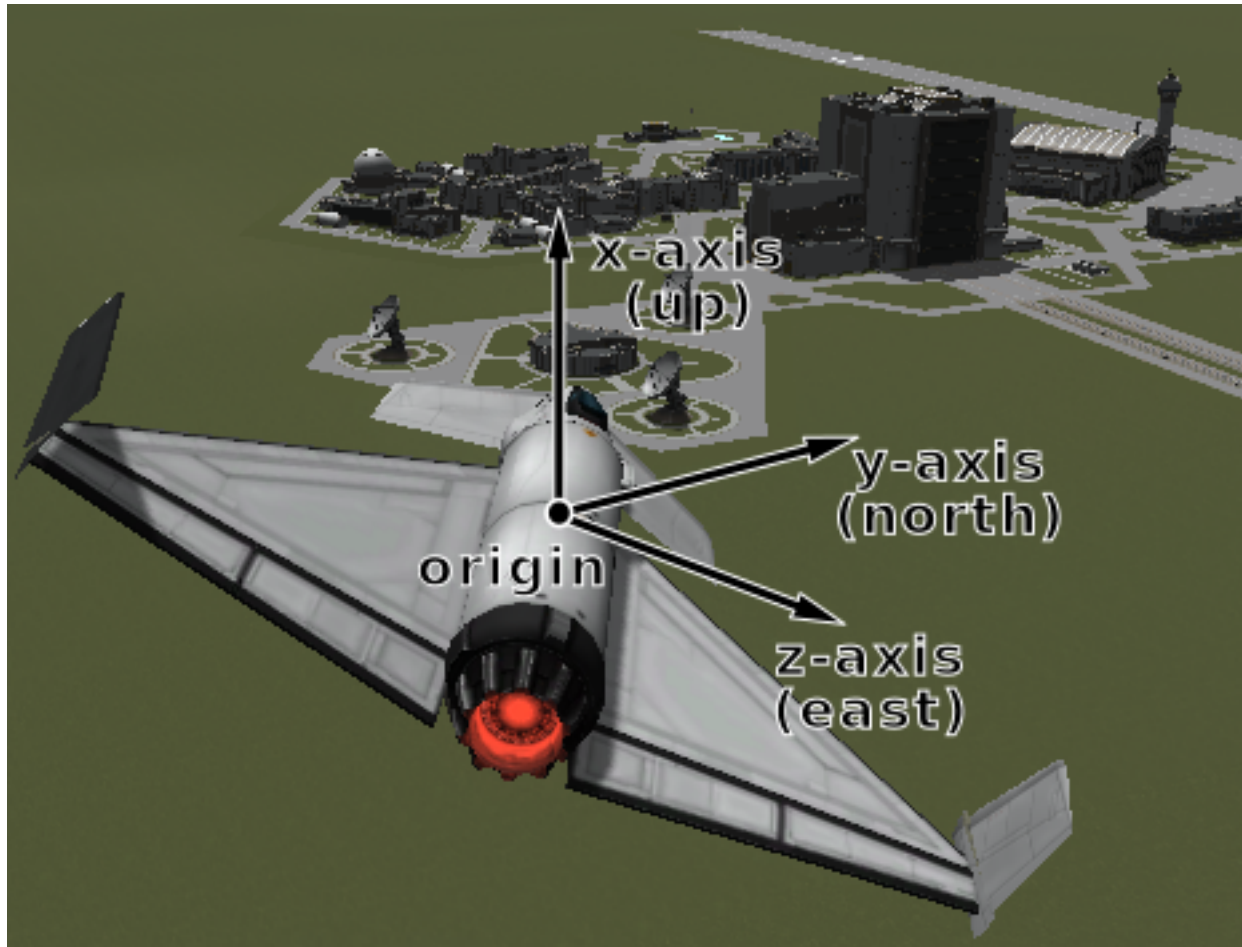


Fig. 5.4: Vessel surface reference frame origin and axes



- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel's velocity vector.
- The y-axis points in the direction of the vessel's velocity vector, relative to the surface of the body being orbited.
- The z-axis is in the plane of the [astronomical horizon](#).
- The x-axis is orthogonal to the other two axes.

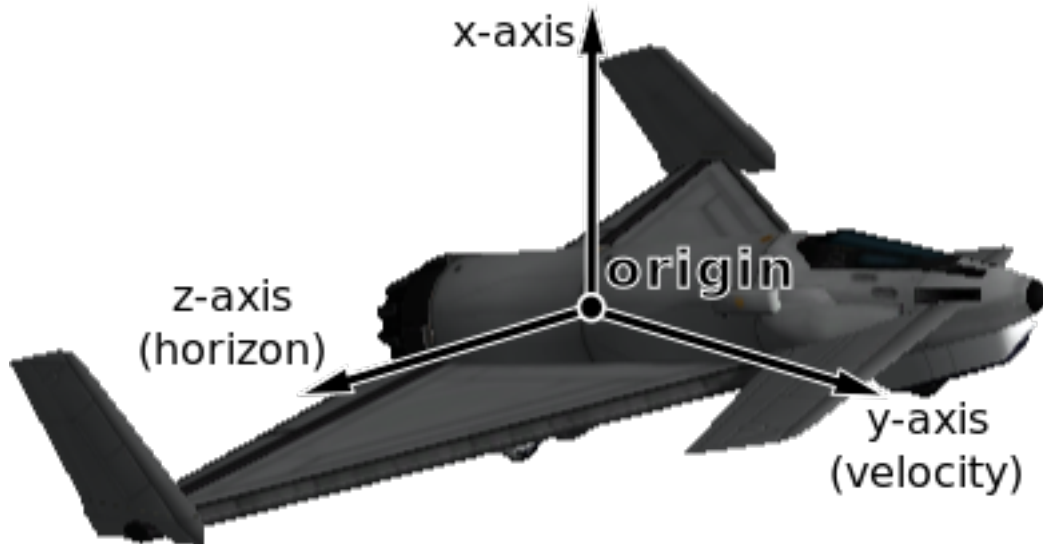


Fig. 5.5: Vessel surface velocity reference frame origin and axes

`org.javatuples.Triplet<Double, Double, Double> position (ReferenceFrame referenceFrame)`

The position of the center of mass of the vessel, in the given reference frame.

#### Parameters

- **referenceFrame** (`ReferenceFrame`) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

`org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> boundingBox (ReferenceFrame referenceFrame)`

The axis-aligned bounding box of the vessel in the given reference frame.

#### Parameters

- **referenceFrame** (`ReferenceFrame`) – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

`org.javatuples.Triplet<Double, Double, Double> velocity (ReferenceFrame referenceFrame)`

The velocity of the center of mass of the vessel, in the given reference frame.

#### Parameters



- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

org.javatuples.[Quartet](#)<[Double](#), [Double](#), [Double](#), [Double](#)> **rotation** (*ReferenceFrame* *referenceFrame*)

The rotation of the vessel, in the given reference frame.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

org.javatuples.[Triplet](#)<[Double](#), [Double](#), [Double](#)> **direction** (*ReferenceFrame* *referenceFrame*)

The direction in which the vessel is pointing, in the given reference frame.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

org.javatuples.[Triplet](#)<[Double](#), [Double](#), [Double](#)> **angularVelocity** (*ReferenceFrame* *referenceFrame*)

The angular velocity of the vessel, in the given reference frame.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the vessel, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

public enum **VesselType**

The type of a vessel. See *Vessel.getType()*.

public *VesselType* **BASE**  
Base.

public *VesselType* **DEBRIS**  
Debris.

public *VesselType* **LANDER**  
Lander.

public *VesselType* **PLANE**  
Plane.

public *VesselType* **PROBE**  
Probe.

public *VesselType* **RELAY**  
Relay.

public *VesselType* **ROVER**  
Rover.

```
public VesselType SHIP
    Ship.

public VesselType STATION
    Station.

public enum VesselSituation
    The situation a vessel is in. See Vessel.getSituation().

    public VesselSituation DOCKED
        Vessel is docked to another.

    public VesselSituation ESCAPING
        Escaping.

    public VesselSituation FLYING
        Vessel is flying through an atmosphere.

    public VesselSituation LANDED
        Vessel is landed on the surface of a body.

    public VesselSituation ORBITING
        Vessel is orbiting a body.

    public VesselSituation PRE_LAUNCH
        Vessel is awaiting launch.

    public VesselSituation SPLASHED
        Vessel has splashed down in an ocean.

    public VesselSituation SUB_ORBITAL
        Vessel is on a sub-orbital trajectory.
```

### 5.3.3 CelestialBody

```
public class CelestialBody
    Represents a celestial body (such as a planet or moon). See getBodies().

    String getName()
        The name of the body.

    java.util.List<CelestialBody> getSatellites()
        A list of celestial bodies that are in orbit around this celestial body.

    Orbit getOrbit()
        The orbit of the body.

    float getMass()
        The mass of the body, in kilograms.

    float getGravitationalParameter()
        The standard gravitational parameter of the body in  $m^3s^{-2}$ .

    float getSurfaceGravity()
        The acceleration due to gravity at sea level (mean altitude) on the body, in  $m/s^2$ .

    float getRotationalPeriod()
        The sidereal rotational period of the body, in seconds.

    float getRotationalSpeed()
        The rotational speed of the body, in radians per second.
```

double **getRotationAngle** ()

The current rotation angle of the body, in radians. A value between 0 and  $2\pi$

double **getInitialRotation** ()

The initial rotation angle of the body (at UT 0), in radians. A value between 0 and  $2\pi$

float **getEquatorialRadius** ()

The equatorial radius of the body, in meters.

double **surfaceHeight** (double *latitude*, double *longitude*)

The height of the surface relative to mean sea level, in meters, at the given position. When over water this is equal to 0.

#### Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.

double **bedrockHeight** (double *latitude*, double *longitude*)

The height of the surface relative to mean sea level, in meters, at the given position. When over water, this is the height of the sea-bed and is therefore negative value.

#### Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.

org.javatuples.Triplet<Double, Double, Double> **mSLPosition** (double *latitude*, double *longitude*,  
*ReferenceFrame referenceFrame*)

The position at mean sea level at the given latitude and longitude, in the given reference frame.

#### Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **referenceFrame** (*ReferenceFrame*) – Reference frame for the returned position vector.

**Returns** Position as a vector.

org.javatuples.Triplet<Double, Double, Double> **surfacePosition** (double *latitude*, double *longitude*,  
*ReferenceFrame referenceFrame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position of the surface of the water.

#### Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **referenceFrame** (*ReferenceFrame*) – Reference frame for the returned position vector.

**Returns** Position as a vector.

org.javatuples.Triplet<Double, Double, Double> **bedrockPosition** (double *latitude*, double *longitude*,  
*ReferenceFrame referenceFrame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position at the bottom of the sea-bed.

**Parameters**

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **referenceFrame** (*ReferenceFrame*) – Reference frame for the returned position vector.

**Returns** Position as a vector.

`org.javatuples.Triplet<Double, Double, Double> positionAtAltitude (double latitude, double longitude, double altitude, ReferenceFrame referenceFrame)`

The position at the given latitude, longitude and altitude, in the given reference frame.

**Parameters**

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **altitude** (*double*) – Altitude in meters above sea level.
- **referenceFrame** (*ReferenceFrame*) – Reference frame for the returned position vector.

**Returns** Position as a vector.

`double altitudeAtPosition (org.javatuples.Triplet<Double, Double, Double> position, ReferenceFrame referenceFrame)`

The altitude, in meters, of the given position in the given reference frame.

**Parameters**

- **position** (*org.javatuples.Triplet<Double, Double, Double>*) – Position as a vector.
- **referenceFrame** (*ReferenceFrame*) – Reference frame for the position vector.

`double latitudeAtPosition (org.javatuples.Triplet<Double, Double, Double> position, ReferenceFrame referenceFrame)`

The latitude of the given position, in the given reference frame.

**Parameters**

- **position** (*org.javatuples.Triplet<Double, Double, Double>*) – Position as a vector.
- **referenceFrame** (*ReferenceFrame*) – Reference frame for the position vector.

`double longitudeAtPosition (org.javatuples.Triplet<Double, Double, Double> position, ReferenceFrame referenceFrame)`

The longitude of the given position, in the given reference frame.

**Parameters**

- **position** (*org.javatuples.Triplet<Double, Double, Double>*) – Position as a vector.
- **referenceFrame** (*ReferenceFrame*) – Reference frame for the position vector.

`float getSphereOfInfluence ()`

The radius of the sphere of influence of the body, in meters.

boolean **getHasAtmosphere** ()  
 true if the body has an atmosphere.

float **getAtmosphereDepth** ()  
 The depth of the atmosphere, in meters.

double **atmosphericDensityAtPosition** (org.javatuples.Triplet<Double, Double, Double> *position*, ReferenceFrame *referenceFrame*)  
 The atmospheric density at the given position, in  $kg/m^3$ , in the given reference frame.

#### Parameters

- **position** (org.javatuples.Triplet<Double, Double, Double>) – The position vector at which to measure the density.
- **referenceFrame** (ReferenceFrame) – Reference frame that the position vector is in.

boolean **getHasAtmosphericOxygen** ()  
 true if there is oxygen in the atmosphere, required for air-breathing engines.

double **temperatureAt** (org.javatuples.Triplet<Double, Double, Double> *position*, ReferenceFrame *referenceFrame*)  
 The temperature on the body at the given position, in the given reference frame.

#### Parameters

- **position** (org.javatuples.Triplet<Double, Double, Double>) – Position as a vector.
- **referenceFrame** (ReferenceFrame) – The reference frame that the position is in.

---

**Note:** This calculation is performed using the bodies current position, which means that the value could be wrong if you want to know the temperature in the far future.

---

double **densityAt** (double *altitude*)  
 Gets the air density, in  $kg/m^3$ , for the specified altitude above sea level, in meters.

#### Parameters

- **altitude** (double) –

---

**Note:** This is an approximation, because actual calculations, taking sun exposure into account to compute air temperature, require us to know the exact point on the body where the density is to be computed (knowing the altitude is not enough). However, the difference is small for high altitudes, so it makes very little difference for trajectory prediction.

---

double **pressureAt** (double *altitude*)  
 Gets the air pressure, in Pascals, for the specified altitude above sea level, in meters.

#### Parameters

- **altitude** (double) –

java.util.Set<String> **getBiomes** ()  
 The biomes present on this body.

String **biomeAt** (double *latitude*, double *longitude*)  
 The biome at the given latitude and longitude, in degrees.

#### Parameters

- **latitude** (*double*) –
- **longitude** (*double*) –

float **getFlyingHighAltitudeThreshold** ()

The altitude, in meters, above which a vessel is considered to be flying “high” when doing science.

float **getSpaceHighAltitudeThreshold** ()

The altitude, in meters, above which a vessel is considered to be in “high” space when doing science.

*ReferenceFrame* **getReferenceFrame** ()

The reference frame that is fixed relative to the celestial body.

- The origin is at the center of the body.
- The axes rotate with the body.
- The x-axis points from the center of the body towards the intersection of the prime meridian and equator (the position at 0° longitude, 0° latitude).
- The y-axis points from the center of the body towards the north pole.
- The z-axis points from the center of the body towards the equator at 90°E longitude.

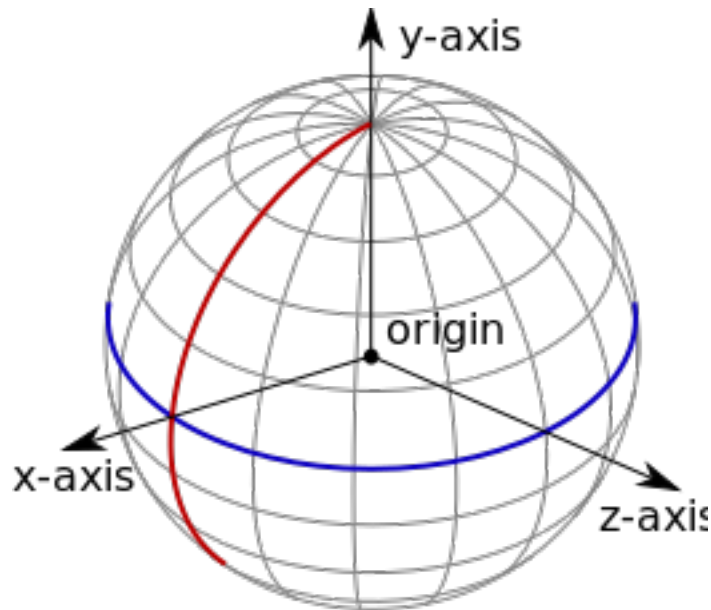


Fig. 5.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

*ReferenceFrame* **getNonRotatingReferenceFrame** ()

The reference frame that is fixed relative to this celestial body, and orientated in a fixed direction (it does not rotate with the body).

- The origin is at the center of the body.
- The axes do not rotate.
- The x-axis points in an arbitrary direction through the equator.
- The y-axis points from the center of the body towards the north pole.
- The z-axis points in an arbitrary direction through the equator.

*ReferenceFrame* **getOrbitalReferenceFrame** ()

The reference frame that is fixed relative to this celestial body, but orientated with the body's orbital prograde/normal/radial directions.

- The origin is at the center of the body.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

org.javatuples.Triplet<Double, Double, Double> **position** (*ReferenceFrame referenceFrame*)

The position of the center of the body, in the specified reference frame.

**Parameters**

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

org.javatuples.Triplet<Double, Double, Double> **velocity** (*ReferenceFrame referenceFrame*)

The linear velocity of the body, in the specified reference frame.

**Parameters**

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

org.javatuples.Quartet<Double, Double, Double, Double> **rotation** (*ReferenceFrame referenceFrame*)

The rotation of the body, in the specified reference frame.

**Parameters**

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

org.javatuples.Triplet<Double, Double, Double> **direction** (*ReferenceFrame referenceFrame*)

The direction in which the north pole of the celestial body is pointing, in the specified reference frame.

**Parameters**

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **angularVelocity** (*ReferenceFrame referenceFrame*)

The angular velocity of the body in the specified reference frame.

**Parameters**

- **referenceFrame** (*ReferenceFrame*) – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the body, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

### 5.3.4 Flight

public class **Flight**

Used to get flight telemetry for a vessel, by calling *Vessel.flight(ReferenceFrame)*. All of the information returned by this class is given in the reference frame passed to that method. Obtained by calling *Vessel.flight(ReferenceFrame)*.

---

**Note:** To get orbital information, such as the apoapsis or inclination, see *Orbit*.

---

float **getGForce** ()

The current G force acting on the vessel in  $m/s^2$ .

double **getMeanAltitude** ()

The altitude above sea level, in meters. Measured from the center of mass of the vessel.

double **getSurfaceAltitude** ()

The altitude above the surface of the body or sea level, whichever is closer, in meters. Measured from the center of mass of the vessel.

double **getBedrockAltitude** ()

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor. Measured from the center of mass of the vessel.

double **getElevation** ()

The elevation of the terrain under the vessel, in meters. This is the height of the terrain above sea level, and is negative when the vessel is over the sea.

double **getLatitude** ()

The *latitude* of the vessel for the body being orbited, in degrees.

double **getLongitude** ()

The *longitude* of the vessel for the body being orbited, in degrees.

org.javatuples.Triplet<Double, Double, Double> **getVelocity** ()

The velocity of the vessel, in the reference frame *ReferenceFrame*.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the vessel in meters per second.

double **getSpeed** ()

The speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

double **getHorizontalSpeed** ()

The horizontal speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

double **getVerticalSpeed** ()

The vertical speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

org.javatuples.Triplet<Double, Double, Double> **getCenterOfMass** ()

The position of the center of mass of the vessel, in the reference frame *ReferenceFrame*

**Returns** The position as a vector.

org.javatuples.Quartet<Double, Double, Double, Double> **getRotation** ()

The rotation of the vessel, in the reference frame *ReferenceFrame*



**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

org.javatuples.Triplet<Double, Double, Double> **getDirection** ()

The direction that the vessel is pointing in, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

float **getPitch** ()

The pitch of the vessel relative to the horizon, in degrees. A value between  $-90^\circ$  and  $+90^\circ$ .

float **getHeading** ()

The heading of the vessel (its angle relative to north), in degrees. A value between  $0^\circ$  and  $360^\circ$ .

float **getRoll** ()

The roll of the vessel relative to the horizon, in degrees. A value between  $-180^\circ$  and  $+180^\circ$ .

org.javatuples.Triplet<Double, Double, Double> **getPrograde** ()

The prograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getRetrograde** ()

The retrograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getNormal** ()

The direction normal to the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getAntiNormal** ()

The direction opposite to the normal of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getRadial** ()

The radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getAntiRadial** ()

The direction opposite to the radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Returns** The direction as a unit vector.

float **getAtmosphereDensity** ()

The current density of the atmosphere around the vessel, in  $kg/m^3$ .

float **getDynamicPressure** ()

The dynamic pressure acting on the vessel, in Pascals. This is a measure of the strength of the aerodynamic forces. It is equal to  $\frac{1}{2} \cdot \text{air density} \cdot \text{velocity}^2$ . It is commonly denoted  $Q$ .

float **getStaticPressure** ()

The static atmospheric pressure acting on the vessel, in Pascals.

float **getStaticPressureAtMSL** ()

The static atmospheric pressure at mean sea level, in Pascals.

org.javatuples.Triplet<Double, Double, Double> **getAerodynamicForce** ()

The total aerodynamic forces acting on the vessel, in reference frame *ReferenceFrame*.

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

`org.javatuples.Triplet<Double, Double, Double> simulateAerodynamicForceAt (CelestialBody body, org.javatuples.Triplet<Double, Double, Double> position, org.javatuples.Triplet<Double, Double, Double> velocity)`

Simulate and return the total aerodynamic forces acting on the vessel, if it were to be traveling with the given velocity at the given position in the atmosphere of the given celestial body.

**Parameters**

- **body** (*CelestialBody*) –
- **position** (*org.javatuples.Triplet<Double, Double, Double>*) –
- **velocity** (*org.javatuples.Triplet<Double, Double, Double>*) –

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

`org.javatuples.Triplet<Double, Double, Double> getLift ()`

The *aerodynamic lift* currently acting on the vessel.

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

`org.javatuples.Triplet<Double, Double, Double> getDrag ()`

The *aerodynamic drag* currently acting on the vessel.

**Returns** A vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

`float getSpeedOfSound ()`

The speed of sound, in the atmosphere around the vessel, in *m/s*.

`float getMach ()`

The speed of the vessel, in multiples of the speed of sound.

`float getReynoldsNumber ()`

The vessels Reynolds number.

---

**Note:** Requires *Ferram Aerospace Research*.

---

`float getTrueAirSpeed ()`

The *true air speed* of the vessel, in meters per second.

`float getEquivalentAirSpeed ()`

The *equivalent air speed* of the vessel, in meters per second.

`float getTerminalVelocity ()`

An estimate of the current terminal velocity of the vessel, in meters per second. This is the speed at which the drag forces cancel out the force of gravity.

`float getAngleOfAttack ()`

The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

`float getSideslipAngle ()`

The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

float **getTotalAirTemperature** ()

The **total air temperature** of the atmosphere around the vessel, in Kelvin. This includes the *Flight.getStaticAirTemperature* () and the vessel's kinetic energy.

float **getStaticAirTemperature** ()

The **static (ambient) temperature** of the atmosphere around the vessel, in Kelvin.

float **getStallFraction** ()

The current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

float **getDragCoefficient** ()

The coefficient of drag. This is the amount of drag produced by the vessel. It depends on air speed, air density and wing area.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

float **getLiftCoefficient** ()

The coefficient of lift. This is the amount of lift produced by the vessel, and depends on air speed, air density and wing area.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

float **getBallisticCoefficient** ()

The **ballistic coefficient**.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

float **getThrustSpecificFuelConsumption** ()

The thrust specific fuel consumption for the jet engines on the vessel. This is a measure of the efficiency of the engines, with a lower value indicating a more efficient vessel. This value is the number of Newtons of fuel that are burned, per hour, to produce one newton of thrust.

---

**Note:** Requires [Ferram Aerospace Research](#).

---

### 5.3.5 Orbit

public class **Orbit**

Describes an orbit. For example, the orbit of a vessel, obtained by calling *Vessel.getOrbit* (), or a celestial body, obtained by calling *CelestialBody.getOrbit* () .

*CelestialBody* **getBody** ()

The celestial body (e.g. planet or moon) around which the object is orbiting.

double **getApoapsis** ()

Gets the apoapsis of the orbit, in meters, from the center of mass of the body being orbited.

---

**Note:** For the apoapsis altitude reported on the in-game map view, use *Orbit.getApoapsisAltitude()*.

---

double **getPeriapsis** ()

The periapsis of the orbit, in meters, from the center of mass of the body being orbited.

---

**Note:** For the periapsis altitude reported on the in-game map view, use *Orbit.getPeriapsisAltitude()*.

---

double **getApoapsisAltitude** ()

The apoapsis of the orbit, in meters, above the sea level of the body being orbited.

---

**Note:** This is equal to *Orbit.getApoapsis()* minus the equatorial radius of the body.

---

double **getPeriapsisAltitude** ()

The periapsis of the orbit, in meters, above the sea level of the body being orbited.

---

**Note:** This is equal to *Orbit.getPeriapsis()* minus the equatorial radius of the body.

---

double **getSemiMajorAxis** ()

The semi-major axis of the orbit, in meters.

double **getSemiMinorAxis** ()

The semi-minor axis of the orbit, in meters.

double **getRadius** ()

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

---

**Note:** This value will change over time if the orbit is elliptical.

---

double **radiusAt** (double *ut*)

The orbital radius at the given time, in meters.

#### Parameters

- **ut** (*double*) – The universal time to measure the radius at.

org.javatuples.Triplet<Double, Double, Double> **positionAt** (double *ut*, *ReferenceFrame* *referenceFrame*)

The position at a given time, in the specified reference frame.

#### Parameters

- **ut** (*double*) – The universal time to measure the position at.
- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

double **getSpeed** ()

The current orbital speed of the object in meters per second.

---

**Note:** This value will change over time if the orbit is elliptical.

---

double **getPeriod** ()

The orbital period, in seconds.

double **getTimeToApoapsis** ()

The time until the object reaches apoapsis, in seconds.

double **getTimeToPeriapsis** ()

The time until the object reaches periapsis, in seconds.

double **getEccentricity** ()

The *eccentricity* of the orbit.

double **getInclination** ()

The *inclination* of the orbit, in radians.

double **getLongitudeOfAscendingNode** ()

The *longitude of the ascending node*, in radians.

double **getArgumentOfPeriapsis** ()

The *argument of periapsis*, in radians.

double **getMeanAnomalyAtEpoch** ()

The *mean anomaly at epoch*.

double **getEpoch** ()

The time since the epoch (the point at which the *mean anomaly at epoch* was measured, in seconds.

double **getMeanAnomaly** ()

The *mean anomaly*.

double **meanAnomalyAtUT** (double *ut*)

The mean anomaly at the given time.

#### Parameters

- **ut** (*double*) – The universal time in seconds.

double **getEccentricAnomaly** ()

The *eccentric anomaly*.

double **eccentricAnomalyAtUT** (double *ut*)

The eccentric anomaly at the given universal time.

#### Parameters

- **ut** (*double*) – The universal time, in seconds.

double **getTrueAnomaly** ()

The *true anomaly*.

double **trueAnomalyAtUT** (double *ut*)

The true anomaly at the given time.

#### Parameters

- **ut** (*double*) – The universal time in seconds.

double **trueAnomalyAtRadius** (double *radius*)

The true anomaly at the given orbital radius.

#### Parameters

- **radius** (*double*) – The orbital radius in meters.

double **uTAtTrueAnomaly** (double *trueAnomaly*)

The universal time, in seconds, corresponding to the given true anomaly.

#### Parameters

- **trueAnomaly** (*double*) – True anomaly.

double **radiusAtTrueAnomaly** (double *trueAnomaly*)

The orbital radius at the point in the orbit given by the true anomaly.

#### Parameters

- **trueAnomaly** (*double*) – The true anomaly.

double **trueAnomalyAtAN** (*Vessel target*)

The true anomaly of the ascending node with the given target vessel.

#### Parameters

- **target** (*Vessel*) – Target vessel.

double **trueAnomalyAtDN** (*Vessel target*)

The true anomaly of the descending node with the given target vessel.

#### Parameters

- **target** (*Vessel*) – Target vessel.

double **getOrbitalSpeed** ()

The current orbital speed in meters per second.

double **orbitalSpeedAt** (double *time*)

The orbital speed at the given time, in meters per second.

#### Parameters

- **time** (*double*) – Time from now, in seconds.

static org.javatuples.Triplet<Double, Double, Double> **referencePlaneNormal** (*Connection connection, ReferenceFrame referenceFrame*)

The direction that is normal to the orbits reference plane, in the given reference frame. The reference plane is the plane from which the orbits inclination is measured.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

static org.javatuples.Triplet<Double, Double, Double> **referencePlaneDirection** (*Connection connection, ReferenceFrame referenceFrame*)

The direction from which the orbits longitude of ascending node is measured, in the given reference frame.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

double **relativeInclination** (*Vessel target*)

Relative inclination of this orbit and the orbit of the given target vessel, in radians.

**Parameters**

- **target** (*Vessel*) – Target vessel.

double **getTimeToSOIChange** ()

The time until the object changes sphere of influence, in seconds. Returns NaN if the object is not going to change sphere of influence.

*Orbit* **getNextOrbit** ()

If the object is going to change sphere of influence in the future, returns the new orbit after the change. Otherwise returns null.

double **timeOfClosestApproach** (*Vessel target*)

Estimates and returns the time at closest approach to a target vessel.

**Parameters**

- **target** (*Vessel*) – Target vessel.

**Returns** The universal time at closest approach, in seconds.

double **distanceAtClosestApproach** (*Vessel target*)

Estimates and returns the distance at closest approach to a target vessel, in meters.

**Parameters**

- **target** (*Vessel*) – Target vessel.

java.util.List<java.util.List<Double>> **listClosestApproaches** (*Vessel target*, int *orbits*)

Returns the times at closest approach and corresponding distances, to a target vessel.

**Parameters**

- **target** (*Vessel*) – Target vessel.
- **orbits** (*int*) – The number of future orbits to search.

**Returns** A list of two lists. The first is a list of times at closest approach, as universal times in seconds. The second is a list of corresponding distances at closest approach, in meters.

### 5.3.6 Control

public class **Control**

Used to manipulate the controls of a vessel. This includes adjusting the throttle, enabling/disabling systems such as SAS and RCS, or altering the direction in which the vessel is pointing. Obtained by calling *Vessel.getControl()*.

---

**Note:** Control inputs (such as pitch, yaw and roll) are zeroed when all clients that have set one or more of these inputs are no longer connected.

---

*ControlSource* **getSource** ()

The source of the vessels control, for example by a kerbal or a probe core.

*ControlState* **getState** ()

The control state of the vessel.

boolean **getSAS** ()

void **setSAS** (boolean *value*)  
The state of SAS.

---

**Note:** Equivalent to *AutoPilot.getSAS()*

---

*SASMode* **getSASMode** ()

void **setSASMode** (*SASMode value*)  
The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

---

**Note:** Equivalent to *AutoPilot.getSASMode()*

---

*SpeedMode* **getSpeedMode** ()

void **setSpeedMode** (*SpeedMode value*)  
The current *SpeedMode* of the navball. This is the mode displayed next to the speed at the top of the navball.

boolean **getRCS** ()

void **setRCS** (boolean *value*)  
The state of RCS.

boolean **getReactionWheels** ()

void **setReactionWheels** (boolean *value*)  
Returns whether all reactive wheels on the vessel are active, and sets the active state of all reaction wheels. See *ReactionWheel.getActive()*.

boolean **getGear** ()

void **setGear** (boolean *value*)  
The state of the landing gear/legs.

boolean **getLegs** ()

void **setLegs** (boolean *value*)  
Returns whether all landing legs on the vessel are deployed, and sets the deployment state of all landing legs. Does not include wheels (for example landing gear). See *Leg.getDeployed()*.

boolean **getWheels** ()

void **setWheels** (boolean *value*)  
Returns whether all wheels on the vessel are deployed, and sets the deployment state of all wheels. Does not include landing legs. See *Wheel.getDeployed()*.

boolean **getLights** ()

void **setLights** (boolean *value*)  
The state of the lights.

boolean **getBrakes** ()

void **setBrakes** (boolean *value*)  
The state of the wheel brakes.

boolean **getAntennas** ()



void **setAntennas** (boolean *value*)  
Returns whether all antennas on the vessel are deployed, and sets the deployment state of all antennas. See *Antenna.getDeployed()*.

boolean **getCargoBays** ()

void **setCargoBays** (boolean *value*)  
Returns whether any of the cargo bays on the vessel are open, and sets the open state of all cargo bays. See *CargoBay.getOpen()*.

boolean **getIntakes** ()

void **setIntakes** (boolean *value*)  
Returns whether all of the air intakes on the vessel are open, and sets the open state of all air intakes. See *Intake.getOpen()*.

boolean **getParachutes** ()

void **setParachutes** (boolean *value*)  
Returns whether all parachutes on the vessel are deployed, and sets the deployment state of all parachutes. Cannot be set to *false*. See *Parachute.getDeployed()*.

boolean **getRadiators** ()

void **setRadiators** (boolean *value*)  
Returns whether all radiators on the vessel are deployed, and sets the deployment state of all radiators. See *Radiator.getDeployed()*.

boolean **getResourceHarvesters** ()

void **setResourceHarvesters** (boolean *value*)  
Returns whether all of the resource harvesters on the vessel are deployed, and sets the deployment state of all resource harvesters. See *ResourceHarvester.getDeployed()*.

boolean **getResourceHarvestersActive** ()

void **setResourceHarvestersActive** (boolean *value*)  
Returns whether any of the resource harvesters on the vessel are active, and sets the active state of all resource harvesters. See *ResourceHarvester.getActive()*.

boolean **getSolarPanels** ()

void **setSolarPanels** (boolean *value*)  
Returns whether all solar panels on the vessel are deployed, and sets the deployment state of all solar panels. See *SolarPanel.getDeployed()*.

boolean **getAbort** ()

void **setAbort** (boolean *value*)  
The state of the abort action group.

float **getThrottle** ()

void **setThrottle** (float *value*)  
The state of the throttle. A value between 0 and 1.

*ControlInputMode* **getInputMode** ()

void **setInputMode** (*ControlInputMode* *value*)  
Sets the behavior of the pitch, yaw, roll and translation control inputs. When set to additive, these inputs are added to the vessels current inputs. This mode is the default. When set to override, these inputs (if non-zero) override the vessels inputs. This mode prevents keyboard control, or SAS, from interfering with the controls when they are set.

float **getPitch** ()

void **setPitch** (float *value*)

The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

float **getYaw** ()

void **setYaw** (float *value*)

The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

float **getRoll** ()

void **setRoll** (float *value*)

The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

float **getForward** ()

void **setForward** (float *value*)

The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

float **getUp** ()

void **setUp** (float *value*)

The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

float **getRight** ()

void **setRight** (float *value*)

The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

float **getWheelThrottle** ()

void **setWheelThrottle** (float *value*)

The state of the wheel throttle. A value between -1 and 1. A value of 1 rotates the wheels forwards, a value of -1 rotates the wheels backwards.

float **getWheelSteering** ()

void **setWheelSteering** (float *value*)

The state of the wheel steering. A value between -1 and 1. A value of 1 steers to the left, and a value of -1 steers to the right.

int **getCurrentStage** ()

The current stage of the vessel. Corresponds to the stage number in the in-game UI.

java.util.List<Vessel> **activateNextStage** ()

Activates the next stage. Equivalent to pressing the space bar in-game.

**Returns** A list of vessel objects that are jettisoned from the active vessel.

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *getActiveVessel* () no longer refer to the active vessel.

---

boolean **getActionGroup** (int *group*)

Returns `true` if the given action group is enabled.

**Parameters**

- **group** (*int*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.

void **setActionGroup** (int *group*, boolean *state*)

Sets the state of the given action group.

**Parameters**

- **group** (*int*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.
- **state** (*boolean*) –

void **toggleActionGroup** (*int group*)  
Toggles the state of the given action group.

**Parameters**

- **group** (*int*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.

*Node* **addNode** (*double ut*, *float prograde*, *float normal*, *float radial*)

Creates a maneuver node at the given universal time, and returns a *Node* object that can be used to modify it. Optionally sets the magnitude of the delta-v for the maneuver node in the prograde, normal and radial directions.

**Parameters**

- **ut** (*double*) – Universal time of the maneuver node.
- **prograde** (*float*) – Delta-v in the prograde direction.
- **normal** (*float*) – Delta-v in the normal direction.
- **radial** (*float*) – Delta-v in the radial direction.

java.util.List<*Node*> **getNodes** ()

Returns a list of all existing maneuver nodes, ordered by time from first to last.

void **removeNodes** ()  
Remove all maneuver nodes.

public enum **ControlState**  
The control state of a vessel. See *Control.getState()*.

public *ControlState* **FULL**  
Full controllable.

public *ControlState* **PARTIAL**  
Partially controllable.

public *ControlState* **NONE**  
Not controllable.

public enum **ControlSource**  
The control source of a vessel. See *Control.getSource()*.

public *ControlSource* **KERBAL**  
Vessel is controlled by a Kerbal.

public *ControlSource* **PROBE**  
Vessel is controlled by a probe core.

public *ControlSource* **NONE**  
Vessel is not controlled.

public enum **SASMode**  
The behavior of the SAS auto-pilot. See *AutoPilot.getSASMode()*.

public *SASMode* **STABILITY\_ASSIST**  
Stability assist mode. Dampen out any rotation.

```
public SASMode MANEUVER
    Point in the burn direction of the next maneuver node.

public SASMode PROGRADE
    Point in the prograde direction.

public SASMode RETROGRADE
    Point in the retrograde direction.

public SASMode NORMAL
    Point in the orbit normal direction.

public SASMode ANTI_NORMAL
    Point in the orbit anti-normal direction.

public SASMode RADIAL
    Point in the orbit radial direction.

public SASMode ANTI_RADIAL
    Point in the orbit anti-radial direction.

public SASMode TARGET
    Point in the direction of the current target.

public SASMode ANTI_TARGET
    Point away from the current target.

public enum SpeedMode
    The mode of the speed reported in the navball. See Control.getSpeedMode().

    public SpeedMode ORBIT
        Speed is relative to the vessel's orbit.

    public SpeedMode SURFACE
        Speed is relative to the surface of the body being orbited.

    public SpeedMode TARGET
        Speed is relative to the current target.

public enum ControlInputMode
    See Control.getInputMode().

    public ControlInputMode ADDITIVE
        Control inputs are added to the vessels current control inputs.

    public ControlInputMode OVERRIDE
        Control inputs (when they are non-zero) override the vessels current control inputs.
```

### 5.3.7 Communications

```
public class Comms
    Used to interact with CommNet for a given vessel. Obtained by calling Vessel.getComms().

    boolean getCanCommunicate()
        Whether the vessel can communicate with KSC.

    boolean getCanTransmitScience()
        Whether the vessel can transmit science data to KSC.

    double getSignalStrength()
        Signal strength to KSC.
```

```

double getSignalDelay ()
    Signal delay to KSC in seconds.

double getPower ()
    The combined power of all active antennae on the vessel.

java.util.List<CommLink> getControlPath ()
    The communication path used to control the vessel.

public class CommLink
    Represents a communication node in the network. For example, a vessel or the KSC.

    CommLinkType getType ()
        The type of link.

    double getSignalStrength ()
        Signal strength of the link.

    CommNode getStart ()
        Start point of the link.

    CommNode getEnd ()
        Start point of the link.

public enum CommLinkType
    The type of a communication link. See CommLink.getType().

    public CommLinkType HOME
        Link is to a base station on Kerbin.

    public CommLinkType CONTROL
        Link is to a control source, for example a manned spacecraft.

    public CommLinkType RELAY
        Link is to a relay satellite.

public class CommNode
    Represents a communication node in the network. For example, a vessel or the KSC.

    String getName ()
        Name of the communication node.

    boolean getIsHome ()
        Whether the communication node is on Kerbin.

    boolean getIsControlPoint ()
        Whether the communication node is a control point, for example a manned vessel.

    boolean getIsVessel ()
        Whether the communication node is a vessel.

    Vessel getVessel ()
        The vessel for this communication node.

```

### 5.3.8 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*

- *Part*
- *Module*
- *Specific Types of Part*
  - *Antenna*
  - *Cargo Bay*
  - *Control Surface*
  - *Decoupler*
  - *Docking Port*
  - *Engine*
  - *Experiment*
  - *Fairing*
  - *Intake*
  - *Leg*
  - *Launch Clamp*
  - *Light*
  - *Parachute*
  - *Radiator*
  - *Resource Converter*
  - *Resource Harvester*
  - *Reaction Wheel*
  - *RCS*
  - *Sensor*
  - *Solar Panel*
  - *Thruster*
  - *Wheel*
- *Trees of Parts*
  - *Traversing the Tree*
  - *Attachment Modes*
- *Fuel Lines*
- *Staging*

## Parts

public class **Parts**

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling `Vessel.getParts()`.

```
java.util.List<Part> getAll()
```

A list of all of the vessels parts.

*Part* **getRoot** ()

The vessels root part.

---

**Note:** See the discussion on *Trees of Parts*.

---

*Part* **getControlling** ()

void **setControlling** (*Part* value)

The part from which the vessel is controlled.

java.util.List<*Part*> **withName** (*String* name)

A list of parts whose *Part.getName()* is *name*.

#### Parameters

- **name** (*String*) –

java.util.List<*Part*> **withTitle** (*String* title)

A list of all parts whose *Part.getTitle()* is *title*.

#### Parameters

- **title** (*String*) –

java.util.List<*Part*> **withTag** (*String* tag)

A list of all parts whose *Part.getTag()* is *tag*.

#### Parameters

- **tag** (*String*) –

java.util.List<*Part*> **withModule** (*String* moduleName)

A list of all parts that contain a *Module* whose *Module.getName()* is *moduleName*.

#### Parameters

- **moduleName** (*String*) –

java.util.List<*Part*> **inStage** (int stage)

A list of all parts that are activated in the given *stage*.

#### Parameters

- **stage** (*int*) –

---

**Note:** See the discussion on *Staging*.

---

java.util.List<*Part*> **inDecoupleStage** (int stage)

A list of all parts that are decoupled in the given *stage*.

#### Parameters

- **stage** (*int*) –

---

**Note:** See the discussion on *Staging*.

---

java.util.List<*Module*> **modulesWithName** (*String* moduleName)

A list of modules (combined across all parts in the vessel) whose *Module.getName()* is *moduleName*.

### Parameters

- `moduleName` (*String*) –

`java.util.List<Antenna> getAntennas ()`  
A list of all antennas in the vessel.

`java.util.List<CargoBay> getCargoBays ()`  
A list of all cargo bays in the vessel.

`java.util.List<ControlSurface> getControlSurfaces ()`  
A list of all control surfaces in the vessel.

`java.util.List<Decoupler> getDecouplers ()`  
A list of all decouplers in the vessel.

`java.util.List<DockingPort> getDockingPorts ()`  
A list of all docking ports in the vessel.

`java.util.List<Engine> getEngines ()`  
A list of all engines in the vessel.

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters, jet engines and RCS thrusters.

---

`java.util.List<Experiment> getExperiments ()`  
A list of all science experiments in the vessel.

`java.util.List<Fairing> getFairings ()`  
A list of all fairings in the vessel.

`java.util.List<Intake> getIntakes ()`  
A list of all intakes in the vessel.

`java.util.List<Leg> getLegs ()`  
A list of all landing legs attached to the vessel.

`java.util.List<LaunchClamp> getLaunchClamps ()`  
A list of all launch clamps attached to the vessel.

`java.util.List<Light> getLights ()`  
A list of all lights in the vessel.

`java.util.List<Parachute> getParachutes ()`  
A list of all parachutes in the vessel.

`java.util.List<Radiator> getRadiators ()`  
A list of all radiators in the vessel.

`java.util.List<RCS> getRCS ()`  
A list of all RCS blocks/thrusters in the vessel.

`java.util.List<ReactionWheel> getReactionWheels ()`  
A list of all reaction wheels in the vessel.

`java.util.List<ResourceConverter> getResourceConverters ()`  
A list of all resource converters in the vessel.

`java.util.List<ResourceHarvester> getResourceHarvesters ()`  
A list of all resource harvesters in the vessel.



```
java.util.List<Sensor> getSensors ()
```

A list of all sensors in the vessel.

```
java.util.List<SolarPanel> getSolarPanels ()
```

A list of all solar panels in the vessel.

```
java.util.List<Wheel> getWheels ()
```

A list of all wheels in the vessel.

## Part

public class **Part**

Represents an individual part. Vessels are made up of multiple parts. Instances of this class can be obtained by several methods in *Parts*.

```
String getName ()
```

Internal name of the part, as used in *part* *cfg* files. For example “Mark1-2Pod”.

```
String getTitle ()
```

Title of the part, as shown when the part is right clicked in-game. For example “Mk1-2 Command Pod”.

```
String getTag ()
```

```
void setTag (String value)
```

The name tag for the part. Can be set to a custom string using the in-game user interface.

---

**Note:** This requires either the *NameTag* or *kOS* mod to be installed.

---

```
boolean getHighlighted ()
```

```
void setHighlighted (boolean value)
```

Whether the part is highlighted.

```
org.javatuples.Triplet<Double, Double, Double> getHighlightColor ()
```

```
void setHighlightColor (org.javatuples.Triplet<Double, Double, Double> value)
```

The color used to highlight the part, as an RGB triple.

```
double getCost ()
```

The cost of the part, in units of funds.

```
Vessel getVessel ()
```

The vessel that contains this part.

```
Part getParent ()
```

The parts parent. Returns *null* if the part does not have a parent. This, in combination with *Part.getChildren()*, can be used to traverse the vessels parts tree.

---

**Note:** See the discussion on *Trees of Parts*.

---

```
java.util.List<Part> getChildren ()
```

The parts children. Returns an empty list if the part has no children. This, in combination with *Part.getParent()*, can be used to traverse the vessels parts tree.

---

**Note:** See the discussion on *Trees of Parts*.

---

boolean **getAxiallyAttached** ()

Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns `false`.

---

**Note:** See the discussion on *Attachment Modes*.

---

boolean **getRadiallyAttached** ()

Whether the part is radially attached to its parent, i.e. on the side of its parent. If the part has no parent, returns `false`.

---

**Note:** See the discussion on *Attachment Modes*.

---

int **getStage** ()

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

---

**Note:** See the discussion on *Staging*.

---

int **getDecoupleStage** ()

The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

---

**Note:** See the discussion on *Staging*.

---

boolean **getMassless** ()

Whether the part is `massless`.

double **getMass** ()

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

double **getDryMass** ()

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

boolean **getShielded** ()

Whether the part is shielded from the exterior of the vessel, for example by a fairing.

float **getDynamicPressure** ()

The dynamic pressure acting on the part, in Pascals.

double **getImpactTolerance** ()

The impact tolerance of the part, in meters per second.

double **getTemperature** ()

Temperature of the part, in Kelvin.

double **getSkinTemperature** ()

Temperature of the skin of the part, in Kelvin.

double **getMaxTemperature** ()

Maximum temperature that the part can survive, in Kelvin.

double **getMaxSkinTemperature** ()

Maximum temperature that the skin of the part can survive, in Kelvin.

float **getThermalMass** ()  
 A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

float **getThermalSkinMass** ()  
 A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

float **getThermalResourceMass** ()  
 A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

float **getThermalConductionFlux** ()  
 The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalConvectionFlux** ()  
 The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalRadiationFlux** ()  
 The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalInternalFlux** ()  
 The rate at which heat energy is begin generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalSkinToInternalFlux** ()  
 The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

*Resources* **getResources** ()  
 A *Resources* object for the part.

boolean **getCrossfeed** ()  
 Whether this part is crossfeed capable.

boolean **getIsFuelLine** ()  
 Whether this part is a fuel line.

java.util.List<Part> **getFuelLinesFrom** ()  
 The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

---

**Note:** See the discussion on *Fuel Lines*.

---

java.util.List<Part> **getFuelLinesTo** ()  
 The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

---

**Note:** See the discussion on *Fuel Lines*.

---

java.util.List<Module> **getModules** ()  
 The modules for this part.

*Antenna* **getAntenna** ()

A *Antenna* if the part is an antenna, otherwise null.

*CargoBay* **getCargoBay** ()

A *CargoBay* if the part is a cargo bay, otherwise null.

*ControlSurface* **getControlSurface** ()

A *ControlSurface* if the part is an aerodynamic control surface, otherwise null.

*Decoupler* **getDecoupler** ()

A *Decoupler* if the part is a decoupler, otherwise null.

*DockingPort* **getDockingPort** ()

A *DockingPort* if the part is a docking port, otherwise null.

*Engine* **getEngine** ()

An *Engine* if the part is an engine, otherwise null.

*Experiment* **getExperiment** ()

An *Experiment* if the part is a science experiment, otherwise null.

*Fairing* **getFairing** ()

A *Fairing* if the part is a fairing, otherwise null.

*Intake* **getIntake** ()

An *Intake* if the part is an intake, otherwise null.

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters and jet engines. For RCS thrusters see *RCS*.

---

*Leg* **getLeg** ()

A *Leg* if the part is a landing leg, otherwise null.

*LaunchClamp* **getLaunchClamp** ()

A *LaunchClamp* if the part is a launch clamp, otherwise null.

*Light* **getLight** ()

A *Light* if the part is a light, otherwise null.

*Parachute* **getParachute** ()

A *Parachute* if the part is a parachute, otherwise null.

*Radiator* **getRadiator** ()

A *Radiator* if the part is a radiator, otherwise null.

*RCS* **getRCS** ()

A *RCS* if the part is an RCS block/thruster, otherwise null.

*ReactionWheel* **getReactionWheel** ()

A *ReactionWheel* if the part is a reaction wheel, otherwise null.

*ResourceConverter* **getResourceConverter** ()

A *ResourceConverter* if the part is a resource converter, otherwise null.

*ResourceHarvester* **getResourceHarvester** ()

A *ResourceHarvester* if the part is a resource harvester, otherwise null.

*Sensor* **getSensor** ()

A *Sensor* if the part is a sensor, otherwise null.

*SolarPanel* **getSolarPanel** ()

A *SolarPanel* if the part is a solar panel, otherwise null.

*Wheel* **getWheel** ()

A *Wheel* if the part is a wheel, otherwise null.

org.javatuples.Triplet<Double, Double, Double> **position** (ReferenceFrame referenceFrame)

The position of the part in the given reference frame.

#### Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

---

**Note:** This is a fixed position in the part, defined by the parts model. It s not necessarily the same as the parts center of mass. Use *Part.centerOfMass (ReferenceFrame)* to get the parts center of mass.

---

org.javatuples.Triplet<Double, Double, Double> **centerOfMass** (ReferenceFrame referenceFrame)

The position of the parts center of mass in the given reference frame. If the part is physicsless, this is equivalent to *Part.position (ReferenceFrame)*.

#### Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **boundingBox** (ReferenceFrame referenceFrame)

The axis-aligned bounding box of the part in the given reference frame.

#### Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

---

**Note:** This is computed from the collision mesh of the part. If the part is not collidable, the box has zero volume and is centered on the *Part.position (ReferenceFrame)* of the part.

---

org.javatuples.Triplet<Double, Double, Double> **direction** (ReferenceFrame referenceFrame)

The direction the part points in, in the given reference frame.

#### Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **velocity** (ReferenceFrame referenceFrame)

The linear velocity of the part in the given reference frame.

#### Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

`org.javatuples.Quartet<Double, Double, Double, Double> rotation (ReferenceFrame reference-Frame)`

The rotation of the part, in the given reference frame.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

`org.javatuples.Triplet<Double, Double, Double> getMomentOfInertia ()`

The moment of inertia of the part in  $kg.m^2$  around its center of mass in the parts reference frame (*ReferenceFrame*).

`java.util.List<Double> getInertiaTensor ()`

The inertia tensor of the part in the parts reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

*ReferenceFrame* **getReferenceFrame ()**

The reference frame that is fixed relative to this part, and centered on a fixed position within the part, defined by the parts model.

- The origin is at the position of the part, as returned by `Part.position (ReferenceFrame)`.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by `DockingPort.getReferenceFrame ()`.

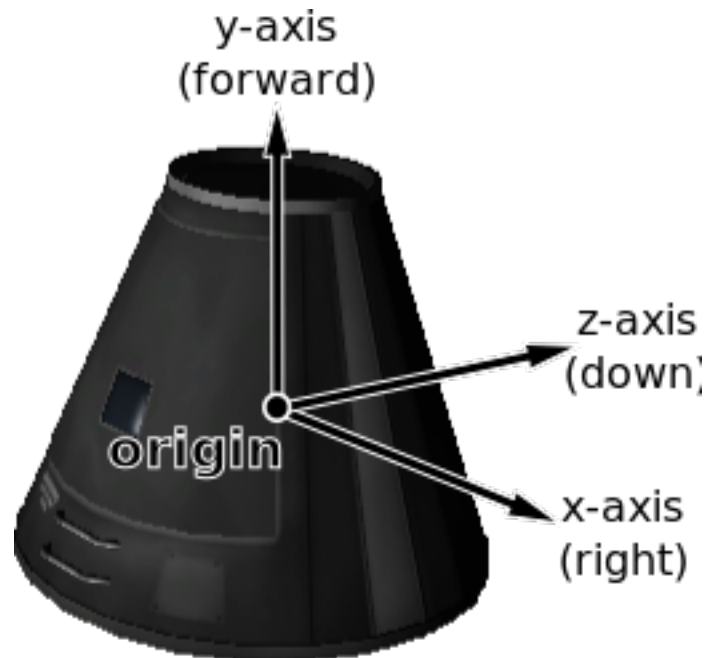


Fig. 5.7: Mk1 Command Pod reference frame origin and axes

*ReferenceFrame* **getCenterOfMassReferenceFrame** ()

The reference frame that is fixed relative to this part, and centered on its center of mass.

- The origin is at the center of mass of the part, as returned by *Part.centerOfMass* (*ReferenceFrame*).
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

---

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *DockingPort.getReferenceFrame* ().

---

*Force* **addForce** (*org.javatuples.Triplet*<*Double*, *Double*, *Double*> *force*,  
*org.javatuples.Triplet*<*Double*, *Double*, *Double*> *position*, *ReferenceFrame* *referenceFrame*)

Exert a constant force on the part, acting at the given position.

#### Parameters

- **force** (*org.javatuples.Triplet*<*Double*, *Double*, *Double*>) – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** (*org.javatuples.Triplet*<*Double*, *Double*, *Double*>) – The position at which the force acts, as a vector.
- **referenceFrame** (*ReferenceFrame*) – The reference frame that the force and position are in.

**Returns** An object that can be used to remove or modify the force.

*void* **instantaneousForce** (*org.javatuples.Triplet*<*Double*, *Double*, *Double*> *force*,  
*org.javatuples.Triplet*<*Double*, *Double*, *Double*> *position*, *ReferenceFrame* *referenceFrame*)

Exert an instantaneous force on the part, acting at the given position.

#### Parameters

- **force** (*org.javatuples.Triplet*<*Double*, *Double*, *Double*>) – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** (*org.javatuples.Triplet*<*Double*, *Double*, *Double*>) – The position at which the force acts, as a vector.
- **referenceFrame** (*ReferenceFrame*) – The reference frame that the force and position are in.

---

**Note:** The force is applied instantaneously in a single physics update.

---

public class **Force**

Obtained by calling *Part.addForce* (*org.javatuples.Triplet*<*Double*, *Double*, *Double*>, *org.javatuples.Triplet*<*Double*, *Double*, *Double*>, *ReferenceFrame*).

*Part* **getPart** ()

The part that this force is applied to.

*org.javatuples.Triplet*<*Double*, *Double*, *Double*> **getForceVector** ()

void **setForceVector** (org.javatuples.Triplet<Double, Double, Double> *value*)

The force vector, in Newtons.

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

org.javatuples.Triplet<Double, Double, Double> **getPosition** ()

void **setPosition** (org.javatuples.Triplet<Double, Double, Double> *value*)

The position at which the force acts, in reference frame *ReferenceFrame*.

**Returns** The position as a vector.

*ReferenceFrame* **getReferenceFrame** ()

void **setReferenceFrame** (*ReferenceFrame* *value*)

The reference frame of the force vector and position.

void **remove** ()

Remove the force.

## Module

public class **Module**

This can be used to interact with a specific part module. This includes part modules in stock KSP, and those added by mods.

In KSP, each part has zero or more *PartModules* associated with it. Each one contains some of the functionality of the part. For example, an engine has a “ModuleEngines” part module that contains all the functionality of an engine.

*String* **getName** ()

Name of the PartModule. For example, “ModuleEngines”.

*Part* **getPart** ()

The part that contains this module.

java.util.Map<*String*, *String*> **getFields** ()

The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

boolean **hasField** (*String* *name*)

Returns `true` if the module has a field with the given name.

### Parameters

- **name** (*String*) – Name of the field.

*String* **getField** (*String* *name*)

Returns the value of a field.

### Parameters

- **name** (*String*) – Name of the field.

void **setFieldInt** (*String* *name*, int *value*)

Set the value of a field to the given integer number.

### Parameters

- **name** (*String*) – Name of the field.
- **value** (*int*) – Value to set.



void **setFieldFloat** (*String* name, float value)  
Set the value of a field to the given floating point number.

**Parameters**

- **name** (*String*) – Name of the field.
- **value** (*float*) – Value to set.

void **setFieldString** (*String* name, *String* value)  
Set the value of a field to the given string.

**Parameters**

- **name** (*String*) – Name of the field.
- **value** (*String*) – Value to set.

void **resetField** (*String* name)  
Set the value of a field to its original value.

**Parameters**

- **name** (*String*) – Name of the field.

java.util.List<*String*> **getEvents** ()  
A list of the names of all of the modules events. Events are the clickable buttons visible in the right-click menu of the part.

boolean **hasEvent** (*String* name)  
true if the module has an event with the given name.

**Parameters**

- **name** (*String*) –

void **triggerEvent** (*String* name)  
Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

**Parameters**

- **name** (*String*) –

java.util.List<*String*> **getActions** ()  
A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

boolean **hasAction** (*String* name)  
true if the part has an action with the given name.

**Parameters**

- **name** (*String*) –

void **setAction** (*String* name, boolean value)  
Set the value of an action with the given name.

**Parameters**

- **name** (*String*) –
- **value** (*boolean*) –

## Specific Types of Part

The following classes provide functionality for specific types of part.

- *Antenna*
- *Cargo Bay*
- *Control Surface*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Experiment*
- *Fairing*
- *Intake*
- *Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *RCS*
- *Sensor*
- *Solar Panel*
- *Thruster*
- *Wheel*

## Antenna

public class **Antenna**

An antenna. Obtained by calling *Part.getAntenna()*.

*Part* **getPart** ()

The part object for this antenna.

*AntennaState* **getState** ()

The current state of the antenna.

boolean **getDeployable** ()

Whether the antenna is deployable.

boolean **getDeployed** ()

void **setDeployed** (boolean *value*)  
Whether the antenna is deployed.

---

**Note:** Fixed antennas are always deployed. Returns an error if you try to deploy a fixed antenna.

---

boolean **getCanTransmit** ()  
Whether data can be transmitted by this antenna.

void **transmit** ()  
Transmit data.

void **cancel** ()  
Cancel current transmission of data.

boolean **getAllowPartial** ()

void **setAllowPartial** (boolean *value*)  
Whether partial data transmission is permitted.

double **getPower** ()  
The power of the antenna.

boolean **getCombinable** ()  
Whether the antenna can be combined with other antennae on the vessel to boost the power.

double **getCombinableExponent** ()  
Exponent used to calculate the combined power of multiple antennae on a vessel.

float **getPacketInterval** ()  
Interval between sending packets in seconds.

float **getPacketSize** ()  
Amount of data sent per packet in Mits.

double **getPacketResourceCost** ()  
Units of electric charge consumed per packet sent.

public enum **AntennaState**  
The state of an antenna. See *Antenna.getState()*.

public *AntennaState* **DEPLOYED**  
Antenna is fully deployed.

public *AntennaState* **RETRACTED**  
Antenna is fully retracted.

public *AntennaState* **DEPLOYING**  
Antenna is being deployed.

public *AntennaState* **RETRACTING**  
Antenna is being retracted.

public *AntennaState* **BROKEN**  
Antenna is broken.

## Cargo Bay

public class **CargoBay**  
A cargo bay. Obtained by calling *Part.getCargoBay()*.

*Part* **getPart** ()

The part object for this cargo bay.

*CargoBayState* **getState** ()

The state of the cargo bay.

boolean **getOpen** ()

void **setOpen** (boolean *value*)

Whether the cargo bay is open.

public enum **CargoBayState**

The state of a cargo bay. See *CargoBay.getState()*.

public *CargoBayState* **OPEN**

Cargo bay is fully open.

public *CargoBayState* **CLOSED**

Cargo bay closed and locked.

public *CargoBayState* **OPENING**

Cargo bay is opening.

public *CargoBayState* **CLOSING**

Cargo bay is closing.

## Control Surface

public class **ControlSurface**

An aerodynamic control surface. Obtained by calling *Part.getControlSurface()*.

*Part* **getPart** ()

The part object for this control surface.

boolean **getPitchEnabled** ()

void **setPitchEnabled** (boolean *value*)

Whether the control surface has pitch control enabled.

boolean **getYawEnabled** ()

void **setYawEnabled** (boolean *value*)

Whether the control surface has yaw control enabled.

boolean **getRollEnabled** ()

void **setRollEnabled** (boolean *value*)

Whether the control surface has roll control enabled.

float **getAuthorityLimiter** ()

void **setAuthorityLimiter** (float *value*)

The authority limiter for the control surface, which controls how far the control surface will move.

boolean **getInverted** ()

void **setInverted** (boolean *value*)

Whether the control surface movement is inverted.

boolean **getDeployed** ()

void **setDeployed** (boolean *value*)

Whether the control surface has been fully deployed.

float **getSurfaceArea** ()

Surface area of the control surface in  $m^2$ .

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAv**

The available torque, in Newton meters, that can be produced by this control surface, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.getReferenceFrame()*.

## Decoupler

public class **Decoupler**

A decoupler. Obtained by calling *Part.getDecoupler()*

*Part* **getPart** ()

The part object for this decoupler.

*Vessel* **decouple** ()

Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *getActiveVessel()* no longer refer to the active vessel.

---

boolean **getDecoupled** ()

Whether the decoupler has fired.

boolean **getStaged** ()

Whether the decoupler is enabled in the staging sequence.

float **getImpulse** ()

The impulse that the decoupler imparts when it is fired, in Newton seconds.

## Docking Port

public class **DockingPort**

A docking port. Obtained by calling *Part.getDockingPort()*

*Part* **getPart** ()

The part object for this docking port.

*DockingPortState* **getState** ()

The current state of the docking port.

*Part* **getDockedPart** ()

The part that this docking port is docked to. Returns `null` if this docking port is not docked to anything.

*Vessel* **undock** ()

Undocks the docking port and returns the new *Vessel* that is created. This method can be called for either docking port in a docked pair. Throws an exception if the docking port is not docked to anything.

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *getActiveVessel()* no longer refer to the active vessel.

---

float **getReengageDistance** ()

The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

boolean **getHasShield** ()

Whether the docking port has a shield.

boolean **getShielded** ()

void **setShielded** (boolean *value*)

The state of the docking ports shield, if it has one.

Returns `true` if the docking port has a shield, and the shield is closed. Otherwise returns `false`. When set to `true`, the shield is closed, and when set to `false` the shield is opened. If the docking port does not have a shield, setting this attribute has no effect.

org.javatuples.Triplet<Double, Double, Double> **position** (ReferenceFrame *referenceFrame*)

The position of the docking port, in the given reference frame.

**Parameters**

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

org.javatuples.Triplet<Double, Double, Double> **direction** (ReferenceFrame *referenceFrame*)

The direction that docking port points in, in the given reference frame.

**Parameters**

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

org.javatuples.Quartet<Double, Double, Double, Double> **rotation** (ReferenceFrame *referenceFrame*)

The rotation of the docking port, in the given reference frame.

**Parameters**

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

ReferenceFrame **getReferenceFrame** ()

The reference frame that is fixed relative to this docking port, and oriented with the port.

- The origin is at the position of the docking port.
- The axes rotate with the docking port.
- The x-axis points out to the right side of the docking port.
- The y-axis points in the direction the docking port is facing.
- The z-axis points out of the bottom off the docking port.

---

**Note:** This reference frame is not necessarily equivalent to the reference frame for the part, returned by `Part.getReferenceFrame()`.

---

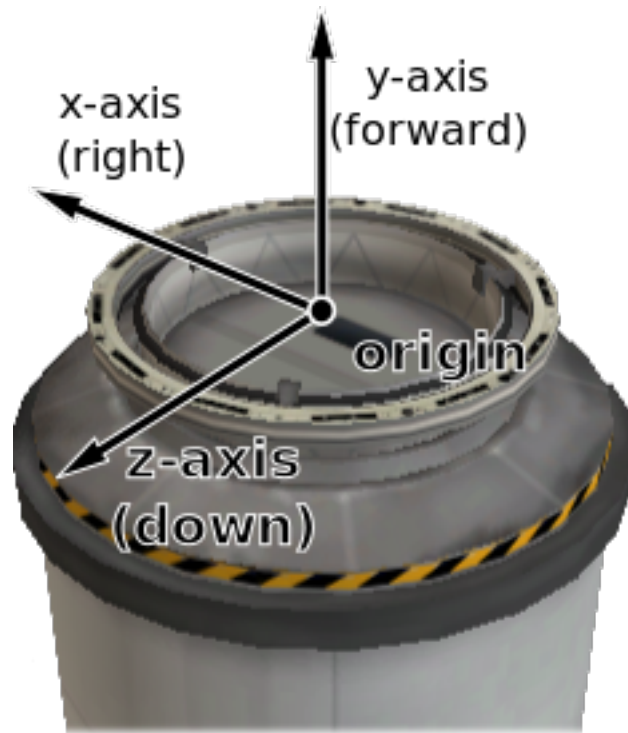


Fig. 5.8: Docking port reference frame origin and axes

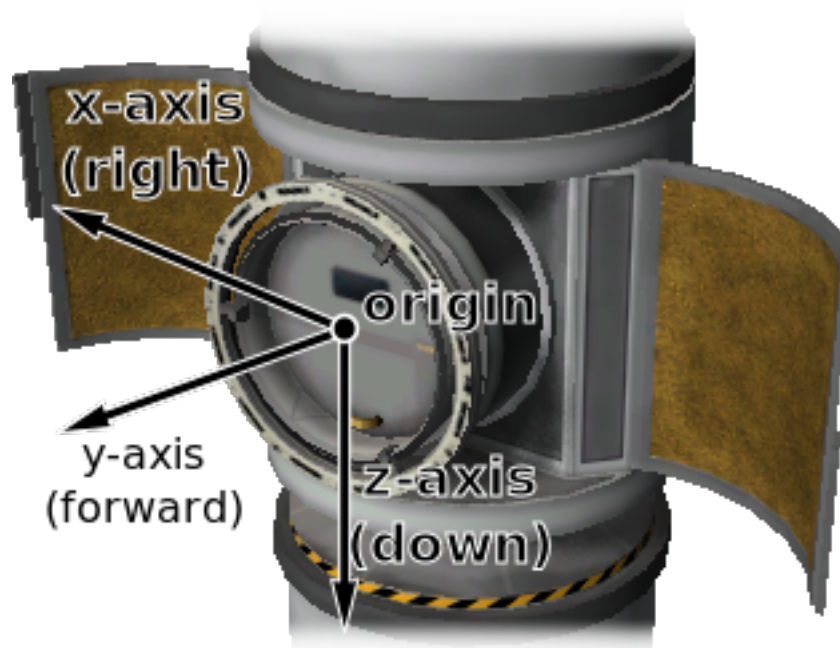


Fig. 5.9: Inline docking port reference frame origin and axes

public enum **DockingPortState**

The state of a docking port. See *DockingPort.getState()*.

public *DockingPortState* **READY**

The docking port is ready to dock to another docking port.

public *DockingPortState* **DOCKED**

The docking port is docked to another docking port, or docked to another part (from the VAB/SPH).

public *DockingPortState* **DOCKING**

The docking port is very close to another docking port, but has not docked. It is using magnetic force to acquire a solid dock.

public *DockingPortState* **UNDOCKING**

The docking port has just been undocked from another docking port, and is disabled until it moves away by a sufficient distance (*DockingPort.getReengageDistance()*).

public *DockingPortState* **SHIELDED**

The docking port has a shield, and the shield is closed.

public *DockingPortState* **MOVING**

The docking ports shield is currently opening/closing.

## Engine

public class **Engine**

An engine, including ones of various types. For example liquid fuelled gimballed engines, solid rocket boosters and jet engines. Obtained by calling *Part.getEngine()*.

---

**Note:** For RCS thrusters *Part.getRCS()*.

---

*Part* **getPart()**

The part object for this engine.

boolean **getActive()**

void **setActive**(boolean *value*)

Whether the engine is active. Setting this attribute may have no effect, depending on *Engine.getCanShutdown()* and *Engine.getCanRestart()*.

float **getThrust()**

The current amount of thrust being produced by the engine, in Newtons.

float **getAvailableThrust()**

The amount of thrust, in Newtons, that would be produced by the engine when activated and with its throttle set to 100%. Returns zero if the engine does not have any fuel. Takes the engine's current *Engine.getThrustLimit()* and atmospheric conditions into account.

float **getMaxThrust()**

The amount of thrust, in Newtons, that would be produced by the engine when activated and fueled, with its throttle and throttle limiter set to 100%.

float **getMaxVacuumThrust()**

The maximum amount of thrust that can be produced by the engine in a vacuum, in Newtons. This is the amount of thrust produced by the engine when activated, *Engine.getThrustLimit()* is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

float **getThrustLimit()**



void **setThrustLimit** (float *value*)

The thrust limiter of the engine. A value between 0 and 1. Setting this attribute may have no effect, for example the thrust limit for a solid rocket booster cannot be changed in flight.

java.util.List<Thruster> **getThrusters** ()

The components of the engine that generate thrust.

---

**Note:** For example, this corresponds to the rocket nozzle on a solid rocket booster, or the individual nozzles on a RAPIER engine. The overall thrust produced by the engine, as reported by *Engine.getAvailableThrust()*, *Engine.getMaxThrust()* and others, is the sum of the thrust generated by each thruster.

---

float **getSpecificImpulse** ()

The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

float **getVacuumSpecificImpulse** ()

The vacuum specific impulse of the engine, in seconds.

float **getKerbinSeaLevelSpecificImpulse** ()

The specific impulse of the engine at sea level on Kerbin, in seconds.

java.util.List<String> **getPropellantNames** ()

The names of the propellants that the engine consumes.

java.util.Map<String, Single> **getPropellantRatios** ()

The ratio of resources that the engine consumes. A dictionary mapping resource names to the ratio at which they are consumed by the engine.

---

**Note:** For example, if the ratios are 0.6 for LiquidFuel and 0.4 for Oxidizer, then for every 0.6 units of LiquidFuel that the engine burns, it will burn 0.4 units of Oxidizer.

---

java.util.List<Propellant> **getPropellants** ()

The propellants that the engine consumes.

boolean **getHasFuel** ()

Whether the engine has any fuel available.

---

**Note:** The engine must be activated for this property to update correctly.

---

float **getThrottle** ()

The current throttle setting for the engine. A value between 0 and 1. This is not necessarily the same as the vessel's main throttle setting, as some engines take time to adjust their throttle (such as jet engines).

boolean **getThrottleLocked** ()

Whether the *Control.getThrottle()* affects the engine. For example, this is `true` for liquid fueled rockets, and `false` for solid rocket boosters.

boolean **getCanRestart** ()

Whether the engine can be restarted once shutdown. If the engine cannot be shutdown, returns `false`. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

boolean **getCanShutdown** ()

Whether the engine can be shutdown once activated. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

boolean **getHasModes** ()

Whether the engine has multiple modes of operation.

String **getMode** ()

void **setMode** (String value)

The name of the current engine mode.

java.util.Map<String, Engine> **getModes** ()

The available modes for the engine. A dictionary mapping mode names to *Engine* objects.

void **toggleMode** ()

Toggle the current engine mode.

boolean **getAutoModeSwitch** ()

void **setAutoModeSwitch** (boolean value)

Whether the engine will automatically switch modes.

boolean **getGimballed** ()

Whether the engine is gimballed.

float **getGimbalRange** ()

The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

boolean **getGimbalLocked** ()

void **setGimbalLocked** (boolean value)

Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

float **getGimbalLimit** ()

void **setGimbalLimit** (float value)

The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAvailableTorque** ()

The available torque, in Newton meters, that can be produced by this engine, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel*. *getReferenceFrame* (). Returns zero if the engine is inactive, or not gimballed.

public class **Propellant**

A propellant for an engine. Obtains by calling *Engine.getPropellants* ().

String **getName** ()

The name of the propellant.

double **getCurrentAmount** ()

The current amount of propellant.

double **getCurrentRequirement** ()

The required amount of propellant.

double **getTotalResourceAvailable** ()

The total amount of the underlying resource currently reachable given resource flow rules.

double **getTotalResourceCapacity** ()

The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

boolean **getIgnoreForIsp** ()

If this propellant should be ignored when calculating required mass flow given specific impulse.

boolean **getIgnoreForThrustCurve** ()

If this propellant should be ignored for thrust curve calculations.

boolean **getDrawStackGauge** ()  
 If this propellant has a stack gauge or not.

boolean **getIsDeprived** ()  
 If this propellant is deprived.

float **getRatio** ()  
 The propellant ratio.

## Experiment

public class **Experiment**  
 Obtained by calling *Part.getExperiment()*.

*Part* **getPart** ()  
 The part object for this experiment.

void **run** ()  
 Run the experiment.

void **transmit** ()  
 Transmit all experimental data contained by this part.

void **dump** ()  
 Dump the experimental data contained by the experiment.

void **reset** ()  
 Reset the experiment.

boolean **getDeployed** ()  
 Whether the experiment has been deployed.

boolean **getRerunnable** ()  
 Whether the experiment can be re-run.

boolean **getInoperable** ()  
 Whether the experiment is inoperable.

boolean **getHasData** ()  
 Whether the experiment contains data.

java.util.List<ScienceData> **getData** ()  
 The data contained in this experiment.

String **getBiome** ()  
 The name of the biome the experiment is currently in.

boolean **getAvailable** ()  
 Determines if the experiment is available given the current conditions.

ScienceSubject **getScienceSubject** ()  
 Containing information on the corresponding specific science result for the current conditions. Returns null if the experiment is unavailable.

public class **ScienceData**  
 Obtained by calling *Experiment.getData()*.

float **getDataAmount** ()  
 Data amount.

float **getScienceValue** ()  
 Science value.

float **getTransmitValue** ()  
Transmit value.

public class **ScienceSubject**  
Obtained by calling *Experiment.getScienceSubject()*.

String **getTitle** ()  
Title of science subject, displayed in science archives

boolean **getIsComplete** ()  
Whether the experiment has been completed.

float **getScience** ()  
Amount of science already earned from this subject, not updated until after transmission/recovery.

float **getScienceCap** ()  
Total science allowable for this subject.

float **getDataScale** ()  
Multiply science value by this to determine data amount in mits.

float **getSubjectValue** ()  
Multiplier for specific Celestial Body/Experiment Situation combination.

float **getScientificValue** ()  
Diminishing value multiplier for decreasing the science value returned from repeated experiments.

## Fairing

public class **Fairing**  
A fairing. Obtained by calling *Part.getFairing()*.

Part **getPart** ()  
The part object for this fairing.

void **jettison** ()  
Jettison the fairing. Has no effect if it has already been jettisoned.

boolean **getJettisoned** ()  
Whether the fairing has been jettisoned.

## Intake

public class **Intake**  
An air intake. Obtained by calling *Part.getIntake()*.

Part **getPart** ()  
The part object for this intake.

boolean **getOpen** ()

void **setOpen** (boolean *value*)  
Whether the intake is open.

float **getSpeed** ()  
Speed of the flow into the intake, in *m/s*.

float **getFlow** ()  
The rate of flow into the intake, in units of resource per second.

float **getArea** ()  
 The area of the intake's opening, in square meters.

## Leg

public class **Leg**  
 A landing leg. Obtained by calling *Part.getLeg()*.

*Part* **getPart** ()  
 The part object for this landing leg.

*LegState* **getState** ()  
 The current state of the landing leg.

boolean **getDeployable** ()  
 Whether the leg is deployable.

boolean **getDeployed** ()

void **setDeployed** (boolean *value*)  
 Whether the landing leg is deployed.

---

**Note:** Fixed landing legs are always deployed. Returns an error if you try to deploy fixed landing gear.

---

boolean **getIsGrounded** ()  
 Returns whether the leg is touching the ground.

public enum **LegState**  
 The state of a landing leg. See *Leg.getState()*.

public *LegState* **DEPLOYED**  
 Landing leg is fully deployed.

public *LegState* **RETRACTED**  
 Landing leg is fully retracted.

public *LegState* **DEPLOYING**  
 Landing leg is being deployed.

public *LegState* **RETRACTING**  
 Landing leg is being retracted.

public *LegState* **BROKEN**  
 Landing leg is broken.

## Launch Clamp

public class **LaunchClamp**  
 A launch clamp. Obtained by calling *Part.getLaunchClamp()*.

*Part* **getPart** ()  
 The part object for this launch clamp.

void **release** ()  
 Releases the docking clamp. Has no effect if the clamp has already been released.

## Light

```
public class Light
    A light. Obtained by calling Part.getLight().

    Part getPart ()
        The part object for this light.

    boolean getActive ()

    void setActive (boolean value)
        Whether the light is switched on.

    org.javatuples.Triplet<Single, Single, Single> getColor ()

    void setColor (org.javatuples.Triplet<Single, Single, Single> value)
        The color of the light, as an RGB triple.

    float getPowerUsage ()
        The current power usage, in units of charge per second.
```

## Parachute

```
public class Parachute
    A parachute. Obtained by calling Part.getParachute().

    Part getPart ()
        The part object for this parachute.

    void deploy ()
        Deploys the parachute. This has no effect if the parachute has already been deployed.

    boolean getDeployed ()
        Whether the parachute has been deployed.

    void arm ()
        Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

    boolean getArmed ()
        Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

    ParachuteState getState ()
        The current state of the parachute.

    float getDeployAltitude ()

    void setDeployAltitude (float value)
        The altitude at which the parachute will full deploy, in meters. Only applicable to stock parachutes.

    float getDeployMinPressure ()

    void setDeployMinPressure (float value)
        The minimum pressure at which the parachute will semi-deploy, in atmospheres. Only applicable to stock parachutes.

public enum ParachuteState
    The state of a parachute. See Parachute.getState().

    public ParachuteState STOWED
        The parachute is safely tucked away inside its housing.
```

```

public ParachuteState ARMED
    The parachute is armed for deployment. (RealChutes only)

public ParachuteState ACTIVE
    The parachute is still stowed, but ready to semi-deploy. (Stock parachutes only)

public ParachuteState SEMI_DEPLOYED
    The parachute has been deployed and is providing some drag, but is not fully deployed yet. (Stock
    parachutes only)

public ParachuteState DEPLOYED
    The parachute is fully deployed.

public ParachuteState CUT
    The parachute has been cut.

```

## Radiator

```

public class Radiator
    A radiator. Obtained by calling Part.getRadiator().

    Part getPart ()
        The part object for this radiator.

    boolean getDeployable ()
        Whether the radiator is deployable.

    boolean getDeployed ()

    void setDeployed (boolean value)
        For a deployable radiator, true if the radiator is extended. If the radiator is not deployable, this is always
        true.

    RadiatorState getState ()
        The current state of the radiator.

```

---

**Note:** A fixed radiator is always *RadiatorState.EXTENDED*.

---

```

public enum RadiatorState
    The state of a radiator. RadiatorState

    public RadiatorState EXTENDED
        Radiator is fully extended.

    public RadiatorState RETRACTED
        Radiator is fully retracted.

    public RadiatorState EXTENDING
        Radiator is being extended.

    public RadiatorState RETRACTING
        Radiator is being retracted.

    public RadiatorState BROKEN
        Radiator is being broken.

```

## Resource Converter

public class **ResourceConverter**

A resource converter. Obtained by calling *Part.getResourceConverter()*.

*Part* **getPart** ()

The part object for this converter.

int **getCount** ()

The number of converters in the part.

*String* **name** (int *index*)

The name of the specified converter.

### Parameters

- **index** (*int*) – Index of the converter.

boolean **active** (int *index*)

True if the specified converter is active.

### Parameters

- **index** (*int*) – Index of the converter.

void **start** (int *index*)

Start the specified converter.

### Parameters

- **index** (*int*) – Index of the converter.

void **stop** (int *index*)

Stop the specified converter.

### Parameters

- **index** (*int*) – Index of the converter.

*ResourceConverterState* **state** (int *index*)

The state of the specified converter.

### Parameters

- **index** (*int*) – Index of the converter.

*String* **statusInfo** (int *index*)

Status information for the specified converter. This is the full status message shown in the in-game UI.

### Parameters

- **index** (*int*) – Index of the converter.

*java.util.List<String>* **inputs** (int *index*)

List of the names of resources consumed by the specified converter.

### Parameters

- **index** (*int*) – Index of the converter.

*java.util.List<String>* **outputs** (int *index*)

List of the names of resources produced by the specified converter.

### Parameters

- **index** (*int*) – Index of the converter.



public enum **ResourceConverterState**

The state of a resource converter. See *ResourceConverter.state(int)*.

public *ResourceConverterState* **RUNNING**

Converter is running.

public *ResourceConverterState* **IDLE**

Converter is idle.

public *ResourceConverterState* **MISSING\_RESOURCE**

Converter is missing a required resource.

public *ResourceConverterState* **STORAGE\_FULL**

No available storage for output resource.

public *ResourceConverterState* **CAPACITY**

At preset resource capacity.

public *ResourceConverterState* **UNKNOWN**

Unknown state. Possible with modified resource converters. In this case, check *ResourceConverter.statusInfo(int)* for more information.

## Resource Harvester

public class **ResourceHarvester**

A resource harvester (drill). Obtained by calling *Part.getResourceHarvester()*.

*Part* **getPart()**

The part object for this harvester.

*ResourceHarvesterState* **getState()**

The state of the harvester.

boolean **getDeployed()**

void **setDeployed**(boolean *value*)

Whether the harvester is deployed.

boolean **getActive()**

void **setActive**(boolean *value*)

Whether the harvester is actively drilling.

float **getExtractionRate()**

The rate at which the drill is extracting ore, in units per second.

float **getThermalEfficiency()**

The thermal efficiency of the drill, as a percentage of its maximum.

float **getCoreTemperature()**

The core temperature of the drill, in Kelvin.

float **getOptimumCoreTemperature()**

The core temperature at which the drill will operate with peak efficiency, in Kelvin.

public enum **ResourceHarvesterState**

The state of a resource harvester. See *ResourceHarvester.getState()*.

public *ResourceHarvesterState* **DEPLOYING**

The drill is deploying.

public *ResourceHarvesterState* **DEPLOYED**  
The drill is deployed and ready.

public *ResourceHarvesterState* **RETRACTING**  
The drill is retracting.

public *ResourceHarvesterState* **RETRACTED**  
The drill is retracted.

public *ResourceHarvesterState* **ACTIVE**  
The drill is running.

## Reaction Wheel

public class **ReactionWheel**  
A reaction wheel. Obtained by calling *Part.getReactionWheel()*.

*Part* **getPart** ()  
The part object for this reaction wheel.

boolean **getActive** ()

void **setActive** (boolean *value*)  
Whether the reaction wheel is active.

boolean **getBroken** ()  
Whether the reaction wheel is broken.

org.javatuples.*Pair*<org.javatuples.*Triplet*<Double, Double, Double>, org.javatuples.*Triplet*<Double, Double, Double>> **getAvailableTorque** ()  
The available torque, in Newton meters, that can be produced by this reaction wheel, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.getReferenceFrame()*. Returns zero if the reaction wheel is inactive or broken.

org.javatuples.*Pair*<org.javatuples.*Triplet*<Double, Double, Double>, org.javatuples.*Triplet*<Double, Double, Double>> **getMaximumTorque** ()  
The maximum torque, in Newton meters, that can be produced by this reaction wheel, when it is active, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.getReferenceFrame()*.

## RCS

public class **RCS**  
An RCS block or thruster. Obtained by calling *Part.getRCS()*.

*Part* **getPart** ()  
The part object for this RCS.

boolean **getActive** ()  
Whether the RCS thrusters are active. An RCS thruster is inactive if the RCS action group is disabled (*Control.getRCS()*), the RCS thruster itself is not enabled (*RCS.getEnabled()*) or it is covered by a fairing (*Part.getShielded()*).

boolean **getEnabled** ()

void **setEnabled** (boolean *value*)  
Whether the RCS thrusters are enabled.

boolean **getPitchEnabled** ()

```

void setPitchEnabled (boolean value)
    Whether the RCS thruster will fire when pitch control input is given.

boolean getYawEnabled ()

void setYawEnabled (boolean value)
    Whether the RCS thruster will fire when yaw control input is given.

boolean getRollEnabled ()

void setRollEnabled (boolean value)
    Whether the RCS thruster will fire when roll control input is given.

boolean getForwardEnabled ()

void setForwardEnabled (boolean value)
    Whether the RCS thruster will fire when pitch control input is given.

boolean getUpEnabled ()

void setUpEnabled (boolean value)
    Whether the RCS thruster will fire when yaw control input is given.

boolean getRightEnabled ()

void setRightEnabled (boolean value)
    Whether the RCS thruster will fire when roll control input is given.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> getAvailableTorque ()
    The available torque, in Newton meters, that can be produced by this RCS, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the Vessel. getReferenceFrame (). Returns zero if RCS is disable.

float getMaxThrust ()
    The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

float getMaxVacuumThrust ()
    The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

java.util.List<Thruster> getThrusters ()
    A list of thrusters, one of each nozzle in the RCS part.

float getSpecificImpulse ()
    The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

float getVacuumSpecificImpulse ()
    The vacuum specific impulse of the RCS, in seconds.

float getKerbinSeaLevelSpecificImpulse ()
    The specific impulse of the RCS at sea level on Kerbin, in seconds.

java.util.List<String> getPropellants ()
    The names of resources that the RCS consumes.

java.util.Map<String, Single> getPropellantRatios ()
    The ratios of resources that the RCS consumes. A dictionary mapping resource names to the ratios at which they are consumed by the RCS.

boolean getHasFuel ()
    Whether the RCS has fuel available.

```

---

**Note:** The RCS thruster must be activated for this property to update correctly.

---

## Sensor

public class **Sensor**

A sensor, such as a thermometer. Obtained by calling *Part.getSensor()*.

*Part* **getPart** ()

The part object for this sensor.

boolean **getActive** ()

void **setActive** (boolean *value*)

Whether the sensor is active.

*String* **getValue** ()

The current value of the sensor.

## Solar Panel

public class **SolarPanel**

A solar panel. Obtained by calling *Part.getSolarPanel()*.

*Part* **getPart** ()

The part object for this solar panel.

boolean **getDeployable** ()

Whether the solar panel is deployable.

boolean **getDeployed** ()

void **setDeployed** (boolean *value*)

Whether the solar panel is extended.

*SolarPanelState* **getState** ()

The current state of the solar panel.

float **getEnergyFlow** ()

The current amount of energy being generated by the solar panel, in units of charge per second.

float **getSunExposure** ()

The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

public enum **SolarPanelState**

The state of a solar panel. See *SolarPanel.getState()*.

public *SolarPanelState* **EXTENDED**

Solar panel is fully extended.

public *SolarPanelState* **RETRACTED**

Solar panel is fully retracted.

public *SolarPanelState* **EXTENDING**

Solar panel is being extended.

public *SolarPanelState* **RETRACTING**

Solar panel is being retracted.

public *SolarPanelState* **BROKEN**  
 Solar panel is broken.

## Thruster

public class **Thruster**

The component of an *Engine* or *RCS* part that generates thrust. Can obtained by calling *Engine.getThrusters()* or *RCS.getThrusters()*.

---

**Note:** Engines can consist of multiple thrusters. For example, the S3 KS-25x4 “Mammoth” has four rocket nozzels, and so consists of four thrusters.

---

*Part* **getPart()**

The *Part* that contains this thruster.

org.javatuples.Triplet<Double, Double, Double> **thrustPosition** (*ReferenceFrame*      *reference-Frame*)

The position at which the thruster generates thrust, in the given reference frame. For gimballed engines, this takes into account the current rotation of the gimbal.

### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

org.javatuples.Triplet<Double, Double, Double> **thrustDirection** (*ReferenceFrame*      *reference-Frame*)

The direction of the force generated by the thruster, in the given reference frame. This is opposite to the direction in which the thruster expels propellant. For gimballed engines, this takes into account the current rotation of the gimbal.

### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

*ReferenceFrame* **getThrustReferenceFrame()**

A reference frame that is fixed relative to the thruster and orientated with its thrust direction (*Thruster.thrustDirection(ReferenceFrame)*). For gimballed engines, this takes into account the current rotation of the gimbal.

- The origin is at the position of thrust for this thruster (*Thruster.thrustPosition(ReferenceFrame)*).
- The axes rotate with the thrust direction. This is the direction in which the thruster expels propellant, including any gimbaling.
- The y-axis points along the thrust direction.
- The x-axis and z-axis are perpendicular to the thrust direction.

boolean **getGimballed()**

Whether the thruster is gimballed.

org.javatuples.Triplet<Double, Double, Double> **gimbalPosition** (*ReferenceFrame referenceFrame*)

Position around which the gimbal pivots.

**Parameters**

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

org.javatuples.Triplet<Double, Double, Double> **getGimbalAngle** ()

The current gimbal angle in the pitch, roll and yaw axes, in degrees.

org.javatuples.Triplet<Double, Double, Double> **initialThrustPosition** (*ReferenceFrame referenceFrame*)

The position at which the thruster generates thrust, when the engine is in its initial position (no gimballing), in the given reference frame.

**Parameters**

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

---

**Note:** This position can move when the gimbal rotates. This is because the thrust position and gimbal position are not necessarily the same.

---

org.javatuples.Triplet<Double, Double, Double> **initialThrustDirection** (*ReferenceFrame referenceFrame*)

The direction of the force generated by the thruster, when the engine is in its initial position (no gimballing), in the given reference frame. This is opposite to the direction in which the thruster expels propellant.

**Parameters**

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

## Wheel

public class **Wheel**

A wheel. Includes landing gear and rover wheels. Obtained by calling *Part.getWheel()*. Can be used to control the motors, steering and deployment of wheels, among other things.

*Part* **getPart** ()

The part object for this wheel.

*WheelState* **getState** ()

The current state of the wheel.

float **getRadius** ()

Radius of the wheel, in meters.

boolean **getGrounded** ()

Whether the wheel is touching the ground.

boolean **getHasBrakes** ()  
Whether the wheel has brakes.

float **getBrakes** ()

void **setBrakes** (float *value*)  
The braking force, as a percentage of maximum, when the brakes are applied.

boolean **getAutoFrictionControl** ()

void **setAutoFrictionControl** (boolean *value*)  
Whether automatic friction control is enabled.

float **getManualFrictionControl** ()

void **setManualFrictionControl** (float *value*)  
Manual friction control value. Only has an effect if automatic friction control is disabled. A value between 0 and 5 inclusive.

boolean **getDeployable** ()  
Whether the wheel is deployable.

boolean **getDeployed** ()

void **setDeployed** (boolean *value*)  
Whether the wheel is deployed.

boolean **getPowered** ()  
Whether the wheel is powered by a motor.

boolean **getMotorEnabled** ()

void **setMotorEnabled** (boolean *value*)  
Whether the motor is enabled.

boolean **getMotorInverted** ()

void **setMotorInverted** (boolean *value*)  
Whether the direction of the motor is inverted.

*MotorState* **getMotorState** ()  
Whether the direction of the motor is inverted.

float **getMotorOutput** ()  
The output of the motor. This is the torque currently being generated, in Newton meters.

boolean **getTractionControlEnabled** ()

void **setTractionControlEnabled** (boolean *value*)  
Whether automatic traction control is enabled. A wheel only has traction control if it is powered.

float **getTractionControl** ()

void **setTractionControl** (float *value*)  
Setting for the traction control. Only takes effect if the wheel has automatic traction control enabled. A value between 0 and 5 inclusive.

float **getDriveLimiter** ()

void **setDriveLimiter** (float *value*)  
Manual setting for the motor limiter. Only takes effect if the wheel has automatic traction control disabled. A value between 0 and 100 inclusive.

boolean **getSteerable** ()  
Whether the wheel has steering.

```
boolean getSteeringEnabled()
void setSteeringEnabled(boolean value)
    Whether the wheel steering is enabled.

boolean getSteeringInverted()
void setSteeringInverted(boolean value)
    Whether the wheel steering is inverted.

boolean getHasSuspension()
    Whether the wheel has suspension.

float getSuspensionSpringStrength()
    Suspension spring strength, as set in the editor.

float getSuspensionDamperStrength()
    Suspension damper strength, as set in the editor.

boolean getBroken()
    Whether the wheel is broken.

boolean getRepairable()
    Whether the wheel is repairable.

float getStress()
    Current stress on the wheel.

float getStressTolerance()
    Stress tolerance of the wheel.

float getStressPercentage()
    Current stress on the wheel as a percentage of its stress tolerance.

float getDeflection()
    Current deflection of the wheel.

float getSlip()
    Current slip of the wheel.

public enum WheelState
    The state of a wheel. See Wheel.getState().

    public WheelState DEPLOYED
        Wheel is fully deployed.

    public WheelState RETRACTED
        Wheel is fully retracted.

    public WheelState DEPLOYING
        Wheel is being deployed.

    public WheelState RETRACTING
        Wheel is being retracted.

    public WheelState BROKEN
        Wheel is broken.

public enum MotorState
    The state of the motor on a powered wheel. See Wheel.getMotorState().

    public MotorState IDLE
        The motor is idle.
```



```
public MotorState RUNNING
    The motor is running.

public MotorState DISABLED
    The motor is disabled.

public MotorState INOPERABLE
    The motor is inoperable.

public MotorState NOT_ENOUGH_RESOURCES
    The motor does not have enough resources to run.
```

## Trees of Parts

Vessels in KSP are comprised of a number of parts, connected to one another in a *tree* structure. An example vessel is shown in Figure 1, and the corresponding tree of parts in Figure 2. The craft file for this example can also be downloaded [here](#).

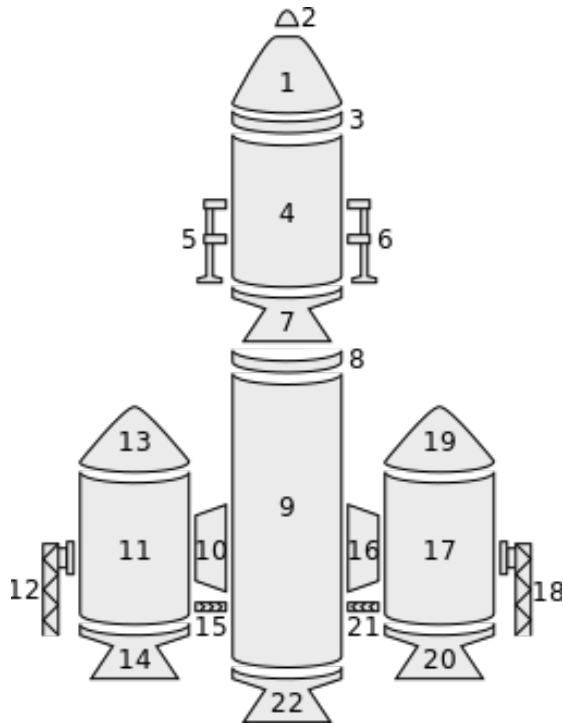


Fig. 5.10: **Figure 1** – Example parts making up a vessel.

## Traversing the Tree

The tree of parts can be traversed using the attributes `Parts.getRoot()`, `Part.getParent()` and `Part.getChildren()`.

The root of the tree is the same as the vessels *root* part (part number 1 in the example above) and can be obtained by calling `Parts.getRoot()`. A parts children can be obtained by calling `Part.getChildren()`. If the part does not have any children, `Part.getChildren()` returns an empty list. A parts parent can be obtained by calling `Part.getParent()`. If the part does not have a parent (as is the case for the root part), `Part.getParent()` returns null.

The following Java example uses these attributes to perform a depth-first traversal over all of the parts in a vessel:

```
import krpc.client.C
import krpc.client.R
import krpc.client.s
import krpc.client.s
import
↳krpc.client.servic

import org.javatuple

import java.io.IOExc
import java.util.Arr
import java.util.Deq

public class TreeTra
    public static vo
    ↳args) throws IOExc
        Connection_
    ↳connection = Conne
        Vessel vesse
    ↳newInstance(conne
```

```

    Part root = ve
Deque<Pair<P
    new ArrayDeque<P
stack.
    push(new Pair<Part
while (stack
    Pair<Part, In
    Part par
    int dept
    String p
    for (int
    pref
    }
    System.
    out.println(prefix
    for (Part chi
    stac
    Pair<Part, Integer
    }
    }
    connection.c
    }

```

When this code is execute using the craft file for the example vessel pictured above, the following is printed out:

```

Command Pod Mk1
TR-18A Stack Decoup
FL-T400 Fuel Tank
LV-909 Liquid Fue
TR-18A Stack Dec
FL-T800 Fuel Ta
LV-909 Liquid
TT-70 Radial D
FL-T400 Fuel
TT18-A Launc
FTX-2 Extern
LV-909 Liqui
Aerodynamic
TT-70 Radial D
FL-T400 Fuel
TT18-A Launc
FTX-2 Extern
LV-909 Liqui
Aerodynamic
LT-1 Landing Stru
LT-1 Landing Stru
Mk16 Parachute

```

## Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

The root part of a vessel (for example the command pod – part 1) does not have a parent part, so does not have an attachment mode. However, the part is consider to be *axially* attached to

nothing.

The following Java example does a depth-first traversal as before, but also prints out the attachment mode used by the part:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Part;
import
↳krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Pair;

import java.io.IOException;
import java.util.ArrayDeque;
import java.util.Deque;

public class AttachmentModes {
    public static void main(String[]
↳args) throws IOException, RPCException {
        Connection
↳connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.
↳newInstance(connection).getActiveVessel();

        Part root = vessel.getParts().getRoot();
        Deque<Pair<Part, Integer>> stack
↳= new ArrayDeque<Pair<Part, Integer>>();
        stack.
↳push(new Pair<Part, Integer>(root, 0));
        while (stack.size() > 0) {
            Pair<Part, Integer> item = stack.pop();
            Part part = item.getValue0();
            int depth = item.getValue1();
            String prefix = "";
            for (int i = 0; i < depth; i++) {
                prefix += " ";
            }
            String attachMode = part.
↳getAxiallyAttached() ? "axial" : "radial";
            System.out.println(prefix
↳+ part.getTitle() + " - " + attachMode);
        }
    }
}
```

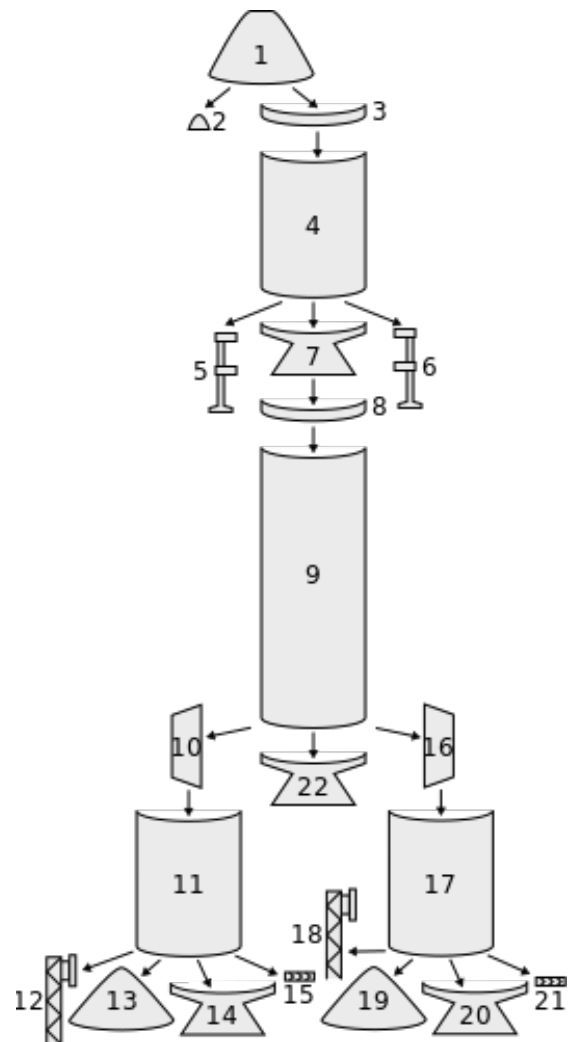


Fig. 5.11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the child part.

```
↪      for (Part child : part.getChildren()) {  
            stack.push(new_  
↪Pair<Part, Integer>(child, depth + 1));  
        }  
    }  
    connection.close();  
}  
}
```

When this code is execute using the craft file for the example vessel pictured above, the following is printed out:

```
Command Pod Mk1 - axial  
TR-18A Stack Decoupler - axial  
FL-T400 Fuel Tank - axial  
LV-909 Liquid Fuel Engine - axial  
TR-18A Stack Decoupler - axial  
FL-T800 Fuel Tank - axial  
LV-909 Liquid Fuel Engine - axial  
TT-70 Radial Decoupler - radial  
FL-T400 Fuel Tank - radial  
  
↪ TT18-A Launch Stability Enhancer - radial  
    FTX-2 External Fuel Duct - radial  
    LV-909 Liquid Fuel Engine - axial  
    Aerodynamic Nose Cone - axial  
    TT-70 Radial Decoupler - radial  
    FL-T400 Fuel Tank - radial  
  
↪ TT18-A Launch Stability Enhancer - radial  
    FTX-2 External Fuel Duct - radial  
    LV-909 Liquid Fuel Engine - axial  
    Aerodynamic Nose Cone - axial  
    LT-1 Landing Struts - radial  
    LT-1 Landing Struts - radial  
Mk16 Parachute - axial
```

## Fuel Lines

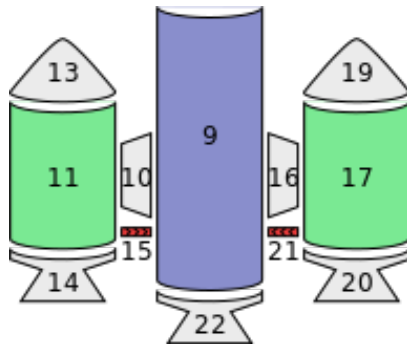


Fig. 5.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.

The attributes `Part.getFuelLinesFrom()` and `Part.getFuelLinesTo()` can be used to discover these connections. In the example in Figure 5, when `Part.getFuelLinesTo()` is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When `Part.getFuelLinesFrom()` is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).

## Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using `Part.getStage()` and `Part.getDecoupleStage()` respectively. For parts that are not activated by staging, `Part.getStage()` returns -1. For parts that are never decoupled, `Part.getDecoupleStage()` returns a value of -1.

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *activated*. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

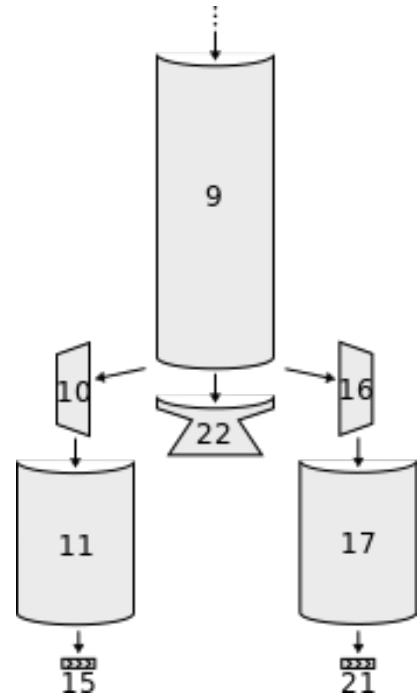


Fig. 5.13: **Figure 4** – A subset of the parts tree from Figure 2 above.

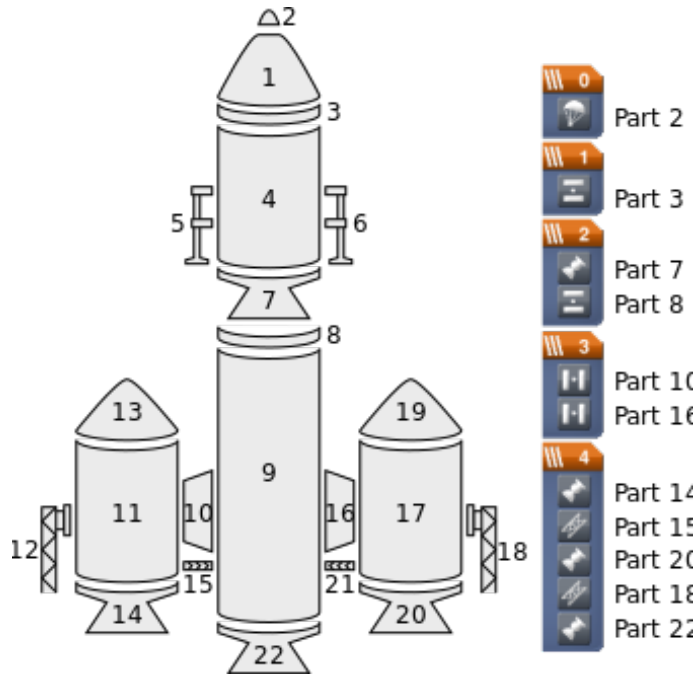


Fig. 5.14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

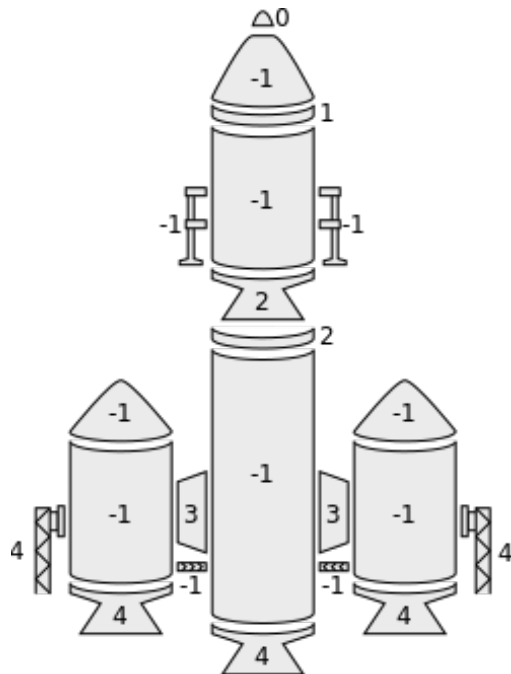
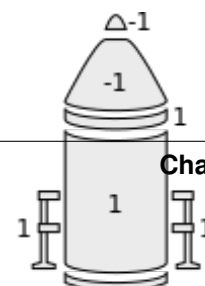


Fig. 5.15: **Figure 7** – The stage in which each part is *activated*.

### 5.3.9 Resources

public class **Resources**

Represents the collection of resources stored in a vessel, stage



or part. Created by calling `Vessel.getResources()`,  
`Vessel.resourcesInDecoupleStage(int, boolean)`  
 or `Part.getResources()`.

`java.util.List<Resource> getAll ()`

All the individual resources that can be stored.

`java.util.List<Resource> withResource (String name)`

All the individual resources with the given name that can be stored.

#### Parameters

- **name** (String) –

`java.util.List<String> getNames ()`

A list of resource names that can be stored.

`boolean hasResource (String name)`

Check whether the named resource can be stored.

#### Parameters

- **name** (String) – The name of the resource.

`float amount (String name)`

Returns the amount of a resource that is currently stored.

#### Parameters

- **name** (String) – The name of the resource.

`float max (String name)`

Returns the amount of a resource that can be stored.

#### Parameters

- **name** (String) – The name of the resource.

`static float density (Connection connection, String name)`

Returns the density of a resource, in kg/l.

#### Parameters

- **name** (String) – The name of the resource.

`static ResourceFlowMode flowMode (Connection connection, String name)`

Returns the flow mode of a resource.

#### Parameters

- **name** (String) – The name of the resource.

`boolean getEnabled ()`

`void setEnabled (boolean value)`

Whether use of all the resources are enabled.

---

**Note:** This is `true` if all of the resources are enabled. If any of the resources are not enabled, this is `false`.

---

public class **Resource**

An individual resource stored within a part. Created using methods in the `Resources` class.

`String getName ()`

The name of the resource.

`Part getPart ()`

The part containing the resource.

`float getAmount ()`

The amount of the resource that is currently stored in the part.

`float getMax ()`

The total amount of the resource that can be stored in the part.

`float getDensity ()`

The density of the resource, in *kg/l*.

`ResourceFlowMode getFlowMode ()`

The flow mode of the resource.

`boolean getEnabled ()`

`void setEnabled (boolean value)`

Whether use of this resource is enabled.

`public class ResourceTransfer`

Transfer resources between parts.

`static ResourceTransfer start (Connection connection, Part fromPart, Part toPart, String resource, float  
maxAmount)`

Start transferring a resource transfer between a pair of parts. The transfer will move at most *maxAmount* units of the resource, depending on how much of the resource is available in the source part and how much storage is available in the destination part. Use *ResourceTransfer.getComplete()* to check if the transfer is complete. Use *ResourceTransfer.getAmount()* to see how much of the resource has been transferred.

#### Parameters

- **fromPart** (*Part*) – The part to transfer to.
- **toPart** (*Part*) – The part to transfer from.
- **resource** (*String*) – The name of the resource to transfer.
- **maxAmount** (*float*) – The maximum amount of resource to transfer.

`float getAmount ()`

The amount of the resource that has been transferred.

`boolean getComplete ()`

Whether the transfer has completed.

`public enum ResourceFlowMode`

The way in which a resource flows between parts. See *Resources.flowMode(String)*.

`public ResourceFlowMode VESSEL`

The resource flows to any part in the vessel. For example, electric charge.

`public ResourceFlowMode STAGE`

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.



public *ResourceFlowMode* **ADJACENT**

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

public *ResourceFlowMode* **NONE**

The resource does not flow. For example, solid fuel.

### 5.3.10 Node

public class **Node**

Represents a maneuver node. Can be created using *Control*.

*addNode(double, float, float, float).*

double **getPrograde**()

void **setPrograde**(double *value*)

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

double **getNormal**()

void **setNormal**(double *value*)

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

double **getRadial**()

void **setRadial**(double *value*)

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

double **getDeltaV**()

void **setDeltaV**(double *value*)

The delta-v of the maneuver node, in meters per second.

---

**Note:** Does not change when executing the maneuver node. See *Node.getRemainingDeltaV()*.

---

double **getRemainingDeltaV**()

Gets the remaining delta-v of the maneuver node, in meters per second. Changes as the node is executed. This is equivalent to the delta-v reported in-game.

org.javatuples.Triplet<Double, Double, Double> **burnVector**(*ReferenceFrame referenceFrame*)

Returns the burn vector for the maneuver node.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned vector is in. Defaults to *Vessel.getOrbitalReferenceFrame()*.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

---

**Note:** Does not change when executing the maneuver node. See `Node.remainingBurnVector(ReferenceFrame)`.

---

`org.javatuples.Triplet<Double, Double, Double> remainingBurnVector(ReferenceFrame referenceFrame)`

Returns the remaining burn vector for the maneuver node.

#### Parameters

- **referenceFrame** (`ReferenceFrame`) – The reference frame that the returned vector is in. Defaults to `Vessel.getOrbitalReferenceFrame()`.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

---

**Note:** Changes as the maneuver node is executed. See `Node.burnVector(ReferenceFrame)`.

---

double **getUT** ()

void **setUT** (double *value*)

The universal time at which the maneuver will occur, in seconds.

double **getTimeTo** ()

The time until the maneuver node will be encountered, in seconds.

Orbit **getOrbit** ()

The orbit that results from executing the maneuver node.

void **remove** ()

Removes the maneuver node.

*ReferenceFrame* **getReferenceFrame** ()

The reference frame that is fixed relative to the maneuver node's burn.

- The origin is at the position of the maneuver node.
- The y-axis points in the direction of the burn.
- The x-axis and z-axis point in arbitrary but fixed directions.

*ReferenceFrame* **getOrbitalReferenceFrame** ()

The reference frame that is fixed relative to the maneuver node, and orientated with the orbital prograde/normal/radial directions of the original orbit at the maneuver node's position.

- The origin is at the position of the maneuver node.
- The x-axis points in the orbital anti-radial direction of the original orbit, at the position of the maneuver node.
- The y-axis points in the orbital prograde direction of the original orbit, at the position of the maneuver node.
- The z-axis points in the orbital normal direction of the original orbit, at the position of the maneuver node.

`org.javatuples.Triplet<Double, Double, Double> position (ReferenceFrame referenceFrame)`  
 The position vector of the maneuver node in the given reference frame.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

`org.javatuples.Triplet<Double, Double, Double> direction (ReferenceFrame referenceFrame)`  
 The direction of the maneuver nodes burn.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

### 5.3.11 ReferenceFrame

public class **ReferenceFrame**

Represents a reference frame for positions, rotations and velocities.  
 Contains:

- The position of the origin.
- The directions of the x, y and z axes.
- The linear velocity of the frame.
- The angular velocity of the frame.

---

**Note:** This class does not contain any properties or methods. It is only used as a parameter to other functions.

---

static *ReferenceFrame* **createRelative** (*Connection* connection, *ReferenceFrame* referenceFrame, *org.javatuples.Triplet<Double, Double, Double>* position, *org.javatuples.Quartet<Double, Double, Double, Double>* rotation, *org.javatuples.Triplet<Double, Double, Double>* velocity, *org.javatuples.Triplet<Double, Double, Double>* angularVelocity)

Create a relative reference frame. This is a custom reference frame whose components offset the components of a parent reference frame.

#### Parameters

- **referenceFrame** (*ReferenceFrame*) – The parent reference frame on which to base this reference frame.
- **position** (*org.javatuples.Triplet<Double, Double, Double>*) – The offset of the position of the origin, as a position vector. Defaults to (0, 0, 0)
- **rotation** (*org.javatuples.Quartet<Double, Double, Double, Double>*) – The rotation to apply to the parent frames rotation, as a quaternion of the form  $(x, y, z, w)$ . Defaults to (0, 0, 0, 1) (i.e. no rotation)

- **velocity** (*org.javatuples.Triplet<Double, Double, Double>*) – The linear velocity to offset the parent frame by, as a vector pointing in the direction of travel, whose magnitude is the speed in meters per second. Defaults to (0, 0, 0).
- **angularVelocity** (*org.javatuples.Triplet<Double, Double, Double>*) – The angular velocity to offset the parent frame by, as a vector. This vector points in the direction of the axis of rotation, and its magnitude is the speed of the rotation in radians per second. Defaults to (0, 0, 0).

static *ReferenceFrame* **createHybrid** (*Connection connection, ReferenceFrame position, ReferenceFrame rotation, ReferenceFrame velocity, ReferenceFrame angularVelocity*)

Create a hybrid reference frame. This is a custom reference frame whose components inherited from other reference frames.

#### Parameters

- **position** (*ReferenceFrame*) – The reference frame providing the position of the origin.
- **rotation** (*ReferenceFrame*) – The reference frame providing the rotation of the frame.
- **velocity** (*ReferenceFrame*) – The reference frame providing the linear velocity of the frame.
- **angularVelocity** (*ReferenceFrame*) – The reference frame providing the angular velocity of the frame.

---

**Note:** The *position* reference frame is required but all other reference frames are optional. If omitted, they are set to the *position* reference frame.

---

### 5.3.12 AutoPilot

public class **AutoPilot**

Provides basic auto-piloting utilities for a vessel. Created by calling *Vessel.getAutoPilot()*.

---

**Note:** If a client engages the auto-pilot and then closes its connection to the server, the auto-pilot will be disengaged and its target reference frame, direction and roll reset to default.

---

void **engage** ()  
Engage the auto-pilot.

void **disengage** ()  
Disengage the auto-pilot.

void **wait\_** ()  
Blocks until the vessel is pointing in the target direction and has the target roll (if set). Throws an exception if the auto-pilot has not been engaged.

float **getError** ()  
The error, in degrees, between the direction the ship has been asked

to point in and the direction it is pointing in. Throws an exception if the auto-pilot has not been engaged and SAS is not enabled or is in stability assist mode.

float **getPitchError** ()

The error, in degrees, between the vessels current and target pitch. Throws an exception if the auto-pilot has not been engaged.

float **getHeadingError** ()

The error, in degrees, between the vessels current and target heading. Throws an exception if the auto-pilot has not been engaged.

float **getRollError** ()

The error, in degrees, between the vessels current and target roll. Throws an exception if the auto-pilot has not been engaged or no target roll is set.

*ReferenceFrame* **getReferenceFrame** ()

void **setReferenceFrame** (*ReferenceFrame* value)

The reference frame for the target direction (*AutoPilot.getTargetDirection*()).

---

**Note:** An error will be thrown if this property is set to a reference frame that rotates with the vessel being controlled, as it is impossible to rotate the vessel in such a reference frame.

---

float **getTargetPitch** ()

void **setTargetPitch** (float value)

The target pitch, in degrees, between -90° and +90°.

float **getTargetHeading** ()

void **setTargetHeading** (float value)

The target heading, in degrees, between 0° and 360°.

float **getTargetRoll** ()

void **setTargetRoll** (float value)

The target roll, in degrees. NaN if no target roll is set.

org.javatuples.Triplet<Double, Double, Double> **getTargetDirection** ()

void **setTargetDirection** (org.javatuples.Triplet<Double, Double, Double> value)

Direction vector corresponding to the target pitch and heading. This is in the reference frame specified by *ReferenceFrame*.

void **targetPitchAndHeading** (float pitch, float heading)

Set target pitch and heading angles.

#### Parameters

- **pitch** (*float*) – Target pitch angle, in degrees between -90° and +90°.
- **heading** (*float*) – Target heading angle, in degrees between 0° and 360°.

boolean **getSAS** ()

void **setSAS** (boolean *value*)

The state of SAS.

---

**Note:** Equivalent to *Control.getSAS()*

---

*SASMode* **getSASMode** ()

void **setSASMode** (*SASMode value*)

The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

---

**Note:** Equivalent to *Control.getSASMode()*

---

double **getRollThreshold** ()

void **setRollThreshold** (double *value*)

The threshold at which the autopilot will try to match the target roll angle, if any. Defaults to 5 degrees.

org.javatuples.Triplet<Double, Double, Double> **getStoppingTime** ()

void **setStoppingTime** (org.javatuples.Triplet<Double, Double, Double> *value*)

The maximum amount of time that the vessel should need to come to a complete stop. This determines the maximum angular velocity of the vessel. A vector of three stopping times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 0.5 seconds for each axis.

org.javatuples.Triplet<Double, Double, Double> **getDecelerationTime** ()

void **setDecelerationTime** (org.javatuples.Triplet<Double, Double, Double> *value*)

The time the vessel should take to come to a stop pointing in the target direction. This determines the angular acceleration used to decelerate the vessel. A vector of three times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 5 seconds for each axis.

org.javatuples.Triplet<Double, Double, Double> **getAttenuationAngle** ()

void **setAttenuationAngle** (org.javatuples.Triplet<Double, Double, Double> *value*)

The angle at which the autopilot considers the vessel to be pointing close to the target. This determines the midpoint of the target velocity attenuation function. A vector of three angles, in degrees, one for each of the pitch, roll and yaw axes. Defaults to 1° for each axis.

boolean **getAutoTune** ()

void **setAutoTune** (boolean *value*)

Whether the rotation rate controllers PID parameters should be automatically tuned using the vessels moment of inertia and available torque. Defaults to `true`. See *AutoPilot.getTimeToPeak()* and *AutoPilot.getOvershoot()*.

org.javatuples.Triplet<Double, Double, Double> **getTimeToPeak** ()

void **setTimeToPeak** (org.javatuples.Triplet<Double, Double, Double> *value*)

The target time to peak used to autotune the PID controllers. A vector of three times, in seconds, for each of the pitch, roll and yaw axes. Defaults to 3 seconds for each axis.

org.javatuples.Triplet<Double, Double, Double> **getOvershoot** ()

void **setOvershoot** (org.javatuples.Triplet<Double, Double, Double> *value*)

The target overshoot percentage used to autotune the PID controllers. A vector of three values, between 0 and 1, for each of the pitch, roll and yaw axes. Defaults to 0.01 for each axis.

org.javatuples.Triplet<Double, Double, Double> **getPitchPIDGains** ()

void **setPitchPIDGains** (org.javatuples.Triplet<Double, Double, Double> *value*)

Gains for the pitch PID controller.

---

**Note:** When *AutoPilot.getAutoTune()* is true, these values are updated automatically, which will overwrite any manual changes.

---

org.javatuples.Triplet<Double, Double, Double> **getRollPIDGains** ()

void **setRollPIDGains** (org.javatuples.Triplet<Double, Double, Double> *value*)

Gains for the roll PID controller.

---

**Note:** When *AutoPilot.getAutoTune()* is true, these values are updated automatically, which will overwrite any manual changes.

---

org.javatuples.Triplet<Double, Double, Double> **getYawPIDGains** ()

void **setYawPIDGains** (org.javatuples.Triplet<Double, Double, Double> *value*)

Gains for the yaw PID controller.

---

**Note:** When *AutoPilot.getAutoTune()* is true, these values are updated automatically, which will overwrite any manual changes.

---

### 5.3.13 Camera

public class **Camera**

Controls the game's camera. Obtained by calling *getCamera()*.

*CameraMode* **getMode()**

void **setMode** (*CameraMode value*)

The current mode of the camera.

float **getPitch()**

void **setPitch** (float *value*)

The pitch of the camera, in degrees. A value between *Camera.getMinPitch()* and *Camera.getMaxPitch()*

float **getHeading()**

void **setHeading** (float *value*)

The heading of the camera, in degrees.

float **getDistance()**

void **setDistance** (float *value*)

The distance from the camera to the subject, in meters. A value between *Camera.getMinDistance()* and *Camera.getMaxDistance()*.

float **getMinPitch()**

The minimum pitch of the camera.

float **getMaxPitch()**

The maximum pitch of the camera.

float **getMinDistance()**

Minimum distance from the camera to the subject, in meters.

float **getMaxDistance()**

Maximum distance from the camera to the subject, in meters.

float **getDefaultDistance()**

Default distance from the camera to the subject, in meters.

*CelestialBody* **getFocussedBody()**

void **setFocussedBody** (*CelestialBody value*)

In map mode, the celestial body that the camera is focussed on. Returns *null* if the camera is not focussed on a celestial body. Returns an error if the camera is not in map mode.

*Vessel* **getFocussedVessel()**

void **setFocussedVessel** (*Vessel value*)

In map mode, the vessel that the camera is focussed on. Returns *null* if the camera is not focussed on a vessel. Returns an error if the camera is not in map mode.



*Node* **getFocussedNode** ()

void **setFocussedNode** (*Node value*)

In map mode, the maneuver node that the camera is focussed on.

Returns `null` if the camera is not focussed on a maneuver node.

Returns an error if the camera is not in map mode.

public enum **CameraMode**

See *Camera.getMode()*.

public *CameraMode* **AUTOMATIC**

The camera is showing the active vessel, in “auto” mode.

public *CameraMode* **FREE**

The camera is showing the active vessel, in “free” mode.

public *CameraMode* **CHASE**

The camera is showing the active vessel, in “chase” mode.

public *CameraMode* **LOCKED**

The camera is showing the active vessel, in “locked” mode.

public *CameraMode* **ORBITAL**

The camera is showing the active vessel, in “orbital” mode.

public *CameraMode* **IVA**

The Intra-Vehicular Activity view is being shown.

public *CameraMode* **MAP**

The map view is being shown.

### 5.3.14 Waypoints

public class **WaypointManager**

Waypoints are the location markers you can see on the map view showing you where contracts are targeted for. With this structure,

you can obtain coordinate data for the locations of these waypoints.

Obtained by calling *getWaypointManager()*.

java.util.List<*Waypoint*> **getWaypoints** ()

A list of all existing waypoints.

*Waypoint* **addWaypoint** (double *latitude*, double *longitude*, *CelestialBody* *body*, *String* *name*)

Creates a waypoint at the given position at ground level, and returns

a *Waypoint* object that can be used to modify it.

#### Parameters

- **latitude** (*double*) – Latitude of the waypoint.
- **longitude** (*double*) – Longitude of the waypoint.
- **body** (*CelestialBody*) – Celestial body the waypoint is attached to.
- **name** (*String*) – Name of the waypoint.

*Waypoint* **addWaypointAtAltitude** (double *latitude*, double *longitude*, double *altitude*, *CelestialBody* *body*, *String* *name*)

Creates a waypoint at the given position and altitude, and returns a

*Waypoint* object that can be used to modify it.

## Parameters

- **latitude** (*double*) – Latitude of the waypoint.
- **longitude** (*double*) – Longitude of the waypoint.
- **altitude** (*double*) – Altitude (above sea level) of the waypoint.
- **body** (*CelestialBody*) – Celestial body the waypoint is attached to.
- **name** (*String*) – Name of the waypoint.

`java.util.Map<String, Integer> getColors ()`

An example map of known color - seed pairs. Any other integers may be used as seed.

`java.util.List<String> getIcons ()`

Returns all available icons (from “Game-Data/Squad/Contracts/Icons”).

`public class Waypoint`

Represents a waypoint. Can be created using `WaypointManager.addWaypoint(double, double, CelestialBody, String)`.

`CelestialBody getBody ()`

`void setBody (CelestialBody value)`

The celestial body the waypoint is attached to.

`String getName ()`

`void setName (String value)`

The name of the waypoint as it appears on the map and the contract.

`int getColor ()`

`void setColor (int value)`

The seed of the icon color. See `WaypointManager.getColors()` for example colors.

`String getIcon ()`

`void setIcon (String value)`

The icon of the waypoint.

`double getLatitude ()`

`void setLatitude (double value)`

The latitude of the waypoint.

`double getLongitude ()`

`void setLongitude (double value)`

The longitude of the waypoint.

`double getMeanAltitude ()`

void **setMeanAltitude** (double *value*)

The altitude of the waypoint above sea level, in meters.

double **getSurfaceAltitude** ()

void **setSurfaceAltitude** (double *value*)

The altitude of the waypoint above the surface of the body or sea level, whichever is closer, in meters.

double **getBedrockAltitude** ()

void **setBedrockAltitude** (double *value*)

The altitude of the waypoint above the surface of the body, in meters. When over water, this is the altitude above the sea floor.

boolean **getNearSurface** ()

true if the waypoint is near to the surface of a body.

boolean **getGrounded** ()

true if the waypoint is attached to the ground.

int **getIndex** ()

The integer index of this waypoint within its cluster of sibling waypoints. In other words, when you have a cluster of waypoints called “Somewhere Alpha”, “Somewhere Beta” and “Somewhere Gamma”, the alpha site has index 0, the beta site has index 1 and the gamma site has index 2. When *Waypoint.getClustered()* is false, this is zero.

boolean **getClustered** ()

true if this waypoint is part of a set of clustered waypoints with greek letter names appended (Alpha, Beta, Gamma, etc). If true, there is a one-to-one correspondence with the greek letter name and the *Waypoint.getIndex()*.

boolean **getHasContract** ()

Whether the waypoint belongs to a contract.

*Contract* **getContract** ()

The associated contract.

void **remove** ()

Removes the waypoint.

### 5.3.15 Contracts

public class **ContractManager**

Contracts manager. Obtained by calling *getWaypointManager()*.

java.util.Set<String> **getTypes** ()

A list of all contract types.

java.util.List<Contract> **getAllContracts** ()

A list of all contracts.

java.util.List<Contract> **getActiveContracts** ()

A list of all active contracts.

`java.util.List<Contract> getOfferedContracts ()`

A list of all offered, but unaccepted, contracts.

`java.util.List<Contract> getCompletedContracts ()`

A list of all completed contracts.

`java.util.List<Contract> getFailedContracts ()`

A list of all failed contracts.

public class **Contract**

A contract. Can be accessed using `getContractManager ()`.

`String getType ()`

Type of the contract.

`String getTitle ()`

Title of the contract.

`String getDescription ()`

Description of the contract.

`String getNotes ()`

Notes for the contract.

`String getSynopsis ()`

Synopsis for the contract.

`java.util.List<String> getKeywords ()`

Keywords for the contract.

`ContractState getState ()`

State of the contract.

`boolean getSeen ()`

Whether the contract has been seen.

`boolean getRead ()`

Whether the contract has been read.

`boolean getActive ()`

Whether the contract is active.

`boolean getFailed ()`

Whether the contract has been failed.

`boolean getCanBeCanceled ()`

Whether the contract can be canceled.

`boolean getCanBeDeclined ()`

Whether the contract can be declined.

`boolean getCanBeFailed ()`

Whether the contract can be failed.

`void accept ()`

Accept an offered contract.

`void cancel ()`

Cancel an active contract.

`void decline ()`

Decline an offered contract.

```

double getFundsAdvance ()
    Funds received when accepting the contract.

double getFundsCompletion ()
    Funds received on completion of the contract.

double getFundsFailure ()
    Funds lost if the contract is failed.

double getReputationCompletion ()
    Reputation gained on completion of the contract.

double getReputationFailure ()
    Reputation lost if the contract is failed.

double getScienceCompletion ()
    Science gained on completion of the contract.

java.util.List<ContractParameter> getParameters ()
    Parameters for the contract.

public enum ContractState
    The state of a contract. See Contract.getState().

public ContractState ACTIVE
    The contract is active.

public ContractState CANCELED
    The contract has been canceled.

public ContractState COMPLETED
    The contract has been completed.

public ContractState DEADLINE_EXPIRED
    The deadline for the contract has expired.

public ContractState DECLINED
    The contract has been declined.

public ContractState FAILED
    The contract has been failed.

public ContractState GENERATED
    The contract has been generated.

public ContractState OFFERED
    The contract has been offered to the player.

public ContractState OFFER_EXPIRED
    The contract was offered to the player, but the offer expired.

public ContractState WITHDRAWN
    The contract has been withdrawn.

public class ContractParameter
    A contract parameter. See Contract.getParameters().

String getTitle ()
    Title of the parameter.

String getNotes ()
    Notes for the parameter.

```

`java.util.List<ContractParameter> getChildren ()`  
Child contract parameters.

`boolean getCompleted ()`  
Whether the parameter has been completed.

`boolean getFailed ()`  
Whether the parameter has been failed.

`boolean getOptional ()`  
Whether the contract parameter is optional.

`double getFundsCompletion ()`  
Funds received on completion of the contract parameter.

`double getFundsFailure ()`  
Funds lost if the contract parameter is failed.

`double getReputationCompletion ()`  
Reputation gained on completion of the contract parameter.

`double getReputationFailure ()`  
Reputation lost if the contract parameter is failed.

`double getScienceCompletion ()`  
Science gained on completion of the contract parameter.

## 5.3.16 Geometry Types

### Vectors

3-dimensional vectors are represented as a 3-tuple. For example:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Triplet;

import java.io.IOException;

public class Vector3 {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.newInstance(connection).getActiveVessel();
        Triplet<Double, Double, Double> v = vessel.flight(null).getPrograde();
        System.out.println(v.getValue0() + ", " + v.getValue1() + ", " + v.getValue2());
        connection.close();
    }
}
```

## Quaternions

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Quartet;

import java.io.IOException;

public class Quaternion {
    public static void main(String[] args)
        throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.newInstance(connection).getActiveVessel();
        Quartet<Double, Double, Double, Double> q = vessel.flight(null).getRotation();
        System.out.println(q.getValue0() + ", " + q.getValue1() + ", " + q.getValue2() + ", " + q.getValue3());
        connection.close();
    }
}
```

## 5.4 Drawing API

### 5.4.1 Drawing

public class **Drawing**

Provides functionality for drawing objects in the flight scene.

Line **addLine** (org.javatuples.Triplet<Double, Double, Double> start, org.javatuples.Triplet<Double, Double, Double> end, SpaceCenter.ReferenceFrame referenceFrame, boolean visible)

Draw a line in the scene.

#### Parameters

- **start** (org.javatuples.Triplet<Double, Double, Double>) – Position of the start of the line.
- **end** (org.javatuples.Triplet<Double, Double, Double>) – Position of the end of the line.
- **referenceFrame** (SpaceCenter.ReferenceFrame) – Reference frame that the positions are in.
- **visible** (boolean) – Whether the line is visible.

Line **addDirection** (org.javatuples.Triplet<Double, Double, Double> direction, SpaceCenter.ReferenceFrame referenceFrame, float length, boolean visible)

Draw a direction vector in the scene, from the center of mass of the active vessel.

### Parameters

- **direction** (*org.javatuples.Triplet<Double, Double, Double>*) – Direction to draw the line in.
- **referenceFrame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the direction is in.
- **length** (*float*) – The length of the line.
- **visible** (*boolean*) – Whether the line is visible.

*Polygon* **addPolygon** (*java.util.List<org.javatuples.Triplet<Double, Double, Double>> vertices, SpaceCenter.ReferenceFrame referenceFrame, boolean visible*)  
Draw a polygon in the scene, defined by a list of vertices.

### Parameters

- **vertices** (*java.util.List<org.javatuples.Triplet<Double, Double, Double>>*) – Vertices of the polygon.
- **referenceFrame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the vertices are in.
- **visible** (*boolean*) – Whether the polygon is visible.

*Text* **addText** (*String text, SpaceCenter.ReferenceFrame referenceFrame, org.javatuples.Triplet<Double, Double, Double> position, org.javatuples.Quartet<Double, Double, Double, Double> rotation, boolean visible*)  
Draw text in the scene.

### Parameters

- **text** (*String*) – The string to draw.
- **referenceFrame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the text position is in.
- **position** (*org.javatuples.Triplet<Double, Double, Double>*) – Position of the text.
- **rotation** (*org.javatuples.Quartet<Double, Double, Double, Double>*) – Rotation of the text, as a quaternion.
- **visible** (*boolean*) – Whether the text is visible.

*void* **clear** (*boolean clientOnly*)  
Remove all objects being drawn.

### Parameters

- **clientOnly** (*boolean*) – If true, only remove objects created by the calling client.

## 5.4.2 Line

*public class* **Line**  
A line. Created using *addLine(org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>, SpaceCenter.ReferenceFrame, boolean)*.  
*org.javatuples.Triplet<Double, Double, Double>* **getStart** ()



void **setStart** (org.javatuples.Triplet<Double, Double, Double> *value*)  
 Start position of the line.

org.javatuples.Triplet<Double, Double, Double> **getEnd** ()

void **setEnd** (org.javatuples.Triplet<Double, Double, Double> *value*)  
 End position of the line.

*SpaceCenter.ReferenceFrame* **getReferenceFrame** ()

void **setReferenceFrame** (*SpaceCenter.ReferenceFrame value*)  
 Reference frame for the positions of the object.

boolean **getVisible** ()

void **setVisible** (boolean *value*)  
 Whether the object is visible.

org.javatuples.Triplet<Double, Double, Double> **getColor** ()

void **setColor** (org.javatuples.Triplet<Double, Double, Double> *value*)  
 Set the color

*String* **getMaterial** ()

void **setMaterial** (*String value*)  
 Material used to render the object. Creates the material from a  
 shader with the given name.

float **getThickness** ()

void **setThickness** (float *value*)  
 Set the thickness

void **remove** ()  
 Remove the object.

### 5.4.3 Polygon

public class **Polygon**  
 A polygon. Created using `addPolygon(java.util.  
 List<org.javatuples.Triplet<Double, Double,  
 Double>>, SpaceCenter.ReferenceFrame,  
 boolean)`.

java.util.List<org.javatuples.Triplet<Double, Double, Double>> **getVertices** ()

void **setVertices** (java.util.List<org.javatuples.Triplet<Double, Double, Double>> *value*)  
 Vertices for the polygon.

*SpaceCenter.ReferenceFrame* **getReferenceFrame** ()

void **setReferenceFrame** (*SpaceCenter.ReferenceFrame value*)  
 Reference frame for the positions of the object.

boolean **getVisible** ()

void **setVisible** (boolean *value*)

Whether the object is visible.

void **remove** ()

Remove the object.

org.javatuples.Triplet<Double, Double, Double> **getColor** ()

void **setColor** (org.javatuples.Triplet<Double, Double, Double> *value*)

Set the color

String **getMaterial** ()

void **setMaterial** (String *value*)

Material used to render the object. Creates the material from a shader with the given name.

float **getThickness** ()

void **setThickness** (float *value*)

Set the thickness

## 5.4.4 Text

public class **Text**

Text. Created using *addText (String, SpaceCenter.ReferenceFrame, org.javatuples.Triplet<Double,Double,Double>, org.javatuples.Quartet<Double,Double,Double,Double>, boolean)*.

org.javatuples.Triplet<Double, Double, Double> **getPosition** ()

void **setPosition** (org.javatuples.Triplet<Double, Double, Double> *value*)

Position of the text.

org.javatuples.Quartet<Double, Double, Double, Double> **getRotation** ()

void **setRotation** (org.javatuples.Quartet<Double, Double, Double, Double> *value*)

Rotation of the text as a quaternion.

SpaceCenter.ReferenceFrame **getReferenceFrame** ()

void **setReferenceFrame** (SpaceCenter.ReferenceFrame *value*)

Reference frame for the positions of the object.

boolean **getVisible** ()

void **setVisible** (boolean *value*)

Whether the object is visible.

```

void remove ()
    Remove the object.

String getContent ()

void setContent (String value)
    The text string

String getFont ()

void setFont (String value)
    Name of the font

static java.util.List<String> availableFonts (Connection connection)
    A list of all available fonts.

int getSize ()

void setSize (int value)
    Font size.

float getCharacterSize ()

void setCharacterSize (float value)
    Character size.

UI.FontStyle getStyle ()

void setStyle (UI.FontStyle value)
    Font style.

org.javatuples.Triplet<Double, Double, Double> getColor ()

void setColor (org.javatuples.Triplet<Double, Double, Double> value)
    Set the color

String getMaterial ()

void setMaterial (String value)
    Material used to render the object.  Creates the material from a
    shader with the given name.

UI.TextAlignment getAlignment ()

void setAlignment (UI.TextAlignment value)
    Alignment.

float getLineSpacing ()

void setLineSpacing (float value)
    Line spacing.

UI.TextAnchor getAnchor ()

```

void **setAnchor** (*UI.TextAnchor value*)  
Anchor.

## 5.5 InfernalRobotics API

Provides RPCs to interact with the [InfernalRobotics](#) mod. Provides the following classes:

### 5.5.1 InfernalRobotics

public class **InfernalRobotics**  
This service provides functionality to interact with [InfernalRobotics](#).

boolean **getAvailable** ()  
Whether Infernal Robotics is installed.

java.util.List<*ServoGroup*> **servoGroups** (*SpaceCenter.Vessel vessel*)  
A list of all the servo groups in the given *vessel*.

#### Parameters

- **vessel** (*SpaceCenter.Vessel*) –

*ServoGroup* **servoGroupWithName** (*SpaceCenter.Vessel vessel*, [String](#) *name*)  
Returns the servo group in the given *vessel* with the given *name*, or `null` if none exists. If multiple servo groups have the same name, only one of them is returned.

#### Parameters

- **vessel** (*SpaceCenter.Vessel*) – Vessel to check.
- **name** ([String](#)) – Name of servo group to find.

*Servo* **servoWithName** (*SpaceCenter.Vessel vessel*, [String](#) *name*)  
Returns the servo in the given *vessel* with the given *name* or `null` if none exists. If multiple servos have the same name, only one of them is returned.

#### Parameters

- **vessel** (*SpaceCenter.Vessel*) – Vessel to check.
- **name** ([String](#)) – Name of the servo to find.

### 5.5.2 ServoGroup

public class **ServoGroup**  
A group of servos, obtained by calling *servoGroups* (*SpaceCenter.Vessel*) or *servoGroupWithName* (*SpaceCenter.Vessel*, *String*). Represents the “Servo Groups” in the InfernalRobotics UI.

[String](#) **getName** ()

void **setName** (*String value*)

The name of the group.

*String* **getForwardKey** ()

void **setForwardKey** (*String value*)

The key assigned to be the “forward” key for the group.

*String* **getReverseKey** ()

void **setReverseKey** (*String value*)

The key assigned to be the “reverse” key for the group.

float **getSpeed** ()

void **setSpeed** (float *value*)

The speed multiplier for the group.

boolean **getExpanded** ()

void **setExpanded** (boolean *value*)

Whether the group is expanded in the InfernalRobotics UI.

java.util.List<*Servo*> **getServos** ()

The servos that are in the group.

*Servo* **servoWithName** (*String name*)

Returns the servo with the given *name* from this group, or null if none exists.

#### Parameters

- **name** (*String*) – Name of servo to find.

java.util.List<*SpaceCenter.Part*> **getParts** ()

The parts containing the servos in the group.

void **moveRight** ()

Moves all of the servos in the group to the right.

void **moveLeft** ()

Moves all of the servos in the group to the left.

void **moveCenter** ()

Moves all of the servos in the group to the center.

void **moveNextPreset** ()

Moves all of the servos in the group to the next preset.

void **movePrevPreset** ()

Moves all of the servos in the group to the previous preset.

void **stop** ()

Stops the servos in the group.

### 5.5.3 Servo

public class **Servo**

Represents a servo. Obtained using *ServoGroup*.

*getServos()*, *ServoGroup.servoWithName(String)*  
or *servoWithName(SpaceCenter.Vessel, String)*.

**String getName()**

**void setName(String value)**

The name of the servo.

**SpaceCenter.Part getPart()**

The part containing the servo.

**void setHighlight(boolean value)**

Whether the servo should be highlighted in-game.

**float getPosition()**

The position of the servo.

**float getMinConfigPosition()**

The minimum position of the servo, specified by the part configuration.

**float getMaxConfigPosition()**

The maximum position of the servo, specified by the part configuration.

**float getMinPosition()**

**void setMinPosition(float value)**

The minimum position of the servo, specified by the in-game tweak menu.

**float getMaxPosition()**

**void setMaxPosition(float value)**

The maximum position of the servo, specified by the in-game tweak menu.

**float getConfigSpeed()**

The speed multiplier of the servo, specified by the part configuration.

**float getSpeed()**

**void setSpeed(float value)**

The speed multiplier of the servo, specified by the in-game tweak menu.

**float getCurrentSpeed()**

**void setCurrentSpeed(float value)**

The current speed at which the servo is moving.

**float getAcceleration()**

**void setAcceleration(float value)**

The current speed multiplier set in the UI.

boolean **getIsMoving** ()

Whether the servo is moving.

boolean **getIsFreeMoving** ()

Whether the servo is freely moving.

boolean **getIsLocked** ()

void **setIsLocked** (boolean *value*)

Whether the servo is locked.

boolean **getIsAxisInverted** ()

void **setIsAxisInverted** (boolean *value*)

Whether the servos axis is inverted.

void **moveRight** ()

Moves the servo to the right.

void **moveLeft** ()

Moves the servo to the left.

void **moveCenter** ()

Moves the servo to the center.

void **moveNextPreset** ()

Moves the servo to the next preset.

void **movePrevPreset** ()

Moves the servo to the previous preset.

void **moveTo** (float *position*, float *speed*)

Moves the servo to *position* and sets the speed multiplier to *speed*.

#### Parameters

- **position** (*float*) – The position to move the servo to.
- **speed** (*float*) – Speed multiplier for the movement.

void **stop** ()

Stops the servo.

### 5.5.4 Example

The following example gets the control group named “MyGroup”, prints out the names and positions of all of the servos in the group, then moves all of the servos to the right for 1 second.

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.InfernalRobotics;
import
↳krpc.client.services.InfernalRobotics.Servo;
import krpc.
↳client.services.InfernalRobotics.ServoGroup;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;
```

```

import java.io.IOException;

public class InfernalRoboticsExample {
    public static
    ↪void main(String[] args) throws IOException,
    ↪RPCException, InterruptedException {
        Connection connection = Connection.
    ↪newInstance("InfernalRobotics Example");
        Vessel vessel = SpaceCenter.
    ↪newInstance(connection).getActiveVessel();
        InfernalRobotics
    ↪ir = InfernalRobotics.newInstance(connection);

        ServoGroup group
    ↪= ir.servoGroupWithName(vessel, "MyGroup");
        if (group == null) {
            System.out.println("Group not found");
            return;
        }

        for (Servo servo : group.getServos()) {
            System.out.println(servo.
    ↪getName() + " " + servo.getPosition());
        }

        group.moveRight();
        Thread.sleep(1000);
        group.stop();
        connection.close();
    }
}

```

## 5.6 Kerbal Alarm Clock API

Provides RPCs to interact with the [Kerbal Alarm Clock](#) mod. Provides the following classes:

### 5.6.1 KerbalAlarmClock

public class **KerbalAlarmClock**

This service provides functionality to interact with [Kerbal Alarm Clock](#).

boolean **getAvailable()**

Whether Kerbal Alarm Clock is available.

java.util.List<Alarm> **getAlarms()**

A list of all the alarms.

Alarm **alarmWithName(String name)**

Get the alarm with the given *name*, or null if no alarms have that name. If more than one alarm has the name, only returns one of them.

#### Parameters



- **name** (*String*) – Name of the alarm to search for.

`java.util.List<Alarm> alarmsWithType (AlarmType type)`

Get a list of alarms of the specified *type*.

#### Parameters

- **type** (*AlarmType*) – Type of alarm to return.

*Alarm* **createAlarm** (*AlarmType type*, *String name*, *double ut*)

Create a new alarm and return it.

#### Parameters

- **type** (*AlarmType*) – Type of the new alarm.
- **name** (*String*) – Name of the new alarm.
- **ut** (*double*) – Time at which the new alarm should trigger.

## 5.6.2 Alarm

public class **Alarm**

Represents an alarm. Obtained by calling  
`getAlarms()`, `alarmWithName (String)` or  
`alarmsWithType (AlarmType)`.

*AlarmAction* **getAction** ()

void **setAction** (*AlarmAction value*)

The action that the alarm triggers.

double **getMargin** ()

void **setMargin** (*double value*)

The number of seconds before the event that the alarm will fire.

double **getTime** ()

void **setTime** (*double value*)

The time at which the alarm will fire.

*AlarmType* **getType** ()

The type of the alarm.

*String* **getID** ()

The unique identifier for the alarm.

*String* **getName** ()

void **setName** (*String value*)

The short name of the alarm.

*String* **getNotes** ()

void **setNotes** (*String value*)

The long description of the alarm.

double **getRemaining** ()  
The number of seconds until the alarm will fire.

boolean **getRepeat** ()

void **setRepeat** (boolean *value*)  
Whether the alarm will be repeated after it has fired.

double **getRepeatPeriod** ()

void **setRepeatPeriod** (double *value*)  
The time delay to automatically create an alarm after it has fired.

*SpaceCenter.Vessel* **getVessel** ()

void **setVessel** (*SpaceCenter.Vessel value*)  
The vessel that the alarm is attached to.

*SpaceCenter.CelestialBody* **getXferOriginBody** ()

void **setXferOriginBody** (*SpaceCenter.CelestialBody value*)  
The celestial body the vessel is departing from.

*SpaceCenter.CelestialBody* **getXferTargetBody** ()

void **setXferTargetBody** (*SpaceCenter.CelestialBody value*)  
The celestial body the vessel is arriving at.

void **remove** ()  
Removes the alarm.

### 5.6.3 AlarmType

public enum **AlarmType**  
The type of an alarm.

public *AlarmType* **RAW**  
An alarm for a specific date/time or a specific period in the future.

public *AlarmType* **MANEUVER**  
An alarm based on the next maneuver node on the current ships flight path. This node will be stored and can be restored when you come back to the ship.

public *AlarmType* **MANEUVER\_AUTO**  
See *AlarmType.MANEUVER*.

public *AlarmType* **APOAPSIS**  
An alarm for furthest part of the orbit from the planet.

public *AlarmType* **PERIAPSIS**  
An alarm for nearest part of the orbit from the planet.

public *AlarmType* **ASCENDING\_NODE**  
Ascending node for the targeted object, or equatorial ascending node.

public *AlarmType* **DESCENDING\_NODE**  
 Descending node for the targeted object, or equatorial descending node.

public *AlarmType* **CLOSEST**  
 An alarm based on the closest approach of this vessel to the targeted vessel, some number of orbits into the future.

public *AlarmType* **CONTRACT**  
 An alarm based on the expiry or deadline of contracts in career modes.

public *AlarmType* **CONTRACT\_AUTO**  
 See *AlarmType.CONTRACT*.

public *AlarmType* **CREW**  
 An alarm that is attached to a crew member.

public *AlarmType* **DISTANCE**  
 An alarm that is triggered when a selected target comes within a chosen distance.

public *AlarmType* **EARTH\_TIME**  
 An alarm based on the time in the “Earth” alternative Universe (aka the Real World).

public *AlarmType* **LAUNCH\_RENDEVOUS**  
 An alarm that fires as your landed craft passes under the orbit of your target.

public *AlarmType* **SOI\_CHANGE**  
 An alarm manually based on when the next SOI point is on the flight path or set to continually monitor the active flight path and add alarms as it detects SOI changes.

public *AlarmType* **SOI\_CHANGE\_AUTO**  
 See *AlarmType.SOI\_CHANGE*.

public *AlarmType* **TRANSFER**  
 An alarm based on Interplanetary Transfer Phase Angles, i.e. when should I launch to planet X? Based on Kosmo Not’s post and used in Olex’s Calculator.

public *AlarmType* **TRANSFER\_MODELLED**  
 See *AlarmType.TRANSFER*.

### 5.6.4 AlarmAction

public enum **AlarmAction**  
 The action performed by an alarm when it fires.

public *AlarmAction* **DO\_NOTHING**  
 Don’t do anything at all. . .

public *AlarmAction* **DO\_NOTHING\_DELETE\_WHEN\_PASSED**  
 Don’t do anything, and delete the alarm.

public *AlarmAction* **KILL\_WARP**  
 Drop out of time warp.

public *AlarmAction* **KILL\_WARP\_ONLY**

Drop out of time warp.

public *AlarmAction* **MESSAGE\_ONLY**

Display a message.

public *AlarmAction* **PAUSE\_GAME**

Pause the game.

## 5.6.5 Example

The following example creates a new alarm for the active vessel. The alarm is set to trigger after 10 seconds have passed, and display a message.

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KerbalAlarmClock;
import _
↳krpc.client.services.KerbalAlarmClock.Alarm;
import krpc.
↳client.services.KerbalAlarmClock.AlarmAction;
import _
↳krpc.client.services.KerbalAlarmClock.AlarmType;
import krpc.client.services.SpaceCenter;

import java.io.IOException;

public class KerbalAlarmClockExample {
    public static void main(String[] _
↳args) throws IOException, RPCException {
        Connection connection = Connection.
↳newInstance("Kerbal Alarm Clock Example");
        KerbalAlarmClock _
↳kac = KerbalAlarmClock.newInstance(connection);
        Alarm alarm = kac.createAlarm(AlarmType.
↳RAW, "My New Alarm", SpaceCenter.
↳newInstance(connection).getUT() + 10);
        alarm.setNotes("10 seconds _
↳have now passed since the alarm was created.");
        alarm.setAction(AlarmAction.MESSAGE_ONLY);
        connection.close();
    }
}
```

## 5.7 RemoteTech API

Provides RPCs to interact with the `RemoteTech` mod. Provides the following classes:

### 5.7.1 RemoteTech

public class **RemoteTech**

This service provides functionality to interact with `RemoteTech`.

boolean **getAvailable** ()

Whether RemoteTech is installed.

java.util.List<String> **getGroundStations** ()

The names of the ground stations.

*Antenna* **antenna** (*SpaceCenter.Part part*)

Get the antenna object for a particular part.

#### Parameters

- **part** (*SpaceCenter.Part*) –

*Comms* **comms** (*SpaceCenter.Vessel vessel*)

Get a communications object, representing the communication capability of a particular vessel.

#### Parameters

- **vessel** (*SpaceCenter.Vessel*) –

## 5.7.2 Comms

public class **Comms**

Communications for a vessel.

*SpaceCenter.Vessel* **getVessel** ()

Get the vessel.

boolean **getHasLocalControl** ()

Whether the vessel can be controlled locally.

boolean **getHasFlightComputer** ()

Whether the vessel has a flight computer on board.

boolean **getHasConnection** ()

Whether the vessel has any connection.

boolean **getHasConnectionToGroundStation** ()

Whether the vessel has a connection to a ground station.

double **getSignalDelay** ()

The shortest signal delay to the vessel, in seconds.

double **getSignalDelayToGroundStation** ()

The signal delay between the vessel and the closest ground station, in seconds.

double **signalDelayToVessel** (*SpaceCenter.Vessel other*)

The signal delay between the this vessel and another vessel, in seconds.

#### Parameters

- **other** (*SpaceCenter.Vessel*) –

java.util.List<*Antenna*> **getAntennas** ()

The antennas for this vessel.

### 5.7.3 Antenna

public class **Antenna**  
A RemoteTech antenna. Obtained by calling *Comms.getAntennas()* or *antenna(SpaceCenter.Part)*.

*SpaceCenter.Part* **getPart** ()  
Get the part containing this antenna.

boolean **getHasConnection** ()  
Whether the antenna has a connection.

*Target* **getTarget** ()

void **setTarget** (*Target value*)  
The object that the antenna is targetting. This property can be used to set the target to *Target.NONE* or *Target.ACTIVE\_VESSEL*. To set the target to a celestial body, ground station or vessel see *Antenna.getTargetBody()*, *Antenna.getTargetGroundStation()* and *Antenna.getTargetVessel()*.

*SpaceCenter.CelestialBody* **getTargetBody** ()

void **setTargetBody** (*SpaceCenter.CelestialBody value*)  
The celestial body the antenna is targetting.

*String* **getTargetGroundStation** ()

void **setTargetGroundStation** (*String value*)  
The ground station the antenna is targetting.

*SpaceCenter.Vessel* **getTargetVessel** ()

void **setTargetVessel** (*SpaceCenter.Vessel value*)  
The vessel the antenna is targetting.

public enum **Target**  
The type of object an antenna is targetting. See *Antenna.getTarget()*.

public *Target* **ACTIVE\_VESSEL**  
The active vessel.

public *Target* **CELESTIAL\_BODY**  
A celestial body.

public *Target* **GROUND\_STATION**  
A ground station.

public *Target* **VESSEL**  
A specific vessel.

public *Target* **NONE**  
No target.

## 5.7.4 Example

The following example sets the target of a dish on the active vessel then prints out the signal delay to the active vessel.

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.RemoteTech;
import krpc.client.services.RemoteTech.Antenna;
import krpc.client.services.RemoteTech.Comms;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Part;
import krpc.client.services.SpaceCenter.Vessel;

import java.io.IOException;

public class RemoteTechExample {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("RemoteTech Example");
        SpaceCenter sc = SpaceCenter.newInstance(connection);
        RemoteTech rt = RemoteTech.newInstance(connection);
        Vessel vessel = sc.getActiveVessel();

        // Set a dish target
        Part part = vessel.getParts().withTitle("Reflectron KR-7").get(0);
        Antenna antenna = rt.antenna(part);

        setTargetBody(sc.getBodies().get("Jool"));

        // Get info about the vessels communications
        Comms comms = rt.comms(vessel);
        System.out.printf("Signal delay\n", comms.getSignalDelay());
        connection.close();
    }
}
```

## 5.8 User Interface API

### 5.8.1 UI

public class **UI**

Provides functionality for drawing and interacting with in-game user interface elements.

Canvas **getStockCanvas()**

The stock UI canvas.

Canvas **addCanvas()**

Add a new canvas.

---

**Note:** If you want to add UI elements to KSPs stock UI canvas, use `getStockCanvas()`.

---

void **message** (*String* content, float duration, *MessagePosition* position)  
Display a message on the screen.

**Parameters**

- **content** (*String*) – Message content.
- **duration** (*float*) – Duration before the message disappears, in seconds.
- **position** (*MessagePosition*) – Position to display the message.

---

**Note:** The message appears just like a stock message, for example quicksave or quickload messages.

---

void **clear** (boolean clientOnly)  
Remove all user interface elements.

**Parameters**

- **clientOnly** (*boolean*) – If true, only remove objects created by the calling client.

public enum **MessagePosition**  
Message position.

public *MessagePosition* **TOP\_LEFT**  
Top left.

public *MessagePosition* **TOP\_CENTER**  
Top center.

public *MessagePosition* **TOP\_RIGHT**  
Top right.

public *MessagePosition* **BOTTOM\_CENTER**  
Bottom center.

## 5.8.2 Canvas

public class **Canvas**  
A canvas for user interface elements. See `getStockCanvas()` and `addCanvas()`.

*RectTransform* **getRectTransform()**  
The rect transform for the canvas.

boolean **getVisible()**

void **setVisible** (boolean value)  
Whether the UI object is visible.

*Panel* **addPanel** (boolean visible)  
Create a new container for user interface elements.

**Parameters**



- **visible** (*boolean*) – Whether the panel is visible.

*Text* **addText** (*String content*, *boolean visible*)  
Add text to the canvas.

#### Parameters

- **content** (*String*) – The text.
- **visible** (*boolean*) – Whether the text is visible.

*InputField* **addInputField** (*boolean visible*)  
Add an input field to the canvas.

#### Parameters

- **visible** (*boolean*) – Whether the input field is visible.

*Button* **addButton** (*String content*, *boolean visible*)  
Add a button to the canvas.

#### Parameters

- **content** (*String*) – The label for the button.
- **visible** (*boolean*) – Whether the button is visible.

*void* **remove** ()  
Remove the UI object.

### 5.8.3 Panel

public class **Panel**  
A container for user interface elements. See *Canvas*.  
*addPanel* (*boolean*).

*RectTransform* **getRectTransform** ()  
The rect transform for the panel.

*boolean* **getVisible** ()

*void* **setVisible** (*boolean value*)  
Whether the UI object is visible.

*Panel* **addPanel** (*boolean visible*)  
Create a panel within this panel.

#### Parameters

- **visible** (*boolean*) – Whether the new panel is visible.

*Text* **addText** (*String content*, *boolean visible*)  
Add text to the panel.

#### Parameters

- **content** (*String*) – The text.
- **visible** (*boolean*) – Whether the text is visible.

*InputField* **addInputField** (*boolean visible*)  
Add an input field to the panel.

#### Parameters

- **visible** (*boolean*) – Whether the input field is visible.

Button **addButton** (*String content*, *boolean visible*)

Add a button to the panel.

#### Parameters

- **content** (*String*) – The label for the button.
- **visible** (*boolean*) – Whether the button is visible.

void **remove** ()

Remove the UI object.

### 5.8.4 Text

public class **Text**

A text label. See *Panel.addText (String, boolean)*.

*RectTransform* **getRectTransform** ()

The rect transform for the text.

boolean **getVisible** ()

void **setVisible** (*boolean value*)

Whether the UI object is visible.

*String* **getContent** ()

void **setContent** (*String value*)

The text string

*String* **getFont** ()

void **setFont** (*String value*)

Name of the font

*java.util.List<String>* **getAvailableFonts** ()

A list of all available fonts.

int **getSize** ()

void **setSize** (*int value*)

Font size.

*FontStyle* **getStyle** ()

void **setStyle** (*FontStyle value*)

Font style.

*org.javatuples.Triplet<Double, Double, Double>* **getColor** ()

void **setColor** (*org.javatuples.Triplet<Double, Double, Double> value*)

Set the color

*TextAnchor* **getAlignment** ()

```

void setAlignment (TextAnchor value)
    Alignment.

float getLineSpacing ()

void setLineSpacing (float value)
    Line spacing.

void remove ()
    Remove the UI object.

public enum FontStyle
    Font style.

public FontStyle NORMAL
    Normal.

public FontStyle BOLD
    Bold.

public FontStyle ITALIC
    Italic.

public FontStyle BOLD_AND_ITALIC
    Bold and italic.

public enum TextAlignment
    Text alignment.

public TextAlignment LEFT
    Left aligned.

public TextAlignment RIGHT
    Right aligned.

public TextAlignment CENTER
    Center aligned.

public enum TextAnchor
    Text alignment.

public TextAnchor LOWER_CENTER
    Lower center.

public TextAnchor LOWER_LEFT
    Lower left.

public TextAnchor LOWER_RIGHT
    Lower right.

public TextAnchor MIDDLE_CENTER
    Middle center.

public TextAnchor MIDDLE_LEFT
    Middle left.

public TextAnchor MIDDLE_RIGHT
    Middle right.

public TextAnchor UPPER_CENTER
    Upper center.

```

public *TextAnchor* **UPPER\_LEFT**  
Upper left.

public *TextAnchor* **UPPER\_RIGHT**  
Upper right.

### 5.8.5 Button

public class **Button**  
A text label. See *Panel.addButton(String, boolean)*.

*RectTransform* **getRectTransform()**  
The rect transform for the text.

boolean **getVisible()**

void **setVisible**(boolean *value*)  
Whether the UI object is visible.

*Text* **getText()**  
The text for the button.

boolean **getClicked()**

void **setClicked**(boolean *value*)  
Whether the button has been clicked.

---

**Note:** This property is set to true when the user clicks the button.  
A client script should reset the property to false in order to detect subsequent button presses.

---

void **remove()**  
Remove the UI object.

### 5.8.6 InputField

public class **InputField**  
An input field. See *Panel.addInputField(boolean)*.

*RectTransform* **getRectTransform()**  
The rect transform for the input field.

boolean **getVisible()**

void **setVisible**(boolean *value*)  
Whether the UI object is visible.

*String* **getValue()**

void **setValue**(*String value*)  
The value of the input field.

*Text* **getText** ()

The text component of the input field.

---

**Note:** Use *InputField.getValue()* to get and set the value in the field. This object can be used to alter the style of the input field's text.

---

boolean **getChanged** ()

void **setChanged** (boolean *value*)

Whether the input field has been changed.

---

**Note:** This property is set to true when the user modifies the value of the input field. A client script should reset the property to false in order to detect subsequent changes.

---

void **remove** ()

Remove the UI object.

### 5.8.7 Rect Transform

public class **RectTransform**

A Unity engine Rect Transform for a UI object. See the [Unity manual](#) for more details.

org.javatuples.[Pair](#)<[Double](#), [Double](#)> **getPosition** ()

void **setPosition** (org.javatuples.[Pair](#)<[Double](#), [Double](#)> *value*)

Position of the rectangles pivot point relative to the anchors.

org.javatuples.[Triplet](#)<[Double](#), [Double](#), [Double](#)> **getLocalPosition** ()

void **setLocalPosition** (org.javatuples.[Triplet](#)<[Double](#), [Double](#), [Double](#)> *value*)

Position of the rectangles pivot point relative to the anchors.

org.javatuples.[Pair](#)<[Double](#), [Double](#)> **getSize** ()

void **setSize** (org.javatuples.[Pair](#)<[Double](#), [Double](#)> *value*)

Width and height of the rectangle.

org.javatuples.[Pair](#)<[Double](#), [Double](#)> **getUpperRight** ()

void **setUpperRight** (org.javatuples.[Pair](#)<[Double](#), [Double](#)> *value*)

Position of the rectangles upper right corner relative to the anchors.

org.javatuples.[Pair](#)<[Double](#), [Double](#)> **getLowerLeft** ()

void **setLowerLeft** (org.javatuples.[Pair](#)<[Double](#), [Double](#)> *value*)

Position of the rectangles lower left corner relative to the anchors.

void **setAnchor** (org.javatuples.Pair<Double, Double> *value*)  
Set the minimum and maximum anchor points as a fraction of the size of the parent rectangle.

org.javatuples.Pair<Double, Double> **getAnchorMax** ()

void **setAnchorMax** (org.javatuples.Pair<Double, Double> *value*)  
The anchor point for the lower left corner of the rectangle defined as a fraction of the size of the parent rectangle.

org.javatuples.Pair<Double, Double> **getAnchorMin** ()

void **setAnchorMin** (org.javatuples.Pair<Double, Double> *value*)  
The anchor point for the upper right corner of the rectangle defined as a fraction of the size of the parent rectangle.

org.javatuples.Pair<Double, Double> **getPivot** ()

void **setPivot** (org.javatuples.Pair<Double, Double> *value*)  
Location of the pivot point around which the rectangle rotates, defined as a fraction of the size of the rectangle itself.

org.javatuples.Quartet<Double, Double, Double, Double> **getRotation** ()

void **setRotation** (org.javatuples.Quartet<Double, Double, Double, Double> *value*)  
Rotation, as a quaternion, of the object around its pivot point.

org.javatuples.Triplet<Double, Double, Double> **getScale** ()

void **setScale** (org.javatuples.Triplet<Double, Double, Double> *value*)  
Scale factor applied to the object in the x, y and z dimensions.

## 6.1 Lua Client

This client provides functionality to interact with a kRPC server from programs written in Lua. It can be [installed](#) using [LuaRocks](#) or downloaded from [GitHub](#).

### 6.1.1 Installing the Library

The Lua client and all of its dependencies can be installed using `luarocks` with a single command:

```
luarocks install krpc
```

### 6.1.2 Using the Library

Once it's installed, simply `require 'krpc'` and you are good to go!

### 6.1.3 Connecting to the Server

To connect to a server, use the `krpc.connect()` function. This returns a connection object through which you can interact with the server. For example to connect to a server running on the local machine:

```
local krpc = require 'krpc'  
local conn = krpc.connect('Example')  
print(conn.krpc:get_status().version)
```

This function also accepts arguments that specify what address and port numbers to connect to. For example:

```
local krpc = require 'krpc'  
local conn = krpc.connect('Remote example', 'my.domain.name', 1000, 1001)  
print(conn.krpc:get_status().version)
```

### 6.1.4 Interacting with the Server

Interaction with the server is performed via the client object (of type `krpc.Client`) returned when connecting to the server using `krpc.connect()`.

Upon connecting, the client interrogates the server to find out what functionality it provides and dynamically adds all of the classes, methods, properties to the client object.

For example, all of the functionality provided by the SpaceCenter service is accessible via `conn.space_center` and the functionality provided by the InfernalRobotics service is accessible via `conn.infernal_robotics`.

Calling methods, getting or setting properties, etc. are mapped to remote procedure calls and passed to the server by the lua client.

## 6.1.5 Streaming Data from the Server

Streams are not yet supported by the Lua client.

## 6.1.6 Reference

**connect** (*[name=nil]*, *[address='127.0.0.1']*, *[rpc\_port=50000]*, *[stream\_port=50001]*)

This function creates a connection to a kRPC server. It returns a *krpc.Client* object, through which the server can be communicated with.

### Parameters

- **name** (*string*) – A descriptive name for the connection. This is passed to the server and appears, for example, in the client connection dialog on the in-game server window.
- **address** (*string*) – The address of the server to connect to. Can either be a hostname or an IP address in dotted decimal notation. Defaults to '127.0.0.1'.
- **rpc\_port** (*number*) – The port number of the RPC Server. Defaults to 50000.
- **stream\_port** (*number*) – The port number of the Stream Server. Defaults to 50001.

### class Client

This class provides the interface for communicating with the server. It is dynamically populated with all the functionality provided by the server. Instances of this class should be obtained by calling *krpc.connect()*.

#### close()

Closes the connection to the server.

### krpc

The built-in KRPC class, providing basic interactions with the server.

**Return type** *krpc.KRPC*

### class KRPC

This class provides access to the basic server functionality provided by the KRPC service. An instance can be obtained by calling *krpc.Client.krpc*. Most of this functionality is used internally by the lua client and therefore does not need to be used directly from application code. The only exception that may be useful is:

#### get\_status()

Gets a status message from the server containing information including the server's version string and performance statistics.

For example, the following prints out the version string for the server:

```
local krpc = require 'krpc'
local conn = krpc.connect()
print('Server version = ' .. conn.krpc:get_status().version)
```

Or to get the rate at which the server is sending and receiving data over the network:



```

local krpc = require 'krpc'
local conn = krpc.connect()
local status = conn.krpc.get_status()
print(string.format('Data in = %.2f KB/s', status.bytes_read_rate/1024))
print(string.format('Data out = %.2f KB/s', status.bytes_written_rate/1024))

```

## 6.2 KRPC API

### 6.2.1 KRPC

None None None None Main kRPC service, used by clients to interact with basic server functionality.

**static** `get_client_id()`

Returns the identifier for the current client.

**Return type** string

**static** `get_client_name()`

Returns the name of the current client. This is an empty string if the client has no name.

**Return type** string

**clients**

A list of RPC clients that are currently connected to the server. Each entry in the list is a clients identifier, name and address.

**Attribute** Read-only, cannot be set

**Return type** List of Tuple of (string, string, string)

**static** `get_status()`

Returns some information about the server, such as the version.

**Return type** `krpc.schema.KRPC.Status`

**static** `get_services()`

Returns information on all services, procedures, classes, properties etc. provided by the server. Can be used by client libraries to automatically create functionality such as stubs.

**Return type** `krpc.schema.KRPC.Services`

**current\_game\_scene**

Get the current game scene.

**Attribute** Read-only, cannot be set

**Return type** `KRPC.GameScene`

**paused**

Whether the game is paused.

**Attribute** Can be read or written

**Return type** boolean

**class** `GameScene`

The game scene. See `KRPC.current_game_scene`.

**space\_center**

The game scene showing the Kerbal Space Center buildings.

**flight**

The game scene showing a vessel in flight (or on the launchpad/runway).

**tracking\_station**

The tracking station.

**editor\_vab**

The Vehicle Assembly Building.

**editor\_sph**

The Space Plane Hangar.

**class InvalidOperationException**

A method call was made to a method that is invalid given the current state of the object.

**class ArgumentException**

A method was invoked where at least one of the passed arguments does not meet the parameter specification of the method.

**class ArgumentNullException**

A null reference was passed to a method that does not accept it as a valid argument.

**class ArgumentOutOfRangeException**

The value of an argument is outside the allowable range of values as defined by the invoked method.

## 6.2.2 Expressions

**class Expression**

A server side expression.

**static constant\_double** (*value*)

A constant value of type double.

**Parameters** *value* (*number*) –

**Return type** *KRPC.Expression*

**static constant\_float** (*value*)

A constant value of type float.

**Parameters** *value* (*number*) –

**Return type** *KRPC.Expression*

**static constant\_int** (*value*)

A constant value of type int.

**Parameters** *value* (*number*) –

**Return type** *KRPC.Expression*

**static constant\_string** (*value*)

A constant value of type string.

**Parameters** *value* (*string*) –

**Return type** *KRPC.Expression*

**static call** (*call*)

An RPC call.

**Parameters** *call* (*krcp.schema.KRPC.ProcedureCall*) –

**Return type** *KRPC.Expression*

**static equal** (*arg0*, *arg1*)

Equality comparison.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static not\_equal** (*arg0*, *arg1*)

Inequality comparison.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static greater\_than** (*arg0*, *arg1*)

Greater than numerical comparison.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static greater\_than\_or\_equal** (*arg0*, *arg1*)

Greater than or equal numerical comparison.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static less\_than** (*arg0*, *arg1*)

Less than numerical comparison.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static less\_than\_or\_equal** (*arg0*, *arg1*)

Less than or equal numerical comparison.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static and** (*arg0*, *arg1*)

Boolean and operator.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression***static or** (*arg0, arg1*)

Boolean or operator.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression***static exclusive\_or** (*arg0, arg1*)

Boolean exclusive-or operator.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression***static not** (*arg*)

Boolean negation operator.

**Parameters** **arg** (KRPC.Expression) –**Return type** *KRPC.Expression***static add** (*arg0, arg1*)

Numerical addition.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression***static subtract** (*arg0, arg1*)

Numerical subtraction.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression***static multiply** (*arg0, arg1*)

Numerical multiplication.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static divide** (*arg0*, *arg1*)

Numerical division.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static modulo** (*arg0*, *arg1*)

Numerical modulo operator.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Returns** The remainder of *arg0* divided by *arg1*

**Return type** *KRPC.Expression*

**static power** (*arg0*, *arg1*)

Numerical power operator.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Returns** *arg0* raised to the power of *arg1*

**Return type** *KRPC.Expression*

**static left\_shift** (*arg0*, *arg1*)

Bitwise left shift.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static right\_shift** (*arg0*, *arg1*)

Bitwise right shift.

**Parameters**

- **arg0** (KRPC.Expression) –
- **arg1** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static to\_double** (*arg*)

Convert to a double type.

**Parameters** **arg** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static to\_float** (*arg*)

Convert to a float type.

**Parameters** **arg** (KRPC.Expression) –

**Return type** *KRPC.Expression*

**static to\_int** (*arg*)

Convert to an int type.

**Parameters** **arg** (KRPC.Expression) –

**Return type** *KRPC.Expression*

## 6.3 SpaceCenter API

### 6.3.1 SpaceCenter

Provides functionality to interact with Kerbal Space Program. This includes controlling the active vessel, managing its resources, planning maneuver nodes and auto-piloting.

**active\_vessel**

The currently active vessel.

**Attribute** Can be read or written

**Return type** *SpaceCenter.Vessel*

**vessels**

A list of all the vessels in the game.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Vessel*

**bodies**

A dictionary of all celestial bodies (planets, moons, etc.) in the game, keyed by the name of the body.

**Attribute** Read-only, cannot be set

**Return type** Map from string to *SpaceCenter.CelestialBody*

**target\_body**

The currently targeted celestial body.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CelestialBody*

**target\_vessel**

The currently targeted vessel.

**Attribute** Can be read or written

**Return type** *SpaceCenter.Vessel*

**target\_docking\_port**

The currently targeted docking port.

**Attribute** Can be read or written

**Return type** *SpaceCenter.DockingPort*

**static clear\_target** ()

Clears the current target.

---

**static launchable\_vessels** (*craft\_directory*)

Returns a list of vessels from the given *craft\_directory* that can be launched.

**Parameters** **craft\_directory** (*string*) – Name of the directory in the current saves “Ships” directory. For example "VAB" or "SPH".

**Return type** List of string

**static launch\_vessel** (*craft\_directory*, *name*, *launch\_site*)

Launch a vessel.

**Parameters**

- **craft\_directory** (*string*) – Name of the directory in the current saves “Ships” directory, that contains the craft file. For example "VAB" or "SPH".
- **name** (*string*) – Name of the vessel to launch. This is the name of the “.craft” file in the save directory, without the “.craft” file extension.
- **launch\_site** (*string*) – Name of the launch site. For example "LaunchPad" or "Runway".

**static launch\_vessel\_from\_vab** (*name*)

Launch a new vessel from the VAB onto the launchpad.

**Parameters** **name** (*string*) – Name of the vessel to launch.

---

**Note:** This is equivalent to calling *SpaceCenter.launch\_vessel()* with the craft directory set to “VAB” and the launch site set to “LaunchPad”.

---

**static launch\_vessel\_from\_sph** (*name*)

Launch a new vessel from the SPH onto the runway.

**Parameters** **name** (*string*) – Name of the vessel to launch.

---

**Note:** This is equivalent to calling *SpaceCenter.launch\_vessel()* with the craft directory set to “SPH” and the launch site set to “Runway”.

---

**static save** (*name*)

Save the game with a given name. This will create a save file called *name.sfs* in the folder of the current save game.

**Parameters** **name** (*string*) –

**static load** (*name*)

Load the game with the given name. This will create a load a save file called *name.sfs* from the folder of the current save game.

**Parameters** **name** (*string*) –

**static quicksave** ()

Save a quicksave.

---

**Note:** This is the same as calling *SpaceCenter.save()* with the name “quicksave”.

---

**static quickload** ()

Load a quicksave.

---

**Note:** This is the same as calling `SpaceCenter.load()` with the name “quicksave”.

---

**ui\_visible**

Whether the UI is visible.

**Attribute** Can be read or written

**Return type** boolean

**navball**

Whether the navball is visible.

**Attribute** Can be read or written

**Return type** boolean

**ut**

The current universal time in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**g**

The value of the [gravitational constant](#)  $G$  in  $N(m/kg)^2$ .

**Attribute** Read-only, cannot be set

**Return type** number

**warp\_rate**

The current warp rate. This is the rate at which time is passing for either on-rails or physical time warp. For example, a value of 10 means time is passing 10x faster than normal. Returns 1 if time warp is not active.

**Attribute** Read-only, cannot be set

**Return type** number

**warp\_factor**

The current warp factor. This is the index of the rate at which time is passing for either regular “on-rails” or physical time warp. Returns 0 if time warp is not active. When in on-rails time warp, this is equal to `SpaceCenter.rails_warp_factor`, and in physics time warp, this is equal to `SpaceCenter.physics_warp_factor`.

**Attribute** Read-only, cannot be set

**Return type** number

**rails\_warp\_factor**

The time warp rate, using regular “on-rails” time warp. A value between 0 and 7 inclusive. 0 means no time warp. Returns 0 if physical time warp is active.

If requested time warp factor cannot be set, it will be set to the next lowest possible value. For example, if the vessel is too close to a planet. See [the KSP wiki](#) for details.

**Attribute** Can be read or written

**Return type** number

**physics\_warp\_factor**

The physical time warp rate. A value between 0 and 3 inclusive. 0 means no time warp. Returns 0 if regular “on-rails” time warp is active.

**Attribute** Can be read or written



**Return type** number

**static can\_rails\_warp\_at** (*[factor = 1]*)

Returns `True` if regular “on-rails” time warp can be used, at the specified warp *factor*. The maximum time warp rate is limited by various things, including how close the active vessel is to a planet. See [the KSP wiki](#) for details.

**Parameters** **factor** (*number*) – The warp factor to check.

**Return type** boolean

**maximum\_rails\_warp\_factor**

The current maximum regular “on-rails” warp factor that can be set. A value between 0 and 7 inclusive. See [the KSP wiki](#) for details.

**Attribute** Read-only, cannot be set

**Return type** number

**static warp\_to** (*ut* [*, max\_rails\_rate = 100000.0*] [*, max\_physics\_rate = 2.0*])

Uses time acceleration to warp forward to a time in the future, specified by universal time *ut*. This call blocks until the desired time is reached. Uses regular “on-rails” or physical time warp as appropriate. For example, physical time warp is used when the active vessel is traveling through an atmosphere. When using regular “on-rails” time warp, the warp rate is limited by *max\_rails\_rate*, and when using physical time warp, the warp rate is limited by *max\_physics\_rate*.

**Parameters**

- **ut** (*number*) – The universal time to warp to, in seconds.
- **max\_rails\_rate** (*number*) – The maximum warp rate in regular “on-rails” time warp.
- **max\_physics\_rate** (*number*) – The maximum warp rate in physical time warp.

**Returns** When the time warp is complete.

**static transform\_position** (*position, from, to*)

Converts a position from one reference frame to another.

**Parameters**

- **position** (*Tuple*) – Position, as a vector, in reference frame *from*.
- **from** (`SpaceCenter.ReferenceFrame`) – The reference frame that the position is in.
- **to** (`SpaceCenter.ReferenceFrame`) – The reference frame to convert the position to.

**Returns** The corresponding position, as a vector, in reference frame *to*.

**Return type** Tuple of (number, number, number)

**static transform\_direction** (*direction, from, to*)

Converts a direction from one reference frame to another.

**Parameters**

- **direction** (*Tuple*) – Direction, as a vector, in reference frame *from*.
- **from** (`SpaceCenter.ReferenceFrame`) – The reference frame that the direction is in.
- **to** (`SpaceCenter.ReferenceFrame`) – The reference frame to convert the direction to.

**Returns** The corresponding direction, as a vector, in reference frame *to*.

**Return type** Tuple of (number, number, number)

**static transform\_rotation** (*rotation, from, to*)

Converts a rotation from one reference frame to another.

**Parameters**

- **rotation** (*Tuple*) – Rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *from*.
- **from** (*SpaceCenter.ReferenceFrame*) – The reference frame that the rotation is in.
- **to** (*SpaceCenter.ReferenceFrame*) – The reference frame to convert the rotation to.

**Returns** The corresponding rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *to*.

**Return type** Tuple of (number, number, number, number)

**static transform\_velocity** (*position, velocity, from, to*)

Converts a velocity (acting at the specified position) from one reference frame to another. The position is required to take the relative angular velocity of the reference frames into account.

**Parameters**

- **position** (*Tuple*) – Position, as a vector, in reference frame *from*.
- **velocity** (*Tuple*) – Velocity, as a vector that points in the direction of travel and whose magnitude is the speed in meters per second, in reference frame *from*.
- **from** (*SpaceCenter.ReferenceFrame*) – The reference frame that the position and velocity are in.
- **to** (*SpaceCenter.ReferenceFrame*) – The reference frame to convert the velocity to.

**Returns** The corresponding velocity, as a vector, in reference frame *to*.

**Return type** Tuple of (number, number, number)

**far\_available**

Whether [Ferram Aerospace Research](#) is installed.

**Attribute** Read-only, cannot be set

**Return type** boolean

**warp\_mode**

The current time warp mode. Returns *SpaceCenter.WarpMode.none* if time warp is not active, *SpaceCenter.WarpMode.rails* if regular “on-rails” time warp is active, or *SpaceCenter.WarpMode.physics* if physical time warp is active.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.WarpMode*

**camera**

An object that can be used to control the camera.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Camera*

**waypoint\_manager**

The waypoint manager.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.WaypointManager*

**contract\_manager**

The contract manager.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ContractManager*

**class WarpMode**

The time warp mode. Returned by *SpaceCenter.WarpMode*

**rails**

Time warp is active, and in regular “on-rails” mode.

**physics**

Time warp is active, and in physical time warp mode.

**none**

Time warp is not active.

## 6.3.2 Vessel

**class Vessel**

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources. Created using *SpaceCenter.active\_vessel* or *SpaceCenter.vessels*.

**name**

The name of the vessel.

**Attribute** Can be read or written

**Return type** string

**type**

The type of the vessel.

**Attribute** Can be read or written

**Return type** *SpaceCenter.VesselType*

**situation**

The situation the vessel is in.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.VesselSituation*

**recoverable**

Whether the vessel is recoverable.

**Attribute** Read-only, cannot be set

**Return type** boolean

**recover()**

Recover the vessel.

**met**

The mission elapsed time in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**biome**

The name of the biome the vessel is currently in.

**Attribute** Read-only, cannot be set

**Return type** string

**flight** (*[reference\_frame = None]*)

Returns a *SpaceCenter.Flight* object that can be used to get flight telemetry for the vessel, in the specified reference frame.

**Parameters** **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame. Defaults to the vessel's surface reference frame (*SpaceCenter.Vessel.surface\_reference\_frame*).

**Return type** *SpaceCenter.Flight*

---

**Note:** When this is called with no arguments, the vessel's surface reference frame is used. This reference frame moves with the vessel, therefore velocities and speeds returned by the flight object will be zero. See the *reference frames tutorial* for examples of getting *the orbital and surface speeds of a vessel*.

---

**orbit**

The current orbit of the vessel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Orbit*

**control**

Returns a *SpaceCenter.Control* object that can be used to manipulate the vessel's control inputs. For example, its pitch/yaw/roll controls, RCS and thrust.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Control*

**comms**

Returns a *SpaceCenter.Comms* object that can be used to interact with CommNet for this vessel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Comms*

**auto\_pilot**

An *SpaceCenter.AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.AutoPilot*

**resources**

A *SpaceCenter.Resources* object, that can be used to get information about resources stored in the vessel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Resources*

**resources\_in\_decouple\_stage** (*stage[, cumulative = True]*)

Returns a *SpaceCenter.Resources* object, that can be used to get information about resources stored in a given *stage*.

**Parameters**

- **stage** (*number*) – Get resources for parts that are decoupled in this stage.
- **cumulative** (*boolean*) – When `False`, returns the resources for parts decoupled in just the given stage. When `True` returns the resources decoupled in the given stage and all subsequent stages combined.

**Return type** *SpaceCenter.Resources*

---

**Note:** For details on stage numbering, see the discussion on *Staging*.

---

### **parts**

A *SpaceCenter.Parts* object, that can be used to interact with the parts that make up this vessel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Parts*

### **mass**

The total mass of the vessel, including resources, in kg.

**Attribute** Read-only, cannot be set

**Return type** `number`

### **dry\_mass**

The total mass of the vessel, excluding resources, in kg.

**Attribute** Read-only, cannot be set

**Return type** `number`

### **thrust**

The total thrust currently being produced by the vessel's engines, in Newtons. This is computed by summing *SpaceCenter.Engine.thrust* for every engine in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `number`

### **available\_thrust**

Gets the total available thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *SpaceCenter.Engine.available\_thrust* for every active engine in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `number`

### **max\_thrust**

The total maximum thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *SpaceCenter.Engine.max\_thrust* for every active engine.

**Attribute** Read-only, cannot be set

**Return type** `number`

### **max\_vacuum\_thrust**

The total maximum thrust that can be produced by the vessel's active engines when the vessel is in a vacuum, in Newtons. This is computed by summing *SpaceCenter.Engine.max\_vacuum\_thrust* for every active engine.

**Attribute** Read-only, cannot be set

**Return type** `number`

**specific\_impulse**

The combined specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

**Attribute** Read-only, cannot be set

**Return type** number

**vacuum\_specific\_impulse**

The combined vacuum specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

**Attribute** Read-only, cannot be set

**Return type** number

**kerbin\_sea\_level\_specific\_impulse**

The combined specific impulse of all active engines at sea level on Kerbin, in seconds. This is computed using the formula [described here](#).

**Attribute** Read-only, cannot be set

**Return type** number

**moment\_of\_inertia**

The moment of inertia of the vessel around its center of mass in  $kg.m^2$ . The inertia values in the returned 3-tuple are around the pitch, roll and yaw directions respectively. This corresponds to the vessels reference frame (*SpaceCenter.ReferenceFrame*).

**Attribute** Read-only, cannot be set

**Return type** Tuple of (number, number, number)

**inertia\_tensor**

The inertia tensor of the vessel around its center of mass, in the vessels reference frame (*SpaceCenter.ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

**Attribute** Read-only, cannot be set

**Return type** List of number

**available\_torque**

The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gimballing engines and aerodynamic control surfaces. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*SpaceCenter.ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

**available\_reaction\_wheel\_torque**

The maximum torque that the currently active and powered reaction wheels can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*SpaceCenter.ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

**available\_rcs\_torque**

The maximum torque that the currently active RCS thrusters can generate. Returns the torques in  $N.m$

around each of the coordinate axes of the vessels reference frame (*SpaceCenter.ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

#### **available\_engine\_torque**

The maximum torque that the currently active and gimballed engines can generate. Returns the torques in *N.m* around each of the coordinate axes of the vessels reference frame (*SpaceCenter.ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

#### **available\_control\_surface\_torque**

The maximum torque that the aerodynamic control surfaces can generate. Returns the torques in *N.m* around each of the coordinate axes of the vessels reference frame (*SpaceCenter.ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

#### **available\_other\_torque**

The maximum torque that parts (excluding reaction wheels, gimballed engines, RCS and control surfaces) can generate. Returns the torques in *N.m* around each of the coordinate axes of the vessels reference frame (*SpaceCenter.ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

#### **reference\_frame**

The reference frame that is fixed relative to the vessel, and orientated with the vessel.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel.
- The x-axis points out to the right of the vessel.
- The y-axis points in the forward direction of the vessel.
- The z-axis points out of the bottom off the vessel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

#### **orbital\_reference\_frame**

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel.
- The axes rotate with the orbital prograde/normal/radial directions.

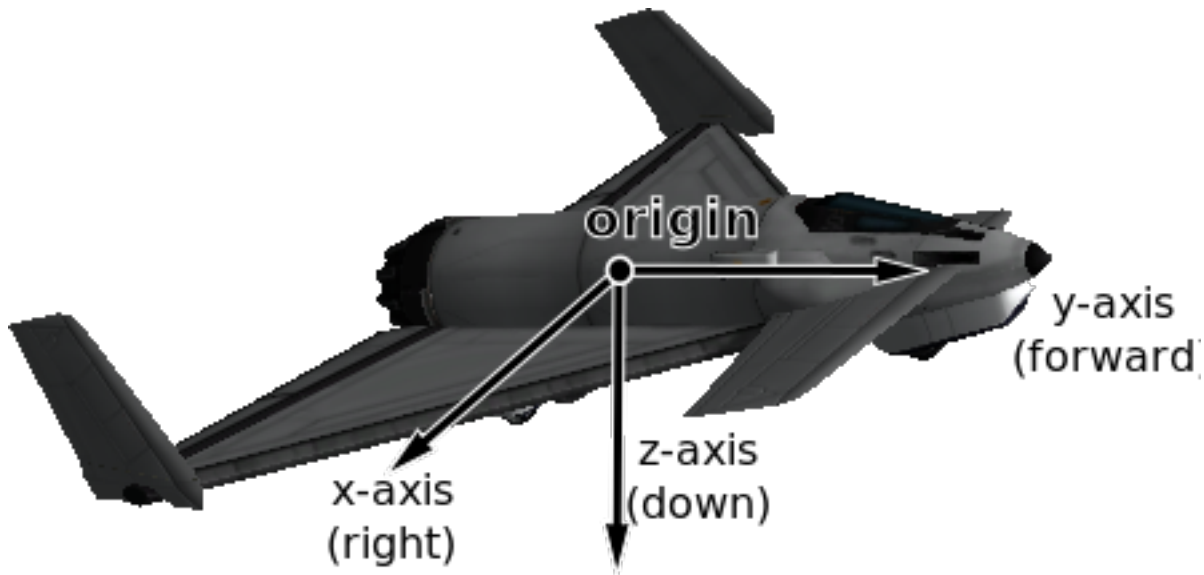


Fig. 6.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

---

**Note:** Be careful not to confuse this with 'orbit' mode on the navball.

---

#### **surface\_reference\_frame**

The reference frame that is fixed relative to the vessel, and orientated with the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the north and up directions on the surface of the body.
- The x-axis points in the [zenith](#) direction (upwards, normal to the body being orbited, from the center of the body towards the center of mass of the vessel).
- The y-axis points northwards towards the [astronomical horizon](#) (north, and tangential to the surface of the body – the direction in which a compass would point when on the surface).
- The z-axis points eastwards towards the [astronomical horizon](#) (east, and tangential to the surface of the body – east on a compass when on the surface).

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*



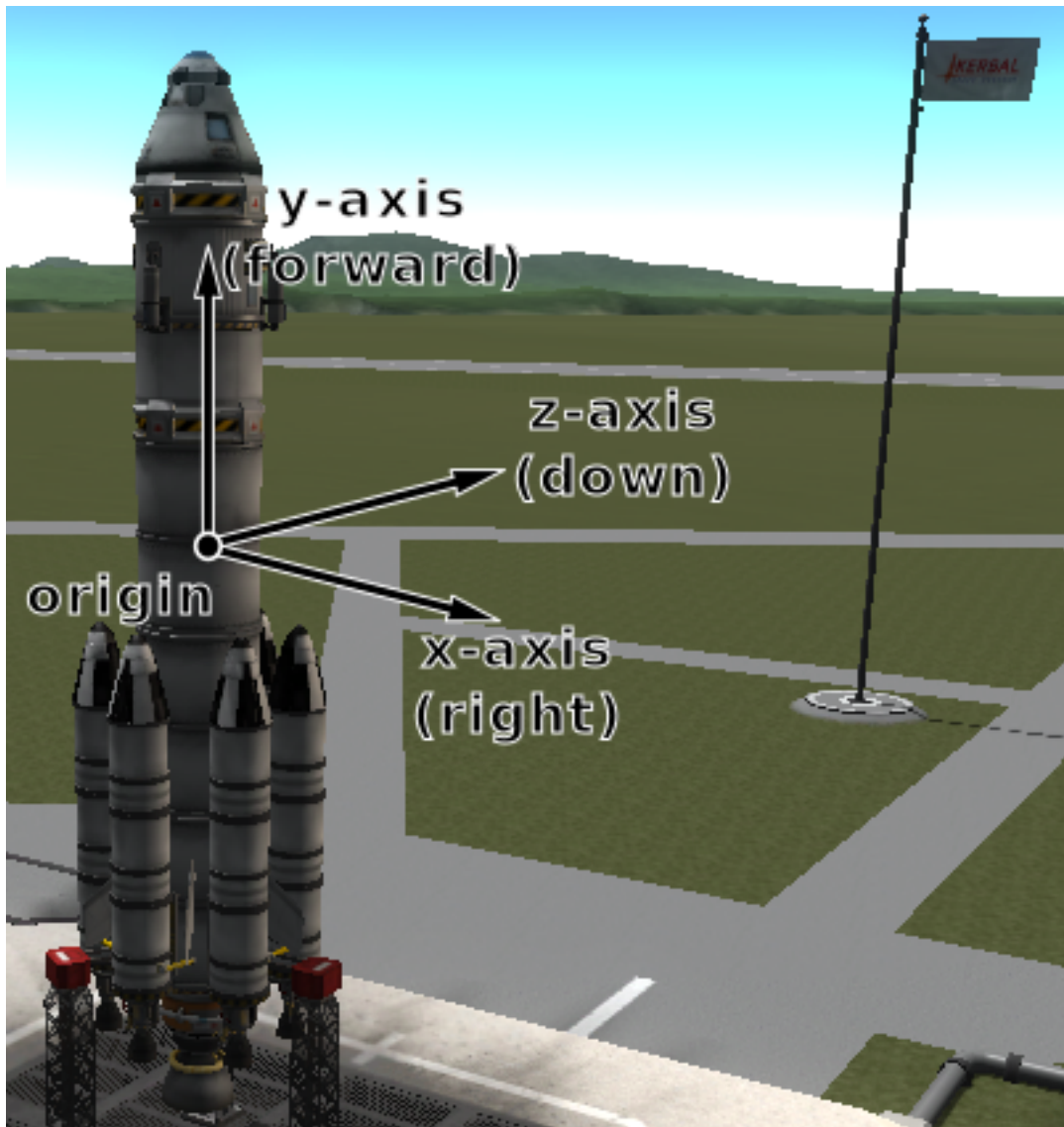


Fig. 6.2: Vessel reference frame origin and axes for the Kerbal-X rocket

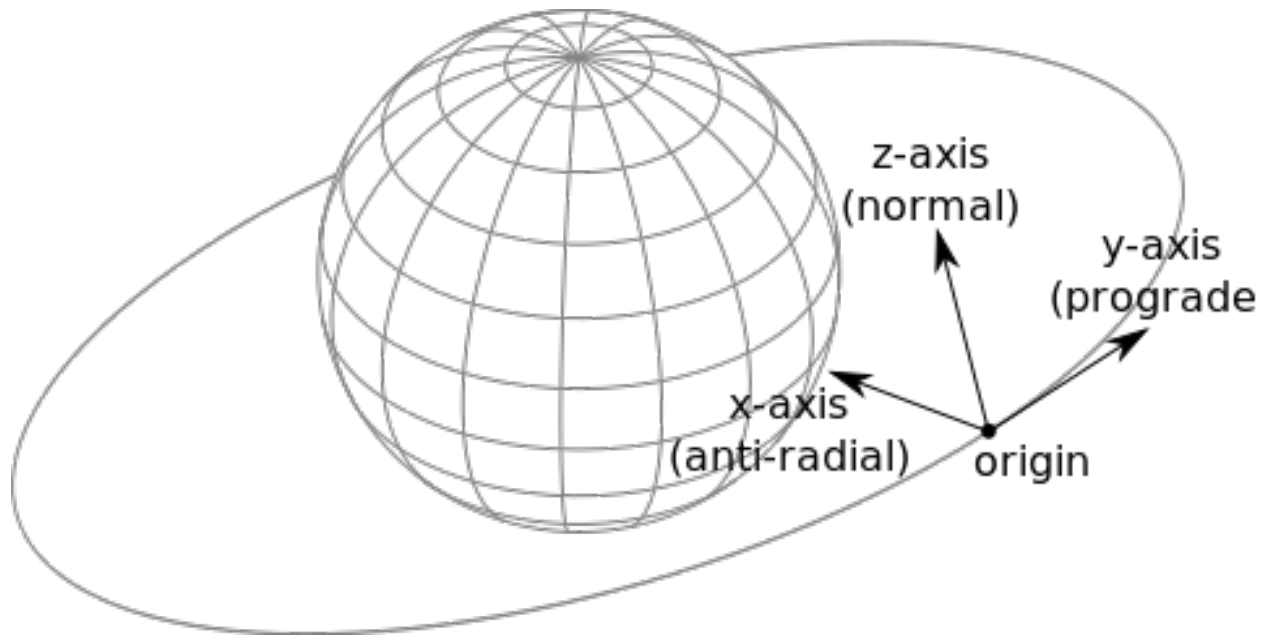


Fig. 6.3: Vessel orbital reference frame origin and axes

---

**Note:** Be careful not to confuse this with ‘surface’ mode on the navball.

---

#### **surface\_velocity\_reference\_frame**

The reference frame that is fixed relative to the vessel, and orientated with the velocity vector of the vessel relative to the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel’s velocity vector.
- The y-axis points in the direction of the vessel’s velocity vector, relative to the surface of the body being orbited.
- The z-axis is in the plane of the [astronomical horizon](#).
- The x-axis is orthogonal to the other two axes.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

#### **position** (*reference\_frame*)

The position of the center of mass of the vessel, in the given reference frame.

**Parameters** **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

#### **bounding\_box** (*reference\_frame*)

The axis-aligned bounding box of the vessel in the given reference frame.

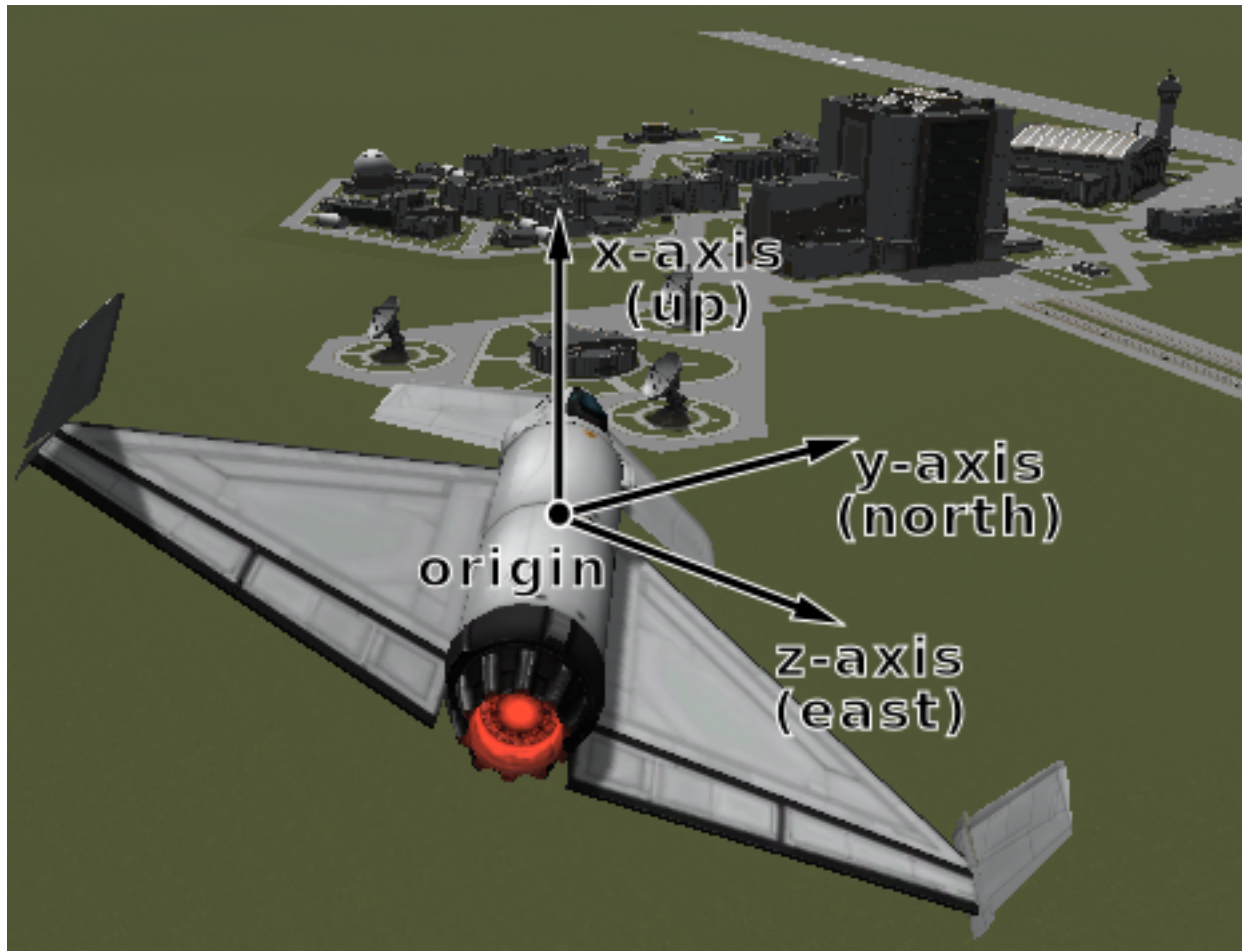


Fig. 6.4: Vessel surface reference frame origin and axes

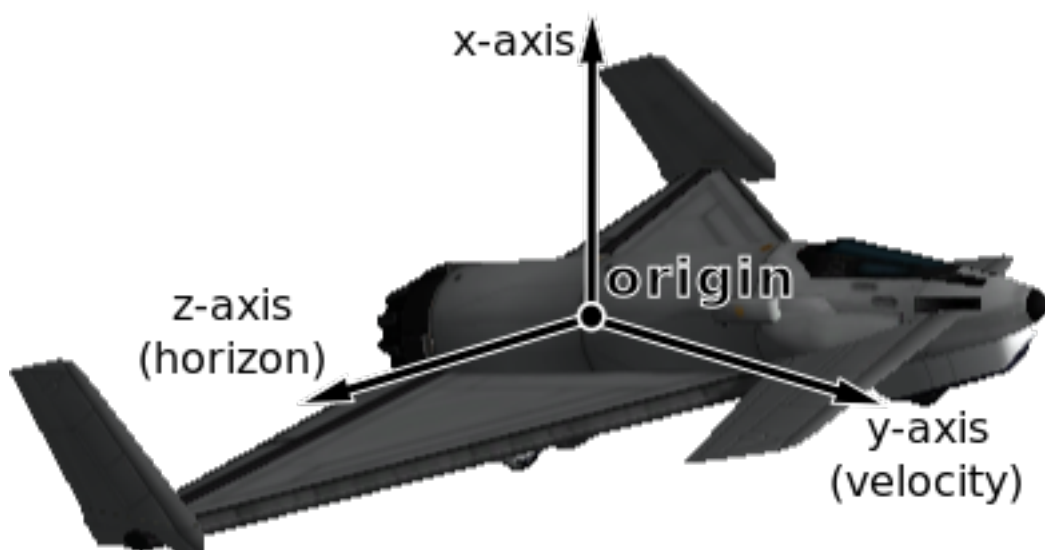


Fig. 6.5: Vessel surface velocity reference frame origin and axes

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

**velocity** (*reference\_frame*)

The velocity of the center of mass of the vessel, in the given reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

**Return type** Tuple of (number, number, number)

**rotation** (*reference\_frame*)

The rotation of the vessel, in the given reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** Tuple of (number, number, number, number)

**direction** (*reference\_frame*)

The direction in which the vessel is pointing, in the given reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**angular\_velocity** (*reference\_frame*)

The angular velocity of the vessel, in the given reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the vessel, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

**Return type** Tuple of (number, number, number)

**class VesselType**

The type of a vessel. See `SpaceCenter.Vessel.type`.

**base**  
Base.

**debris**  
Debris.

**lander**  
Lander.

**plane**  
Plane.

**probe**  
Probe.

**relay**  
Relay.

**rover**  
Rover.

**ship**  
Ship.

**station**  
Station.

#### **class VesselSituation**

The situation a vessel is in. See *SpaceCenter.Vessel.situation*.

**docked**  
Vessel is docked to another.

**escaping**  
Escaping.

**flying**  
Vessel is flying through an atmosphere.

**landed**  
Vessel is landed on the surface of a body.

**orbiting**  
Vessel is orbiting a body.

**pre\_launch**  
Vessel is awaiting launch.

**splashed**  
Vessel has splashed down in an ocean.

**sub\_orbital**  
Vessel is on a sub-orbital trajectory.

### 6.3.3 CelestialBody

#### **class CelestialBody**

Represents a celestial body (such as a planet or moon). See *SpaceCenter.bodies*.

**name**  
The name of the body.

**Attribute** Read-only, cannot be set

**Return type** string

**satellites**  
A list of celestial bodies that are in orbit around this celestial body.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.CelestialBody*

**orbit**  
The orbit of the body.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Orbit*

**mass**

The mass of the body, in kilograms.

**Attribute** Read-only, cannot be set

**Return type** number

**gravitational\_parameter**

The [standard gravitational parameter](#) of the body in  $m^3s^{-2}$ .

**Attribute** Read-only, cannot be set

**Return type** number

**surface\_gravity**

The acceleration due to gravity at sea level (mean altitude) on the body, in  $m/s^2$ .

**Attribute** Read-only, cannot be set

**Return type** number

**rotational\_period**

The sidereal rotational period of the body, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**rotational\_speed**

The rotational speed of the body, in radians per second.

**Attribute** Read-only, cannot be set

**Return type** number

**rotation\_angle**

The current rotation angle of the body, in radians. A value between 0 and  $2\pi$

**Attribute** Read-only, cannot be set

**Return type** number

**initial\_rotation**

The initial rotation angle of the body (at UT 0), in radians. A value between 0 and  $2\pi$

**Attribute** Read-only, cannot be set

**Return type** number

**equatorial\_radius**

The equatorial radius of the body, in meters.

**Attribute** Read-only, cannot be set

**Return type** number

**surface\_height** (*latitude, longitude*)

The height of the surface relative to mean sea level, in meters, at the given position. When over water this is equal to 0.

**Parameters**

- **latitude** (*number*) – Latitude in degrees.

- **longitude** (*number*) – Longitude in degrees.

**Return type** number

**bedrock\_height** (*latitude, longitude*)

The height of the surface relative to mean sea level, in meters, at the given position. When over water, this is the height of the sea-bed and is therefore negative value.

**Parameters**

- **latitude** (*number*) – Latitude in degrees.
- **longitude** (*number*) – Longitude in degrees.

**Return type** number

**msl\_position** (*latitude, longitude, reference\_frame*)

The position at mean sea level at the given latitude and longitude, in the given reference frame.

**Parameters**

- **latitude** (*number*) – Latitude in degrees.
- **longitude** (*number*) – Longitude in degrees.
- **reference\_frame** (`SpaceCenter.ReferenceFrame`) – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Return type** Tuple of (number, number, number)

**surface\_position** (*latitude, longitude, reference\_frame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position of the surface of the water.

**Parameters**

- **latitude** (*number*) – Latitude in degrees.
- **longitude** (*number*) – Longitude in degrees.
- **reference\_frame** (`SpaceCenter.ReferenceFrame`) – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Return type** Tuple of (number, number, number)

**bedrock\_position** (*latitude, longitude, reference\_frame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position at the bottom of the sea-bed.

**Parameters**

- **latitude** (*number*) – Latitude in degrees.
- **longitude** (*number*) – Longitude in degrees.
- **reference\_frame** (`SpaceCenter.ReferenceFrame`) – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Return type** Tuple of (number, number, number)

**position\_at\_altitude** (*latitude, longitude, altitude, reference\_frame*)

The position at the given latitude, longitude and altitude, in the given reference frame.

**Parameters**

- **latitude** (*number*) – Latitude in degrees.
- **longitude** (*number*) – Longitude in degrees.
- **altitude** (*number*) – Altitude in meters above sea level.
- **reference\_frame** (`SpaceCenter.ReferenceFrame`) – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Return type** Tuple of (number, number, number)

**altitude\_at\_position** (*position*, *reference\_frame*)

The altitude, in meters, of the given position in the given reference frame.

**Parameters**

- **position** (*Tuple*) – Position as a vector.
- **reference\_frame** (`SpaceCenter.ReferenceFrame`) – Reference frame for the position vector.

**Return type** number

**latitude\_at\_position** (*position*, *reference\_frame*)

The latitude of the given position, in the given reference frame.

**Parameters**

- **position** (*Tuple*) – Position as a vector.
- **reference\_frame** (`SpaceCenter.ReferenceFrame`) – Reference frame for the position vector.

**Return type** number

**longitude\_at\_position** (*position*, *reference\_frame*)

The longitude of the given position, in the given reference frame.

**Parameters**

- **position** (*Tuple*) – Position as a vector.
- **reference\_frame** (`SpaceCenter.ReferenceFrame`) – Reference frame for the position vector.

**Return type** number

**sphere\_of\_influence**

The radius of the sphere of influence of the body, in meters.

**Attribute** Read-only, cannot be set

**Return type** number

**has\_atmosphere**

True if the body has an atmosphere.

**Attribute** Read-only, cannot be set

**Return type** boolean

**atmosphere\_depth**

The depth of the atmosphere, in meters.



**Attribute** Read-only, cannot be set

**Return type** number

**atmospheric\_density\_at\_position** (*position*, *reference\_frame*)

The atmospheric density at the given position, in  $kg/m^3$ , in the given reference frame.

**Parameters**

- **position** (*Tuple*) – The position vector at which to measure the density.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the position vector is in.

**Return type** number

**has\_atmospheric\_oxygen**

True if there is oxygen in the atmosphere, required for air-breathing engines.

**Attribute** Read-only, cannot be set

**Return type** boolean

**temperature\_at** (*position*, *reference\_frame*)

The temperature on the body at the given position, in the given reference frame.

**Parameters**

- **position** (*Tuple*) – Position as a vector.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The reference frame that the position is in.

**Return type** number

---

**Note:** This calculation is performed using the bodies current position, which means that the value could be wrong if you want to know the temperature in the far future.

---

**density\_at** (*altitude*)

Gets the air density, in  $kg/m^3$ , for the specified altitude above sea level, in meters.

**Parameters** **altitude** (*number*) –

**Return type** number

---

**Note:** This is an approximation, because actual calculations, taking sun exposure into account to compute air temperature, require us to know the exact point on the body where the density is to be computed (knowing the altitude is not enough). However, the difference is small for high altitudes, so it makes very little difference for trajectory prediction.

---

**pressure\_at** (*altitude*)

Gets the air pressure, in Pascals, for the specified altitude above sea level, in meters.

**Parameters** **altitude** (*number*) –

**Return type** number

**biomes**

The biomes present on this body.

**Attribute** Read-only, cannot be set

**Return type** Set of string

**biome\_at** (*latitude*, *longitude*)

The biome at the given latitude and longitude, in degrees.

**Parameters**

- **latitude** (*number*) –
- **longitude** (*number*) –

**Return type** string

**flying\_high\_altitude\_threshold**

The altitude, in meters, above which a vessel is considered to be flying “high” when doing science.

**Attribute** Read-only, cannot be set

**Return type** number

**space\_high\_altitude\_threshold**

The altitude, in meters, above which a vessel is considered to be in “high” space when doing science.

**Attribute** Read-only, cannot be set

**Return type** number

**reference\_frame**

The reference frame that is fixed relative to the celestial body.

- The origin is at the center of the body.
- The axes rotate with the body.
- The x-axis points from the center of the body towards the intersection of the prime meridian and equator (the position at 0° longitude, 0° latitude).
- The y-axis points from the center of the body towards the north pole.
- The z-axis points from the center of the body towards the equator at 90°E longitude.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

**non\_rotating\_reference\_frame**

The reference frame that is fixed relative to this celestial body, and orientated in a fixed direction (it does not rotate with the body).

- The origin is at the center of the body.
- The axes do not rotate.
- The x-axis points in an arbitrary direction through the equator.
- The y-axis points from the center of the body towards the north pole.
- The z-axis points in an arbitrary direction through the equator.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

**orbital\_reference\_frame**

The reference frame that is fixed relative to this celestial body, but orientated with the body’s orbital prograde/normal/radial directions.

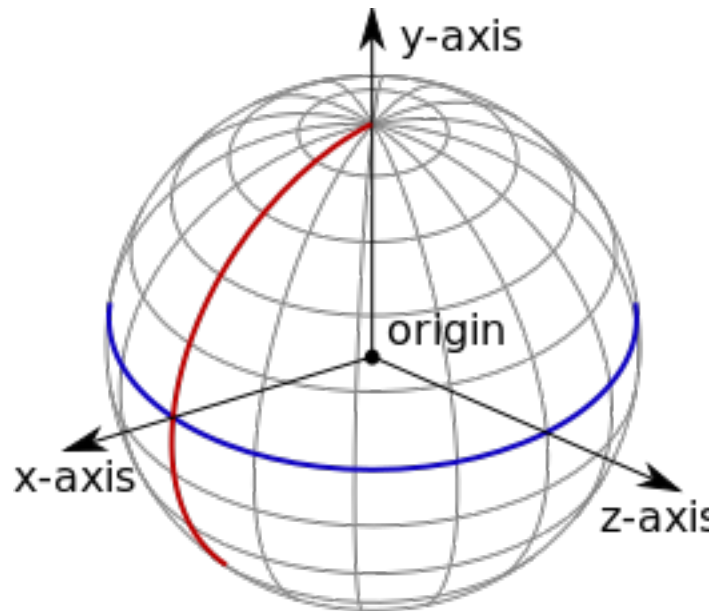


Fig. 6.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

- The origin is at the center of the body.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

**Attribute** Read-only, cannot be set

**Return type** `SpaceCenter.ReferenceFrame`

**position** (*reference\_frame*)

The position of the center of the body, in the specified reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

**velocity** (*reference\_frame*)

The linear velocity of the body, in the specified reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

**Return type** Tuple of (number, number, number)

**rotation** (*reference\_frame*)

The rotation of the body, in the specified reference frame.

**Parameters** `reference_frame` (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** Tuple of (number, number, number, number)

**direction** (`reference_frame`)

The direction in which the north pole of the celestial body is pointing, in the specified reference frame.

**Parameters** `reference_frame` (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**angular\_velocity** (`reference_frame`)

The angular velocity of the body in the specified reference frame.

**Parameters** `reference_frame` (`SpaceCenter.ReferenceFrame`) – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the body, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

**Return type** Tuple of (number, number, number)

## 6.3.4 Flight

### **class Flight**

Used to get flight telemetry for a vessel, by calling `SpaceCenter.Vessel.flight()`. All of the information returned by this class is given in the reference frame passed to that method. Obtained by calling `SpaceCenter.Vessel.flight()`.

---

**Note:** To get orbital information, such as the apoapsis or inclination, see `SpaceCenter.Orbit`.

---

#### **g\_force**

The current G force acting on the vessel in  $m/s^2$ .

**Attribute** Read-only, cannot be set

**Return type** number

#### **mean\_altitude**

The altitude above sea level, in meters. Measured from the center of mass of the vessel.

**Attribute** Read-only, cannot be set

**Return type** number

#### **surface\_altitude**

The altitude above the surface of the body or sea level, whichever is closer, in meters. Measured from the center of mass of the vessel.

**Attribute** Read-only, cannot be set

**Return type** number

**bedrock\_altitude**

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor. Measured from the center of mass of the vessel.

**Attribute** Read-only, cannot be set

**Return type** number

**elevation**

The elevation of the terrain under the vessel, in meters. This is the height of the terrain above sea level, and is negative when the vessel is over the sea.

**Attribute** Read-only, cannot be set

**Return type** number

**latitude**

The [latitude](#) of the vessel for the body being orbited, in degrees.

**Attribute** Read-only, cannot be set

**Return type** number

**longitude**

The [longitude](#) of the vessel for the body being orbited, in degrees.

**Attribute** Read-only, cannot be set

**Return type** number

**velocity**

The velocity of the vessel, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the vessel in meters per second.

**Return type** Tuple of (number, number, number)

**speed**

The speed of the vessel in meters per second, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Return type** number

**horizontal\_speed**

The horizontal speed of the vessel in meters per second, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Return type** number

**vertical\_speed**

The vertical speed of the vessel in meters per second, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Return type** number

**center\_of\_mass**

The position of the center of mass of the vessel, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

**rotation**

The rotation of the vessel, in the reference frame *SpaceCenter.ReferenceFrame*

**Attribute** Read-only, cannot be set

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** Tuple of (number, number, number, number)

**direction**

The direction that the vessel is pointing in, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**pitch**

The pitch of the vessel relative to the horizon, in degrees. A value between -90° and +90°.

**Attribute** Read-only, cannot be set

**Return type** number

**heading**

The heading of the vessel (its angle relative to north), in degrees. A value between 0° and 360°.

**Attribute** Read-only, cannot be set

**Return type** number

**roll**

The roll of the vessel relative to the horizon, in degrees. A value between -180° and +180°.

**Attribute** Read-only, cannot be set

**Return type** number

**prograde**

The prograde direction of the vessels orbit, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**retrograde**

The retrograde direction of the vessels orbit, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**normal**

The direction normal to the vessels orbit, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**anti\_normal**

The direction opposite to the normal of the vessels orbit, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**radial**

The radial direction of the vessels orbit, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**anti\_radial**

The direction opposite to the radial direction of the vessels orbit, in the reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**atmosphere\_density**

The current density of the atmosphere around the vessel, in  $kg/m^3$ .

**Attribute** Read-only, cannot be set

**Return type** number

**dynamic\_pressure**

The dynamic pressure acting on the vessel, in Pascals. This is a measure of the strength of the aerodynamic forces. It is equal to  $\frac{1}{2} \cdot \text{air density} \cdot \text{velocity}^2$ . It is commonly denoted  $Q$ .

**Attribute** Read-only, cannot be set

**Return type** number

**static\_pressure**

The static atmospheric pressure acting on the vessel, in Pascals.

**Attribute** Read-only, cannot be set

**Return type** number

**static\_pressure\_at\_msl**

The static atmospheric pressure at mean sea level, in Pascals.

**Attribute** Read-only, cannot be set

**Return type** number

**aerodynamic\_force**

The total aerodynamic forces acting on the vessel, in reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Return type** Tuple of (number, number, number)

**simulate\_aerodynamic\_force\_at** (*body, position, velocity*)

Simulate and return the total aerodynamic forces acting on the vessel, if it were to be traveling with the given velocity at the given position in the atmosphere of the given celestial body.

**Parameters**

- **body** (*SpaceCenter.CelestialBody*) –
- **position** (*Tuple*) –
- **velocity** (*Tuple*) –

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Return type** Tuple of (number, number, number)

**lift**

The *aerodynamic lift* currently acting on the vessel.

**Attribute** Read-only, cannot be set

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Return type** Tuple of (number, number, number)

**drag**

The *aerodynamic drag* currently acting on the vessel.

**Attribute** Read-only, cannot be set

**Returns** A vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

**Return type** Tuple of (number, number, number)

**speed\_of\_sound**

The speed of sound, in the atmosphere around the vessel, in *m/s*.

**Attribute** Read-only, cannot be set

**Return type** number

**mach**

The speed of the vessel, in multiples of the speed of sound.

**Attribute** Read-only, cannot be set

**Return type** number

**reynolds\_number**

The vessels Reynolds number.

**Attribute** Read-only, cannot be set



**Return type** number

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**true\_air\_speed**

The [true air speed](#) of the vessel, in meters per second.

**Attribute** Read-only, cannot be set

**Return type** number

**equivalent\_air\_speed**

The [equivalent air speed](#) of the vessel, in meters per second.

**Attribute** Read-only, cannot be set

**Return type** number

**terminal\_velocity**

An estimate of the current terminal velocity of the vessel, in meters per second. This is the speed at which the drag forces cancel out the force of gravity.

**Attribute** Read-only, cannot be set

**Return type** number

**angle\_of\_attack**

The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

**Attribute** Read-only, cannot be set

**Return type** number

**sideslip\_angle**

The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

**Attribute** Read-only, cannot be set

**Return type** number

**total\_air\_temperature**

The [total air temperature](#) of the atmosphere around the vessel, in Kelvin. This includes the *SpaceCenter.Flight.static\_air\_temperature* and the vessel's kinetic energy.

**Attribute** Read-only, cannot be set

**Return type** number

**static\_air\_temperature**

The [static \(ambient\) temperature](#) of the atmosphere around the vessel, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

**stall\_fraction**

The current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**drag\_coefficient**

The coefficient of drag. This is the amount of drag produced by the vessel. It depends on air speed, air density and wing area.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**lift\_coefficient**

The coefficient of lift. This is the amount of lift produced by the vessel, and depends on air speed, air density and wing area.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**ballistic\_coefficient**

The [ballistic coefficient](#).

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**thrust\_specific\_fuel\_consumption**

The thrust specific fuel consumption for the jet engines on the vessel. This is a measure of the efficiency of the engines, with a lower value indicating a more efficient vessel. This value is the number of Newtons of fuel that are burned, per hour, to produce one newton of thrust.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** Requires [Ferram Aerospace Research](#).

---

## 6.3.5 Orbit

**class Orbit**

Describes an orbit. For example, the orbit of a vessel, obtained by calling *SpaceCenter.Vessel.orbit*, or a celestial body, obtained by calling *SpaceCenter.CelestialBody.orbit*.

**body**

The celestial body (e.g. planet or moon) around which the object is orbiting.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.CelestialBody*

#### **apoapsis**

Gets the apoapsis of the orbit, in meters, from the center of mass of the body being orbited.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** For the apoapsis altitude reported on the in-game map view, use *SpaceCenter.Orbit.apoapsis\_altitude*.

---

#### **periapsis**

The periapsis of the orbit, in meters, from the center of mass of the body being orbited.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** For the periapsis altitude reported on the in-game map view, use *SpaceCenter.Orbit.periapsis\_altitude*.

---

#### **apoapsis\_altitude**

The apoapsis of the orbit, in meters, above the sea level of the body being orbited.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** This is equal to *SpaceCenter.Orbit.apoapsis* minus the equatorial radius of the body.

---

#### **periapsis\_altitude**

The periapsis of the orbit, in meters, above the sea level of the body being orbited.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** This is equal to *SpaceCenter.Orbit.periapsis* minus the equatorial radius of the body.

---

#### **semi\_major\_axis**

The semi-major axis of the orbit, in meters.

**Attribute** Read-only, cannot be set

**Return type** number

#### **semi\_minor\_axis**

The semi-minor axis of the orbit, in meters.

**Attribute** Read-only, cannot be set

**Return type** number

#### **radius**

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** This value will change over time if the orbit is elliptical.

---

**radius\_at** (*ut*)

The orbital radius at the given time, in meters.

**Parameters** **ut** (*number*) – The universal time to measure the radius at.

**Return type** number

**position\_at** (*ut*, *reference\_frame*)

The position at a given time, in the specified reference frame.

**Parameters**

- **ut** (*number*) – The universal time to measure the position at.
- **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

**speed**

The current orbital speed of the object in meters per second.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** This value will change over time if the orbit is elliptical.

---

**period**

The orbital period, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**time\_to\_apoapsis**

The time until the object reaches apoapsis, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**time\_to\_periapsis**

The time until the object reaches periapsis, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**eccentricity**

The [eccentricity](#) of the orbit.

**Attribute** Read-only, cannot be set

**Return type** number

**inclination**

The [inclination](#) of the orbit, in radians.

**Attribute** Read-only, cannot be set

**Return type** number

**longitude\_of\_ascending\_node**

The [longitude of the ascending node](#), in radians.

**Attribute** Read-only, cannot be set

**Return type** number

**argument\_of\_periapsis**

The [argument of periapsis](#), in radians.

**Attribute** Read-only, cannot be set

**Return type** number

**mean\_anomaly\_at\_epoch**

The [mean anomaly at epoch](#).

**Attribute** Read-only, cannot be set

**Return type** number

**epoch**

The time since the epoch (the point at which the [mean anomaly at epoch](#) was measured, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**mean\_anomaly**

The [mean anomaly](#).

**Attribute** Read-only, cannot be set

**Return type** number

**mean\_anomaly\_at\_ut (ut)**

The mean anomaly at the given time.

**Parameters** **ut** (*number*) – The universal time in seconds.

**Return type** number

**eccentric\_anomaly**

The [eccentric anomaly](#).

**Attribute** Read-only, cannot be set

**Return type** number

**eccentric\_anomaly\_at\_ut (ut)**

The eccentric anomaly at the given universal time.

**Parameters** **ut** (*number*) – The universal time, in seconds.

**Return type** number

**true\_anomaly**

The [true anomaly](#).

**Attribute** Read-only, cannot be set

**Return type** number

**true\_anomaly\_at\_ut** (*ut*)

The true anomaly at the given time.

**Parameters** **ut** (*number*) – The universal time in seconds.

**Return type** number

**true\_anomaly\_at\_radius** (*radius*)

The true anomaly at the given orbital radius.

**Parameters** **radius** (*number*) – The orbital radius in meters.

**Return type** number

**ut\_at\_true\_anomaly** (*true\_anomaly*)

The universal time, in seconds, corresponding to the given true anomaly.

**Parameters** **true\_anomaly** (*number*) – True anomaly.

**Return type** number

**radius\_at\_true\_anomaly** (*true\_anomaly*)

The orbital radius at the point in the orbit given by the true anomaly.

**Parameters** **true\_anomaly** (*number*) – The true anomaly.

**Return type** number

**true\_anomaly\_at\_an** (*target*)

The true anomaly of the ascending node with the given target vessel.

**Parameters** **target** (`SpaceCenter.Vessel`) – Target vessel.

**Return type** number

**true\_anomaly\_at\_dn** (*target*)

The true anomaly of the descending node with the given target vessel.

**Parameters** **target** (`SpaceCenter.Vessel`) – Target vessel.

**Return type** number

**orbital\_speed**

The current orbital speed in meters per second.

**Attribute** Read-only, cannot be set

**Return type** number

**orbital\_speed\_at** (*time*)

The orbital speed at the given time, in meters per second.

**Parameters** **time** (*number*) – Time from now, in seconds.

**Return type** number

**static reference\_plane\_normal** (*reference\_frame*)

The direction that is normal to the orbits reference plane, in the given reference frame. The reference plane is the plane from which the orbits inclination is measured.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**static reference\_plane\_direction** (*reference\_frame*)

The direction from which the orbits longitude of ascending node is measured, in the given reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**relative\_inclination** (*target*)

Relative inclination of this orbit and the orbit of the given target vessel, in radians.

**Parameters** **target** (`SpaceCenter.Vessel`) – Target vessel.

**Return type** number

**time\_to\_soi\_change**

The time until the object changes sphere of influence, in seconds. Returns NaN if the object is not going to change sphere of influence.

**Attribute** Read-only, cannot be set

**Return type** number

**next\_orbit**

If the object is going to change sphere of influence in the future, returns the new orbit after the change. Otherwise returns `nil`.

**Attribute** Read-only, cannot be set

**Return type** `SpaceCenter.Orbit`

**time\_of\_closest\_approach** (*target*)

Estimates and returns the time at closest approach to a target vessel.

**Parameters** **target** (`SpaceCenter.Vessel`) – Target vessel.

**Returns** The universal time at closest approach, in seconds.

**Return type** number

**distance\_at\_closest\_approach** (*target*)

Estimates and returns the distance at closest approach to a target vessel, in meters.

**Parameters** **target** (`SpaceCenter.Vessel`) – Target vessel.

**Return type** number

**list\_closest\_approaches** (*target, orbits*)

Returns the times at closest approach and corresponding distances, to a target vessel.

**Parameters**

- **target** (`SpaceCenter.Vessel`) – Target vessel.
- **orbits** (*number*) – The number of future orbits to search.

**Returns** A list of two lists. The first is a list of times at closest approach, as universal times in seconds. The second is a list of corresponding distances at closest approach, in meters.

**Return type** List of List of number

### 6.3.6 Control

**class Control**

Used to manipulate the controls of a vessel. This includes adjusting the throttle, enabling/disabling systems such as SAS and RCS, or altering the direction in which the vessel is pointing. Obtained by calling *SpaceCenter.Vessel.control*.

---

**Note:** Control inputs (such as pitch, yaw and roll) are zeroed when all clients that have set one or more of these inputs are no longer connected.

---

**source**

The source of the vessels control, for example by a kerbal or a probe core.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ControlSource*

**state**

The control state of the vessel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ControlState*

**sas**

The state of SAS.

**Attribute** Can be read or written

**Return type** boolean

---

**Note:** Equivalent to *SpaceCenter.AutoPilot.sas*

---

**sas\_mode**

The current *SpaceCenter.SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

**Attribute** Can be read or written

**Return type** *SpaceCenter.SASMode*

---

**Note:** Equivalent to *SpaceCenter.AutoPilot.sas\_mode*

---

**speed\_mode**

The current *SpaceCenter.SpeedMode* of the navball. This is the mode displayed next to the speed at the top of the navball.

**Attribute** Can be read or written

**Return type** *SpaceCenter.SpeedMode*

**rcs**

The state of RCS.

**Attribute** Can be read or written

**Return type** boolean



**reaction\_wheels**

Returns whether all reactive wheels on the vessel are active, and sets the active state of all reaction wheels. See *SpaceCenter.ReactionWheel.active*.

**Attribute** Can be read or written

**Return type** boolean

**gear**

The state of the landing gear/legs.

**Attribute** Can be read or written

**Return type** boolean

**legs**

Returns whether all landing legs on the vessel are deployed, and sets the deployment state of all landing legs. Does not include wheels (for example landing gear). See *SpaceCenter.Leg.deployed*.

**Attribute** Can be read or written

**Return type** boolean

**wheels**

Returns whether all wheels on the vessel are deployed, and sets the deployment state of all wheels. Does not include landing legs. See *SpaceCenter.Wheel.deployed*.

**Attribute** Can be read or written

**Return type** boolean

**lights**

The state of the lights.

**Attribute** Can be read or written

**Return type** boolean

**brakes**

The state of the wheel brakes.

**Attribute** Can be read or written

**Return type** boolean

**antennas**

Returns whether all antennas on the vessel are deployed, and sets the deployment state of all antennas. See *SpaceCenter.Antenna.deployed*.

**Attribute** Can be read or written

**Return type** boolean

**cargo\_bays**

Returns whether any of the cargo bays on the vessel are open, and sets the open state of all cargo bays. See *SpaceCenter.CargoBay.open*.

**Attribute** Can be read or written

**Return type** boolean

**intakes**

Returns whether all of the air intakes on the vessel are open, and sets the open state of all air intakes. See *SpaceCenter.Intake.open*.

**Attribute** Can be read or written

**Return type** boolean

**parachutes**

Returns whether all parachutes on the vessel are deployed, and sets the deployment state of all parachutes. Cannot be set to `False`. See *SpaceCenter.Parachute.deployed*.

**Attribute** Can be read or written

**Return type** boolean

**radiators**

Returns whether all radiators on the vessel are deployed, and sets the deployment state of all radiators. See *SpaceCenter.Radiator.deployed*.

**Attribute** Can be read or written

**Return type** boolean

**resource\_harvesters**

Returns whether all of the resource harvesters on the vessel are deployed, and sets the deployment state of all resource harvesters. See *SpaceCenter.ResourceHarvester.deployed*.

**Attribute** Can be read or written

**Return type** boolean

**resource\_harvesters\_active**

Returns whether any of the resource harvesters on the vessel are active, and sets the active state of all resource harvesters. See *SpaceCenter.ResourceHarvester.active*.

**Attribute** Can be read or written

**Return type** boolean

**solar\_panels**

Returns whether all solar panels on the vessel are deployed, and sets the deployment state of all solar panels. See *SpaceCenter.SolarPanel.deployed*.

**Attribute** Can be read or written

**Return type** boolean

**abort**

The state of the abort action group.

**Attribute** Can be read or written

**Return type** boolean

**throttle**

The state of the throttle. A value between 0 and 1.

**Attribute** Can be read or written

**Return type** number

**input\_mode**

Sets the behavior of the pitch, yaw, roll and translation control inputs. When set to additive, these inputs are added to the vessels current inputs. This mode is the default. When set to override, these inputs (if non-zero) override the vessels inputs. This mode prevents keyboard control, or SAS, from interfering with the controls when they are set.

**Attribute** Can be read or written

**Return type** *SpaceCenter.ControlInputMode*

**pitch**

The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

**Attribute** Can be read or written

**Return type** number

**yaw**

The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

**Attribute** Can be read or written

**Return type** number

**roll**

The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

**Attribute** Can be read or written

**Return type** number

**forward**

The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

**Attribute** Can be read or written

**Return type** number

**up**

The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

**Attribute** Can be read or written

**Return type** number

**right**

The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

**Attribute** Can be read or written

**Return type** number

**wheel\_throttle**

The state of the wheel throttle. A value between -1 and 1. A value of 1 rotates the wheels forwards, a value of -1 rotates the wheels backwards.

**Attribute** Can be read or written

**Return type** number

**wheel\_steering**

The state of the wheel steering. A value between -1 and 1. A value of 1 steers to the left, and a value of -1 steers to the right.

**Attribute** Can be read or written

**Return type** number

**current\_stage**

The current stage of the vessel. Corresponds to the stage number in the in-game UI.

**Attribute** Read-only, cannot be set

**Return type** number

**activate\_next\_stage()**

Activates the next stage. Equivalent to pressing the space bar in-game.

**Returns** A list of vessel objects that are jettisoned from the active vessel.

**Return type** List of *SpaceCenter.Vessel*

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *SpaceCenter.active\_vessel* no longer refer to the active vessel.

---

**get\_action\_group** (*group*)

Returns `True` if the given action group is enabled.

**Parameters** **group** (*number*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.

**Return type** boolean

**set\_action\_group** (*group*, *state*)

Sets the state of the given action group.

**Parameters**

- **group** (*number*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.
- **state** (*boolean*) –

**toggle\_action\_group** (*group*)

Toggles the state of the given action group.

**Parameters** **group** (*number*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.

**add\_node** (*ut* [, *prograde* = 0.0] [, *normal* = 0.0] [, *radial* = 0.0])

Creates a maneuver node at the given universal time, and returns a *SpaceCenter.Node* object that can be used to modify it. Optionally sets the magnitude of the delta-v for the maneuver node in the prograde, normal and radial directions.

**Parameters**

- **ut** (*number*) – Universal time of the maneuver node.
- **prograde** (*number*) – Delta-v in the prograde direction.
- **normal** (*number*) – Delta-v in the normal direction.
- **radial** (*number*) – Delta-v in the radial direction.

**Return type** *SpaceCenter.Node*

**nodes**

Returns a list of all existing maneuver nodes, ordered by time from first to last.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Node*

**remove\_nodes** ()

Remove all maneuver nodes.

**class ControlState**

The control state of a vessel. See *SpaceCenter.Control.state*.

**full**

Full controllable.

**partial**

Partially controllable.

**none**

Not controllable.

**class ControlSource**The control source of a vessel. See *SpaceCenter.Control.source*.**kerbal**

Vessel is controlled by a Kerbal.

**probe**

Vessel is controlled by a probe core.

**none**

Vessel is not controlled.

**class SASMode**The behavior of the SAS auto-pilot. See *SpaceCenter.AutoPilot.sas\_mode*.**stability\_assist**

Stability assist mode. Dampen out any rotation.

**maneuver**

Point in the burn direction of the next maneuver node.

**prograde**

Point in the prograde direction.

**retrograde**

Point in the retrograde direction.

**normal**

Point in the orbit normal direction.

**anti\_normal**

Point in the orbit anti-normal direction.

**radial**

Point in the orbit radial direction.

**anti\_radial**

Point in the orbit anti-radial direction.

**target**

Point in the direction of the current target.

**anti\_target**

Point away from the current target.

**class SpeedMode**The mode of the speed reported in the navball. See *SpaceCenter.Control.speed\_mode*.**orbit**

Speed is relative to the vessel's orbit.

**surface**

Speed is relative to the surface of the body being orbited.

**target**

Speed is relative to the current target.

**class ControlInputMode**

See *SpaceCenter.Control.input\_mode*.

**additive**

Control inputs are added to the vessels current control inputs.

**override**

Control inputs (when they are non-zero) override the vessels current control inputs.

## 6.3.7 Communications

**class Comms**

Used to interact with CommNet for a given vessel. Obtained by calling *SpaceCenter.Vessel.comms*.

**can\_communicate**

Whether the vessel can communicate with KSC.

**Attribute** Read-only, cannot be set

**Return type** boolean

**can\_transmit\_science**

Whether the vessel can transmit science data to KSC.

**Attribute** Read-only, cannot be set

**Return type** boolean

**signal\_strength**

Signal strength to KSC.

**Attribute** Read-only, cannot be set

**Return type** number

**signal\_delay**

Signal delay to KSC in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**power**

The combined power of all active antennae on the vessel.

**Attribute** Read-only, cannot be set

**Return type** number

**control\_path**

The communication path used to control the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.CommLink*

**class CommLink**

Represents a communication node in the network. For example, a vessel or the KSC.

**type**

The type of link.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.CommLinkType*

**signal\_strength**

Signal strength of the link.

**Attribute** Read-only, cannot be set

**Return type** number

**start**

Start point of the link.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.CommNode*

**end**

Start point of the link.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.CommNode*

**class CommLinkType**

The type of a communication link. See *SpaceCenter.CommLink.type*.

**home**

Link is to a base station on Kerbin.

**control**

Link is to a control source, for example a manned spacecraft.

**relay**

Link is to a relay satellite.

**class CommNode**

Represents a communication node in the network. For example, a vessel or the KSC.

**name**

Name of the communication node.

**Attribute** Read-only, cannot be set

**Return type** string

**is\_home**

Whether the communication node is on Kerbin.

**Attribute** Read-only, cannot be set

**Return type** boolean

**is\_control\_point**

Whether the communication node is a control point, for example a manned vessel.

**Attribute** Read-only, cannot be set

**Return type** boolean

**is\_vessel**

Whether the communication node is a vessel.

**Attribute** Read-only, cannot be set

**Return type** boolean

**vessel**

The vessel for this communication node.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Vessel*

### 6.3.8 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*
  - *Antenna*
  - *Cargo Bay*
  - *Control Surface*
  - *Decoupler*
  - *Docking Port*
  - *Engine*
  - *Experiment*
  - *Fairing*
  - *Intake*
  - *Leg*
  - *Launch Clamp*
  - *Light*
  - *Parachute*
  - *Radiator*
  - *Resource Converter*
  - *Resource Harvester*
  - *Reaction Wheel*
  - *RCS*
  - *Sensor*
  - *Solar Panel*
  - *Thruster*
  - *Wheel*
- *Trees of Parts*
  - *Traversing the Tree*
  - *Attachment Modes*



- *Fuel Lines*
- *Staging*

## Parts

### class Parts

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling *SpaceCenter.Vessel.parts*.

#### all

A list of all of the vessels parts.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Part*

#### root

The vessels root part.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

---

**Note:** See the discussion on *Trees of Parts*.

---

#### controlling

The part from which the vessel is controlled.

**Attribute** Can be read or written

**Return type** *SpaceCenter.Part*

#### with\_name (name)

A list of parts whose *SpaceCenter.Part.name* is *name*.

**Parameters** **name** (*string*) –

**Return type** List of *SpaceCenter.Part*

#### with\_title (title)

A list of all parts whose *SpaceCenter.Part.title* is *title*.

**Parameters** **title** (*string*) –

**Return type** List of *SpaceCenter.Part*

#### with\_tag (tag)

A list of all parts whose *SpaceCenter.Part.tag* is *tag*.

**Parameters** **tag** (*string*) –

**Return type** List of *SpaceCenter.Part*

#### with\_module (module\_name)

A list of all parts that contain a *SpaceCenter.Module* whose *SpaceCenter.Module.name* is *module\_name*.

**Parameters** **module\_name** (*string*) –

**Return type** List of *SpaceCenter.Part*

**in\_stage** (*stage*)

A list of all parts that are activated in the given *stage*.

**Parameters** **stage** (*number*) –

**Return type** List of *SpaceCenter.Part*

---

**Note:** See the discussion on *Staging*.

---

**in\_decouple\_stage** (*stage*)

A list of all parts that are decoupled in the given *stage*.

**Parameters** **stage** (*number*) –

**Return type** List of *SpaceCenter.Part*

---

**Note:** See the discussion on *Staging*.

---

**modules\_with\_name** (*module\_name*)

A list of modules (combined across all parts in the vessel) whose *SpaceCenter.Module.name* is *module\_name*.

**Parameters** **module\_name** (*string*) –

**Return type** List of *SpaceCenter.Module*

**antennas**

A list of all antennas in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Antenna*

**cargo\_bays**

A list of all cargo bays in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.CargoBay*

**control\_surfaces**

A list of all control surfaces in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.ControlSurface*

**decouplers**

A list of all decouplers in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Decoupler*

**docking\_ports**

A list of all docking ports in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.DockingPort*

**engines**

A list of all engines in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Engine*

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters, jet engines and RCS thrusters.

---

#### **experiments**

A list of all science experiments in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Experiment*

#### **fairings**

A list of all fairings in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Fairing*

#### **intakes**

A list of all intakes in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Intake*

#### **legs**

A list of all landing legs attached to the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Leg*

#### **launch\_clamps**

A list of all launch clamps attached to the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.LaunchClamp*

#### **lights**

A list of all lights in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Light*

#### **parachutes**

A list of all parachutes in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Parachute*

#### **radiators**

A list of all radiators in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Radiator*

#### **rcs**

A list of all RCS blocks/thrusters in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.RCS*

**reaction\_wheels**

A list of all reaction wheels in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.ReactionWheel*

**resource\_converters**

A list of all resource converters in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.ResourceConverter*

**resource\_harvesters**

A list of all resource harvesters in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.ResourceHarvester*

**sensors**

A list of all sensors in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Sensor*

**solar\_panels**

A list of all solar panels in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.SolarPanel*

**wheels**

A list of all wheels in the vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Wheel*

## Part

### class Part

Represents an individual part. Vessels are made up of multiple parts. Instances of this class can be obtained by several methods in *SpaceCenter.Parts*.

**name**

Internal name of the part, as used in [part cfg files](#). For example “Mark1-2Pod”.

**Attribute** Read-only, cannot be set

**Return type** string

**title**

Title of the part, as shown when the part is right clicked in-game. For example “Mk1-2 Command Pod”.

**Attribute** Read-only, cannot be set

**Return type** string

**tag**

The name tag for the part. Can be set to a custom string using the in-game user interface.

**Attribute** Can be read or written

**Return type** string

---

**Note:** This requires either the [NameTag](#) or [kOS](#) mod to be installed.

---

**highlighted**

Whether the part is highlighted.

**Attribute** Can be read or written

**Return type** boolean

**highlight\_color**

The color used to highlight the part, as an RGB triple.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**cost**

The cost of the part, in units of funds.

**Attribute** Read-only, cannot be set

**Return type** number

**vessel**

The vessel that contains this part.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Vessel*

**parent**

The parts parent. Returns *nil* if the part does not have a parent. This, in combination with *SpaceCenter.Part.children*, can be used to traverse the vessels parts tree.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

---

**Note:** See the discussion on *Trees of Parts*.

---

**children**

The parts children. Returns an empty list if the part has no children. This, in combination with *SpaceCenter.Part.parent*, can be used to traverse the vessels parts tree.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Part*

---

**Note:** See the discussion on *Trees of Parts*.

---

**axially\_attached**

Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns *False*.

**Attribute** Read-only, cannot be set

**Return type** boolean

---

**Note:** See the discussion on *Attachment Modes*.

---

#### **radially\_attached**

Whether the part is radially attached to its parent, i.e. on the side of its parent. If the part has no parent, returns `False`.

**Attribute** Read-only, cannot be set

**Return type** boolean

---

**Note:** See the discussion on *Attachment Modes*.

---

#### **stage**

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** See the discussion on *Staging*.

---

#### **decouple\_stage**

The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

**Attribute** Read-only, cannot be set

**Return type** number

---

**Note:** See the discussion on *Staging*.

---

#### **massless**

Whether the part is `massless`.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **mass**

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

**Attribute** Read-only, cannot be set

**Return type** number

#### **dry\_mass**

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

**Attribute** Read-only, cannot be set

**Return type** number

**shielded**

Whether the part is shielded from the exterior of the vessel, for example by a fairing.

**Attribute** Read-only, cannot be set

**Return type** boolean

**dynamic\_pressure**

The dynamic pressure acting on the part, in Pascals.

**Attribute** Read-only, cannot be set

**Return type** number

**impact\_tolerance**

The impact tolerance of the part, in meters per second.

**Attribute** Read-only, cannot be set

**Return type** number

**temperature**

Temperature of the part, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

**skin\_temperature**

Temperature of the skin of the part, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

**max\_temperature**

Maximum temperature that the part can survive, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

**max\_skin\_temperature**

Maximum temperature that the skin of the part can survive, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

**thermal\_mass**

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

**thermal\_skin\_mass**

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

**thermal\_resource\_mass**

A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

**thermal\_conduction\_flux**

The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Attribute** Read-only, cannot be set

**Return type** number

**thermal\_convection\_flux**

The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Attribute** Read-only, cannot be set

**Return type** number

**thermal\_radiation\_flux**

The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Attribute** Read-only, cannot be set

**Return type** number

**thermal\_internal\_flux**

The rate at which heat energy is being generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Attribute** Read-only, cannot be set

**Return type** number

**thermal\_skin\_to\_internal\_flux**

The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

**Attribute** Read-only, cannot be set

**Return type** number

**resources**

A *SpaceCenter.Resources* object for the part.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Resources*

**crossfeed**

Whether this part is crossfeed capable.

**Attribute** Read-only, cannot be set

**Return type** boolean

**is\_fuel\_line**

Whether this part is a fuel line.



**Attribute** Read-only, cannot be set

**Return type** boolean

#### **fuel\_lines\_from**

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Part*

---

**Note:** See the discussion on *Fuel Lines*.

---

#### **fuel\_lines\_to**

The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Part*

---

**Note:** See the discussion on *Fuel Lines*.

---

#### **modules**

The modules for this part.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Module*

#### **antenna**

A *SpaceCenter.Antenna* if the part is an antenna, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Antenna*

#### **cargo\_bay**

A *SpaceCenter.CargoBay* if the part is a cargo bay, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.CargoBay*

#### **control\_surface**

A *SpaceCenter.ControlSurface* if the part is an aerodynamic control surface, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ControlSurface*

#### **decoupler**

A *SpaceCenter.Decoupler* if the part is a decoupler, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Decoupler*

#### **docking\_port**

A *SpaceCenter.DockingPort* if the part is a docking port, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.DockingPort*

**engine**

An *SpaceCenter.Engine* if the part is an engine, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Engine*

**experiment**

An *SpaceCenter.Experiment* if the part is a science experiment, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Experiment*

**fairing**

A *SpaceCenter.Fairing* if the part is a fairing, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Fairing*

**intake**

An *SpaceCenter.Intake* if the part is an intake, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Intake*

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters and jet engines. For RCS thrusters see *SpaceCenter.RCS*.

---

**leg**

A *SpaceCenter.Leg* if the part is a landing leg, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Leg*

**launch\_clamp**

A *SpaceCenter.LaunchClamp* if the part is a launch clamp, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.LaunchClamp*

**light**

A *SpaceCenter.Light* if the part is a light, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Light*

**parachute**

A *SpaceCenter.Parachute* if the part is a parachute, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Parachute*

**radiator**

A *SpaceCenter.Radiator* if the part is a radiator, otherwise nil.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Radiator*

**rcs**

A *SpaceCenter.RCS* if the part is an RCS block/thruster, otherwise *nil*.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.RCS*

**reaction\_wheel**

A *SpaceCenter.ReactionWheel* if the part is a reaction wheel, otherwise *nil*.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReactionWheel*

**resource\_converter**

A *SpaceCenter.ResourceConverter* if the part is a resource converter, otherwise *nil*.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ResourceConverter*

**resource\_harvester**

A *SpaceCenter.ResourceHarvester* if the part is a resource harvester, otherwise *nil*.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ResourceHarvester*

**sensor**

A *SpaceCenter.Sensor* if the part is a sensor, otherwise *nil*.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Sensor*

**solar\_panel**

A *SpaceCenter.SolarPanel* if the part is a solar panel, otherwise *nil*.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.SolarPanel*

**wheel**

A *SpaceCenter.Wheel* if the part is a wheel, otherwise *nil*.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Wheel*

**position** (*reference\_frame*)

The position of the part in the given reference frame.

**Parameters** **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

---

**Note:** This is a fixed position in the part, defined by the parts model. It is not necessarily the same as the parts center of mass. Use *SpaceCenter.Part.center\_of\_mass()* to get the parts center of mass.

---

**center\_of\_mass** (*reference\_frame*)

The position of the parts center of mass in the given reference frame. If the part is physicsless, this is equivalent to *SpaceCenter.Part.position()*.

**Parameters** `reference_frame` (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

**bounding\_box** (`reference_frame`)

The axis-aligned bounding box of the part in the given reference frame.

**Parameters** `reference_frame` (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

---

**Note:** This is computed from the collision mesh of the part. If the part is not collidable, the box has zero volume and is centered on the `SpaceCenter.Part.position()` of the part.

---

**direction** (`reference_frame`)

The direction the part points in, in the given reference frame.

**Parameters** `reference_frame` (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**velocity** (`reference_frame`)

The linear velocity of the part in the given reference frame.

**Parameters** `reference_frame` (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

**Return type** Tuple of (number, number, number)

**rotation** (`reference_frame`)

The rotation of the part, in the given reference frame.

**Parameters** `reference_frame` (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** Tuple of (number, number, number, number)

**moment\_of\_inertia**

The moment of inertia of the part in  $kg.m^2$  around its center of mass in the parts reference frame (`SpaceCenter.ReferenceFrame`).

**Attribute** Read-only, cannot be set

**Return type** Tuple of (number, number, number)

**inertia\_tensor**

The inertia tensor of the part in the parts reference frame (`SpaceCenter.ReferenceFrame`). Returns the 3x3 matrix as a list of elements, in row-major order.

**Attribute** Read-only, cannot be set

**Return type** List of number

**reference\_frame**

The reference frame that is fixed relative to this part, and centered on a fixed position within the part, defined by the parts model.

- The origin is at the position of the part, as returned by *SpaceCenter.Part.position()*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

---

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *SpaceCenter.DockingPort.reference\_frame*.

---

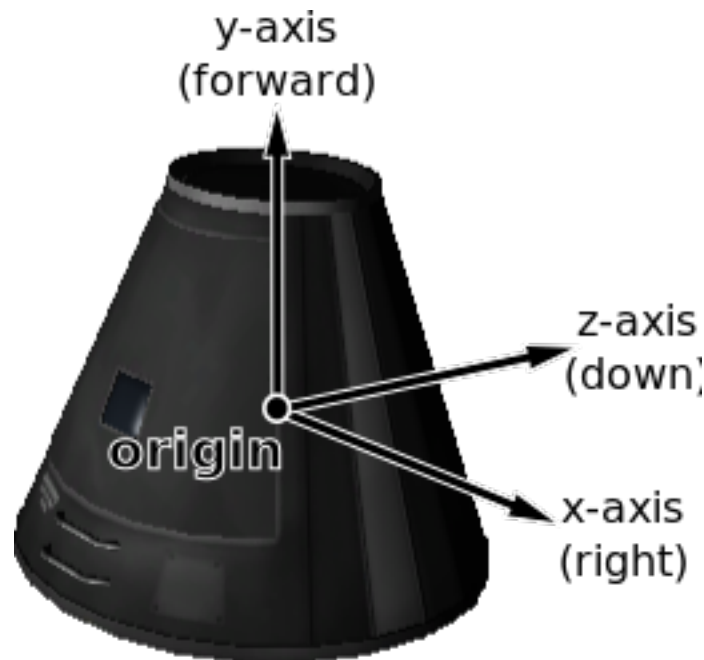


Fig. 6.7: Mk1 Command Pod reference frame origin and axes

**center\_of\_mass\_reference\_frame**

The reference frame that is fixed relative to this part, and centered on its center of mass.

- The origin is at the center of mass of the part, as returned by *SpaceCenter.Part.center\_of\_mass()*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

---

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *SpaceCenter.DockingPort.reference\_frame*.

---

**add\_force** (*force*, *position*, *reference\_frame*)

Exert a constant force on the part, acting at the given position.

**Parameters**

- **force** (*Tuple*) – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** (*Tuple*) – The position at which the force acts, as a vector.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The reference frame that the force and position are in.

**Returns** An object that can be used to remove or modify the force.

**Return type** *SpaceCenter.Force*

**instantaneous\_force** (*force*, *position*, *reference\_frame*)

Exert an instantaneous force on the part, acting at the given position.

**Parameters**

- **force** (*Tuple*) – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** (*Tuple*) – The position at which the force acts, as a vector.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The reference frame that the force and position are in.

---

**Note:** The force is applied instantaneously in a single physics update.

---

**class Force**

Obtained by calling *SpaceCenter.Part.add\_force()*.

**part**

The part that this force is applied to.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**force\_vector**

The force vector, in Newtons.

**Attribute** Can be read or written

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Return type** Tuple of (number, number, number)

**position**

The position at which the force acts, in reference frame *SpaceCenter.ReferenceFrame*.

**Attribute** Can be read or written

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

**reference\_frame**

The reference frame of the force vector and position.

**Attribute** Can be read or written

**Return type** *SpaceCenter.ReferenceFrame*

**remove()**

Remove the force.

## Module

### class Module

This can be used to interact with a specific part module. This includes part modules in stock KSP, and those added by mods.

In KSP, each part has zero or more [PartModules](#) associated with it. Each one contains some of the functionality of the part. For example, an engine has a “ModuleEngines” part module that contains all the functionality of an engine.

**name**

Name of the PartModule. For example, “ModuleEngines”.

**Attribute** Read-only, cannot be set

**Return type** string

**part**

The part that contains this module.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**fields**

The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

**Attribute** Read-only, cannot be set

**Return type** Map from string to string

**has\_field(name)**

Returns `True` if the module has a field with the given name.

**Parameters** **name** (*string*) – Name of the field.

**Return type** boolean

**get\_field(name)**

Returns the value of a field.

**Parameters** **name** (*string*) – Name of the field.

**Return type** string

**set\_field\_int(name, value)**

Set the value of a field to the given integer number.

**Parameters**

- **name** (*string*) – Name of the field.

- **value** (*number*) – Value to set.

**set\_field\_float** (*name*, *value*)

Set the value of a field to the given floating point number.

**Parameters**

- **name** (*string*) – Name of the field.
- **value** (*number*) – Value to set.

**set\_field\_string** (*name*, *value*)

Set the value of a field to the given string.

**Parameters**

- **name** (*string*) – Name of the field.
- **value** (*string*) – Value to set.

**reset\_field** (*name*)

Set the value of a field to its original value.

**Parameters** **name** (*string*) – Name of the field.

**events**

A list of the names of all of the modules events. Events are the clickable buttons visible in the right-click menu of the part.

**Attribute** Read-only, cannot be set

**Return type** List of string

**has\_event** (*name*)

True if the module has an event with the given name.

**Parameters** **name** (*string*) –

**Return type** boolean

**trigger\_event** (*name*)

Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

**Parameters** **name** (*string*) –

**actions**

A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

**Attribute** Read-only, cannot be set

**Return type** List of string

**has\_action** (*name*)

True if the part has an action with the given name.

**Parameters** **name** (*string*) –

**Return type** boolean

**set\_action** (*name* [, *value* = *True* ])

Set the value of an action with the given name.

**Parameters**

- **name** (*string*) –
- **value** (*boolean*) –



## Specific Types of Part

The following classes provide functionality for specific types of part.

- *Antenna*
- *Cargo Bay*
- *Control Surface*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Experiment*
- *Fairing*
- *Intake*
- *Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *RCS*
- *Sensor*
- *Solar Panel*
- *Thruster*
- *Wheel*

## Antenna

### **class** **Antenna**

An antenna. Obtained by calling *SpaceCenter.Part.antenna*.

#### **part**

The part object for this antenna.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

#### **state**

The current state of the antenna.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.AntennaState*

**deployable**

Whether the antenna is deployable.

**Attribute** Read-only, cannot be set

**Return type** boolean

**deployed**

Whether the antenna is deployed.

**Attribute** Can be read or written

**Return type** boolean

---

**Note:** Fixed antennas are always deployed. Returns an error if you try to deploy a fixed antenna.

---

**can\_transmit**

Whether data can be transmitted by this antenna.

**Attribute** Read-only, cannot be set

**Return type** boolean

**transmit ()**

Transmit data.

**cancel ()**

Cancel current transmission of data.

**allow\_partial**

Whether partial data transmission is permitted.

**Attribute** Can be read or written

**Return type** boolean

**power**

The power of the antenna.

**Attribute** Read-only, cannot be set

**Return type** number

**combinable**

Whether the antenna can be combined with other antennae on the vessel to boost the power.

**Attribute** Read-only, cannot be set

**Return type** boolean

**combinable\_exponent**

Exponent used to calculate the combined power of multiple antennae on a vessel.

**Attribute** Read-only, cannot be set

**Return type** number

**packet\_interval**

Interval between sending packets in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**packet\_size**

Amount of data sent per packet in Mits.

**Attribute** Read-only, cannot be set

**Return type** number

**packet\_resource\_cost**

Units of electric charge consumed per packet sent.

**Attribute** Read-only, cannot be set

**Return type** number

**class AntennaState**

The state of an antenna. See *SpaceCenter.Antenna.state*.

**deployed**

Antenna is fully deployed.

**retracted**

Antenna is fully retracted.

**deploying**

Antenna is being deployed.

**retracting**

Antenna is being retracted.

**broken**

Antenna is broken.

**Cargo Bay****class CargoBay**

A cargo bay. Obtained by calling *SpaceCenter.Part.cargo\_bay*.

**part**

The part object for this cargo bay.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**state**

The state of the cargo bay.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.CargoBayState*

**open**

Whether the cargo bay is open.

**Attribute** Can be read or written

**Return type** boolean

**class CargoBayState**

The state of a cargo bay. See *SpaceCenter.CargoBay.state*.

**open**

Cargo bay is fully open.

**closed**

Cargo bay closed and locked.

**opening**

Cargo bay is opening.

**closing**

Cargo bay is closing.

## Control Surface

**class ControlSurface**

An aerodynamic control surface. Obtained by calling *SpaceCenter.Part.control\_surface*.

**part**

The part object for this control surface.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**pitch\_enabled**

Whether the control surface has pitch control enabled.

**Attribute** Can be read or written

**Return type** boolean

**yaw\_enabled**

Whether the control surface has yaw control enabled.

**Attribute** Can be read or written

**Return type** boolean

**roll\_enabled**

Whether the control surface has roll control enabled.

**Attribute** Can be read or written

**Return type** boolean

**authority\_limiter**

The authority limiter for the control surface, which controls how far the control surface will move.

**Attribute** Can be read or written

**Return type** number

**inverted**

Whether the control surface movement is inverted.

**Attribute** Can be read or written

**Return type** boolean

**deployed**

Whether the control surface has been fully deployed.

**Attribute** Can be read or written

**Return type** boolean

**surface\_area**

Surface area of the control surface in  $m^2$ .

**Attribute** Read-only, cannot be set

**Return type** number

#### **available\_torque**

The available torque, in Newton meters, that can be produced by this control surface, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *SpaceCenter.Vessel.reference\_frame*.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

## **Decoupler**

### **class Decoupler**

A decoupler. Obtained by calling *SpaceCenter.Part.decoupler*

#### **part**

The part object for this decoupler.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

#### **decouple ()**

Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

**Return type** *SpaceCenter.Vessel*

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *SpaceCenter.active\_vessel* no longer refer to the active vessel.

---

#### **decoupled**

Whether the decoupler has fired.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **staged**

Whether the decoupler is enabled in the staging sequence.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **impulse**

The impulse that the decoupler imparts when it is fired, in Newton seconds.

**Attribute** Read-only, cannot be set

**Return type** number

## Docking Port

### **class DockingPort**

A docking port. Obtained by calling *SpaceCenter.Part.docking\_port*

#### **part**

The part object for this docking port.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

#### **state**

The current state of the docking port.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.DockingPortState*

#### **docked\_part**

The part that this docking port is docked to. Returns *nil* if this docking port is not docked to anything.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

#### **undock ()**

Undocks the docking port and returns the new *SpaceCenter.Vessel* that is created. This method can be called for either docking port in a docked pair. Throws an exception if the docking port is not docked to anything.

**Return type** *SpaceCenter.Vessel*

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *SpaceCenter.active\_vessel* no longer refer to the active vessel.

---

#### **reengage\_distance**

The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

**Attribute** Read-only, cannot be set

**Return type** number

#### **has\_shield**

Whether the docking port has a shield.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **shielded**

The state of the docking ports shield, if it has one.

Returns *True* if the docking port has a shield, and the shield is closed. Otherwise returns *False*. When set to *True*, the shield is closed, and when set to *False* the shield is opened. If the docking port does not have a shield, setting this attribute has no effect.

**Attribute** Can be read or written

**Return type** boolean

**position** (*reference\_frame*)

The position of the docking port, in the given reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

**direction** (*reference\_frame*)

The direction that docking port points in, in the given reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**rotation** (*reference\_frame*)

The rotation of the docking port, in the given reference frame.

**Parameters** **reference\_frame** (`SpaceCenter.ReferenceFrame`) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** Tuple of (number, number, number, number)

**reference\_frame**

The reference frame that is fixed relative to this docking port, and oriented with the port.

- The origin is at the position of the docking port.
- The axes rotate with the docking port.
- The x-axis points out to the right side of the docking port.
- The y-axis points in the direction the docking port is facing.
- The z-axis points out of the bottom off the docking port.

**Attribute** Read-only, cannot be set

**Return type** `SpaceCenter.ReferenceFrame`

---

**Note:** This reference frame is not necessarily equivalent to the reference frame for the part, returned by `SpaceCenter.Part.reference_frame`.

---

**class DockingPortState**

The state of a docking port. See `SpaceCenter.DockingPort.state`.

**ready**

The docking port is ready to dock to another docking port.

**docked**

The docking port is docked to another docking port, or docked to another part (from the VAB/SPH).

**docking**

The docking port is very close to another docking port, but has not docked. It is using magnetic force to acquire a solid dock.

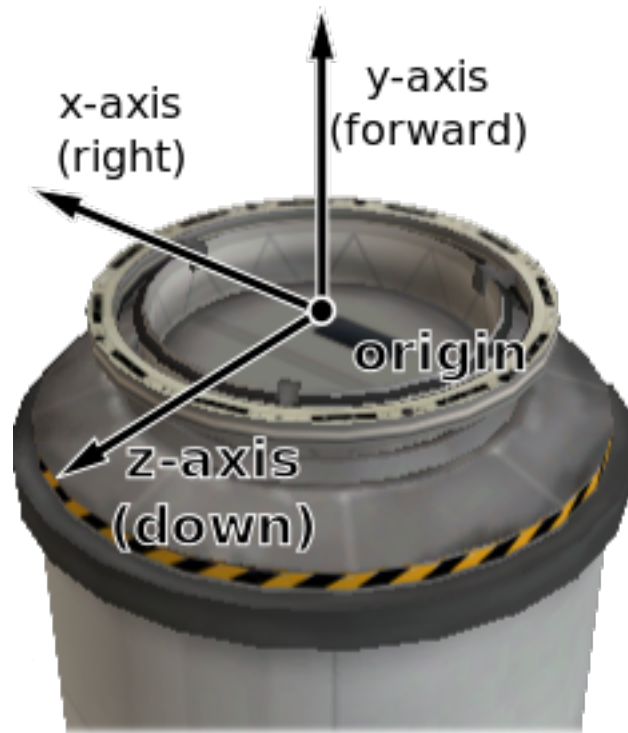


Fig. 6.8: Docking port reference frame origin and axes

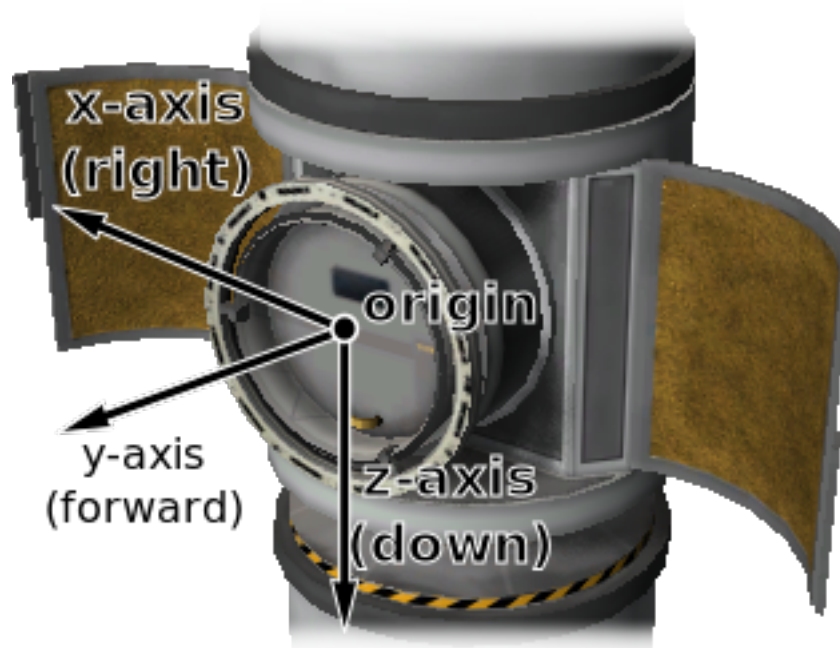


Fig. 6.9: Inline docking port reference frame origin and axes



**undocking**

The docking port has just been undocked from another docking port, and is disabled until it moves away by a sufficient distance (*SpaceCenter.DockingPort.reengage\_distance*).

**shielded**

The docking port has a shield, and the shield is closed.

**moving**

The docking ports shield is currently opening/closing.

## Engine

**class Engine**

An engine, including ones of various types. For example liquid fuelled gimballed engines, solid rocket boosters and jet engines. Obtained by calling *SpaceCenter.Part.engine*.

---

**Note:** For RCS thrusters *SpaceCenter.Part.rcs*.

---

**part**

The part object for this engine.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**active**

Whether the engine is active. Setting this attribute may have no effect, depending on *SpaceCenter.Engine.can\_shutdown* and *SpaceCenter.Engine.can\_restart*.

**Attribute** Can be read or written

**Return type** boolean

**thrust**

The current amount of thrust being produced by the engine, in Newtons.

**Attribute** Read-only, cannot be set

**Return type** number

**available\_thrust**

The amount of thrust, in Newtons, that would be produced by the engine when activated and with its throttle set to 100%. Returns zero if the engine does not have any fuel. Takes the engine's current *SpaceCenter.Engine.thrust\_limit* and atmospheric conditions into account.

**Attribute** Read-only, cannot be set

**Return type** number

**max\_thrust**

The amount of thrust, in Newtons, that would be produced by the engine when activated and fueled, with its throttle and throttle limiter set to 100%.

**Attribute** Read-only, cannot be set

**Return type** number

**max\_vacuum\_thrust**

The maximum amount of thrust that can be produced by the engine in a vacuum, in Newtons. This is the

amount of thrust produced by the engine when activated, *SpaceCenter.Engine.thrust\_limit* is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

**Attribute** Read-only, cannot be set

**Return type** number

#### **thrust\_limit**

The thrust limiter of the engine. A value between 0 and 1. Setting this attribute may have no effect, for example the thrust limit for a solid rocket booster cannot be changed in flight.

**Attribute** Can be read or written

**Return type** number

#### **thrusters**

The components of the engine that generate thrust.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Thruster*

---

**Note:** For example, this corresponds to the rocket nozzle on a solid rocket booster, or the individual nozzles on a RAPIER engine. The overall thrust produced by the engine, as reported by *SpaceCenter.Engine.available\_thrust*, *SpaceCenter.Engine.max\_thrust* and others, is the sum of the thrust generated by each thruster.

---

#### **specific\_impulse**

The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

**Attribute** Read-only, cannot be set

**Return type** number

#### **vacuum\_specific\_impulse**

The vacuum specific impulse of the engine, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

#### **kerbin\_sea\_level\_specific\_impulse**

The specific impulse of the engine at sea level on Kerbin, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

#### **propellant\_names**

The names of the propellants that the engine consumes.

**Attribute** Read-only, cannot be set

**Return type** List of string

#### **propellant\_ratios**

The ratio of resources that the engine consumes. A dictionary mapping resource names to the ratio at which they are consumed by the engine.

**Attribute** Read-only, cannot be set

**Return type** Map from string to number

---

**Note:** For example, if the ratios are 0.6 for LiquidFuel and 0.4 for Oxidizer, then for every 0.6 units of LiquidFuel that the engine burns, it will burn 0.4 units of Oxidizer.

---

**propellants**

The propellants that the engine consumes.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Propellant*

**has\_fuel**

Whether the engine has any fuel available.

**Attribute** Read-only, cannot be set

**Return type** boolean

---

**Note:** The engine must be activated for this property to update correctly.

---

**throttle**

The current throttle setting for the engine. A value between 0 and 1. This is not necessarily the same as the vessel's main throttle setting, as some engines take time to adjust their throttle (such as jet engines).

**Attribute** Read-only, cannot be set

**Return type** number

**throttle\_locked**

Whether the *SpaceCenter.Control.throttle* affects the engine. For example, this is `True` for liquid fueled rockets, and `False` for solid rocket boosters.

**Attribute** Read-only, cannot be set

**Return type** boolean

**can\_restart**

Whether the engine can be restarted once shutdown. If the engine cannot be shutdown, returns `False`. For example, this is `True` for liquid fueled rockets and `False` for solid rocket boosters.

**Attribute** Read-only, cannot be set

**Return type** boolean

**can\_shutdown**

Whether the engine can be shutdown once activated. For example, this is `True` for liquid fueled rockets and `False` for solid rocket boosters.

**Attribute** Read-only, cannot be set

**Return type** boolean

**has\_modes**

Whether the engine has multiple modes of operation.

**Attribute** Read-only, cannot be set

**Return type** boolean

**mode**

The name of the current engine mode.

**Attribute** Can be read or written

**Return type** string

**modes**

The available modes for the engine. A dictionary mapping mode names to *SpaceCenter.Engine* objects.

**Attribute** Read-only, cannot be set

**Return type** Map from string to *SpaceCenter.Engine*

**toggle\_mode()**

Toggle the current engine mode.

**auto\_mode\_switch**

Whether the engine will automatically switch modes.

**Attribute** Can be read or written

**Return type** boolean

**gimballed**

Whether the engine is gimballed.

**Attribute** Read-only, cannot be set

**Return type** boolean

**gimbal\_range**

The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

**Attribute** Read-only, cannot be set

**Return type** number

**gimbal\_locked**

Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

**Attribute** Can be read or written

**Return type** boolean

**gimbal\_limit**

The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

**Attribute** Can be read or written

**Return type** number

**available\_torque**

The available torque, in Newton meters, that can be produced by this engine, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *SpaceCenter.Vessel.reference\_frame*. Returns zero if the engine is inactive, or not gimballed.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

**class Propellant**

A propellant for an engine. Obtains by calling *SpaceCenter.Engine.propellants*.

**name**

The name of the propellant.

**Attribute** Read-only, cannot be set

**Return type** string

**current\_amount**

The current amount of propellant.

**Attribute** Read-only, cannot be set

**Return type** number

**current\_requirement**

The required amount of propellant.

**Attribute** Read-only, cannot be set

**Return type** number

**total\_resource\_available**

The total amount of the underlying resource currently reachable given resource flow rules.

**Attribute** Read-only, cannot be set

**Return type** number

**total\_resource\_capacity**

The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

**Attribute** Read-only, cannot be set

**Return type** number

**ignore\_for\_isp**

If this propellant should be ignored when calculating required mass flow given specific impulse.

**Attribute** Read-only, cannot be set

**Return type** boolean

**ignore\_for\_thrust\_curve**

If this propellant should be ignored for thrust curve calculations.

**Attribute** Read-only, cannot be set

**Return type** boolean

**draw\_stack\_gauge**

If this propellant has a stack gauge or not.

**Attribute** Read-only, cannot be set

**Return type** boolean

**is\_deprived**

If this propellant is deprived.

**Attribute** Read-only, cannot be set

**Return type** boolean

**ratio**

The propellant ratio.

**Attribute** Read-only, cannot be set

**Return type** number

## Experiment

### **class Experiment**

Obtained by calling *SpaceCenter.Part.experiment*.

#### **part**

The part object for this experiment.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

#### **run()**

Run the experiment.

#### **transmit()**

Transmit all experimental data contained by this part.

#### **dump()**

Dump the experimental data contained by the experiment.

#### **reset()**

Reset the experiment.

#### **deployed**

Whether the experiment has been deployed.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **rerunnable**

Whether the experiment can be re-run.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **inoperable**

Whether the experiment is inoperable.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **has\_data**

Whether the experiment contains data.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **data**

The data contained in this experiment.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.ScienceData*

#### **biome**

The name of the biome the experiment is currently in.

**Attribute** Read-only, cannot be set

**Return type** string

**available**

Determines if the experiment is available given the current conditions.

**Attribute** Read-only, cannot be set

**Return type** boolean

**science\_subject**

Containing information on the corresponding specific science result for the current conditions. Returns *nil* if the experiment is unavailable.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ScienceSubject*

**class ScienceData**

Obtained by calling *SpaceCenter.Experiment.data*.

**data\_amount**

Data amount.

**Attribute** Read-only, cannot be set

**Return type** number

**science\_value**

Science value.

**Attribute** Read-only, cannot be set

**Return type** number

**transmit\_value**

Transmit value.

**Attribute** Read-only, cannot be set

**Return type** number

**class ScienceSubject**

Obtained by calling *SpaceCenter.Experiment.science\_subject*.

**title**

Title of science subject, displayed in science archives

**Attribute** Read-only, cannot be set

**Return type** string

**is\_complete**

Whether the experiment has been completed.

**Attribute** Read-only, cannot be set

**Return type** boolean

**science**

Amount of science already earned from this subject, not updated until after transmission/recovery.

**Attribute** Read-only, cannot be set

**Return type** number

**science\_cap**

Total science allowable for this subject.

**Attribute** Read-only, cannot be set

**Return type** number

**data\_scale**

Multiply science value by this to determine data amount in mits.

**Attribute** Read-only, cannot be set

**Return type** number

**subject\_value**

Multiplier for specific Celestial Body/Experiment Situation combination.

**Attribute** Read-only, cannot be set

**Return type** number

**scientific\_value**

Diminishing value multiplier for decreasing the science value returned from repeated experiments.

**Attribute** Read-only, cannot be set

**Return type** number

## Fairing

**class Fairing**

A fairing. Obtained by calling *SpaceCenter.Part.fairing*.

**part**

The part object for this fairing.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**jettison()**

Jettison the fairing. Has no effect if it has already been jettisoned.

**jettisoned**

Whether the fairing has been jettisoned.

**Attribute** Read-only, cannot be set

**Return type** boolean

## Intake

**class Intake**

An air intake. Obtained by calling *SpaceCenter.Part.intake*.

**part**

The part object for this intake.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**open**

Whether the intake is open.

**Attribute** Can be read or written

**Return type** boolean



**speed**

Speed of the flow into the intake, in  $m/s$ .

**Attribute** Read-only, cannot be set

**Return type** number

**flow**

The rate of flow into the intake, in units of resource per second.

**Attribute** Read-only, cannot be set

**Return type** number

**area**

The area of the intake's opening, in square meters.

**Attribute** Read-only, cannot be set

**Return type** number

**Leg****class Leg**

A landing leg. Obtained by calling *SpaceCenter.Part.leg*.

**part**

The part object for this landing leg.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**state**

The current state of the landing leg.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.LegState*

**deployable**

Whether the leg is deployable.

**Attribute** Read-only, cannot be set

**Return type** boolean

**deployed**

Whether the landing leg is deployed.

**Attribute** Can be read or written

**Return type** boolean

---

**Note:** Fixed landing legs are always deployed. Returns an error if you try to deploy fixed landing gear.

---

**is\_grounded**

Returns whether the leg is touching the ground.

**Attribute** Read-only, cannot be set

**Return type** boolean

**class LegState**

The state of a landing leg. See *SpaceCenter.Leg.state*.

**deployed**

Landing leg is fully deployed.

**retracted**

Landing leg is fully retracted.

**deploying**

Landing leg is being deployed.

**retracting**

Landing leg is being retracted.

**broken**

Landing leg is broken.

**Launch Clamp****class LaunchClamp**

A launch clamp. Obtained by calling *SpaceCenter.Part.launch\_clamp*.

**part**

The part object for this launch clamp.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**release ()**

Releases the docking clamp. Has no effect if the clamp has already been released.

**Light****class Light**

A light. Obtained by calling *SpaceCenter.Part.light*.

**part**

The part object for this light.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**active**

Whether the light is switched on.

**Attribute** Can be read or written

**Return type** boolean

**color**

The color of the light, as an RGB triple.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**power\_usage**

The current power usage, in units of charge per second.

**Attribute** Read-only, cannot be set

**Return type** number

## Parachute

### class Parachute

A parachute. Obtained by calling *SpaceCenter.Part.parachute*.

#### part

The part object for this parachute.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

#### deploy()

Deploys the parachute. This has no effect if the parachute has already been deployed.

#### deployed

Whether the parachute has been deployed.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### arm()

Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

#### armed

Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### state

The current state of the parachute.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ParachuteState*

#### deploy\_altitude

The altitude at which the parachute will full deploy, in meters. Only applicable to stock parachutes.

**Attribute** Can be read or written

**Return type** number

#### deploy\_min\_pressure

The minimum pressure at which the parachute will semi-deploy, in atmospheres. Only applicable to stock parachutes.

**Attribute** Can be read or written

**Return type** number

### class ParachuteState

The state of a parachute. See *SpaceCenter.Parachute.state*.

#### stowed

The parachute is safely tucked away inside its housing.

**armed**

The parachute is armed for deployment. (RealChutes only)

**active**

The parachute is still stowed, but ready to semi-deploy. (Stock parachutes only)

**semi\_deployed**

The parachute has been deployed and is providing some drag, but is not fully deployed yet. (Stock parachutes only)

**deployed**

The parachute is fully deployed.

**cut**

The parachute has been cut.

## Radiator

**class Radiator**

A radiator. Obtained by calling *SpaceCenter.Part.radiator*.

**part**

The part object for this radiator.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**deployable**

Whether the radiator is deployable.

**Attribute** Read-only, cannot be set

**Return type** boolean

**deployed**

For a deployable radiator, *True* if the radiator is extended. If the radiator is not deployable, this is always *True*.

**Attribute** Can be read or written

**Return type** boolean

**state**

The current state of the radiator.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.RadiatorState*

---

**Note:** A fixed radiator is always *SpaceCenter.RadiatorState.extended*.

---

**class RadiatorState**

The state of a radiator. *SpaceCenter.RadiatorState*

**extended**

Radiator is fully extended.

**retracted**

Radiator is fully retracted.

**extending**

Radiator is being extended.

**retracting**

Radiator is being retracted.

**broken**

Radiator is being broken.

## Resource Converter

### **class ResourceConverter**

A resource converter. Obtained by calling *SpaceCenter.Part.resource\_converter*.

**part**

The part object for this converter.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**count**

The number of converters in the part.

**Attribute** Read-only, cannot be set

**Return type** number

**name (index)**

The name of the specified converter.

**Parameters** **index** (*number*) – Index of the converter.

**Return type** string

**active (index)**

True if the specified converter is active.

**Parameters** **index** (*number*) – Index of the converter.

**Return type** boolean

**start (index)**

Start the specified converter.

**Parameters** **index** (*number*) – Index of the converter.

**stop (index)**

Stop the specified converter.

**Parameters** **index** (*number*) – Index of the converter.

**state (index)**

The state of the specified converter.

**Parameters** **index** (*number*) – Index of the converter.

**Return type** *SpaceCenter.ResourceConverterState*

**status\_info (index)**

Status information for the specified converter. This is the full status message shown in the in-game UI.

**Parameters** **index** (*number*) – Index of the converter.

**Return type** string

**inputs** (*index*)

List of the names of resources consumed by the specified converter.

**Parameters** **index** (*number*) – Index of the converter.

**Return type** List of string

**outputs** (*index*)

List of the names of resources produced by the specified converter.

**Parameters** **index** (*number*) – Index of the converter.

**Return type** List of string

**class ResourceConverterState**

The state of a resource converter. See *SpaceCenter.ResourceConverter.state()*.

**running**

Converter is running.

**idle**

Converter is idle.

**missing\_resource**

Converter is missing a required resource.

**storage\_full**

No available storage for output resource.

**capacity**

At preset resource capacity.

**unknown**

Unknown state. Possible with modified resource converters. In this case, check *SpaceCenter.ResourceConverter.status\_info()* for more information.

## Resource Harvester

**class ResourceHarvester**

A resource harvester (drill). Obtained by calling *SpaceCenter.Part.resource\_harvester*.

**part**

The part object for this harvester.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**state**

The state of the harvester.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ResourceHarvesterState*

**deployed**

Whether the harvester is deployed.

**Attribute** Can be read or written

**Return type** boolean

**active**

Whether the harvester is actively drilling.

**Attribute** Can be read or written

**Return type** boolean

#### **extraction\_rate**

The rate at which the drill is extracting ore, in units per second.

**Attribute** Read-only, cannot be set

**Return type** number

#### **thermal\_efficiency**

The thermal efficiency of the drill, as a percentage of its maximum.

**Attribute** Read-only, cannot be set

**Return type** number

#### **core\_temperature**

The core temperature of the drill, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

#### **optimum\_core\_temperature**

The core temperature at which the drill will operate with peak efficiency, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** number

### **class ResourceHarvesterState**

The state of a resource harvester. See *SpaceCenter.ResourceHarvester.state*.

#### **deploying**

The drill is deploying.

#### **deployed**

The drill is deployed and ready.

#### **retracting**

The drill is retracting.

#### **retracted**

The drill is retracted.

#### **active**

The drill is running.

## **Reaction Wheel**

### **class ReactionWheel**

A reaction wheel. Obtained by calling *SpaceCenter.Part.reaction\_wheel*.

#### **part**

The part object for this reaction wheel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

#### **active**

Whether the reaction wheel is active.

**Attribute** Can be read or written

**Return type** boolean

**broken**

Whether the reaction wheel is broken.

**Attribute** Read-only, cannot be set

**Return type** boolean

**available\_torque**

The available torque, in Newton meters, that can be produced by this reaction wheel, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *SpaceCenter.Vessel.reference\_frame*. Returns zero if the reaction wheel is inactive or broken.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

**max\_torque**

The maximum torque, in Newton meters, that can be produced by this reaction wheel, when it is active, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *SpaceCenter.Vessel.reference\_frame*.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

## RCS

**class RCS**

An RCS block or thruster. Obtained by calling *SpaceCenter.Part.rcs*.

**part**

The part object for this RCS.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**active**

Whether the RCS thrusters are active. An RCS thruster is inactive if the RCS action group is disabled (*SpaceCenter.Control.rcs*), the RCS thruster itself is not enabled (*SpaceCenter.RCS.enabled*) or it is covered by a fairing (*SpaceCenter.Part.shielded*).

**Attribute** Read-only, cannot be set

**Return type** boolean

**enabled**

Whether the RCS thrusters are enabled.

**Attribute** Can be read or written

**Return type** boolean

**pitch\_enabled**

Whether the RCS thruster will fire when pitch control input is given.



**Attribute** Can be read or written

**Return type** boolean

#### **yaw\_enabled**

Whether the RCS thruster will fire when yaw control input is given.

**Attribute** Can be read or written

**Return type** boolean

#### **roll\_enabled**

Whether the RCS thruster will fire when roll control input is given.

**Attribute** Can be read or written

**Return type** boolean

#### **forward\_enabled**

Whether the RCS thruster will fire when pitch control input is given.

**Attribute** Can be read or written

**Return type** boolean

#### **up\_enabled**

Whether the RCS thruster will fire when yaw control input is given.

**Attribute** Can be read or written

**Return type** boolean

#### **right\_enabled**

Whether the RCS thruster will fire when roll control input is given.

**Attribute** Can be read or written

**Return type** boolean

#### **available\_torque**

The available torque, in Newton meters, that can be produced by this RCS, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *SpaceCenter.Vessel.reference\_frame*. Returns zero if RCS is disabled.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (Tuple of (number, number, number), Tuple of (number, number, number))

#### **max\_thrust**

The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

**Attribute** Read-only, cannot be set

**Return type** number

#### **max\_vacuum\_thrust**

The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

**Attribute** Read-only, cannot be set

**Return type** number

#### **thrusters**

A list of thrusters, one of each nozzle in the RCS part.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Thruster*

**specific\_impulse**

The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

**Attribute** Read-only, cannot be set

**Return type** number

**vacuum\_specific\_impulse**

The vacuum specific impulse of the RCS, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**kerbin\_sea\_level\_specific\_impulse**

The specific impulse of the RCS at sea level on Kerbin, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**propellants**

The names of resources that the RCS consumes.

**Attribute** Read-only, cannot be set

**Return type** List of string

**propellant\_ratios**

The ratios of resources that the RCS consumes. A dictionary mapping resource names to the ratios at which they are consumed by the RCS.

**Attribute** Read-only, cannot be set

**Return type** Map from string to number

**has\_fuel**

Whether the RCS has fuel available.

**Attribute** Read-only, cannot be set

**Return type** boolean

---

**Note:** The RCS thruster must be activated for this property to update correctly.

---

## Sensor

**class Sensor**

A sensor, such as a thermometer. Obtained by calling *SpaceCenter.Part.sensor*.

**part**

The part object for this sensor.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**active**

Whether the sensor is active.

**Attribute** Can be read or written

**Return type** boolean

**value**

The current value of the sensor.

**Attribute** Read-only, cannot be set

**Return type** string

## Solar Panel

### class SolarPanel

A solar panel. Obtained by calling *SpaceCenter.Part.solar\_panel*.

**part**

The part object for this solar panel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**deployable**

Whether the solar panel is deployable.

**Attribute** Read-only, cannot be set

**Return type** boolean

**deployed**

Whether the solar panel is extended.

**Attribute** Can be read or written

**Return type** boolean

**state**

The current state of the solar panel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.SolarPanelState*

**energy\_flow**

The current amount of energy being generated by the solar panel, in units of charge per second.

**Attribute** Read-only, cannot be set

**Return type** number

**sun\_exposure**

The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

**Attribute** Read-only, cannot be set

**Return type** number

### class SolarPanelState

The state of a solar panel. See *SpaceCenter.SolarPanel.state*.

**extended**

Solar panel is fully extended.

**retracted**

Solar panel is fully retracted.

**extending**

Solar panel is being extended.

**retracting**

Solar panel is being retracted.

**broken**

Solar panel is broken.

## Thruster

**class Thruster**

The component of an *SpaceCenter.Engine* or *SpaceCenter.RCS* part that generates thrust. Can obtained by calling *SpaceCenter.Engine.thrusters* or *SpaceCenter.RCS.thrusters*.

---

**Note:** Engines can consist of multiple thrusters. For example, the S3 KS-25x4 “Mammoth” has four rocket nozzels, and so consists of four thrusters.

---

**part**

The *SpaceCenter.Part* that contains this thruster.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**thrust\_position** (*reference\_frame*)

The position at which the thruster generates thrust, in the given reference frame. For gimballed engines, this takes into account the current rotation of the gimbal.

**Parameters** **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

**thrust\_direction** (*reference\_frame*)

The direction of the force generated by the thruster, in the given reference frame. This is opposite to the direction in which the thruster expels propellant. For gimballed engines, this takes into account the current rotation of the gimbal.

**Parameters** **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

**thrust\_reference\_frame**

A reference frame that is fixed relative to the thruster and orientated with its thrust direction (*SpaceCenter.Thruster.thrust\_direction()*). For gimballed engines, this takes into account the current rotation of the gimbal.

- The origin is at the position of thrust for this thruster (*SpaceCenter.Thruster.thrust\_position()*).

- The axes rotate with the thrust direction. This is the direction in which the thruster expels propellant, including any gimbaling.
- The y-axis points along the thrust direction.
- The x-axis and z-axis are perpendicular to the thrust direction.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

#### **`gimballed`**

Whether the thruster is gimballed.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **`gimbal_position`** (*reference\_frame*)

Position around which the gimbal pivots.

**Parameters** **`reference_frame`** (*SpaceCenter.ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

#### **`gimbal_angle`**

The current gimbal angle in the pitch, roll and yaw axes, in degrees.

**Attribute** Read-only, cannot be set

**Return type** Tuple of (number, number, number)

#### **`initial_thrust_position`** (*reference\_frame*)

The position at which the thruster generates thrust, when the engine is in its initial position (no gimbaling), in the given reference frame.

**Parameters** **`reference_frame`** (*SpaceCenter.ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

---

**Note:** This position can move when the gimbal rotates. This is because the thrust position and gimbal position are not necessarily the same.

---

#### **`initial_thrust_direction`** (*reference\_frame*)

The direction of the force generated by the thruster, when the engine is in its initial position (no gimbaling), in the given reference frame. This is opposite to the direction in which the thruster expels propellant.

**Parameters** **`reference_frame`** (*SpaceCenter.ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

## Wheel

### **class Wheel**

A wheel. Includes landing gear and rover wheels. Obtained by calling *SpaceCenter.Part.wheel*. Can be used to control the motors, steering and deployment of wheels, among other things.

#### **part**

The part object for this wheel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

#### **state**

The current state of the wheel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.WheelState*

#### **radius**

Radius of the wheel, in meters.

**Attribute** Read-only, cannot be set

**Return type** number

#### **grounded**

Whether the wheel is touching the ground.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **has\_brakes**

Whether the wheel has brakes.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **brakes**

The braking force, as a percentage of maximum, when the brakes are applied.

**Attribute** Can be read or written

**Return type** number

#### **auto\_friction\_control**

Whether automatic friction control is enabled.

**Attribute** Can be read or written

**Return type** boolean

#### **manual\_friction\_control**

Manual friction control value. Only has an effect if automatic friction control is disabled. A value between 0 and 5 inclusive.

**Attribute** Can be read or written

**Return type** number

#### **deployable**

Whether the wheel is deployable.

**Attribute** Read-only, cannot be set

**Return type** boolean

**deployed**

Whether the wheel is deployed.

**Attribute** Can be read or written

**Return type** boolean

**powered**

Whether the wheel is powered by a motor.

**Attribute** Read-only, cannot be set

**Return type** boolean

**motor\_enabled**

Whether the motor is enabled.

**Attribute** Can be read or written

**Return type** boolean

**motor\_inverted**

Whether the direction of the motor is inverted.

**Attribute** Can be read or written

**Return type** boolean

**motor\_state**

Whether the direction of the motor is inverted.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.MotorState*

**motor\_output**

The output of the motor. This is the torque currently being generated, in Newton meters.

**Attribute** Read-only, cannot be set

**Return type** number

**traction\_control\_enabled**

Whether automatic traction control is enabled. A wheel only has traction control if it is powered.

**Attribute** Can be read or written

**Return type** boolean

**traction\_control**

Setting for the traction control. Only takes effect if the wheel has automatic traction control enabled. A value between 0 and 5 inclusive.

**Attribute** Can be read or written

**Return type** number

**drive\_limiter**

Manual setting for the motor limiter. Only takes effect if the wheel has automatic traction control disabled. A value between 0 and 100 inclusive.

**Attribute** Can be read or written

**Return type** number

**steerable**

Whether the wheel has steering.

**Attribute** Read-only, cannot be set

**Return type** boolean

**steering\_enabled**

Whether the wheel steering is enabled.

**Attribute** Can be read or written

**Return type** boolean

**steering\_inverted**

Whether the wheel steering is inverted.

**Attribute** Can be read or written

**Return type** boolean

**has\_suspension**

Whether the wheel has suspension.

**Attribute** Read-only, cannot be set

**Return type** boolean

**suspension\_spring\_strength**

Suspension spring strength, as set in the editor.

**Attribute** Read-only, cannot be set

**Return type** number

**suspension\_damper\_strength**

Suspension damper strength, as set in the editor.

**Attribute** Read-only, cannot be set

**Return type** number

**broken**

Whether the wheel is broken.

**Attribute** Read-only, cannot be set

**Return type** boolean

**repairable**

Whether the wheel is repairable.

**Attribute** Read-only, cannot be set

**Return type** boolean

**stress**

Current stress on the wheel.

**Attribute** Read-only, cannot be set

**Return type** number

**stress\_tolerance**

Stress tolerance of the wheel.

**Attribute** Read-only, cannot be set



**Return type** number

**stress\_percentage**

Current stress on the wheel as a percentage of its stress tolerance.

**Attribute** Read-only, cannot be set

**Return type** number

**deflection**

Current deflection of the wheel.

**Attribute** Read-only, cannot be set

**Return type** number

**slip**

Current slip of the wheel.

**Attribute** Read-only, cannot be set

**Return type** number

**class WheelState**

The state of a wheel. See *SpaceCenter.Wheel.state*.

**deployed**

Wheel is fully deployed.

**retracted**

Wheel is fully retracted.

**deploying**

Wheel is being deployed.

**retracting**

Wheel is being retracted.

**broken**

Wheel is broken.

**class MotorState**

The state of the motor on a powered wheel. See *SpaceCenter.Wheel.motor\_state*.

**idle**

The motor is idle.

**running**

The motor is running.

**disabled**

The motor is disabled.

**inoperable**

The motor is inoperable.

**not\_enough\_resources**

The motor does not have enough resources to run.

## Trees of Parts

Vessels in KSP are comprised of a number of parts, connected to one another in a *tree* structure. An example vessel is shown in Figure 1, and the corresponding tree of parts in Figure 2. The craft file for this example can also be [downloaded here](#).

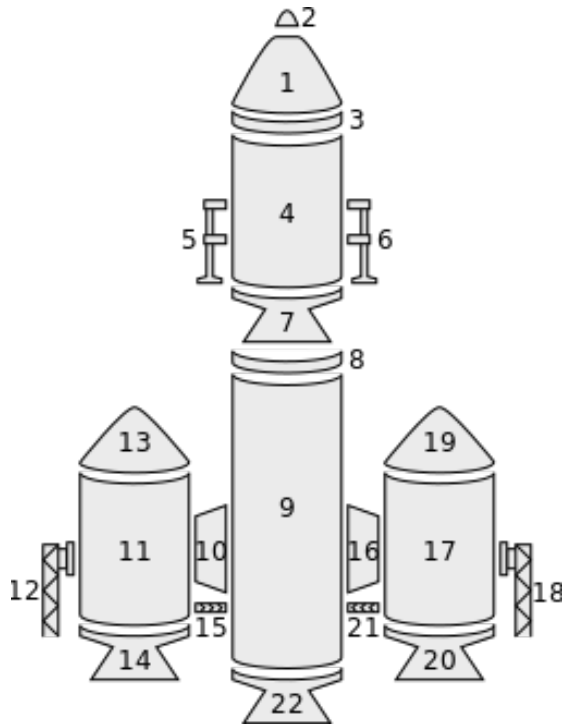


Fig. 6.10: **Figure 1** – Example parts making up a vessel.

### Traversing the Tree

The tree of parts can be traversed using the attributes `SpaceCenter.Parts.root`, `SpaceCenter.Part.parent` and `SpaceCenter.Part.children`.

The root of the tree is the same as the vessel's *root* part (part number 1 in the example above) and can be obtained by calling `SpaceCenter.Parts.root`. A part's children can be obtained by calling `SpaceCenter.Part.children`. If the part does not have any children, `SpaceCenter.Part.children` returns an empty list. A part's parent can be obtained by calling `SpaceCenter.Part.parent`. If the part does not have a parent (as is the case for the root part), `SpaceCenter.Part.parent` returns `nil`.

The following Lua example uses these attributes to perform a depth-first traversal over all of the parts in a vessel:

```
local krpc = require 'krpc'
local conn = krpc.connect()
local vessel = conn.vessel

local root = vessel.root
local stack = {root}
while #stack > 0 do
    local part = stack.pop()
    print(string.rep(' ', part.depth) .. part.name)
    for _, child in ipairs(part.children) do
        stack.push(child)
    end
end
```

When this code is executed using the craft file for the example vessel pictured above, the following is printed out:

```
Command Pod Mk1
TR-18A Stack Decoupler
FL-T400 Fuel Tank
LV-909 Liquid Fuel Tank
TR-18A Stack Decoupler
FL-T800 Fuel Tank
LV-909 Liquid Fuel Tank
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launcher
FTX-2 External Tank
LV-909 Liquid Fuel Tank
Aerodynamic Surface
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launcher
FTX-2 External Tank
LV-909 Liquid Fuel Tank
```

	Aerodynamic
LT-1	Landing Stru
LT-1	Landing Stru
Mk16	Parachute

Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

The root part of a vessel (for example the command pod – part 1) does not have a parent part, so does not have an attachment mode. However, the part is consider to be *axially* attached to nothing.

The following Lua example does a depth-first traversal as before, but also prints out the attachment mode used by the part:

```

local krpc = require 'krpc'
local conn = krpc.connect()
local vessel = conn.space_center.active_vessel

local root = vessel.parts.root
local stack = {{root, 0}}
while #stack > 0 do
    local
    ↪part,depth = unpack(table.remove(stack))
    local attach_mode
    if part.axially_attached then
        attach_mode = 'axial'
    else -- radially_attached
        attach_mode = 'radial'
    end
    print(string.rep(' ', depth) ..
    ↪.. part.title .. ' - ' .. attach_mode)
    for _,child in ipairs(part.children) do
        table.insert(stack, {child, depth+1})
    end
end
end

```

When this code is execute using the craft file for the example vessel pictured above, the following is printed out:

```

Command Pod Mk1 - axial
TR-18A Stack Decoupler - axial
FL-T400 Fuel Tank - axial
  LV-909 Liquid Fuel Engine - axial
  TR-18A Stack Decoupler - axial
  FL-T800 Fuel Tank - axial
    LV-909 Liquid Fuel Engine - axial
    TT-70 Radial Decoupler - radial

```

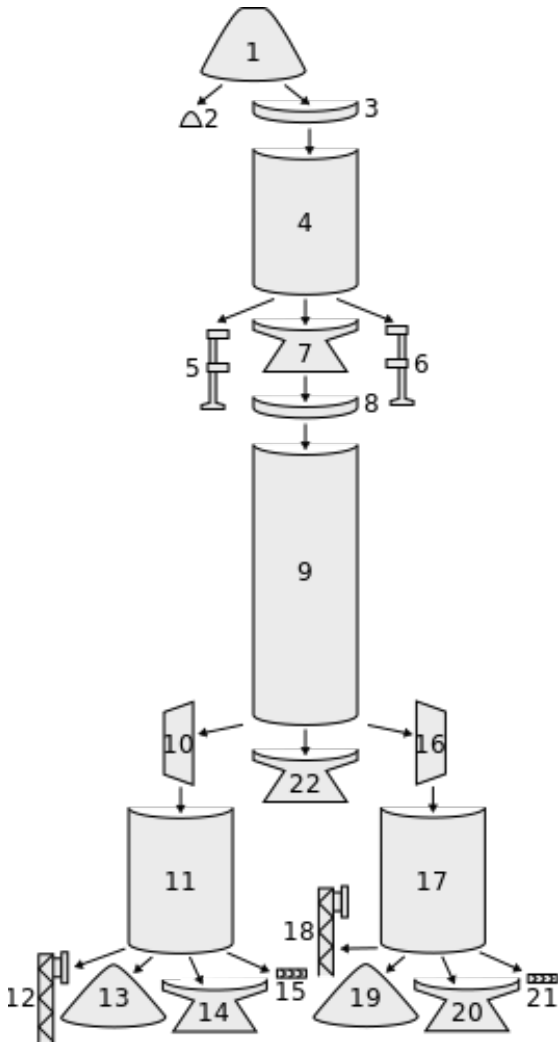


Figure 6.11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the child part.

```

    FL-T400 Fuel Tank - radial
  ↳ TT18-A Launch Stability Enhancer - radial
    FTX-2 External Fuel Duct - radial
    LV-909 Liquid Fuel Engine - axial
    Aerodynamic Nose Cone - axial
    TT-70 Radial Decoupler - radial
    FL-T400 Fuel Tank - radial
  ↳ TT18-A Launch Stability Enhancer - radial
    FTX-2 External Fuel Duct - radial
    LV-909 Liquid Fuel Engine - axial
    Aerodynamic Nose Cone - axial
    LT-1 Landing Struts - radial
    LT-1 Landing Struts - radial
    Mk16 Parachute - axial

```

## Fuel Lines

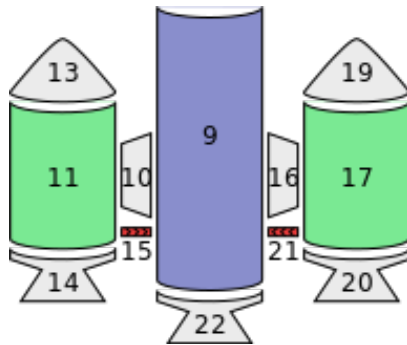
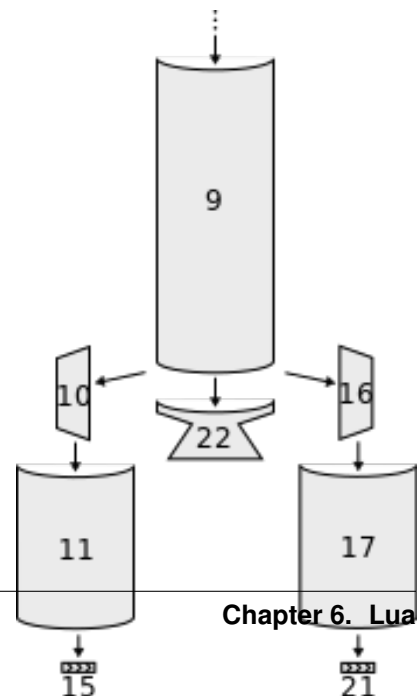


Fig. 6.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.

The attributes `SpaceCenter.Part.fuel_lines_from` and `SpaceCenter.Part.fuel_lines_to` can be used to discover these connections. In the example in Figure 5, when `SpaceCenter.Part.fuel_lines_to` is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When `SpaceCenter.Part.fuel_lines_from` is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).



## Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using `SpaceCenter.Part.stage` and `SpaceCenter.Part.decouple_stage` respectively. For parts that are not activated by staging, `SpaceCenter.Part.stage` returns -1. For parts that are never decoupled, `SpaceCenter.Part.decouple_stage` returns a value of -1.

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *activated*. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

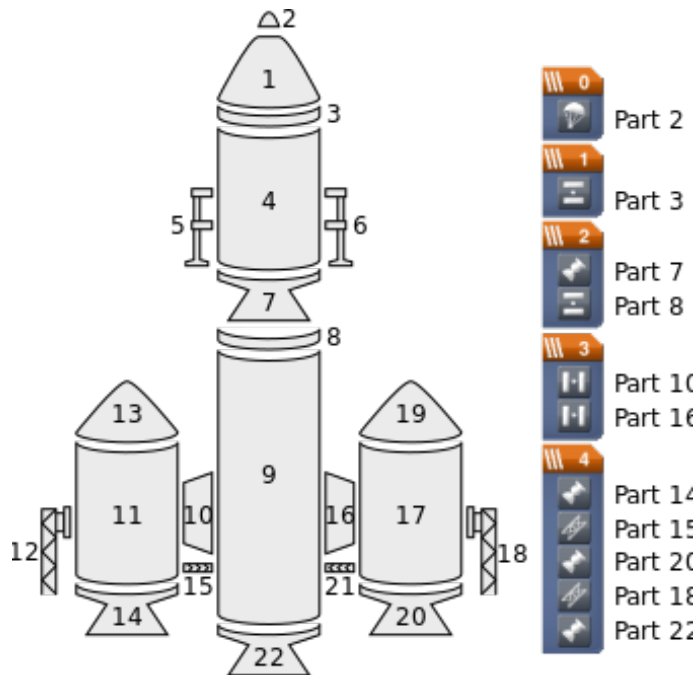


Fig. 6.14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

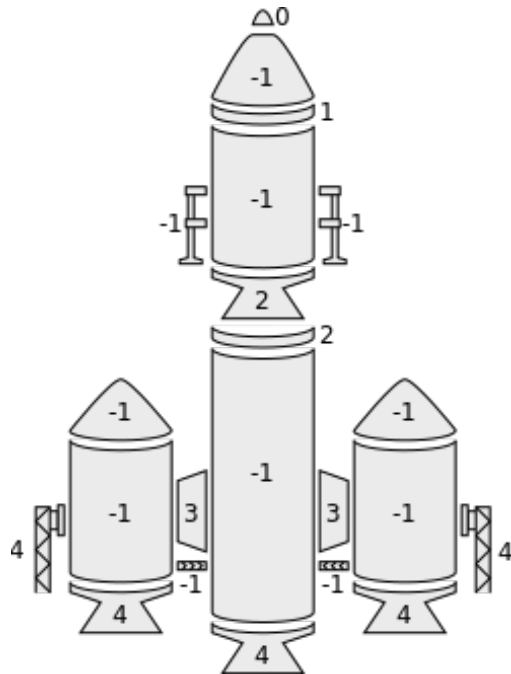


Fig. 6.15: **Figure 7** – The stage in which each part is *activated*.

### 6.3.9 Resources

#### class Resources

Represents the collection of resources stored in a vessel, stage or part. Created by calling `SpaceCenter.Vessel.resources`, `SpaceCenter.Vessel.resources_in_decouple_stage()` or `SpaceCenter.Part.resources`.

#### all

All the individual resources that can be stored.

**Attribute** Read-only, cannot be set

**Return type** List of `SpaceCenter.Resource`

#### with\_resource(name)

All the individual resources with the given name that can be stored.

**Parameters** `name(string)` –

**Return type** List of `SpaceCenter.Resource`

#### names

A list of resource names that can be stored.

**Attribute** Read-only, cannot be set

**Return type** List of string

#### has\_resource(name)

Check whether the named resource can be stored.

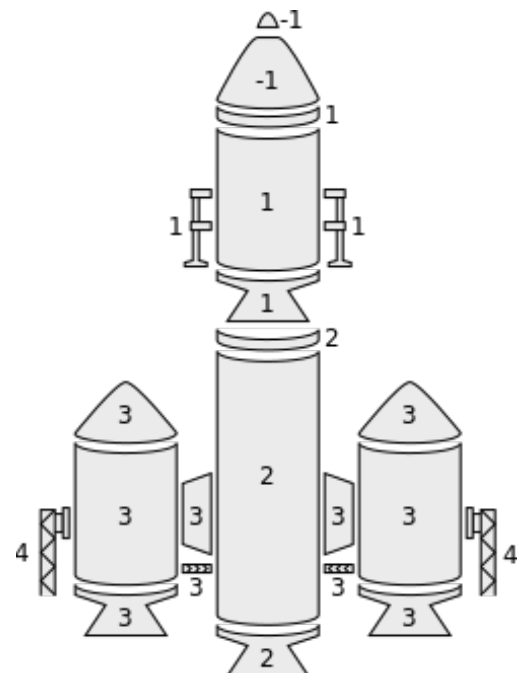


Fig. 6.16: **Figure 8** – The stage in which each part is *decoupled*.

**Parameters** **name** (*string*) – The name of the resource.

**Return type** boolean

**amount** (*name*)

Returns the amount of a resource that is currently stored.

**Parameters** **name** (*string*) – The name of the resource.

**Return type** number

**max** (*name*)

Returns the amount of a resource that can be stored.

**Parameters** **name** (*string*) – The name of the resource.

**Return type** number

**static density** (*name*)

Returns the density of a resource, in *kg/l*.

**Parameters** **name** (*string*) – The name of the resource.

**Return type** number

**static flow\_mode** (*name*)

Returns the flow mode of a resource.

**Parameters** **name** (*string*) – The name of the resource.

**Return type** *SpaceCenter.ResourceFlowMode*

**enabled**

Whether use of all the resources are enabled.

**Attribute** Can be read or written

**Return type** boolean

---

**Note:** This is `True` if all of the resources are enabled. If any of the resources are not enabled, this is `False`.

---

## class Resource

An individual resource stored within a part. Created using methods in the *SpaceCenter.Resources* class.

**name**

The name of the resource.

**Attribute** Read-only, cannot be set

**Return type** string

**part**

The part containing the resource.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**amount**

The amount of the resource that is currently stored in the part.

**Attribute** Read-only, cannot be set

**Return type** number

**max**

The total amount of the resource that can be stored in the part.

**Attribute** Read-only, cannot be set

**Return type** number

**density**

The density of the resource, in *kg/l*.

**Attribute** Read-only, cannot be set

**Return type** number

**flow\_mode**

The flow mode of the resource.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ResourceFlowMode*

**enabled**

Whether use of this resource is enabled.

**Attribute** Can be read or written

**Return type** boolean

**class ResourceTransfer**

Transfer resources between parts.

**static start** (*from\_part, to\_part, resource, max\_amount*)

Start transferring a resource transfer between a pair of parts. The transfer will move at most *max\_amount* units of the resource, depending on how much of the resource is available in the source part and how much storage is available in the destination part. Use *SpaceCenter.ResourceTransfer.complete* to check if the transfer is complete. Use *SpaceCenter.ResourceTransfer.amount* to see how much of the resource has been transferred.

**Parameters**

- **from\_part** (*SpaceCenter.Part*) – The part to transfer to.
- **to\_part** (*SpaceCenter.Part*) – The part to transfer from.
- **resource** (*string*) – The name of the resource to transfer.
- **max\_amount** (*number*) – The maximum amount of resource to transfer.

**Return type** *SpaceCenter.ResourceTransfer*

**amount**

The amount of the resource that has been transferred.

**Attribute** Read-only, cannot be set

**Return type** number

**complete**

Whether the transfer has completed.

**Attribute** Read-only, cannot be set



**Return type** boolean

#### **class ResourceFlowMode**

The way in which a resource flows between parts. See *SpaceCenter.Resources.flow\_mode()*.

#### **vessel**

The resource flows to any part in the vessel. For example, electric charge.

#### **stage**

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

#### **adjacent**

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

#### **none**

The resource does not flow. For example, solid fuel.

## 6.3.10 Node

#### **class Node**

Represents a maneuver node. Can be created using *SpaceCenter.Control.add\_node()*.

#### **prograde**

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

**Attribute** Can be read or written

**Return type** number

#### **normal**

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

**Attribute** Can be read or written

**Return type** number

#### **radial**

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

**Attribute** Can be read or written

**Return type** number

#### **delta\_v**

The delta-v of the maneuver node, in meters per second.

**Attribute** Can be read or written

**Return type** number

---

**Note:** Does not change when executing the maneuver node. See *SpaceCenter.Node.remaining\_delta\_v*.

---

**remaining\_delta\_v**

Gets the remaining delta-v of the maneuver node, in meters per second. Changes as the node is executed. This is equivalent to the delta-v reported in-game.

**Attribute** Read-only, cannot be set

**Return type** number

**burn\_vector** ([*reference\_frame* = None])

Returns the burn vector for the maneuver node.

**Parameters** **reference\_frame** (SpaceCenter.ReferenceFrame) – The reference frame that the returned vector is in. Defaults to *SpaceCenter.Vessel.orbital\_reference\_frame*.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

**Return type** Tuple of (number, number, number)

---

**Note:** Does not change when executing the maneuver node. See *SpaceCenter.Node.remaining\_burn\_vector()*.

---

**remaining\_burn\_vector** ([*reference\_frame* = None])

Returns the remaining burn vector for the maneuver node.

**Parameters** **reference\_frame** (SpaceCenter.ReferenceFrame) – The reference frame that the returned vector is in. Defaults to *SpaceCenter.Vessel.orbital\_reference\_frame*.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

**Return type** Tuple of (number, number, number)

---

**Note:** Changes as the maneuver node is executed. See *SpaceCenter.Node.burn\_vector()*.

---

**ut**

The universal time at which the maneuver will occur, in seconds.

**Attribute** Can be read or written

**Return type** number

**time\_to**

The time until the maneuver node will be encountered, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**orbit**

The orbit that results from executing the maneuver node.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Orbit*

**remove** ()

Removes the maneuver node.

**reference\_frame**

The reference frame that is fixed relative to the maneuver node's burn.

- The origin is at the position of the maneuver node.
- The y-axis points in the direction of the burn.
- The x-axis and z-axis point in arbitrary but fixed directions.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

**orbital\_reference\_frame**

The reference frame that is fixed relative to the maneuver node, and orientated with the orbital prograde/normal/radial directions of the original orbit at the maneuver node's position.

- The origin is at the position of the maneuver node.
- The x-axis points in the orbital anti-radial direction of the original orbit, at the position of the maneuver node.
- The y-axis points in the orbital prograde direction of the original orbit, at the position of the maneuver node.
- The z-axis points in the orbital normal direction of the original orbit, at the position of the maneuver node.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ReferenceFrame*

**position** (*reference\_frame*)

The position vector of the maneuver node in the given reference frame.

**Parameters** **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** Tuple of (number, number, number)

**direction** (*reference\_frame*)

The direction of the maneuver nodes burn.

**Parameters** **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** Tuple of (number, number, number)

### 6.3.11 ReferenceFrame

**class ReferenceFrame**

Represents a reference frame for positions, rotations and velocities.

Contains:

- The position of the origin.
- The directions of the x, y and z axes.
- The linear velocity of the frame.
- The angular velocity of the frame.

---

**Note:** This class does not contain any properties or methods. It is only used as a parameter to other functions.

---

```
static create_relative (reference_frame [, position = (0.0, 0.0, 0.0) ] [, rotation = (0.0, 0.0, 0.0, 1.0) ] [, velocity = (0.0, 0.0, 0.0) ] [, angular_velocity = (0.0, 0.0, 0.0) ] )
```

Create a relative reference frame. This is a custom reference frame whose components offset the components of a parent reference frame.

**Parameters**

- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – The parent reference frame on which to base this reference frame.
- **position** (*Tuple*) – The offset of the position of the origin, as a position vector. Defaults to (0, 0, 0)
- **rotation** (*Tuple*) – The rotation to apply to the parent frames rotation, as a quaternion of the form (*x, y, z, w*). Defaults to (0, 0, 0, 1) (i.e. no rotation)
- **velocity** (*Tuple*) – The linear velocity to offset the parent frame by, as a vector pointing in the direction of travel, whose magnitude is the speed in meters per second. Defaults to (0, 0, 0).
- **angular\_velocity** (*Tuple*) – The angular velocity to offset the parent frame by, as a vector. This vector points in the direction of the axis of rotation, and its magnitude is the speed of the rotation in radians per second. Defaults to (0, 0, 0).

**Return type** *SpaceCenter.ReferenceFrame*

```
static create_hybrid (position [, rotation = None ] [, velocity = None ] [, angular_velocity = None ] )
```

Create a hybrid reference frame. This is a custom reference frame whose components inherited from other reference frames.

**Parameters**

- **position** (*SpaceCenter.ReferenceFrame*) – The reference frame providing the position of the origin.
- **rotation** (*SpaceCenter.ReferenceFrame*) – The reference frame providing the rotation of the frame.
- **velocity** (*SpaceCenter.ReferenceFrame*) – The reference frame providing the linear velocity of the frame.

- **angular\_velocity** (`SpaceCenter.ReferenceFrame`) – The reference frame providing the angular velocity of the frame.

**Return type** `SpaceCenter.ReferenceFrame`

---

**Note:** The *position* reference frame is required but all other reference frames are optional. If omitted, they are set to the *position* reference frame.

---

### 6.3.12 AutoPilot

#### **class** `AutoPilot`

Provides basic auto-piloting utilities for a vessel. Created by calling `SpaceCenter.Vessel.auto_pilot`.

---

**Note:** If a client engages the auto-pilot and then closes its connection to the server, the auto-pilot will be disengaged and its target reference frame, direction and roll reset to default.

---

#### **engage** ()

Engage the auto-pilot.

#### **disengage** ()

Disengage the auto-pilot.

#### **wait** ()

Blocks until the vessel is pointing in the target direction and has the target roll (if set). Throws an exception if the auto-pilot has not been engaged.

#### **error**

The error, in degrees, between the direction the ship has been asked to point in and the direction it is pointing in. Throws an exception if the auto-pilot has not been engaged and SAS is not enabled or is in stability assist mode.

**Attribute** Read-only, cannot be set

**Return type** number

#### **pitch\_error**

The error, in degrees, between the vessels current and target pitch. Throws an exception if the auto-pilot has not been engaged.

**Attribute** Read-only, cannot be set

**Return type** number

#### **heading\_error**

The error, in degrees, between the vessels current and target heading. Throws an exception if the auto-pilot has not been engaged.

**Attribute** Read-only, cannot be set

**Return type** number

**roll\_error**

The error, in degrees, between the vessels current and target roll.  
Throws an exception if the auto-pilot has not been engaged or no target roll is set.

**Attribute** Read-only, cannot be set

**Return type** number

**reference\_frame**

The reference frame for the target direction (*SpaceCenter.AutoPilot.target\_direction*).

**Attribute** Can be read or written

**Return type** *SpaceCenter.ReferenceFrame*

---

**Note:** An error will be thrown if this property is set to a reference frame that rotates with the vessel being controlled, as it is impossible to rotate the vessel in such a reference frame.

---

**target\_pitch**

The target pitch, in degrees, between -90° and +90°.

**Attribute** Can be read or written

**Return type** number

**target\_heading**

The target heading, in degrees, between 0° and 360°.

**Attribute** Can be read or written

**Return type** number

**target\_roll**

The target roll, in degrees. NaN if no target roll is set.

**Attribute** Can be read or written

**Return type** number

**target\_direction**

Direction vector corresponding to the target pitch and heading.  
This is in the reference frame specified by *SpaceCenter.ReferenceFrame*.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**target\_pitch\_and\_heading** (*pitch, heading*)

Set target pitch and heading angles.

**Parameters**

- **pitch** (*number*) – Target pitch angle, in degrees between -90° and +90°.
- **heading** (*number*) – Target heading angle, in degrees between 0° and 360°.

**sas**

The state of SAS.

**Attribute** Can be read or written

**Return type** boolean

---

**Note:** Equivalent to *SpaceCenter.Control.sas*

---

#### **sas\_mode**

The current *SpaceCenter.SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

**Attribute** Can be read or written

**Return type** *SpaceCenter.SASMode*

---

**Note:** Equivalent to *SpaceCenter.Control.sas\_mode*

---

#### **roll\_threshold**

The threshold at which the autopilot will try to match the target roll angle, if any. Defaults to 5 degrees.

**Attribute** Can be read or written

**Return type** number

#### **stopping\_time**

The maximum amount of time that the vessel should need to come to a complete stop. This determines the maximum angular velocity of the vessel. A vector of three stopping times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 0.5 seconds for each axis.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

#### **deceleration\_time**

The time the vessel should take to come to a stop pointing in the target direction. This determines the angular acceleration used to decelerate the vessel. A vector of three times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 5 seconds for each axis.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

#### **attenuation\_angle**

The angle at which the autopilot considers the vessel to be pointing close to the target. This determines the midpoint of the target velocity attenuation function. A vector of three angles, in degrees, one for each of the pitch, roll and yaw axes. Defaults to 1° for each axis.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

#### **auto\_tune**

Whether the rotation rate controllers PID parameters should be automatically tuned using the vessels moment of inertia and

available torque. Defaults to `True`. See `SpaceCenter.AutoPilot.time_to_peak` and `SpaceCenter.AutoPilot.overshoot`.

**Attribute** Can be read or written

**Return type** `boolean`

#### **time\_to\_peak**

The target time to peak used to autotune the PID controllers. A vector of three times, in seconds, for each of the pitch, roll and yaw axes. Defaults to 3 seconds for each axis.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

#### **overshoot**

The target overshoot percentage used to autotune the PID controllers. A vector of three values, between 0 and 1, for each of the pitch, roll and yaw axes. Defaults to 0.01 for each axis.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

#### **pitch\_pid\_gains**

Gains for the pitch PID controller.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

---

**Note:** When `SpaceCenter.AutoPilot.auto_tune` is `true`, these values are updated automatically, which will overwrite any manual changes.

---

#### **roll\_pid\_gains**

Gains for the roll PID controller.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

---

**Note:** When `SpaceCenter.AutoPilot.auto_tune` is `true`, these values are updated automatically, which will overwrite any manual changes.

---

#### **yaw\_pid\_gains**

Gains for the yaw PID controller.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

---

**Note:** When `SpaceCenter.AutoPilot.auto_tune` is `true`, these values are updated automatically, which will overwrite any manual changes.

---



### 6.3.13 Camera

#### **class Camera**

Controls the game's camera. Obtained by calling *SpaceCenter.camera*.

#### **mode**

The current mode of the camera.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CameraMode*

#### **pitch**

The pitch of the camera, in degrees. A value between *SpaceCenter.Camera.min\_pitch* and *SpaceCenter.Camera.max\_pitch*

**Attribute** Can be read or written

**Return type** number

#### **heading**

The heading of the camera, in degrees.

**Attribute** Can be read or written

**Return type** number

#### **distance**

The distance from the camera to the subject, in meters. A value between *SpaceCenter.Camera.min\_distance* and *SpaceCenter.Camera.max\_distance*.

**Attribute** Can be read or written

**Return type** number

#### **min\_pitch**

The minimum pitch of the camera.

**Attribute** Read-only, cannot be set

**Return type** number

#### **max\_pitch**

The maximum pitch of the camera.

**Attribute** Read-only, cannot be set

**Return type** number

#### **min\_distance**

Minimum distance from the camera to the subject, in meters.

**Attribute** Read-only, cannot be set

**Return type** number

#### **max\_distance**

Maximum distance from the camera to the subject, in meters.

**Attribute** Read-only, cannot be set

**Return type** number

**default\_distance**

Default distance from the camera to the subject, in meters.

**Attribute** Read-only, cannot be set

**Return type** number

**focussed\_body**

In map mode, the celestial body that the camera is focussed on.  
Returns `nil` if the camera is not focussed on a celestial body.  
Returns an error if the camera is not in map mode.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CelestialBody*

**focussed\_vessel**

In map mode, the vessel that the camera is focussed on. Returns `nil` if the camera is not focussed on a vessel. Returns an error if the camera is not in map mode.

**Attribute** Can be read or written

**Return type** *SpaceCenter.Vessel*

**focussed\_node**

In map mode, the maneuver node that the camera is focussed on. Returns `nil` if the camera is not focussed on a maneuver node. Returns an error if the camera is not in map mode.

**Attribute** Can be read or written

**Return type** *SpaceCenter.Node*

**class CameraMode**

See *SpaceCenter.Camera.mode*.

**automatic**

The camera is showing the active vessel, in “auto” mode.

**free**

The camera is showing the active vessel, in “free” mode.

**chase**

The camera is showing the active vessel, in “chase” mode.

**locked**

The camera is showing the active vessel, in “locked” mode.

**orbital**

The camera is showing the active vessel, in “orbital” mode.

**iva**

The Intra-Vehicular Activity view is being shown.

**map**

The map view is being shown.

## 6.3.14 Waypoints

**class WaypointManager**

Waypoints are the location markers you can see on the map view showing you where contracts are targeted for. With this structure,

you can obtain coordinate data for the locations of these waypoints.  
Obtained by calling *SpaceCenter.waypoint\_manager*.

### **waypoints**

A list of all existing waypoints.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Waypoint*

### **add\_waypoint** (*latitude, longitude, body, name*)

Creates a waypoint at the given position at ground level, and returns  
a *SpaceCenter.Waypoint* object that can be used to modify it.

#### **Parameters**

- **latitude** (*number*) – Latitude of the waypoint.
- **longitude** (*number*) – Longitude of the waypoint.
- **body** (*SpaceCenter.CelestialBody*) – Celestial body the waypoint is attached to.
- **name** (*string*) – Name of the waypoint.

**Return type** *SpaceCenter.Waypoint*

### **add\_waypoint\_at\_altitude** (*latitude, longitude, altitude, body, name*)

Creates a waypoint at the given position and altitude, and returns a  
*SpaceCenter.Waypoint* object that can be used to modify it.

#### **Parameters**

- **latitude** (*number*) – Latitude of the waypoint.
- **longitude** (*number*) – Longitude of the waypoint.
- **altitude** (*number*) – Altitude (above sea level) of the waypoint.
- **body** (*SpaceCenter.CelestialBody*) – Celestial body the waypoint is attached to.
- **name** (*string*) – Name of the waypoint.

**Return type** *SpaceCenter.Waypoint*

### **colors**

An example map of known color - seed pairs. Any other integers  
may be used as seed.

**Attribute** Read-only, cannot be set

**Return type** Map from string to number

### **icons**

Returns all available icons (from “Game-  
Data/Squad/Contracts/Icons”).

**Attribute** Read-only, cannot be set

**Return type** List of string

### **class Waypoint**

Represents a waypoint. Can be created using *SpaceCenter.WaypointManager.add\_waypoint()*.

**body**

The celestial body the waypoint is attached to.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CelestialBody*

**name**

The name of the waypoint as it appears on the map and the contract.

**Attribute** Can be read or written

**Return type** string

**color**

The seed of the icon color. See *SpaceCenter.WaypointManager.colors* for example colors.

**Attribute** Can be read or written

**Return type** number

**icon**

The icon of the waypoint.

**Attribute** Can be read or written

**Return type** string

**latitude**

The latitude of the waypoint.

**Attribute** Can be read or written

**Return type** number

**longitude**

The longitude of the waypoint.

**Attribute** Can be read or written

**Return type** number

**mean\_altitude**

The altitude of the waypoint above sea level, in meters.

**Attribute** Can be read or written

**Return type** number

**surface\_altitude**

The altitude of the waypoint above the surface of the body or sea level, whichever is closer, in meters.

**Attribute** Can be read or written

**Return type** number

**bedrock\_altitude**

The altitude of the waypoint above the surface of the body, in meters.  
When over water, this is the altitude above the sea floor.

**Attribute** Can be read or written

**Return type** number

**near\_surface**

True if the waypoint is near to the surface of a body.

**Attribute** Read-only, cannot be set

**Return type** boolean

**grounded**

True if the waypoint is attached to the ground.

**Attribute** Read-only, cannot be set

**Return type** boolean

**index**

The integer index of this waypoint within its cluster of sibling waypoints. In other words, when you have a cluster of waypoints called “Somewhere Alpha”, “Somewhere Beta” and “Somewhere Gamma”, the alpha site has index 0, the beta site has index 1 and the gamma site has index 2. When *SpaceCenter.Waypoint.clustered* is False, this is zero.

**Attribute** Read-only, cannot be set

**Return type** number

**clustered**

True if this waypoint is part of a set of clustered waypoints with greek letter names appended (Alpha, Beta, Gamma, etc). If True, there is a one-to-one correspondence with the greek letter name and the *SpaceCenter.Waypoint.index*.

**Attribute** Read-only, cannot be set

**Return type** boolean

**has\_contract**

Whether the waypoint belongs to a contract.

**Attribute** Read-only, cannot be set

**Return type** boolean

**contract**

The associated contract.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Contract*

**remove()**

Removes the waypoint.

### 6.3.15 Contracts

**class ContractManager**

Contracts manager. Obtained by calling *SpaceCenter.waypoint\_manager*.

**types**

A list of all contract types.

**Attribute** Read-only, cannot be set

**Return type** Set of string

**all\_contracts**

A list of all contracts.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Contract*

**active\_contracts**

A list of all active contracts.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Contract*

**offered\_contracts**

A list of all offered, but unaccepted, contracts.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Contract*

**completed\_contracts**

A list of all completed contracts.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Contract*

**failed\_contracts**

A list of all failed contracts.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Contract*

**class Contract**

A contract. Can be accessed using *SpaceCenter.contract\_manager*.

**type**

Type of the contract.

**Attribute** Read-only, cannot be set

**Return type** string

**title**

Title of the contract.

**Attribute** Read-only, cannot be set

**Return type** string

**description**

Description of the contract.

**Attribute** Read-only, cannot be set

**Return type** string

**notes**

Notes for the contract.

**Attribute** Read-only, cannot be set

**Return type** string

**synopsis**

Synopsis for the contract.

**Attribute** Read-only, cannot be set

**Return type** string

**keywords**

Keywords for the contract.

**Attribute** Read-only, cannot be set

**Return type** List of string

**state**

State of the contract.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.ContractState*

**seen**

Whether the contract has been seen.

**Attribute** Read-only, cannot be set

**Return type** boolean

**read**

Whether the contract has been read.

**Attribute** Read-only, cannot be set

**Return type** boolean

**active**

Whether the contract is active.

**Attribute** Read-only, cannot be set

**Return type** boolean

**failed**

Whether the contract has been failed.

**Attribute** Read-only, cannot be set

**Return type** boolean

**can\_be\_canceled**

Whether the contract can be canceled.

**Attribute** Read-only, cannot be set

**Return type** boolean

**can\_be\_declined**

Whether the contract can be declined.

**Attribute** Read-only, cannot be set

**Return type** boolean

**can\_be\_failed**

Whether the contract can be failed.

**Attribute** Read-only, cannot be set

**Return type** boolean

**accept** ()

Accept an offered contract.

**cancel** ()

Cancel an active contract.

**decline** ()

Decline an offered contract.

**funds\_advance**

Funds received when accepting the contract.

**Attribute** Read-only, cannot be set

**Return type** number

**funds\_completion**

Funds received on completion of the contract.

**Attribute** Read-only, cannot be set

**Return type** number

**funds\_failure**

Funds lost if the contract is failed.

**Attribute** Read-only, cannot be set

**Return type** number

**reputation\_completion**

Reputation gained on completion of the contract.

**Attribute** Read-only, cannot be set

**Return type** number

**reputation\_failure**

Reputation lost if the contract is failed.

**Attribute** Read-only, cannot be set

**Return type** number

**science\_completion**

Science gained on completion of the contract.

**Attribute** Read-only, cannot be set

**Return type** number

**parameters**

Parameters for the contract.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.ContractParameter*

**class ContractState**

The state of a contract. See *SpaceCenter.Contract.state*.

**active**

The contract is active.



**canceled**

The contract has been canceled.

**completed**

The contract has been completed.

**deadline\_expired**

The deadline for the contract has expired.

**declined**

The contract has been declined.

**failed**

The contract has been failed.

**generated**

The contract has been generated.

**offered**

The contract has been offered to the player.

**offer\_expired**

The contract was offered to the player, but the offer expired.

**withdrawn**

The contract has been withdrawn.

**class ContractParameter**

A contract parameter. See *SpaceCenter.Contract.parameters*.

**title**

Title of the parameter.

**Attribute** Read-only, cannot be set

**Return type** string

**notes**

Notes for the parameter.

**Attribute** Read-only, cannot be set

**Return type** string

**children**

Child contract parameters.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.ContractParameter*

**completed**

Whether the parameter has been completed.

**Attribute** Read-only, cannot be set

**Return type** boolean

**failed**

Whether the parameter has been failed.

**Attribute** Read-only, cannot be set

**Return type** boolean

**optional**

Whether the contract parameter is optional.

**Attribute** Read-only, cannot be set

**Return type** boolean

**funds\_completion**

Funds received on completion of the contract parameter.

**Attribute** Read-only, cannot be set

**Return type** number

**funds\_failure**

Funds lost if the contract parameter is failed.

**Attribute** Read-only, cannot be set

**Return type** number

**reputation\_completion**

Reputation gained on completion of the contract parameter.

**Attribute** Read-only, cannot be set

**Return type** number

**reputation\_failure**

Reputation lost if the contract parameter is failed.

**Attribute** Read-only, cannot be set

**Return type** number

**science\_completion**

Science gained on completion of the contract parameter.

**Attribute** Read-only, cannot be set

**Return type** number

## 6.3.16 Geometry Types

### Vectors

3-dimensional vectors are represented as a 3-tuple. For example:

```
local krpc = require 'krpc'
local conn = krpc.connect()
local v = conn.
    ↪space_center.active_vessel:flight().prograde
print(v[1], v[2], v[3])
```

### Quaternions

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

```

local krpc = require 'krpc'
local conn = krpc.connect()
local q = conn.
↪space_center.active_vessel:flight().rotation
print(q[1], q[2], q[3], q[4])

```

## 6.4 Drawing API

### 6.4.1 Drawing

Provides functionality for drawing objects in the flight scene.

**static add\_line** (*start*, *end*, *reference\_frame* [, *visible* = *True* ])  
 Draw a line in the scene.

#### Parameters

- **start** (*Tuple*) – Position of the start of the line.
- **end** (*Tuple*) – Position of the end of the line.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the positions are in.
- **visible** (*boolean*) – Whether the line is visible.

**Return type** *Drawing.Line*

**static add\_direction** (*direction*, *reference\_frame* [, *length* = *10.0* ] [, *visible* = *True* ])  
 Draw a direction vector in the scene, from the center of mass of the active vessel.

#### Parameters

- **direction** (*Tuple*) – Direction to draw the line in.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the direction is in.
- **length** (*number*) – The length of the line.
- **visible** (*boolean*) – Whether the line is visible.

**Return type** *Drawing.Line*

**static add\_polygon** (*vertices*, *reference\_frame* [, *visible* = *True* ])  
 Draw a polygon in the scene, defined by a list of vertices.

#### Parameters

- **vertices** (*List*) – Vertices of the polygon.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the vertices are in.
- **visible** (*boolean*) – Whether the polygon is visible.

**Return type** *Drawing.Polygon*

**static add\_text** (*text*, *reference\_frame*, *position*, *rotation* [, *visible* = *True* ])  
 Draw text in the scene.

**Parameters**

- **text** (*string*) – The string to draw.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the text position is in.
- **position** (*Tuple*) – Position of the text.
- **rotation** (*Tuple*) – Rotation of the text, as a quaternion.
- **visible** (*boolean*) – Whether the text is visible.

**Return type** *Drawing.Text*

**static clear** (*[client\_only = False]*)  
Remove all objects being drawn.

**Parameters** **client\_only** (*boolean*) – If true, only remove objects created by the calling client.

## 6.4.2 Line

**class Line**

A line. Created using *Drawing.add\_line()*.

**start**

Start position of the line.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**end**

End position of the line.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**reference\_frame**

Reference frame for the positions of the object.

**Attribute** Can be read or written

**Return type** *SpaceCenter.ReferenceFrame*

**visible**

Whether the object is visible.

**Attribute** Can be read or written

**Return type** boolean

**color**

Set the color

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**material**

Material used to render the object. Creates the material from a shader with the given name.

**Attribute** Can be read or written

**Return type** string

**thickness**

Set the thickness

**Attribute** Can be read or written

**Return type** number

**remove ()**

Remove the object.

### 6.4.3 Polygon

**class Polygon**

A polygon. Created using *Drawing.add\_polygon()*.

**vertices**

Vertices for the polygon.

**Attribute** Can be read or written

**Return type** List of Tuple of (number, number, number)

**reference\_frame**

Reference frame for the positions of the object.

**Attribute** Can be read or written

**Return type** *SpaceCenter.ReferenceFrame*

**visible**

Whether the object is visible.

**Attribute** Can be read or written

**Return type** boolean

**remove ()**

Remove the object.

**color**

Set the color

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**material**

Material used to render the object. Creates the material from a shader with the given name.

**Attribute** Can be read or written

**Return type** string

**thickness**

Set the thickness

**Attribute** Can be read or written

**Return type** number

### 6.4.4 Text

**class Text**

Text. Created using *Drawing.add\_text()*.

**position**

Position of the text.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**rotation**

Rotation of the text as a quaternion.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number, number)

**reference\_frame**

Reference frame for the positions of the object.

**Attribute** Can be read or written

**Return type** *SpaceCenter.ReferenceFrame*

**visible**

Whether the object is visible.

**Attribute** Can be read or written

**Return type** boolean

**remove()**

Remove the object.

**content**

The text string

**Attribute** Can be read or written

**Return type** string

**font**

Name of the font

**Attribute** Can be read or written

**Return type** string

**static available\_fonts()**

A list of all available fonts.

**Return type** List of string

**size**

Font size.

**Attribute** Can be read or written

**Return type** number

**character\_size**

Character size.

**Attribute** Can be read or written

**Return type** number

**style**

Font style.

**Attribute** Can be read or written

**Return type** *UI.FontStyle*

**color**

Set the color

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**material**

Material used to render the object. Creates the material from a shader with the given name.

**Attribute** Can be read or written

**Return type** string

**alignment**

Alignment.

**Attribute** Can be read or written

**Return type** *UI.TextAlignment*

**line\_spacing**

Line spacing.

**Attribute** Can be read or written

**Return type** number

**anchor**

Anchor.

**Attribute** Can be read or written

**Return type** *UI.TextAnchor*

## 6.5 InfernalRobotics API

Provides RPCs to interact with the [InfernalRobotics](#) mod. Provides the following classes:

### 6.5.1 InfernalRobotics

This service provides functionality to interact with [InfernalRobotics](#).

**available**

Whether Infernal Robotics is installed.

**Attribute** Read-only, cannot be set

**Return type** boolean

**static servo\_groups** (*vessel*)

A list of all the servo groups in the given *vessel*.

**Parameters** **vessel** (`SpaceCenter.Vessel`) –

**Return type** List of `InfernalRobotics.ServoGroup`

**static servo\_group\_with\_name** (*vessel*, *name*)

Returns the servo group in the given *vessel* with the given *name*, or `nil` if none exists. If multiple servo groups have the same name, only one of them is returned.

**Parameters**

- **vessel** (`SpaceCenter.Vessel`) – Vessel to check.
- **name** (*string*) – Name of servo group to find.

**Return type** `InfernalRobotics.ServoGroup`

**static servo\_with\_name** (*vessel*, *name*)

Returns the servo in the given *vessel* with the given *name* or `nil` if none exists. If multiple servos have the same name, only one of them is returned.

**Parameters**

- **vessel** (`SpaceCenter.Vessel`) – Vessel to check.
- **name** (*string*) – Name of the servo to find.

**Return type** `InfernalRobotics.Servo`

## 6.5.2 ServoGroup

**class ServoGroup**

A group of servos, obtained by calling `InfernalRobotics.servo_groups()` or `InfernalRobotics.servo_group_with_name()`. Represents the “Servo Groups” in the InfernalRobotics UI.

**name**

The name of the group.

**Attribute** Can be read or written

**Return type** `string`

**forward\_key**

The key assigned to be the “forward” key for the group.

**Attribute** Can be read or written

**Return type** `string`

**reverse\_key**

The key assigned to be the “reverse” key for the group.

**Attribute** Can be read or written

**Return type** `string`

**speed**

The speed multiplier for the group.



**Attribute** Can be read or written

**Return type** number

#### expanded

Whether the group is expanded in the InfernalRobotics UI.

**Attribute** Can be read or written

**Return type** boolean

#### servos

The servos that are in the group.

**Attribute** Read-only, cannot be set

**Return type** List of *InfernalRobotics.Servo*

#### servo\_with\_name (name)

Returns the servo with the given *name* from this group, or *nil* if none exists.

**Parameters** *name* (*string*) – Name of servo to find.

**Return type** *InfernalRobotics.Servo*

#### parts

The parts containing the servos in the group.

**Attribute** Read-only, cannot be set

**Return type** List of *SpaceCenter.Part*

#### move\_right ()

Moves all of the servos in the group to the right.

#### move\_left ()

Moves all of the servos in the group to the left.

#### move\_center ()

Moves all of the servos in the group to the center.

#### move\_next\_preset ()

Moves all of the servos in the group to the next preset.

#### move\_prev\_preset ()

Moves all of the servos in the group to the previous preset.

#### stop ()

Stops the servos in the group.

## 6.5.3 Servo

#### class Servo

Represents a servo. Obtained using *InfernalRobotics.ServoGroup.servos*, *InfernalRobotics.ServoGroup.servo\_with\_name ()* or *InfernalRobotics.servo\_with\_name ()*.

#### name

The name of the servo.

**Attribute** Can be read or written

**Return type** string

**part**

The part containing the servo.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**highlight**

Whether the servo should be highlighted in-game.

**Attribute** Write-only, cannot be read

**Return type** boolean

**position**

The position of the servo.

**Attribute** Read-only, cannot be set

**Return type** number

**min\_config\_position**

The minimum position of the servo, specified by the part configuration.

**Attribute** Read-only, cannot be set

**Return type** number

**max\_config\_position**

The maximum position of the servo, specified by the part configuration.

**Attribute** Read-only, cannot be set

**Return type** number

**min\_position**

The minimum position of the servo, specified by the in-game tweak menu.

**Attribute** Can be read or written

**Return type** number

**max\_position**

The maximum position of the servo, specified by the in-game tweak menu.

**Attribute** Can be read or written

**Return type** number

**config\_speed**

The speed multiplier of the servo, specified by the part configuration.

**Attribute** Read-only, cannot be set

**Return type** number

**speed**

The speed multiplier of the servo, specified by the in-game tweak menu.

**Attribute** Can be read or written

**Return type** number

**current\_speed**

The current speed at which the servo is moving.

**Attribute** Can be read or written

**Return type** number

**acceleration**

The current speed multiplier set in the UI.

**Attribute** Can be read or written

**Return type** number

**is\_moving**

Whether the servo is moving.

**Attribute** Read-only, cannot be set

**Return type** boolean

**is\_free\_moving**

Whether the servo is freely moving.

**Attribute** Read-only, cannot be set

**Return type** boolean

**is\_locked**

Whether the servo is locked.

**Attribute** Can be read or written

**Return type** boolean

**is\_axis\_inverted**

Whether the servos axis is inverted.

**Attribute** Can be read or written

**Return type** boolean

**move\_right()**

Moves the servo to the right.

**move\_left()**

Moves the servo to the left.

**move\_center()**

Moves the servo to the center.

**move\_next\_preset()**

Moves the servo to the next preset.

**move\_prev\_preset()**

Moves the servo to the previous preset.

**move\_to(*position*, *speed*)**

Moves the servo to *position* and sets the speed multiplier to *speed*.

**Parameters**

- **position** (*number*) – The position to move the servo to.
- **speed** (*number*) – Speed multiplier for the movement.

**stop()**  
Stops the servo.

### 6.5.4 Example

The following example gets the control group named “MyGroup”, prints out the names and positions of all of the servos in the group, then moves all of the servos to the right for 1 second.

```
local krpc = require 'krpc'

local _
↪conn = krpc.connect('InfernalRobotics Example')
local vessel = conn.space_center.active_vessel

local group = conn.infernal_robotics.
↪servo_group_with_name(vessel, 'MyGroup')
if group == krpc.types.none then
    print('Group not found')
    os.exit(1)
end

for _, servo in ipairs(group.servos) do
    print(servo.name, servo.position)
end

group:move_right()
krpc.platform.sleep(1)
group:stop()
```

## 6.6 Kerbal Alarm Clock API

Provides RPCs to interact with the [Kerbal Alarm Clock](#) mod. Provides the following classes:

### 6.6.1 KerbalAlarmClock

This service provides functionality to interact with [Kerbal Alarm Clock](#).

**available**

Whether Kerbal Alarm Clock is available.

**Attribute** Read-only, cannot be set

**Return type** boolean

**alarms**

A list of all the alarms.

**Attribute** Read-only, cannot be set

**Return type** List of *KerbalAlarmClock.Alarm*

**static alarm\_with\_name** (*name*)

Get the alarm with the given *name*, or `nil` if no alarms have that name. If more than one alarm has the name, only returns one of them.

**Parameters** *name* (*string*) – Name of the alarm to search for.

**Return type** *KerbalAlarmClock.Alarm*

**static alarms\_with\_type** (*type*)

Get a list of alarms of the specified *type*.

**Parameters** *type* (*KerbalAlarmClock.AlarmType*) – Type of alarm to return.

**Return type** List of *KerbalAlarmClock.Alarm*

**static create\_alarm** (*type*, *name*, *ut*)

Create a new alarm and return it.

**Parameters**

- **type** (*KerbalAlarmClock.AlarmType*) – Type of the new alarm.
- **name** (*string*) – Name of the new alarm.
- **ut** (*number*) – Time at which the new alarm should trigger.

**Return type** *KerbalAlarmClock.Alarm*

## 6.6.2 Alarm

**class Alarm**

Represents an alarm. Obtained by calling *KerbalAlarmClock.alarms*, *KerbalAlarmClock.alarm\_with\_name()* or *KerbalAlarmClock.alarms\_with\_type()*.

**action**

The action that the alarm triggers.

**Attribute** Can be read or written

**Return type** *KerbalAlarmClock.AlarmAction*

**margin**

The number of seconds before the event that the alarm will fire.

**Attribute** Can be read or written

**Return type** *number*

**time**

The time at which the alarm will fire.

**Attribute** Can be read or written

**Return type** *number*

**type**

The type of the alarm.

**Attribute** Read-only, cannot be set

**Return type** *KerbalAlarmClock.AlarmType*

**id**

The unique identifier for the alarm.

**Attribute** Read-only, cannot be set

**Return type** string

**name**

The short name of the alarm.

**Attribute** Can be read or written

**Return type** string

**notes**

The long description of the alarm.

**Attribute** Can be read or written

**Return type** string

**remaining**

The number of seconds until the alarm will fire.

**Attribute** Read-only, cannot be set

**Return type** number

**repeat**

Whether the alarm will be repeated after it has fired.

**Attribute** Can be read or written

**Return type** boolean

**repeat\_period**

The time delay to automatically create an alarm after it has fired.

**Attribute** Can be read or written

**Return type** number

**vessel**

The vessel that the alarm is attached to.

**Attribute** Can be read or written

**Return type** *SpaceCenter.Vessel*

**xfer\_origin\_body**

The celestial body the vessel is departing from.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CelestialBody*

**xfer\_target\_body**

The celestial body the vessel is arriving at.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CelestialBody*

**remove ()**

Removes the alarm.

### 6.6.3 AlarmType

#### **class AlarmType**

The type of an alarm.

#### **raw**

An alarm for a specific date/time or a specific period in the future.

#### **maneuver**

An alarm based on the next maneuver node on the current ships flight path. This node will be stored and can be restored when you come back to the ship.

#### **maneuver\_auto**

See *KerbalAlarmClock.AlarmType.maneuver*.

#### **apoapsis**

An alarm for furthest part of the orbit from the planet.

#### **periapsis**

An alarm for nearest part of the orbit from the planet.

#### **ascending\_node**

Ascending node for the targeted object, or equatorial ascending node.

#### **descending\_node**

Descending node for the targeted object, or equatorial descending node.

#### **closest**

An alarm based on the closest approach of this vessel to the targeted vessel, some number of orbits into the future.

#### **contract**

An alarm based on the expiry or deadline of contracts in career modes.

#### **contract\_auto**

See *KerbalAlarmClock.AlarmType.contract*.

#### **crew**

An alarm that is attached to a crew member.

#### **distance**

An alarm that is triggered when a selected target comes within a chosen distance.

#### **earth\_time**

An alarm based on the time in the “Earth” alternative Universe (aka the Real World).

#### **launch\_rendevous**

An alarm that fires as your landed craft passes under the orbit of your target.

#### **soi\_change**

An alarm manually based on when the next SOI point is on the flight path or set to continually monitor the active flight path and add alarms as it detects SOI changes.

**soi\_change\_auto**

See `KerbAlarmClock.AlarmType.soi_change`.

**transfer**

An alarm based on Interplanetary Transfer Phase Angles, i.e. when should I launch to planet X? Based on Kosmo Not's post and used in Olex's Calculator.

**transfer\_modelled**

See `KerbAlarmClock.AlarmType.transfer`.

## 6.6.4 AlarmAction

**class AlarmAction**

The action performed by an alarm when it fires.

**do\_nothing**

Don't do anything at all...

**do\_nothing\_delete\_when\_passed**

Don't do anything, and delete the alarm.

**kill\_warp**

Drop out of time warp.

**kill\_warp\_only**

Drop out of time warp.

**message\_only**

Display a message.

**pause\_game**

Pause the game.

## 6.6.5 Example

The following example creates a new alarm for the active vessel. The alarm is set to trigger after 10 seconds have passed, and display a message.

```
local krpc = require 'krpc'

local conn_
↪= krpc.connect('Kerb Alarm Clock Example')

local_
↪alarm = conn.kerb_alarm_clock.create_alarm(
    conn.kerb_alarm_clock.AlarmType.raw,
    'My New Alarm',
    conn.space_center.ut+10)

alarm.notes = '10 seconds_
↪have now passed since the alarm was created.'
alarm.action =_
↪conn.kerb_alarm_clock.AlarmAction.message_only
```



## 6.7 RemoteTech API

Provides RPCs to interact with the [RemoteTech](#) mod. Provides the following classes:

### 6.7.1 RemoteTech

This service provides functionality to interact with [RemoteTech](#).

#### **available**

Whether RemoteTech is installed.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **ground\_stations**

The names of the ground stations.

**Attribute** Read-only, cannot be set

**Return type** List of string

#### **static antenna** (*part*)

Get the antenna object for a particular part.

**Parameters** **part** (`SpaceCenter.Part`) –

**Return type** *RemoteTech.Antenna*

#### **static comms** (*vessel*)

Get a communications object, representing the communication capability of a particular vessel.

**Parameters** **vessel** (`SpaceCenter.Vessel`) –

**Return type** *RemoteTech.Comms*

### 6.7.2 Comms

#### **class Comms**

Communications for a vessel.

#### **vessel**

Get the vessel.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Vessel*

#### **has\_local\_control**

Whether the vessel can be controlled locally.

**Attribute** Read-only, cannot be set

**Return type** boolean

#### **has\_flight\_computer**

Whether the vessel has a flight computer on board.

**Attribute** Read-only, cannot be set

**Return type** boolean

**has\_connection**

Whether the vessel has any connection.

**Attribute** Read-only, cannot be set

**Return type** boolean

**has\_connection\_to\_ground\_station**

Whether the vessel has a connection to a ground station.

**Attribute** Read-only, cannot be set

**Return type** boolean

**signal\_delay**

The shortest signal delay to the vessel, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**signal\_delay\_to\_ground\_station**

The signal delay between the vessel and the closest ground station, in seconds.

**Attribute** Read-only, cannot be set

**Return type** number

**signal\_delay\_to\_vessel** (*other*)

The signal delay between the this vessel and another vessel, in seconds.

**Parameters** **other** (*SpaceCenter.Vessel*) –

**Return type** number

**antennas**

The antennas for this vessel.

**Attribute** Read-only, cannot be set

**Return type** List of *RemoteTech.Antenna*

### 6.7.3 Antenna

**class Antenna**

A RemoteTech antenna. Obtained by calling *RemoteTech.Comms.antennas* or *RemoteTech.antenna()*.

**part**

Get the part containing this antenna.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**has\_connection**

Whether the antenna has a connection.

**Attribute** Read-only, cannot be set

**Return type** boolean

**target**

The object that the antenna is targetting. This property can be used to set the target to *RemoteTech.Target.none* or *RemoteTech.Target.active\_vessel*. To set the target to a celestial body, ground station or vessel see *RemoteTech.Antenna.target\_body*, *RemoteTech.Antenna.target\_ground\_station* and *RemoteTech.Antenna.target\_vessel*.

**Attribute** Can be read or written

**Return type** *RemoteTech.Target*

**target\_body**

The celestial body the antenna is targetting.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CelestialBody*

**target\_ground\_station**

The ground station the antenna is targetting.

**Attribute** Can be read or written

**Return type** string

**target\_vessel**

The vessel the antenna is targetting.

**Attribute** Can be read or written

**Return type** *SpaceCenter.Vessel*

**class Target**

The type of object an antenna is targetting. See *RemoteTech.Antenna.target*.

**active\_vessel**

The active vessel.

**celestial\_body**

A celestial body.

**ground\_station**

A ground station.

**vessel**

A specific vessel.

**none**

No target.

## 6.7.4 Example

The following example sets the target of a dish on the active vessel then prints out the signal delay to the active vessel.

```
local krpc = require 'krpc'
local math = require 'math'
local conn = krpc.connect('RemoteTech Example')
local vessel = conn.space_center.active_vessel
```

```
-- Set a dish target
local part_
↪= vessel.parts:with_title('Reflectron KR-7')[1]
local antenna = conn.remote_tech:antenna(part)
antenna.
↪target_body = conn.space_center.bodies['Jool']

-- Get info about the vessels communications
local comms = conn.remote_tech:comms(vessel)
print('Signal delay = ' .. comms.signal_delay)
```

## 6.8 User Interface API

### 6.8.1 UI

Provides functionality for drawing and interacting with in-game user interface elements.

#### **stock\_canvas**

The stock UI canvas.

**Attribute** Read-only, cannot be set

**Return type** *UI.Canvas*

#### **static add\_canvas()**

Add a new canvas.

**Return type** *UI.Canvas*

---

**Note:** If you want to add UI elements to KSPs stock UI canvas, use *UI.stock\_canvas*.

---

#### **static message**(*content*[, *duration* = 1.0][, *position* = 1])

Display a message on the screen.

##### **Parameters**

- **content** (*string*) – Message content.
- **duration** (*number*) – Duration before the message disappears, in seconds.
- **position** (*UI.MessagePosition*) – Position to display the message.

---

**Note:** The message appears just like a stock message, for example quicksave or quickload messages.

---

#### **static clear**([*client\_only* = False])

Remove all user interface elements.

**Parameters** **client\_only** (*boolean*) – If true, only remove objects created by the calling client.

**class MessagePosition**

Message position.

**top\_left**

Top left.

**top\_center**

Top center.

**top\_right**

Top right.

**bottom\_center**

Bottom center.

## 6.8.2 Canvas

**class Canvas**

A canvas for user interface elements. See *UI.stock\_canvas* and *UI.add\_canvas()*.

**rect\_transform**

The rect transform for the canvas.

**Attribute** Read-only, cannot be set

**Return type** *UI.RectTransform*

**visible**

Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** boolean

**add\_panel** (*[visible = True]*)

Create a new container for user interface elements.

**Parameters** **visible** (*boolean*) – Whether the panel is visible.

**Return type** *UI.Panel*

**add\_text** (*content[, visible = True]*)

Add text to the canvas.

**Parameters**

- **content** (*string*) – The text.
- **visible** (*boolean*) – Whether the text is visible.

**Return type** *UI.Text*

**add\_input\_field** (*[visible = True]*)

Add an input field to the canvas.

**Parameters** **visible** (*boolean*) – Whether the input field is visible.

**Return type** *UI.InputField*

**add\_button** (*content[, visible = True]*)

Add a button to the canvas.

**Parameters**

- **content** (*string*) – The label for the button.
- **visible** (*boolean*) – Whether the button is visible.

**Return type** *UI.Button*

**remove** ()

Remove the UI object.

### 6.8.3 Panel

**class Panel**

A container for user interface elements. See *UI.Canvas*.

*add\_panel* ().

**rect\_transform**

The rect transform for the panel.

**Attribute** Read-only, cannot be set

**Return type** *UI.RectTransform*

**visible**

Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** *boolean*

**add\_panel** ([*visible = True*])

Create a panel within this panel.

**Parameters** **visible** (*boolean*) – Whether the new panel is visible.

**Return type** *UI.Panel*

**add\_text** (*content* [, *visible = True*])

Add text to the panel.

**Parameters**

- **content** (*string*) – The text.
- **visible** (*boolean*) – Whether the text is visible.

**Return type** *UI.Text*

**add\_input\_field** ([*visible = True*])

Add an input field to the panel.

**Parameters** **visible** (*boolean*) – Whether the input field is visible.

**Return type** *UI.InputField*

**add\_button** (*content* [, *visible = True*])

Add a button to the panel.

**Parameters**

- **content** (*string*) – The label for the button.
- **visible** (*boolean*) – Whether the button is visible.

**Return type** *UI.Button*

**remove ()**

Remove the UI object.

## 6.8.4 Text

**class Text**

A text label. See *UI.Panel.add\_text ()*.

**rect\_transform**

The rect transform for the text.

**Attribute** Read-only, cannot be set

**Return type** *UI.RectTransform*

**visible**

Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** boolean

**content**

The text string

**Attribute** Can be read or written

**Return type** string

**font**

Name of the font

**Attribute** Can be read or written

**Return type** string

**available\_fonts**

A list of all available fonts.

**Attribute** Read-only, cannot be set

**Return type** List of string

**size**

Font size.

**Attribute** Can be read or written

**Return type** number

**style**

Font style.

**Attribute** Can be read or written

**Return type** *UI.FontStyle*

**color**

Set the color

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**alignment**

Alignment.

**Attribute** Can be read or written

**Return type** *UI.TextAnchor*

**line\_spacing**  
Line spacing.

**Attribute** Can be read or written

**Return type** number

**remove** ()  
Remove the UI object.

**class FontStyle**  
Font style.

**normal**  
Normal.

**bold**  
Bold.

**italic**  
Italic.

**bold\_and\_italic**  
Bold and italic.

**class TextAlignment**  
Text alignment.

**left**  
Left aligned.

**right**  
Right aligned.

**center**  
Center aligned.

**class TextAnchor**  
Text alignment.

**lower\_center**  
Lower center.

**lower\_left**  
Lower left.

**lower\_right**  
Lower right.

**middle\_center**  
Middle center.

**middle\_left**  
Middle left.

**middle\_right**  
Middle right.

**upper\_center**  
Upper center.



**upper\_left**  
Upper left.

**upper\_right**  
Upper right.

## 6.8.5 Button

**class Button**  
A text label. See *UI.Panel.add\_button()*.

**rect\_transform**  
The rect transform for the text.

**Attribute** Read-only, cannot be set

**Return type** *UI.RectTransform*

**visible**  
Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** boolean

**text**  
The text for the button.

**Attribute** Read-only, cannot be set

**Return type** *UI.Text*

**clicked**  
Whether the button has been clicked.

**Attribute** Can be read or written

**Return type** boolean

---

**Note:** This property is set to true when the user clicks the button.  
A client script should reset the property to false in order to detect subsequent button presses.

---

**remove()**  
Remove the UI object.

## 6.8.6 InputField

**class InputField**  
An input field. See *UI.Panel.add\_input\_field()*.

**rect\_transform**  
The rect transform for the input field.

**Attribute** Read-only, cannot be set

**Return type** *UI.RectTransform*

**visible**  
Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** boolean

**value**

The value of the input field.

**Attribute** Can be read or written

**Return type** string

**text**

The text component of the input field.

**Attribute** Read-only, cannot be set

**Return type** *UI.Text*

---

**Note:** Use *UI.InputField.value* to get and set the value in the field. This object can be used to alter the style of the input field's text.

---

**changed**

Whether the input field has been changed.

**Attribute** Can be read or written

**Return type** boolean

---

**Note:** This property is set to true when the user modifies the value of the input field. A client script should reset the property to false in order to detect subsequent changes.

---

**remove ()**

Remove the UI object.

## 6.8.7 Rect Transform

**class RectTransform**

A Unity engine Rect Transform for a UI object. See the [Unity manual](#) for more details.

**position**

Position of the rectangles pivot point relative to the anchors.

**Attribute** Can be read or written

**Return type** Tuple of (number, number)

**local\_position**

Position of the rectangles pivot point relative to the anchors.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)

**size**

Width and height of the rectangle.

**Attribute** Can be read or written

**Return type** Tuple of (number, number)

#### **upper\_right**

Position of the rectangles upper right corner relative to the anchors.

**Attribute** Can be read or written

**Return type** Tuple of (number, number)

#### **lower\_left**

Position of the rectangles lower left corner relative to the anchors.

**Attribute** Can be read or written

**Return type** Tuple of (number, number)

#### **anchor**

Set the minimum and maximum anchor points as a fraction of the size of the parent rectangle.

**Attribute** Write-only, cannot be read

**Return type** Tuple of (number, number)

#### **anchor\_max**

The anchor point for the lower left corner of the rectangle defined as a fraction of the size of the parent rectangle.

**Attribute** Can be read or written

**Return type** Tuple of (number, number)

#### **anchor\_min**

The anchor point for the upper right corner of the rectangle defined as a fraction of the size of the parent rectangle.

**Attribute** Can be read or written

**Return type** Tuple of (number, number)

#### **pivot**

Location of the pivot point around which the rectangle rotates, defined as a fraction of the size of the rectangle itself.

**Attribute** Can be read or written

**Return type** Tuple of (number, number)

#### **rotation**

Rotation, as a quaternion, of the object around its pivot point.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number, number)

#### **scale**

Scale factor applied to the object in the x, y and z dimensions.

**Attribute** Can be read or written

**Return type** Tuple of (number, number, number)



## 7.1 Python Client

This client provides a Python API for interacting with a kRPC server. It supports Python 2.7+ and 3.x

### 7.1.1 Installing the Library

The library can be found on [PyPI](#) or [downloaded from GitHub](#).

To install using pip on Linux:

```
pip install krpc
```

Or on Windows:

```
C:\Python27\Scripts\pip.exe install krpc
```

### 7.1.2 Connecting to the Server

The `krpc.connect()` function is used to open a connection to a server. It returns a connection object (of type `krpc.client.Client`) through which you can interact with the server. The following example connects to a server running on the local machine, queries its version and prints it out:

```
import krpc
conn = krpc.connect()
print(conn.krpc.get_status().version)
```

This function also accepts arguments that specify what address and port numbers to connect to, and an optional descriptive name for the connection which is displayed in the kRPC window in the game. For example:

```
import krpc
conn = krpc.connect(
    name='My Example Program',
    address='192.168.1.10',
    rpc_port=1000, stream_port=1001)
print(conn.krpc.get_status().version)
```

### 7.1.3 Calling Remote Procedures

The kRPC server provides *procedures* that a client can run. These procedures are arranged in groups called *services* to keep things organized. When connecting, the Python client interrogates the server to discover what procedures it provides, and dynamically creates class types, methods, properties etc. to call them.

The following example demonstrates how to invoke remote procedures using the Python client. It calls `SpaceCenter.active_vessel` to get an object representing the active vessel (of type `SpaceCenter.Vessel`). It sets the name of the vessel and then prints out its altitude:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel

vessel.name = "My Vessel"

flight_info = vessel.flight()
print(flight_info.mean_altitude)
```

All of the functionality provided by the `SpaceCenter` service is accessible via `conn.space_center`. To explore the functionality provided by a service, you can use the `help()` function from an interactive terminal. For example, running `help(conn.space_center)` will list all of the classes, enumerations, procedures and properties provided by the `SpaceCenter` service. This works similarly for class types, for example: `help(conn.space_center.Vessel)`.

### 7.1.4 Streaming Data from the Server

A common use case for kRPC is to continuously extract data from the game. The naive approach to do this would be to repeatedly call a remote procedure, such as in the following which repeatedly prints the position of the active vessel:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel
refframe = vessel.orbit.body.reference_frame
while True:
    print(vessel.position(refframe))
```

This approach requires significant communication overhead as request/response messages are repeatedly sent between the client and server. kRPC provides a more efficient mechanism to achieve this, called *streams*.

A stream repeatedly executes a procedure on the server (with a fixed set of argument values) and sends the result to the client. It only requires a single message to be sent to the server to establish the stream, which will then continuously send data to the client until the stream is closed.

The following example does the same thing as above using streams:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel
refframe = vessel.orbit.body.reference_frame
position = conn.add_stream(vessel.position, refframe)
while True:
    print(position())
```

It calls `krpc.client.Client.add_stream()` once at the start of the program to create the stream, and then repeatedly prints the position returned by the stream. The stream is automatically closed when the client disconnects.

Streams can also be created using the `with` statement, which ensures that the stream is closed after leaving the block:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel
refframe = vessel.orbit.body.reference_frame
with conn.stream(vessel.position, refframe) as position:
    while True:
        print(position())
```

A stream can be created for any procedure that returns a value. This includes both method calls and attribute accesses. The examples above demonstrated how to stream method calls. Attributes can be streamed as follows:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel
flight_info = vessel.flight()
altitude = conn.add_stream(getattr, flight_info, 'mean_altitude')
while True:
    print(altitude())
```

A stream can be created for any function call (except property setters). The most recent value of a stream can be obtained by calling `krpc.stream.Stream.__call__()`. A stream can be stopped and removed from the server by calling `krpc.stream.Stream.remove()` on the stream object. All of a clients streams are automatically stopped when it disconnects.

### 7.1.5 Synchronizing with Stream Updates

A common use case for kRPC is to wait until the value returned by a method or attribute changes, and then take some action. kRPC provides two mechanisms to do this efficiently: *condition variables* and *callbacks*.

#### Condition Variables

Each stream has a condition variable associated with it, that is notified whenever the value of the stream changes. The condition variables are instances of `threading.Condition` from the Python standard library. These can be used to block the current thread of execution until the value of the stream changes.

The following example waits until the abort button is pressed in game, by waiting for the value of `vessel.control.abort` to change to true:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel
with conn.stream(getattr, vessel.control, 'abort') as abort:
    with abort.condition:
        while not abort():
            abort.wait()
```

This code creates a stream, acquires a lock on the streams condition variable (using a `with` statement) and then repeatedly checks the value of `abort`. It leaves the loop when it changes to true.

The body of the loop calls `wait` on the stream, which causes the program to block until the value changes. This prevents the loop from ‘spinning’ and so it does not consume processing resources whilst waiting.

**Note:** The stream does not start receiving updates until the first call to `wait`. This means that the example code will

not miss any updates to the streams value, as it will have already locked the condition variable before the first stream update is received.

---

The example code above uses a `with` statement to acquire the lock on the condition variable. This can also be done explicitly using `acquire` and `release`:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel
with conn.stream(getattr, vessel.control, 'abort') as abort:
    abort.condition.acquire()
    while not abort():
        abort.wait()
    abort.condition.release()
```

## Callbacks

Streams allow you to register callback functions that are called whenever the value of the stream changes. Callback functions should take a single argument, which is the new value of the stream, and should return nothing.

For example the following program registers two callbacks that are invoked when the value of `vessel.control.abort` changes:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel
abort = conn.add_stream(getattr, vessel.control, 'abort')

def check_abort1(x):
    print 'Abort 1 called with a value of', x

def check_abort2(x):
    print 'Abort 2 called with a value of', x

abort.add_callback(check_abort1)
abort.add_callback(check_abort2)
abort.start()

# Keep the program running...
while True:
    pass
```

---

**Note:** When a stream is created it does not start receiving updates until `start` is called. This is implicitly called when accessing the value of a stream, but as this example does not do this an explicit call to `start` is required.

---

---

**Note:** The callbacks are registered before the call to `start` so that stream updates are not missed.

---

---

**Note:** The callback function may be called from a different thread to that which created the stream. Any changes to shared state must therefore be protected with appropriate synchronization.

---



## 7.1.6 Custom Events

Some procedures return event objects of type `krpc.event.Event`. These allow you to wait until an event occurs, by calling `krpc.event.Event.wait`. Under the hood, these are implemented using streams and condition variables.

Custom events can also be created. An expression API allows you to create code that runs on the server and these can be used to build a custom event. For example, the following creates the expression `mean_altitude > 1000` and then creates an event that will be triggered when the expression returns true:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel
flight = vessel.flight()

# Convert a remote procedure call to a message,
# so it can be passed to the server
mean_altitude = conn.get_call(getattr, flight, 'mean_altitude')

# Create an expression on the server
expr = conn.krpc.Expression.greater_than(
    conn.krpc.Expression.call(mean_altitude),
    conn.krpc.Expression.constant_double(1000))

# Create an event from the expression
event = conn.krpc.add_event(expr)

# Wait on the event
with event.condition:
    event.wait()
    print 'Altitude reached 1000m'
```

## 7.1.7 Client API Reference

`krpc.connect([name=None][, address='127.0.0.1'][, rpc_port=50000][, stream_port=50001])`

This function creates a connection to a kRPC server. It returns a `krpc.client.Client` object, through which the server can be communicated with.

### Parameters

- **name** (*str*) – A descriptive name for the connection. This is passed to the server and appears in the in-game server window.
- **address** (*str*) – The address of the server to connect to. Can either be a hostname or an IP address in dotted decimal notation. Defaults to '127.0.0.1'.
- **rpc\_port** (*int*) – The port number of the RPC Server. Defaults to 50000. This should match the RPC port number of the server you want to connect to.
- **stream\_port** (*int*) – The port number of the Stream Server. Defaults to 50001. This should match the stream port number of the server you want to connect to.

**class** `krpc.client.Client`

This class provides the interface for communicating with the server. It is dynamically populated with all the functionality provided by the server. Instances of this class should be obtained by calling `krpc.connect()`.

**add\_stream** (*func*, \**args*, \*\**kwargs*)

Create a stream for the function *func* called with arguments *args* and *kwargs*. Returns a *krpc.stream.Stream* object.

**stream** (*func*, \**args*, \*\**kwargs*)

Allows use of the `with` statement to create a stream and automatically remove it from the server when it goes out of scope. The function to be streamed should be passed as *func*, and its arguments as *args* and *kwargs*.

**get\_call** (*func*, \**args*, \*\**kwargs*)

Converts a call to function *func* with arguments *args* and *kwargs* into a message object. This allows descriptions of procedure calls to be passed to the server, for example when constructing custom events. See *Custom Events*.

**close** ()

Closes the connection to the server.

**krpc**

The basic KRPC service, providing interaction with basic functionality of the server.

**Return type** *krpc.client.KRPC*

**class** *krpc.client.KRPC*

This class provides access to the basic server functionality provided by the *KRPC* service. An instance can be obtained by calling *krpc.client.Client.krpc*.

See *KRPC* for full documentation of this class.

Some of this functionality is used internally by the python client (for example to create and remove streams) and therefore does not need to be used directly from application code.

**class** *krpc.stream.Stream*

This class represents a stream. See *Streaming Data from the Server*.

**start** (*wait=True*)

Starts the stream. When a stream is created by calling *krpc.client.Client.add\_stream()* it does not start sending updates to the client until this method is called.

If *wait* is true, this method will block until at least one update has been received from the server.

If *wait* is false, the method starts the stream and returns immediately. Subsequent calls to `__call__()` may raise a *StreamError* exception if the stream does not yet contain a value.

**\_\_call\_\_** ()

Returns the most recent value for the stream. If executing the remote procedure for the stream throws an exception, calling this method will rethrow the exception. Raises a *StreamError* exception if no update has been received from the server.

If the stream has not been started this method calls `start(True)` to start the stream and wait until at least one update has been received.

**condition**

A condition variable (of type `threading.Condition`) that is notified whenever the value of the stream changes.

**wait** (*timeout=None*)

This method blocks until the value of the stream changes or the operation times out.

The streams condition variable must be locked before calling this method.

If *timeout* is specified and is not `None`, it should be a floating point number specifying the timeout in seconds for the operation.

If the stream has not been started this method calls `start(False)` to start the stream (without waiting for at least one update to be received).

**add\_callback** (*callback*)

Adds a callback function that is invoked whenever the value of the stream changes. The callback function should take one argument, which is passed the new value of the stream.

---

**Note:** The callback function may be called from a different thread to that which created the stream. Any changes to shared state must therefore be protected with appropriate synchronization.

---

**remove** ()

Removes the stream from the server.

**class** `krpc.event.Event`

This class represents an event. See *Custom Events*. It is wrapper around a stream of type `bool` that indicates when the event occurs.

**start** ()

Starts the event. When an event is created, it will not receive updates from the server until this method is called.

**condition**

The condition variable (of type `threading.Condition`) that is notified whenever the event occurs.

**wait** (*timeout=None*)

This method blocks until the event occurs or the operation times out.

The events condition variable must be locked before calling this method.

If *timeout* is specified and is not `None`, it should be a floating point number specifying the timeout in seconds for the operation.

If the event has not been started this method calls `start()` to start the underlying stream.

**add\_callback** (*callback*)

Adds a callback function that is invoked whenever the event occurs. The callback function should be a function that takes zero arguments.

**remove** ()

Removes the event from the server.

**stream**

Returns the underlying stream for the event.

## 7.2 KRPC API

### 7.2.1 KRPC

None None None None Main kRPC service, used by clients to interact with basic server functionality.

**static** `get_client_id()`

Returns the identifier for the current client.

**Return type** `str`

**static** `get_client_name()`

Returns the name of the current client. This is an empty string if the client has no name.

**Return type** str

#### **clients**

A list of RPC clients that are currently connected to the server. Each entry in the list is a clients identifier, name and address.

**Attribute** Read-only, cannot be set

**Return type** list(tuple(str, str, str))

#### **static get\_status()**

Returns some information about the server, such as the version.

**Return type** krpc.schema.KRPC.Status

#### **static get\_services()**

Returns information on all services, procedures, classes, properties etc. provided by the server. Can be used by client libraries to automatically create functionality such as stubs.

**Return type** krpc.schema.KRPC.Services

#### **current\_game\_scene**

Get the current game scene.

**Attribute** Read-only, cannot be set

**Return type** GameScene

#### **paused**

Whether the game is paused.

**Attribute** Can be read or written

**Return type** bool

#### **class GameScene**

The game scene. See *current\_game\_scene*.

##### **space\_center**

The game scene showing the Kerbal Space Center buildings.

##### **flight**

The game scene showing a vessel in flight (or on the launchpad/runway).

##### **tracking\_station**

The tracking station.

##### **editor\_vab**

The Vehicle Assembly Building.

##### **editor\_sph**

The Space Plane Hangar.

#### **class InvalidOperationException**

A method call was made to a method that is invalid given the current state of the object.

#### **class ArgumentException**

A method was invoked where at least one of the passed arguments does not meet the parameter specification of the method.

#### **class ArgumentNullException**

A null reference was passed to a method that does not accept it as a valid argument.

#### **class ArgumentOutOfRangeException**

The value of an argument is outside the allowable range of values as defined by the invoked method.

## 7.2.2 Expressions

### class **Expression**

A server side expression.

**static constant\_double** (*value*)

A constant value of type double.

**Parameters** *value* (*float*) –

**Return type** *Expression*

**static constant\_float** (*value*)

A constant value of type float.

**Parameters** *value* (*float*) –

**Return type** *Expression*

**static constant\_int** (*value*)

A constant value of type int.

**Parameters** *value* (*int*) –

**Return type** *Expression*

**static constant\_string** (*value*)

A constant value of type string.

**Parameters** *value* (*str*) –

**Return type** *Expression*

**static call** (*call*)

An RPC call.

**Parameters** *call* (*krpc.schema.KRPC.ProcedureCall*) –

**Return type** *Expression*

**static equal** (*arg0*, *arg1*)

Equality comparison.

**Parameters**

- *arg0* (*Expression*) –
- *arg1* (*Expression*) –

**Return type** *Expression*

**static not\_equal** (*arg0*, *arg1*)

Inequality comparison.

**Parameters**

- *arg0* (*Expression*) –
- *arg1* (*Expression*) –

**Return type** *Expression*

**static greater\_than** (*arg0*, *arg1*)

Greater than numerical comparison.

**Parameters**

- *arg0* (*Expression*) –

- **arg1** (Expression)–

**Return type** *Expression*

**static greater\_than\_or\_equal** (*arg0*, *arg1*)

Greater than or equal numerical comparison.

**Parameters**

- **arg0** (Expression)–
- **arg1** (Expression)–

**Return type** *Expression*

**static less\_than** (*arg0*, *arg1*)

Less than numerical comparison.

**Parameters**

- **arg0** (Expression)–
- **arg1** (Expression)–

**Return type** *Expression*

**static less\_than\_or\_equal** (*arg0*, *arg1*)

Less than or equal numerical comparison.

**Parameters**

- **arg0** (Expression)–
- **arg1** (Expression)–

**Return type** *Expression*

**static and** (*arg0*, *arg1*)

Boolean and operator.

**Parameters**

- **arg0** (Expression)–
- **arg1** (Expression)–

**Return type** *Expression*

**static or** (*arg0*, *arg1*)

Boolean or operator.

**Parameters**

- **arg0** (Expression)–
- **arg1** (Expression)–

**Return type** *Expression*

**static exclusive\_or** (*arg0*, *arg1*)

Boolean exclusive-or operator.

**Parameters**

- **arg0** (Expression)–
- **arg1** (Expression)–

**Return type** *Expression*

**static not** (*arg*)  
 Boolean negation operator.

**Parameters** *arg* (Expression) –

**Return type** *Expression*

**static add** (*arg0*, *arg1*)  
 Numerical addition.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

**Return type** *Expression*

**static subtract** (*arg0*, *arg1*)  
 Numerical subtraction.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

**Return type** *Expression*

**static multiply** (*arg0*, *arg1*)  
 Numerical multiplication.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

**Return type** *Expression*

**static divide** (*arg0*, *arg1*)  
 Numerical division.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

**Return type** *Expression*

**static modulo** (*arg0*, *arg1*)  
 Numerical modulo operator.

**Parameters**

- **arg0** (Expression) –
- **arg1** (Expression) –

**Returns** The remainder of *arg0* divided by *arg1*

**Return type** *Expression*

**static power** (*arg0*, *arg1*)  
 Numerical power operator.

**Parameters**

- **arg0** (Expression) –

- **arg1** (Expression) –

**Returns** arg0 raised to the power of arg1

**Return type** *Expression*

**static left\_shift** (*arg0*, *arg1*)

Bitwise left shift.

**Parameters**

- **arg0** (Expression) –

- **arg1** (Expression) –

**Return type** *Expression*

**static right\_shift** (*arg0*, *arg1*)

Bitwise right shift.

**Parameters**

- **arg0** (Expression) –

- **arg1** (Expression) –

**Return type** *Expression*

**static to\_double** (*arg*)

Convert to a double type.

**Parameters** **arg** (Expression) –

**Return type** *Expression*

**static to\_float** (*arg*)

Convert to a float type.

**Parameters** **arg** (Expression) –

**Return type** *Expression*

**static to\_int** (*arg*)

Convert to an int type.

**Parameters** **arg** (Expression) –

**Return type** *Expression*

## 7.3 SpaceCenter API

### 7.3.1 SpaceCenter

Provides functionality to interact with Kerbal Space Program. This includes controlling the active vessel, managing its resources, planning maneuver nodes and auto-piloting.

**active\_vessel**

The currently active vessel.

**Attribute** Can be read or written

**Return type** *Vessel*



**vessels**

A list of all the vessels in the game.

**Attribute** Read-only, cannot be set

**Return type** `list(Vessel)`

**bodies**

A dictionary of all celestial bodies (planets, moons, etc.) in the game, keyed by the name of the body.

**Attribute** Read-only, cannot be set

**Return type** `dict(str, CelestialBody)`

**target\_body**

The currently targeted celestial body.

**Attribute** Can be read or written

**Return type** `CelestialBody`

**target\_vessel**

The currently targeted vessel.

**Attribute** Can be read or written

**Return type** `Vessel`

**target\_docking\_port**

The currently targeted docking port.

**Attribute** Can be read or written

**Return type** `DockingPort`

**static clear\_target()**

Clears the current target.

**static launchable\_vessels(craft\_directory)**

Returns a list of vessels from the given *craft\_directory* that can be launched.

**Parameters** **craft\_directory** (*str*) – Name of the directory in the current saves “Ships” directory. For example “VAB” or “SPH”.

**Return type** `list(str)`

**static launch\_vessel(craft\_directory, name, launch\_site)**

Launch a vessel.

**Parameters**

- **craft\_directory** (*str*) – Name of the directory in the current saves “Ships” directory, that contains the craft file. For example “VAB” or “SPH”.
- **name** (*str*) – Name of the vessel to launch. This is the name of the “.craft” file in the save directory, without the “.craft” file extension.
- **launch\_site** (*str*) – Name of the launch site. For example “LaunchPad” or “Runway”.

**static launch\_vessel\_from\_vab(name)**

Launch a new vessel from the VAB onto the launchpad.

**Parameters** **name** (*str*) – Name of the vessel to launch.

---

**Note:** This is equivalent to calling `launch_vessel()` with the craft directory set to “VAB” and the launch site set to “LaunchPad”.

---

**static launch\_vessel\_from\_sph** (*name*)

Launch a new vessel from the SPH onto the runway.

**Parameters** *name* (*str*) – Name of the vessel to launch.

---

**Note:** This is equivalent to calling `launch_vessel()` with the craft directory set to “SPH” and the launch site set to “Runway”.

---

**static save** (*name*)

Save the game with a given name. This will create a save file called `name.sfs` in the folder of the current save game.

**Parameters** *name* (*str*) –

**static load** (*name*)

Load the game with the given name. This will create a load a save file called `name.sfs` from the folder of the current save game.

**Parameters** *name* (*str*) –

**static quicksave** ()

Save a quicksave.

---

**Note:** This is the same as calling `save()` with the name “quicksave”.

---

**static quickload** ()

Load a quicksave.

---

**Note:** This is the same as calling `load()` with the name “quicksave”.

---

**ui\_visible**

Whether the UI is visible.

**Attribute** Can be read or written

**Return type** bool

**navball**

Whether the navball is visible.

**Attribute** Can be read or written

**Return type** bool

**ut**

The current universal time in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**g**

The value of the [gravitational constant](#)  $G$  in  $N(m/kg)^2$ .

**Attribute** Read-only, cannot be set

**Return type** float

#### **warp\_rate**

The current warp rate. This is the rate at which time is passing for either on-rails or physical time warp. For example, a value of 10 means time is passing 10x faster than normal. Returns 1 if time warp is not active.

**Attribute** Read-only, cannot be set

**Return type** float

#### **warp\_factor**

The current warp factor. This is the index of the rate at which time is passing for either regular “on-rails” or physical time warp. Returns 0 if time warp is not active. When in on-rails time warp, this is equal to *rails\_warp\_factor*, and in physics time warp, this is equal to *physics\_warp\_factor*.

**Attribute** Read-only, cannot be set

**Return type** float

#### **rails\_warp\_factor**

The time warp rate, using regular “on-rails” time warp. A value between 0 and 7 inclusive. 0 means no time warp. Returns 0 if physical time warp is active.

If requested time warp factor cannot be set, it will be set to the next lowest possible value. For example, if the vessel is too close to a planet. See [the KSP wiki](#) for details.

**Attribute** Can be read or written

**Return type** int

#### **physics\_warp\_factor**

The physical time warp rate. A value between 0 and 3 inclusive. 0 means no time warp. Returns 0 if regular “on-rails” time warp is active.

**Attribute** Can be read or written

**Return type** int

#### **static can\_rails\_warp\_at** (*[factor = 1]*)

Returns `True` if regular “on-rails” time warp can be used, at the specified warp *factor*. The maximum time warp rate is limited by various things, including how close the active vessel is to a planet. See [the KSP wiki](#) for details.

**Parameters** **factor** (*int*) – The warp factor to check.

**Return type** bool

#### **maximum\_rails\_warp\_factor**

The current maximum regular “on-rails” warp factor that can be set. A value between 0 and 7 inclusive. See [the KSP wiki](#) for details.

**Attribute** Read-only, cannot be set

**Return type** int

#### **static warp\_to** (*ut*, *[max\_rails\_rate = 100000.0]*, *[max\_physics\_rate = 2.0]*)

Uses time acceleration to warp forward to a time in the future, specified by universal time *ut*. This call blocks until the desired time is reached. Uses regular “on-rails” or physical time warp as appropriate. For example, physical time warp is used when the active vessel is traveling through an atmosphere. When using regular “on-rails” time warp, the warp rate is limited by *max\_rails\_rate*, and when using physical time warp, the warp rate is limited by *max\_physics\_rate*.

**Parameters**

- **ut** (*float*) – The universal time to warp to, in seconds.
- **max\_rails\_rate** (*float*) – The maximum warp rate in regular “on-rails” time warp.
- **max\_physics\_rate** (*float*) – The maximum warp rate in physical time warp.

**Returns** When the time warp is complete.

**static transform\_position** (*position, from, to*)

Converts a position from one reference frame to another.

**Parameters**

- **position** (*tuple*) – Position, as a vector, in reference frame *from*.
- **from** (*ReferenceFrame*) – The reference frame that the position is in.
- **to** (*ReferenceFrame*) – The reference frame to covert the position to.

**Returns** The corresponding position, as a vector, in reference frame *to*.

**Return type** tuple(float, float, float)

**static transform\_direction** (*direction, from, to*)

Converts a direction from one reference frame to another.

**Parameters**

- **direction** (*tuple*) – Direction, as a vector, in reference frame *from*.
- **from** (*ReferenceFrame*) – The reference frame that the direction is in.
- **to** (*ReferenceFrame*) – The reference frame to covert the direction to.

**Returns** The corresponding direction, as a vector, in reference frame *to*.

**Return type** tuple(float, float, float)

**static transform\_rotation** (*rotation, from, to*)

Converts a rotation from one reference frame to another.

**Parameters**

- **rotation** (*tuple*) – Rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *from*.
- **from** (*ReferenceFrame*) – The reference frame that the rotation is in.
- **to** (*ReferenceFrame*) – The reference frame to covert the rotation to.

**Returns** The corresponding rotation, as a quaternion of the form  $(x, y, z, w)$ , in reference frame *to*.

**Return type** tuple(float, float, float, float)

**static transform\_velocity** (*position, velocity, from, to*)

Converts a velocity (acting at the specified position) from one reference frame to another. The position is required to take the relative angular velocity of the reference frames into account.

**Parameters**

- **position** (*tuple*) – Position, as a vector, in reference frame *from*.
- **velocity** (*tuple*) – Velocity, as a vector that points in the direction of travel and whose magnitude is the speed in meters per second, in reference frame *from*.
- **from** (*ReferenceFrame*) – The reference frame that the position and velocity are in.
- **to** (*ReferenceFrame*) – The reference frame to covert the velocity to.

**Returns** The corresponding velocity, as a vector, in reference frame *to*.

**Return type** tuple(float, float, float)

#### **far\_available**

Whether [Ferram Aerospace Research](#) is installed.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **warp\_mode**

The current time warp mode. Returns *WarpMode.none* if time warp is not active, *WarpMode.rails* if regular “on-rails” time warp is active, or *WarpMode.physics* if physical time warp is active.

**Attribute** Read-only, cannot be set

**Return type** *WarpMode*

#### **camera**

An object that can be used to control the camera.

**Attribute** Read-only, cannot be set

**Return type** *Camera*

#### **waypoint\_manager**

The waypoint manager.

**Attribute** Read-only, cannot be set

**Return type** *WaypointManager*

#### **contract\_manager**

The contract manager.

**Attribute** Read-only, cannot be set

**Return type** *ContractManager*

#### **class WarpMode**

The time warp mode. Returned by *WarpMode*

##### **rails**

Time warp is active, and in regular “on-rails” mode.

##### **physics**

Time warp is active, and in physical time warp mode.

##### **none**

Time warp is not active.

## 7.3.2 Vessel

#### **class Vessel**

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources. Created using *active\_vessel* or *vessels*.

##### **name**

The name of the vessel.

**Attribute** Can be read or written

**Return type** str

**type**

The type of the vessel.

**Attribute** Can be read or written

**Return type** *VesselType*

**situation**

The situation the vessel is in.

**Attribute** Read-only, cannot be set

**Return type** *VesselSituation*

**recoverable**

Whether the vessel is recoverable.

**Attribute** Read-only, cannot be set

**Return type** bool

**recover()**

Recover the vessel.

**met**

The mission elapsed time in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**biome**

The name of the biome the vessel is currently in.

**Attribute** Read-only, cannot be set

**Return type** str

**flight** (*[reference\_frame = None]*)

Returns a *Flight* object that can be used to get flight telemetry for the vessel, in the specified reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – Reference frame. Defaults to the vessel's surface reference frame (*Vessel.surface\_reference\_frame*).

**Return type** *Flight*

---

**Note:** When this is called with no arguments, the vessel's surface reference frame is used. This reference frame moves with the vessel, therefore velocities and speeds returned by the flight object will be zero. See the *reference frames tutorial* for examples of getting *the orbital and surface speeds of a vessel*.

---

**orbit**

The current orbit of the vessel.

**Attribute** Read-only, cannot be set

**Return type** *Orbit*

**control**

Returns a *Control* object that can be used to manipulate the vessel's control inputs. For example, its pitch/yaw/roll controls, RCS and thrust.

**Attribute** Read-only, cannot be set

**Return type** *Control*

#### **comms**

Returns a *Comms* object that can be used to interact with CommNet for this vessel.

**Attribute** Read-only, cannot be set

**Return type** *Comms*

#### **auto\_pilot**

An *AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

**Attribute** Read-only, cannot be set

**Return type** *AutoPilot*

#### **resources**

A *Resources* object, that can used to get information about resources stored in the vessel.

**Attribute** Read-only, cannot be set

**Return type** *Resources*

#### **resources\_in\_decouple\_stage** (*stage* [, *cumulative* = *True* ])

Returns a *Resources* object, that can used to get information about resources stored in a given *stage*.

##### **Parameters**

- **stage** (*int*) – Get resources for parts that are decoupled in this stage.
- **cumulative** (*bool*) – When *False*, returns the resources for parts decoupled in just the given stage. When *True* returns the resources decoupled in the given stage and all subsequent stages combined.

**Return type** *Resources*

---

**Note:** For details on stage numbering, see the discussion on *Staging*.

---

#### **parts**

A *Parts* object, that can used to interact with the parts that make up this vessel.

**Attribute** Read-only, cannot be set

**Return type** *Parts*

#### **mass**

The total mass of the vessel, including resources, in kg.

**Attribute** Read-only, cannot be set

**Return type** float

#### **dry\_mass**

The total mass of the vessel, excluding resources, in kg.

**Attribute** Read-only, cannot be set

**Return type** float

#### **thrust**

The total thrust currently being produced by the vessel's engines, in Newtons. This is computed by summing *Engine.thrust* for every engine in the vessel.

**Attribute** Read-only, cannot be set

**Return type** float

**available\_thrust**

Gets the total available thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *Engine.available\_thrust* for every active engine in the vessel.

**Attribute** Read-only, cannot be set

**Return type** float

**max\_thrust**

The total maximum thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *Engine.max\_thrust* for every active engine.

**Attribute** Read-only, cannot be set

**Return type** float

**max\_vacuum\_thrust**

The total maximum thrust that can be produced by the vessel's active engines when the vessel is in a vacuum, in Newtons. This is computed by summing *Engine.max\_vacuum\_thrust* for every active engine.

**Attribute** Read-only, cannot be set

**Return type** float

**specific\_impulse**

The combined specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

**Attribute** Read-only, cannot be set

**Return type** float

**vacuum\_specific\_impulse**

The combined vacuum specific impulse of all active engines, in seconds. This is computed using the formula [described here](#).

**Attribute** Read-only, cannot be set

**Return type** float

**kerbin\_sea\_level\_specific\_impulse**

The combined specific impulse of all active engines at sea level on Kerbin, in seconds. This is computed using the formula [described here](#).

**Attribute** Read-only, cannot be set

**Return type** float

**moment\_of\_inertia**

The moment of inertia of the vessel around its center of mass in  $kg.m^2$ . The inertia values in the returned 3-tuple are around the pitch, roll and yaw directions respectively. This corresponds to the vessels reference frame (*ReferenceFrame*).

**Attribute** Read-only, cannot be set

**Return type** tuple(float, float, float)

**inertia\_tensor**

The inertia tensor of the vessel around its center of mass, in the vessels reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

**Attribute** Read-only, cannot be set



**Return type** list(float)

**available\_torque**

The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gimbaled engines and aerodynamic control surfaces. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**available\_reaction\_wheel\_torque**

The maximum torque that the currently active and powered reaction wheels can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**available\_rcs\_torque**

The maximum torque that the currently active RCS thrusters can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**available\_engine\_torque**

The maximum torque that the currently active and gimbaled engines can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**available\_control\_surface\_torque**

The maximum torque that the aerodynamic control surfaces can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**available\_other\_torque**

The maximum torque that parts (excluding reaction wheels, gimbaled engines, RCS and control surfaces) can generate. Returns the torques in  $N.m$  around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**reference\_frame**

The reference frame that is fixed relative to the vessel, and orientated with the vessel.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel.
- The x-axis points out to the right of the vessel.

- The y-axis points in the forward direction of the vessel.
- The z-axis points out of the bottom of the vessel.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

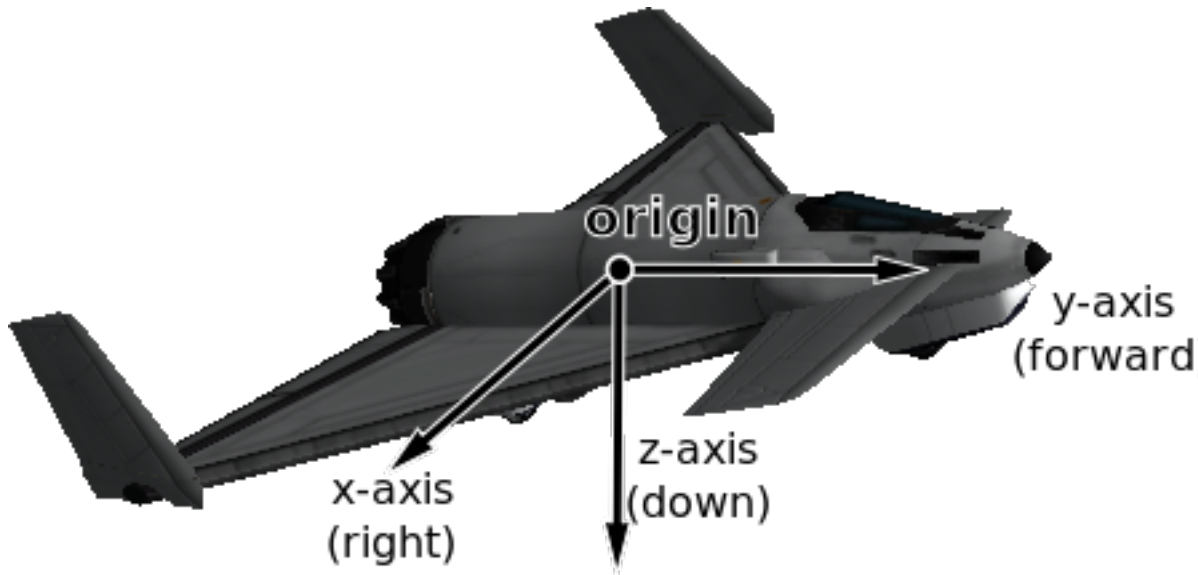


Fig. 7.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

#### **orbital\_reference\_frame**

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

---

**Note:** Be careful not to confuse this with ‘orbit’ mode on the navball.

---

#### **surface\_reference\_frame**

The reference frame that is fixed relative to the vessel, and orientated with the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the north and up directions on the surface of the body.

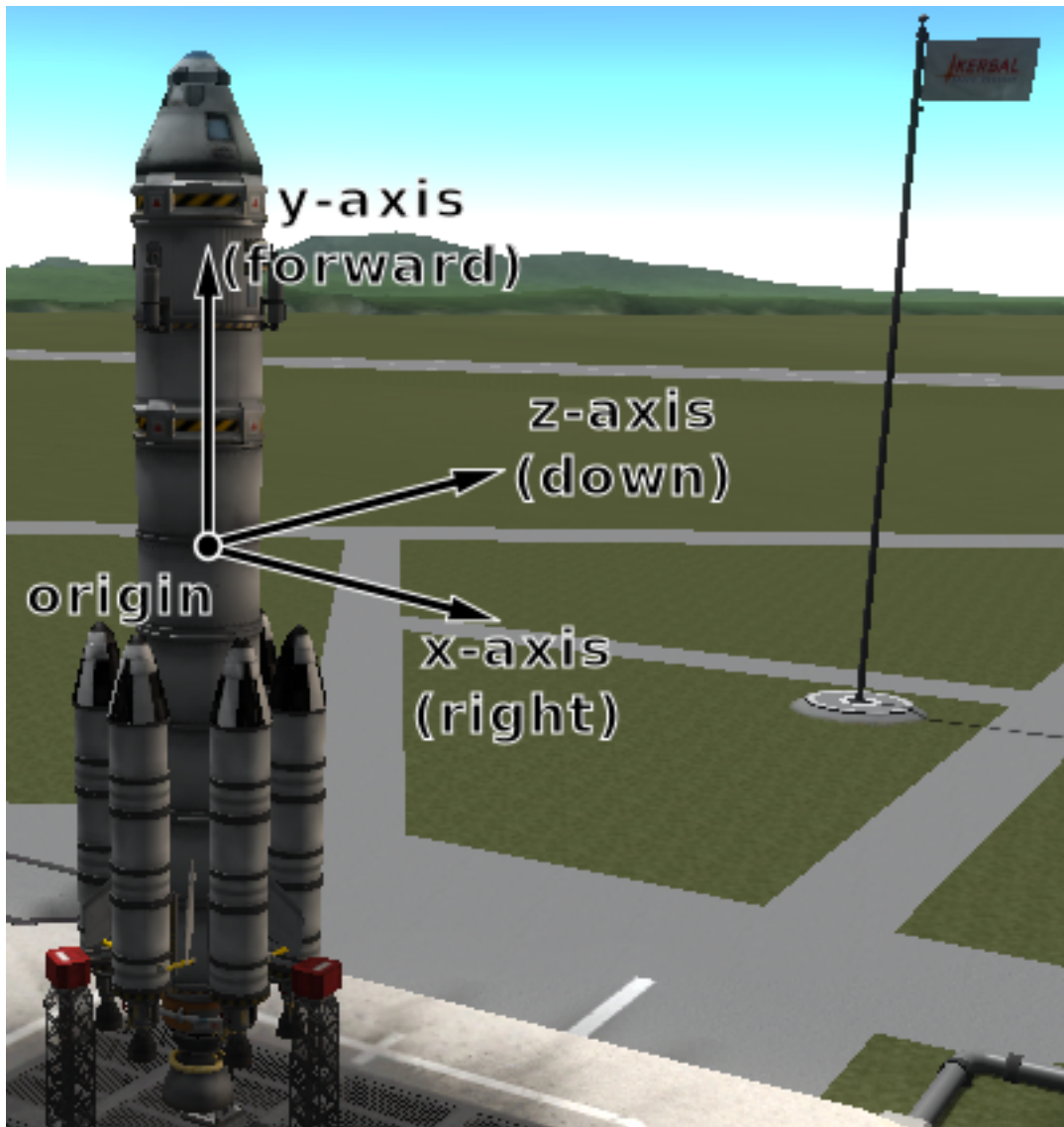


Fig. 7.2: Vessel reference frame origin and axes for the Kerbal-X rocket

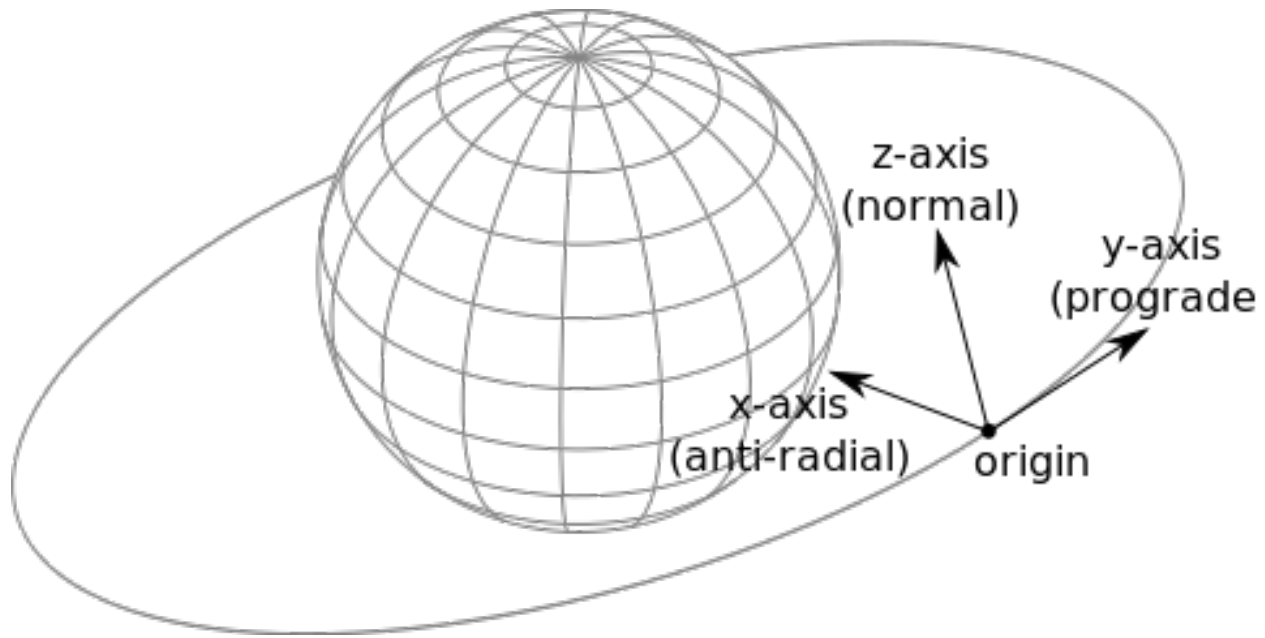


Fig. 7.3: Vessel orbital reference frame origin and axes

- The x-axis points in the [zenith](#) direction (upwards, normal to the body being orbited, from the center of the body towards the center of mass of the vessel).
- The y-axis points northwards towards the [astronomical horizon](#) (north, and tangential to the surface of the body – the direction in which a compass would point when on the surface).
- The z-axis points eastwards towards the [astronomical horizon](#) (east, and tangential to the surface of the body – east on a compass when on the surface).

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

---

**Note:** Be careful not to confuse this with ‘surface’ mode on the navball.

---

#### **surface\_velocity\_reference\_frame**

The reference frame that is fixed relative to the vessel, and orientated with the velocity vector of the vessel relative to the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel’s velocity vector.
- The y-axis points in the direction of the vessel’s velocity vector, relative to the surface of the body being orbited.
- The z-axis is in the plane of the [astronomical horizon](#).
- The x-axis is orthogonal to the other two axes.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

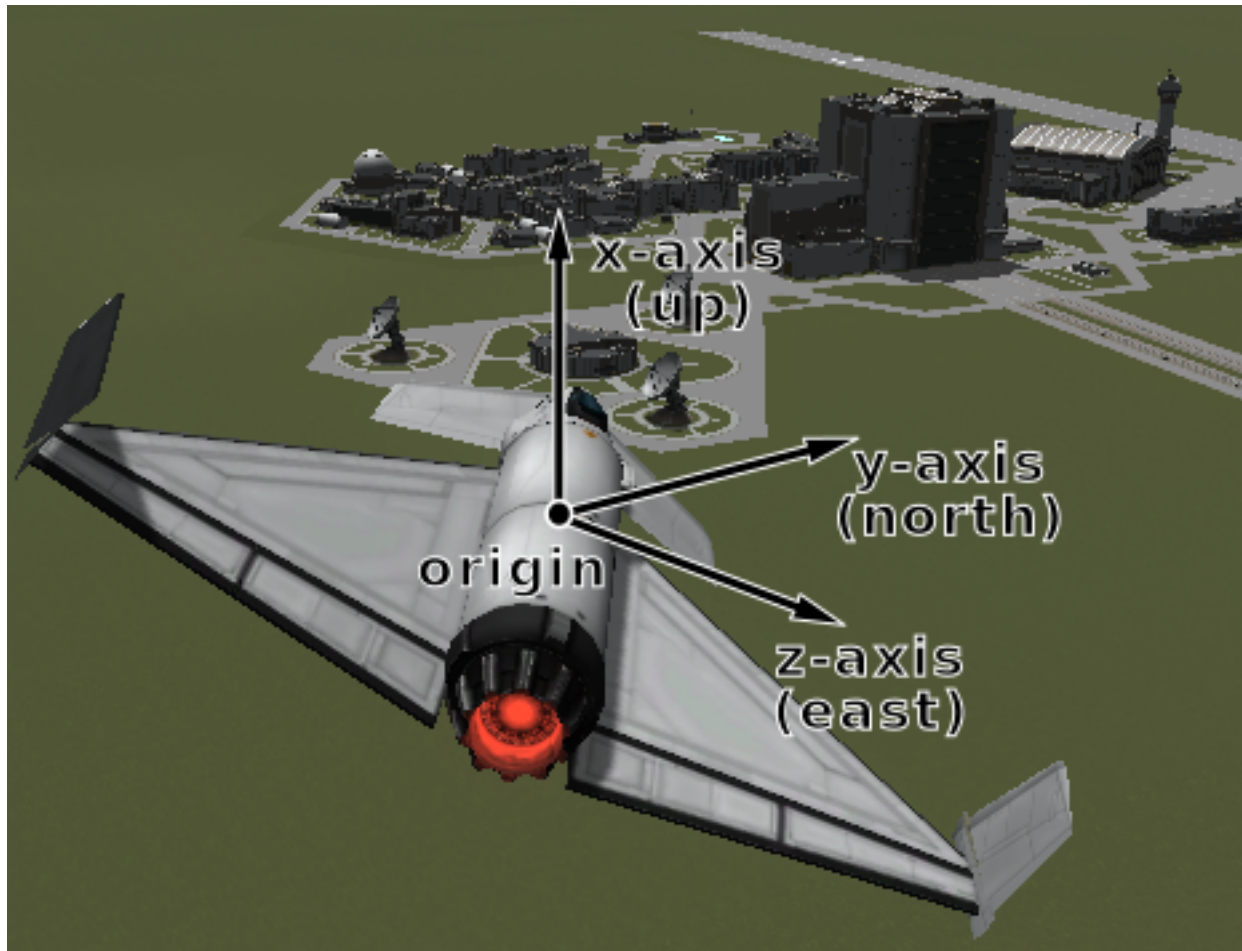


Fig. 7.4: Vessel surface reference frame origin and axes

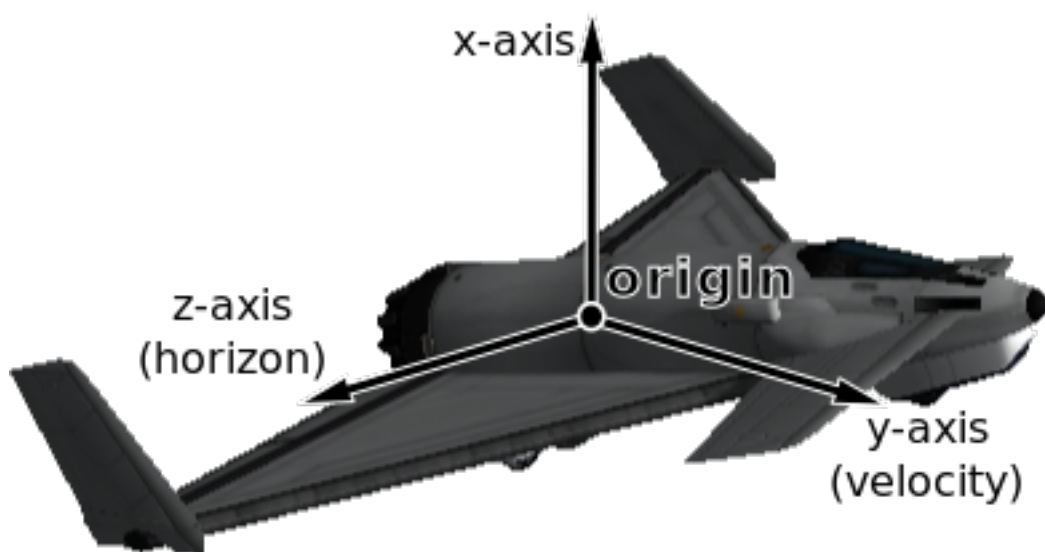


Fig. 7.5: Vessel surface velocity reference frame origin and axes

**position** (*reference\_frame*)

The position of the center of mass of the vessel, in the given reference frame.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** tuple(float, float, float)

**bounding\_box** (*reference\_frame*)

The axis-aligned bounding box of the vessel in the given reference frame.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**velocity** (*reference\_frame*)

The velocity of the center of mass of the vessel, in the given reference frame.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

**Return type** tuple(float, float, float)

**rotation** (*reference\_frame*)

The rotation of the vessel, in the given reference frame.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** tuple(float, float, float, float)

**direction** (*reference\_frame*)

The direction in which the vessel is pointing, in the given reference frame.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

**angular\_velocity** (*reference\_frame*)

The angular velocity of the vessel, in the given reference frame.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the vessel, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

**Return type** tuple(float, float, float)

**class VesselType**

The type of a vessel. See *Vessel.type*.

**base**  
Base.

**debris**  
Debris.

**lander**  
Lander.

**plane**  
Plane.

**probe**  
Probe.

**relay**  
Relay.

**rover**  
Rover.

**ship**  
Ship.

**station**  
Station.

#### **class VesselSituation**

The situation a vessel is in. See *Vessel.situation*.

**docked**  
Vessel is docked to another.

**escaping**  
Escaping.

**flying**  
Vessel is flying through an atmosphere.

**landed**  
Vessel is landed on the surface of a body.

**orbiting**  
Vessel is orbiting a body.

**pre\_launch**  
Vessel is awaiting launch.

**splashed**  
Vessel has splashed down in an ocean.

**sub\_orbital**  
Vessel is on a sub-orbital trajectory.

### 7.3.3 CelestialBody

#### **class CelestialBody**

Represents a celestial body (such as a planet or moon). See *bodies*.

**name**  
The name of the body.

**Attribute** Read-only, cannot be set

**Return type** str

**satellites**

A list of celestial bodies that are in orbit around this celestial body.

**Attribute** Read-only, cannot be set

**Return type** list(*CelestialBody*)

**orbit**

The orbit of the body.

**Attribute** Read-only, cannot be set

**Return type** *Orbit*

**mass**

The mass of the body, in kilograms.

**Attribute** Read-only, cannot be set

**Return type** float

**gravitational\_parameter**

The [standard gravitational parameter](#) of the body in  $m^3s^{-2}$ .

**Attribute** Read-only, cannot be set

**Return type** float

**surface\_gravity**

The acceleration due to gravity at sea level (mean altitude) on the body, in  $m/s^2$ .

**Attribute** Read-only, cannot be set

**Return type** float

**rotational\_period**

The sidereal rotational period of the body, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**rotational\_speed**

The rotational speed of the body, in radians per second.

**Attribute** Read-only, cannot be set

**Return type** float

**rotation\_angle**

The current rotation angle of the body, in radians. A value between 0 and  $2\pi$

**Attribute** Read-only, cannot be set

**Return type** float

**initial\_rotation**

The initial rotation angle of the body (at UT 0), in radians. A value between 0 and  $2\pi$

**Attribute** Read-only, cannot be set

**Return type** float

**equatorial\_radius**

The equatorial radius of the body, in meters.



**Attribute** Read-only, cannot be set

**Return type** float

**surface\_height** (*latitude, longitude*)

The height of the surface relative to mean sea level, in meters, at the given position. When over water this is equal to 0.

**Parameters**

- **latitude** (*float*) – Latitude in degrees.
- **longitude** (*float*) – Longitude in degrees.

**Return type** float

**bedrock\_height** (*latitude, longitude*)

The height of the surface relative to mean sea level, in meters, at the given position. When over water, this is the height of the sea-bed and is therefore negative value.

**Parameters**

- **latitude** (*float*) – Latitude in degrees.
- **longitude** (*float*) – Longitude in degrees.

**Return type** float

**msl\_position** (*latitude, longitude, reference\_frame*)

The position at mean sea level at the given latitude and longitude, in the given reference frame.

**Parameters**

- **latitude** (*float*) – Latitude in degrees.
- **longitude** (*float*) – Longitude in degrees.
- **reference\_frame** (*ReferenceFrame*) – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Return type** tuple(float, float, float)

**surface\_position** (*latitude, longitude, reference\_frame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position of the surface of the water.

**Parameters**

- **latitude** (*float*) – Latitude in degrees.
- **longitude** (*float*) – Longitude in degrees.
- **reference\_frame** (*ReferenceFrame*) – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Return type** tuple(float, float, float)

**bedrock\_position** (*latitude, longitude, reference\_frame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position at the bottom of the sea-bed.

**Parameters**

- **latitude** (*float*) – Latitude in degrees.

- **longitude** (*float*) – Longitude in degrees.
- **reference\_frame** (*ReferenceFrame*) – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Return type** tuple(float, float, float)

**position\_at\_altitude** (*latitude, longitude, altitude, reference\_frame*)

The position at the given latitude, longitude and altitude, in the given reference frame.

**Parameters**

- **latitude** (*float*) – Latitude in degrees.
- **longitude** (*float*) – Longitude in degrees.
- **altitude** (*float*) – Altitude in meters above sea level.
- **reference\_frame** (*ReferenceFrame*) – Reference frame for the returned position vector.

**Returns** Position as a vector.

**Return type** tuple(float, float, float)

**altitude\_at\_position** (*position, reference\_frame*)

The altitude, in meters, of the given position in the given reference frame.

**Parameters**

- **position** (*tuple*) – Position as a vector.
- **reference\_frame** (*ReferenceFrame*) – Reference frame for the position vector.

**Return type** float

**latitude\_at\_position** (*position, reference\_frame*)

The latitude of the given position, in the given reference frame.

**Parameters**

- **position** (*tuple*) – Position as a vector.
- **reference\_frame** (*ReferenceFrame*) – Reference frame for the position vector.

**Return type** float

**longitude\_at\_position** (*position, reference\_frame*)

The longitude of the given position, in the given reference frame.

**Parameters**

- **position** (*tuple*) – Position as a vector.
- **reference\_frame** (*ReferenceFrame*) – Reference frame for the position vector.

**Return type** float

**sphere\_of\_influence**

The radius of the sphere of influence of the body, in meters.

**Attribute** Read-only, cannot be set

**Return type** float

**has\_atmosphere**

True if the body has an atmosphere.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **atmosphere\_depth**

The depth of the atmosphere, in meters.

**Attribute** Read-only, cannot be set

**Return type** float

#### **atmospheric\_density\_at\_position** (*position*, *reference\_frame*)

The atmospheric density at the given position, in  $kg/m^3$ , in the given reference frame.

##### **Parameters**

- **position** (*tuple*) – The position vector at which to measure the density.
- **reference\_frame** (*ReferenceFrame*) – Reference frame that the position vector is in.

**Return type** float

#### **has\_atmospheric\_oxygen**

True if there is oxygen in the atmosphere, required for air-breathing engines.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **temperature\_at** (*position*, *reference\_frame*)

The temperature on the body at the given position, in the given reference frame.

##### **Parameters**

- **position** (*tuple*) – Position as a vector.
- **reference\_frame** (*ReferenceFrame*) – The reference frame that the position is in.

**Return type** float

---

**Note:** This calculation is performed using the bodies current position, which means that the value could be wrong if you want to know the temperature in the far future.

---

#### **density\_at** (*altitude*)

Gets the air density, in  $kg/m^3$ , for the specified altitude above sea level, in meters.

**Parameters** **altitude** (*float*) –

**Return type** float

---

**Note:** This is an approximation, because actual calculations, taking sun exposure into account to compute air temperature, require us to know the exact point on the body where the density is to be computed (knowing the altitude is not enough). However, the difference is small for high altitudes, so it makes very little difference for trajectory prediction.

---

#### **pressure\_at** (*altitude*)

Gets the air pressure, in Pascals, for the specified altitude above sea level, in meters.

**Parameters** **altitude** (*float*) –

**Return type** float

**biomes**

The biomes present on this body.

**Attribute** Read-only, cannot be set

**Return type** set(str)

**biome\_at** (*latitude*, *longitude*)

The biome at the given latitude and longitude, in degrees.

**Parameters**

- **latitude** (*float*) –
- **longitude** (*float*) –

**Return type** str

**flying\_high\_altitude\_threshold**

The altitude, in meters, above which a vessel is considered to be flying “high” when doing science.

**Attribute** Read-only, cannot be set

**Return type** float

**space\_high\_altitude\_threshold**

The altitude, in meters, above which a vessel is considered to be in “high” space when doing science.

**Attribute** Read-only, cannot be set

**Return type** float

**reference\_frame**

The reference frame that is fixed relative to the celestial body.

- The origin is at the center of the body.
- The axes rotate with the body.
- The x-axis points from the center of the body towards the intersection of the prime meridian and equator (the position at 0° longitude, 0° latitude).
- The y-axis points from the center of the body towards the north pole.
- The z-axis points from the center of the body towards the equator at 90°E longitude.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

**non\_rotating\_reference\_frame**

The reference frame that is fixed relative to this celestial body, and orientated in a fixed direction (it does not rotate with the body).

- The origin is at the center of the body.
- The axes do not rotate.
- The x-axis points in an arbitrary direction through the equator.
- The y-axis points from the center of the body towards the north pole.
- The z-axis points in an arbitrary direction through the equator.

**Attribute** Read-only, cannot be set

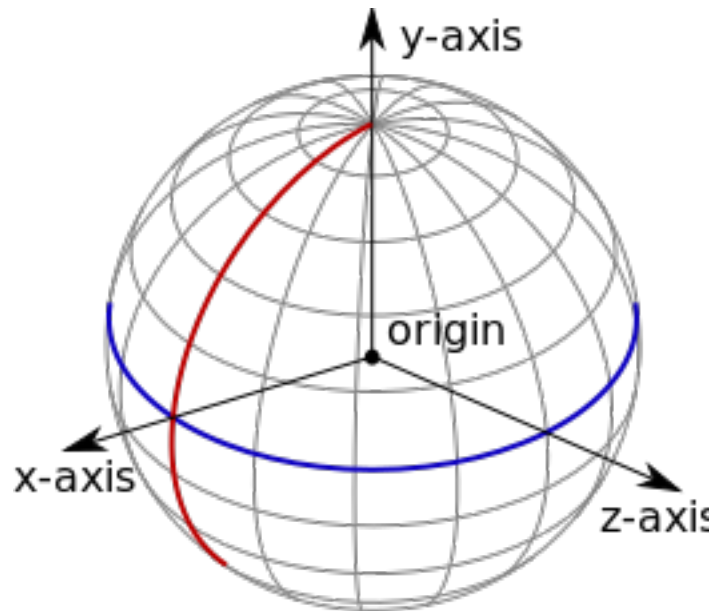


Fig. 7.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

**Return type** *ReferenceFrame*

#### **orbital\_reference\_frame**

The reference frame that is fixed relative to this celestial body, but orientated with the body's orbital prograde/normal/radial directions.

- The origin is at the center of the body.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

#### **position** (*reference\_frame*)

The position of the center of the body, in the specified reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** tuple(float, float, float)

#### **velocity** (*reference\_frame*)

The linear velocity of the body, in the specified reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

**Return type** tuple(float, float, float)

**rotation** (*reference\_frame*)

The rotation of the body, in the specified reference frame.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** tuple(float, float, float, float)

**direction** (*reference\_frame*)

The direction in which the north pole of the celestial body is pointing, in the specified reference frame.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

**angular\_velocity** (*reference\_frame*)

The angular velocity of the body in the specified reference frame.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame the returned angular velocity is in.

**Returns** The angular velocity as a vector. The magnitude of the vector is the rotational speed of the body, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

**Return type** tuple(float, float, float)

### 7.3.4 Flight

**class Flight**

Used to get flight telemetry for a vessel, by calling *Vessel.flight()*. All of the information returned by this class is given in the reference frame passed to that method. Obtained by calling *Vessel.flight()*.

---

**Note:** To get orbital information, such as the apoapsis or inclination, see *Orbit*.

---

**g\_force**

The current G force acting on the vessel in  $m/s^2$ .

**Attribute** Read-only, cannot be set

**Return type** float

**mean\_altitude**

The altitude above sea level, in meters. Measured from the center of mass of the vessel.

**Attribute** Read-only, cannot be set

**Return type** float

**surface\_altitude**

The altitude above the surface of the body or sea level, whichever is closer, in meters. Measured from the center of mass of the vessel.

**Attribute** Read-only, cannot be set

**Return type** float

#### **bedrock\_altitude**

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor. Measured from the center of mass of the vessel.

**Attribute** Read-only, cannot be set

**Return type** float

#### **elevation**

The elevation of the terrain under the vessel, in meters. This is the height of the terrain above sea level, and is negative when the vessel is over the sea.

**Attribute** Read-only, cannot be set

**Return type** float

#### **latitude**

The [latitude](#) of the vessel for the body being orbited, in degrees.

**Attribute** Read-only, cannot be set

**Return type** float

#### **longitude**

The [longitude](#) of the vessel for the body being orbited, in degrees.

**Attribute** Read-only, cannot be set

**Return type** float

#### **velocity**

The velocity of the vessel, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the vessel in meters per second.

**Return type** tuple(float, float, float)

#### **speed**

The speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Return type** float

#### **horizontal\_speed**

The horizontal speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Return type** float

#### **vertical\_speed**

The vertical speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Return type** float

#### **center\_of\_mass**

The position of the center of mass of the vessel, in the reference frame *ReferenceFrame*

**Attribute** Read-only, cannot be set

**Returns** The position as a vector.

**Return type** tuple(float, float, float)

#### **rotation**

The rotation of the vessel, in the reference frame *ReferenceFrame*

**Attribute** Read-only, cannot be set

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** tuple(float, float, float, float)

#### **direction**

The direction that the vessel is pointing in, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

#### **pitch**

The pitch of the vessel relative to the horizon, in degrees. A value between  $-90^\circ$  and  $+90^\circ$ .

**Attribute** Read-only, cannot be set

**Return type** float

#### **heading**

The heading of the vessel (its angle relative to north), in degrees. A value between  $0^\circ$  and  $360^\circ$ .

**Attribute** Read-only, cannot be set

**Return type** float

#### **roll**

The roll of the vessel relative to the horizon, in degrees. A value between  $-180^\circ$  and  $+180^\circ$ .

**Attribute** Read-only, cannot be set

**Return type** float

#### **prograde**

The prograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

#### **retrograde**

The retrograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

#### **normal**

The direction normal to the vessels orbit, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.



**Return type** tuple(float, float, float)

#### **anti\_normal**

The direction opposite to the normal of the vessels orbit, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

#### **radial**

The radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

#### **anti\_radial**

The direction opposite to the radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

#### **atmosphere\_density**

The current density of the atmosphere around the vessel, in  $kg/m^3$ .

**Attribute** Read-only, cannot be set

**Return type** float

#### **dynamic\_pressure**

The dynamic pressure acting on the vessel, in Pascals. This is a measure of the strength of the aerodynamic forces. It is equal to  $\frac{1}{2} \cdot \text{air density} \cdot \text{velocity}^2$ . It is commonly denoted  $Q$ .

**Attribute** Read-only, cannot be set

**Return type** float

#### **static\_pressure**

The static atmospheric pressure acting on the vessel, in Pascals.

**Attribute** Read-only, cannot be set

**Return type** float

#### **static\_pressure\_at\_msl**

The static atmospheric pressure at mean sea level, in Pascals.

**Attribute** Read-only, cannot be set

**Return type** float

#### **aerodynamic\_force**

The total aerodynamic forces acting on the vessel, in reference frame *ReferenceFrame*.

**Attribute** Read-only, cannot be set

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Return type** tuple(float, float, float)

**simulate\_aerodynamic\_force\_at** (*body, position, velocity*)

Simulate and return the total aerodynamic forces acting on the vessel, if it were to be traveling with the given velocity at the given position in the atmosphere of the given celestial body.

**Parameters**

- **body** (*CelestialBody*) –
- **position** (*tuple*) –
- **velocity** (*tuple*) –

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Return type** tuple(float, float, float)

**lift**

The [aerodynamic lift](#) currently acting on the vessel.

**Attribute** Read-only, cannot be set

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Return type** tuple(float, float, float)

**drag**

The [aerodynamic drag](#) currently acting on the vessel.

**Attribute** Read-only, cannot be set

**Returns** A vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

**Return type** tuple(float, float, float)

**speed\_of\_sound**

The speed of sound, in the atmosphere around the vessel, in *m/s*.

**Attribute** Read-only, cannot be set

**Return type** float

**mach**

The speed of the vessel, in multiples of the speed of sound.

**Attribute** Read-only, cannot be set

**Return type** float

**reynolds\_number**

The vessels Reynolds number.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**true\_air\_speed**

The [true air speed](#) of the vessel, in meters per second.

**Attribute** Read-only, cannot be set

**Return type** float

**equivalent\_air\_speed**

The [equivalent air speed](#) of the vessel, in meters per second.

**Attribute** Read-only, cannot be set

**Return type** float

**terminal\_velocity**

An estimate of the current terminal velocity of the vessel, in meters per second. This is the speed at which the drag forces cancel out the force of gravity.

**Attribute** Read-only, cannot be set

**Return type** float

**angle\_of\_attack**

The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

**Attribute** Read-only, cannot be set

**Return type** float

**sideslip\_angle**

The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

**Attribute** Read-only, cannot be set

**Return type** float

**total\_air\_temperature**

The [total air temperature](#) of the atmosphere around the vessel, in Kelvin. This includes the *Flight.static\_air\_temperature* and the vessel's kinetic energy.

**Attribute** Read-only, cannot be set

**Return type** float

**static\_air\_temperature**

The [static \(ambient\) temperature](#) of the atmosphere around the vessel, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**stall\_fraction**

The current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**drag\_coefficient**

The coefficient of drag. This is the amount of drag produced by the vessel. It depends on air speed, air density and wing area.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**lift\_coefficient**

The coefficient of lift. This is the amount of lift produced by the vessel, and depends on air speed, air density and wing area.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**ballistic\_coefficient**

The [ballistic coefficient](#).

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** Requires [Ferram Aerospace Research](#).

---

**thrust\_specific\_fuel\_consumption**

The thrust specific fuel consumption for the jet engines on the vessel. This is a measure of the efficiency of the engines, with a lower value indicating a more efficient vessel. This value is the number of Newtons of fuel that are burned, per hour, to produce one newton of thrust.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** Requires [Ferram Aerospace Research](#).

---

## 7.3.5 Orbit

**class Orbit**

Describes an orbit. For example, the orbit of a vessel, obtained by calling *Vessel.orbit*, or a celestial body, obtained by calling *CelestialBody.orbit*.

**body**

The celestial body (e.g. planet or moon) around which the object is orbiting.

**Attribute** Read-only, cannot be set

**Return type** *CelestialBody*

**apoapsis**

Gets the apoapsis of the orbit, in meters, from the center of mass of the body being orbited.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** For the apoapsis altitude reported on the in-game map view, use *Orbit.apoapsis\_altitude*.

---

**periapsis**

The periapsis of the orbit, in meters, from the center of mass of the body being orbited.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** For the periapsis altitude reported on the in-game map view, use *Orbit.periapsis\_altitude*.

---

**apoapsis\_altitude**

The apoapsis of the orbit, in meters, above the sea level of the body being orbited.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** This is equal to *Orbit.apoapsis* minus the equatorial radius of the body.

---

**periapsis\_altitude**

The periapsis of the orbit, in meters, above the sea level of the body being orbited.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** This is equal to *Orbit.periapsis* minus the equatorial radius of the body.

---

**semi\_major\_axis**

The semi-major axis of the orbit, in meters.

**Attribute** Read-only, cannot be set

**Return type** float

**semi\_minor\_axis**

The semi-minor axis of the orbit, in meters.

**Attribute** Read-only, cannot be set

**Return type** float

**radius**

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** This value will change over time if the orbit is elliptical.

---

**radius\_at (ut)**

The orbital radius at the given time, in meters.

**Parameters** **ut** (*float*) – The universal time to measure the radius at.

**Return type** float

**position\_at** (*ut*, *reference\_frame*)

The position at a given time, in the specified reference frame.

**Parameters**

- **ut** (*float*) – The universal time to measure the position at.
- **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** tuple(float, float, float)

**speed**

The current orbital speed of the object in meters per second.

**Attribute** Read-only, cannot be set

**Return type** float

---

**Note:** This value will change over time if the orbit is elliptical.

---

**period**

The orbital period, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**time\_to\_apoapsis**

The time until the object reaches apoapsis, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**time\_to\_periapsis**

The time until the object reaches periapsis, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**eccentricity**

The [eccentricity](#) of the orbit.

**Attribute** Read-only, cannot be set

**Return type** float

**inclination**

The [inclination](#) of the orbit, in radians.

**Attribute** Read-only, cannot be set

**Return type** float

**longitude\_of\_ascending\_node**

The [longitude of the ascending node](#), in radians.

**Attribute** Read-only, cannot be set

**Return type** float

**argument\_of\_periapsis**

The [argument of periapsis](#), in radians.

**Attribute** Read-only, cannot be set

**Return type** float

**mean\_anomaly\_at\_epoch**

The [mean anomaly at epoch](#).

**Attribute** Read-only, cannot be set

**Return type** float

**epoch**

The time since the epoch (the point at which the [mean anomaly at epoch](#) was measured, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**mean\_anomaly**

The [mean anomaly](#).

**Attribute** Read-only, cannot be set

**Return type** float

**mean\_anomaly\_at\_ut** (*ut*)

The mean anomaly at the given time.

**Parameters** **ut** (*float*) – The universal time in seconds.

**Return type** float

**eccentric\_anomaly**

The [eccentric anomaly](#).

**Attribute** Read-only, cannot be set

**Return type** float

**eccentric\_anomaly\_at\_ut** (*ut*)

The eccentric anomaly at the given universal time.

**Parameters** **ut** (*float*) – The universal time, in seconds.

**Return type** float

**true\_anomaly**

The [true anomaly](#).

**Attribute** Read-only, cannot be set

**Return type** float

**true\_anomaly\_at\_ut** (*ut*)

The true anomaly at the given time.

**Parameters** **ut** (*float*) – The universal time in seconds.

**Return type** float

**true\_anomaly\_at\_radius** (*radius*)

The true anomaly at the given orbital radius.

**Parameters** **radius** (*float*) – The orbital radius in meters.

**Return type** float

**ut\_at\_true\_anomaly** (*true\_anomaly*)

The universal time, in seconds, corresponding to the given true anomaly.

**Parameters** **true\_anomaly** (*float*) – True anomaly.

**Return type** float

**radius\_at\_true\_anomaly** (*true\_anomaly*)

The orbital radius at the point in the orbit given by the true anomaly.

**Parameters** **true\_anomaly** (*float*) – The true anomaly.

**Return type** float

**true\_anomaly\_at\_an** (*target*)

The true anomaly of the ascending node with the given target vessel.

**Parameters** **target** (*Vessel*) – Target vessel.

**Return type** float

**true\_anomaly\_at\_dn** (*target*)

The true anomaly of the descending node with the given target vessel.

**Parameters** **target** (*Vessel*) – Target vessel.

**Return type** float

**orbital\_speed**

The current orbital speed in meters per second.

**Attribute** Read-only, cannot be set

**Return type** float

**orbital\_speed\_at** (*time*)

The orbital speed at the given time, in meters per second.

**Parameters** **time** (*float*) – Time from now, in seconds.

**Return type** float

**static reference\_plane\_normal** (*reference\_frame*)

The direction that is normal to the orbits reference plane, in the given reference frame. The reference plane is the plane from which the orbits inclination is measured.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

**static reference\_plane\_direction** (*reference\_frame*)

The direction from which the orbits longitude of ascending node is measured, in the given reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

**relative\_inclination** (*target*)

Relative inclination of this orbit and the orbit of the given target vessel, in radians.



**Parameters** **target** (*Vessel*) – Target vessel.

**Return type** *float*

**time\_to\_soi\_change**

The time until the object changes sphere of influence, in seconds. Returns *NaN* if the object is not going to change sphere of influence.

**Attribute** Read-only, cannot be set

**Return type** *float*

**next\_orbit**

If the object is going to change sphere of influence in the future, returns the new orbit after the change. Otherwise returns *None*.

**Attribute** Read-only, cannot be set

**Return type** *Orbit*

**time\_of\_closest\_approach** (*target*)

Estimates and returns the time at closest approach to a target vessel.

**Parameters** **target** (*Vessel*) – Target vessel.

**Returns** The universal time at closest approach, in seconds.

**Return type** *float*

**distance\_at\_closest\_approach** (*target*)

Estimates and returns the distance at closest approach to a target vessel, in meters.

**Parameters** **target** (*Vessel*) – Target vessel.

**Return type** *float*

**list\_closest\_approaches** (*target, orbits*)

Returns the times at closest approach and corresponding distances, to a target vessel.

**Parameters**

- **target** (*Vessel*) – Target vessel.
- **orbits** (*int*) – The number of future orbits to search.

**Returns** A list of two lists. The first is a list of times at closest approach, as universal times in seconds. The second is a list of corresponding distances at closest approach, in meters.

**Return type** *list(list(float))*

## 7.3.6 Control

**class Control**

Used to manipulate the controls of a vessel. This includes adjusting the throttle, enabling/disabling systems such as SAS and RCS, or altering the direction in which the vessel is pointing. Obtained by calling *Vessel.control*.

---

**Note:** Control inputs (such as pitch, yaw and roll) are zeroed when all clients that have set one or more of these inputs are no longer connected.

---

**source**

The source of the vessels control, for example by a kerbal or a probe core.

**Attribute** Read-only, cannot be set

**Return type** *ControlSource*

**state**

The control state of the vessel.

**Attribute** Read-only, cannot be set

**Return type** *ControlState*

**sas**

The state of SAS.

**Attribute** Can be read or written

**Return type** bool

---

**Note:** Equivalent to *AutoPilot.sas*

---

**sas\_mode**

The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

**Attribute** Can be read or written

**Return type** *SASMode*

---

**Note:** Equivalent to *AutoPilot.sas\_mode*

---

**speed\_mode**

The current *SpeedMode* of the navball. This is the mode displayed next to the speed at the top of the navball.

**Attribute** Can be read or written

**Return type** *SpeedMode*

**rcs**

The state of RCS.

**Attribute** Can be read or written

**Return type** bool

**reaction\_wheels**

Returns whether all reactive wheels on the vessel are active, and sets the active state of all reaction wheels. See *ReactionWheel.active*.

**Attribute** Can be read or written

**Return type** bool

**gear**

The state of the landing gear/legs.

**Attribute** Can be read or written

**Return type** bool

**legs**

Returns whether all landing legs on the vessel are deployed, and sets the deployment state of all landing legs. Does not include wheels (for example landing gear). See *Leg.deployed*.

**Attribute** Can be read or written

**Return type** bool

**wheels**

Returns whether all wheels on the vessel are deployed, and sets the deployment state of all wheels. Does not include landing legs. See *Wheel.deployed*.

**Attribute** Can be read or written

**Return type** bool

**lights**

The state of the lights.

**Attribute** Can be read or written

**Return type** bool

**brakes**

The state of the wheel brakes.

**Attribute** Can be read or written

**Return type** bool

**antennas**

Returns whether all antennas on the vessel are deployed, and sets the deployment state of all antennas. See *Antenna.deployed*.

**Attribute** Can be read or written

**Return type** bool

**cargo\_bays**

Returns whether any of the cargo bays on the vessel are open, and sets the open state of all cargo bays. See *CargoBay.open*.

**Attribute** Can be read or written

**Return type** bool

**intakes**

Returns whether all of the air intakes on the vessel are open, and sets the open state of all air intakes. See *Intake.open*.

**Attribute** Can be read or written

**Return type** bool

**parachutes**

Returns whether all parachutes on the vessel are deployed, and sets the deployment state of all parachutes. Cannot be set to *False*. See *Parachute.deployed*.

**Attribute** Can be read or written

**Return type** bool

**radiators**

Returns whether all radiators on the vessel are deployed, and sets the deployment state of all radiators. See *Radiator.deployed*.

**Attribute** Can be read or written

**Return type** bool

**resource\_harvesters**

Returns whether all of the resource harvesters on the vessel are deployed, and sets the deployment state of all resource harvesters. See *ResourceHarvester.deployed*.

**Attribute** Can be read or written

**Return type** bool

**resource\_harvesters\_active**

Returns whether any of the resource harvesters on the vessel are active, and sets the active state of all resource harvesters. See *ResourceHarvester.active*.

**Attribute** Can be read or written

**Return type** bool

**solar\_panels**

Returns whether all solar panels on the vessel are deployed, and sets the deployment state of all solar panels. See *SolarPanel.deployed*.

**Attribute** Can be read or written

**Return type** bool

**abort**

The state of the abort action group.

**Attribute** Can be read or written

**Return type** bool

**throttle**

The state of the throttle. A value between 0 and 1.

**Attribute** Can be read or written

**Return type** float

**input\_mode**

Sets the behavior of the pitch, yaw, roll and translation control inputs. When set to additive, these inputs are added to the vessels current inputs. This mode is the default. When set to override, these inputs (if non-zero) override the vessels inputs. This mode prevents keyboard control, or SAS, from interfering with the controls when they are set.

**Attribute** Can be read or written

**Return type** *ControlInputMode*

**pitch**

The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

**Attribute** Can be read or written

**Return type** float

**yaw**

The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

**Attribute** Can be read or written

**Return type** float

**roll**

The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

**Attribute** Can be read or written

**Return type** float

**forward**

The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

**Attribute** Can be read or written

**Return type** float

**up**

The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

**Attribute** Can be read or written

**Return type** float

**right**

The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

**Attribute** Can be read or written

**Return type** float

**wheel\_throttle**

The state of the wheel throttle. A value between -1 and 1. A value of 1 rotates the wheels forwards, a value of -1 rotates the wheels backwards.

**Attribute** Can be read or written

**Return type** float

**wheel\_steering**

The state of the wheel steering. A value between -1 and 1. A value of 1 steers to the left, and a value of -1 steers to the right.

**Attribute** Can be read or written

**Return type** float

**current\_stage**

The current stage of the vessel. Corresponds to the stage number in the in-game UI.

**Attribute** Read-only, cannot be set

**Return type** int

**activate\_next\_stage()**

Activates the next stage. Equivalent to pressing the space bar in-game.

**Returns** A list of vessel objects that are jettisoned from the active vessel.

**Return type** list(*Vessel*)

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *active\_vessel* no longer refer to the active vessel.

---

**get\_action\_group(group)**

Returns *True* if the given action group is enabled.

**Parameters** `group` (*int*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.

**Return type** `bool`

**set\_action\_group** (*group*, *state*)

Sets the state of the given action group.

**Parameters**

- **group** (*int*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.
- **state** (*bool*) –

**toggle\_action\_group** (*group*)

Toggles the state of the given action group.

**Parameters** `group` (*int*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the [Extended Action Groups mod](#) is installed.

**add\_node** (*ut* [, *prograde* = 0.0] [, *normal* = 0.0] [, *radial* = 0.0])

Creates a maneuver node at the given universal time, and returns a *Node* object that can be used to modify it. Optionally sets the magnitude of the delta-v for the maneuver node in the prograde, normal and radial directions.

**Parameters**

- **ut** (*float*) – Universal time of the maneuver node.
- **prograde** (*float*) – Delta-v in the prograde direction.
- **normal** (*float*) – Delta-v in the normal direction.
- **radial** (*float*) – Delta-v in the radial direction.

**Return type** *Node*

**nodes**

Returns a list of all existing maneuver nodes, ordered by time from first to last.

**Attribute** Read-only, cannot be set

**Return type** `list(Node)`

**remove\_nodes** ()

Remove all maneuver nodes.

**class ControlState**

The control state of a vessel. See *Control.state*.

**full**

Full controllable.

**partial**

Partially controllable.

**none**

Not controllable.

**class ControlSource**

The control source of a vessel. See *Control.source*.

**kerbal**

Vessel is controlled by a Kerbal.

**probe**

Vessel is controlled by a probe core.

**none**

Vessel is not controlled.

**class SASMode**

The behavior of the SAS auto-pilot. See *AutoPilot.sas\_mode*.

**stability\_assist**

Stability assist mode. Dampen out any rotation.

**maneuver**

Point in the burn direction of the next maneuver node.

**prograde**

Point in the prograde direction.

**retrograde**

Point in the retrograde direction.

**normal**

Point in the orbit normal direction.

**anti\_normal**

Point in the orbit anti-normal direction.

**radial**

Point in the orbit radial direction.

**anti\_radial**

Point in the orbit anti-radial direction.

**target**

Point in the direction of the current target.

**anti\_target**

Point away from the current target.

**class SpeedMode**

The mode of the speed reported in the navball. See *Control.speed\_mode*.

**orbit**

Speed is relative to the vessel's orbit.

**surface**

Speed is relative to the surface of the body being orbited.

**target**

Speed is relative to the current target.

**class ControlInputMode**

See *Control.input\_mode*.

**additive**

Control inputs are added to the vessels current control inputs.

**override**

Control inputs (when they are non-zero) override the vessels current control inputs.

### 7.3.7 Communications

**class Comms**

Used to interact with CommNet for a given vessel. Obtained by calling *Vessel.comms*.

**can\_communicate**

Whether the vessel can communicate with KSC.

**Attribute** Read-only, cannot be set

**Return type** bool

**can\_transmit\_science**

Whether the vessel can transmit science data to KSC.

**Attribute** Read-only, cannot be set

**Return type** bool

**signal\_strength**

Signal strength to KSC.

**Attribute** Read-only, cannot be set

**Return type** float

**signal\_delay**

Signal delay to KSC in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**power**

The combined power of all active antennae on the vessel.

**Attribute** Read-only, cannot be set

**Return type** float

**control\_path**

The communication path used to control the vessel.

**Attribute** Read-only, cannot be set

**Return type** list(*CommLink*)

**class CommLink**

Represents a communication node in the network. For example, a vessel or the KSC.

**type**

The type of link.

**Attribute** Read-only, cannot be set

**Return type** *CommLinkType*

**signal\_strength**

Signal strength of the link.

**Attribute** Read-only, cannot be set

**Return type** float

**start**

Start point of the link.



```

        Attribute Read-only, cannot be set
        Return type CommNode
    end
    Start point of the link.
        Attribute Read-only, cannot be set
        Return type CommNode
class CommLinkType
    The type of a communication link. See CommLink.type.
    home
        Link is to a base station on Kerbin.
    control
        Link is to a control source, for example a manned spacecraft.
    relay
        Link is to a relay satellite.
class CommNode
    Represents a communication node in the network. For example, a vessel or the KSC.
    name
        Name of the communication node.
        Attribute Read-only, cannot be set
        Return type str
    is_home
        Whether the communication node is on Kerbin.
        Attribute Read-only, cannot be set
        Return type bool
    is_control_point
        Whether the communication node is a control point, for example a manned vessel.
        Attribute Read-only, cannot be set
        Return type bool
    is_vessel
        Whether the communication node is a vessel.
        Attribute Read-only, cannot be set
        Return type bool
    vessel
        The vessel for this communication node.
        Attribute Read-only, cannot be set
        Return type Vessel

```

### 7.3.8 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*
  - *Antenna*
  - *Cargo Bay*
  - *Control Surface*
  - *Decoupler*
  - *Docking Port*
  - *Engine*
  - *Experiment*
  - *Fairing*
  - *Intake*
  - *Leg*
  - *Launch Clamp*
  - *Light*
  - *Parachute*
  - *Radiator*
  - *Resource Converter*
  - *Resource Harvester*
  - *Reaction Wheel*
  - *RCS*
  - *Sensor*
  - *Solar Panel*
  - *Thruster*
  - *Wheel*
- *Trees of Parts*
  - *Traversing the Tree*
  - *Attachment Modes*
- *Fuel Lines*
- *Staging*

## Parts

### **class Parts**

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling

*Vessel.parts.*

**all**

A list of all of the vessels parts.

**Attribute** Read-only, cannot be set

**Return type** *list(Part)*

**root**

The vessels root part.

**Attribute** Read-only, cannot be set

**Return type** *Part*

---

**Note:** See the discussion on *Trees of Parts*.

---

**controlling**

The part from which the vessel is controlled.

**Attribute** Can be read or written

**Return type** *Part*

**with\_name** (*name*)

A list of parts whose *Part.name* is *name*.

**Parameters** **name** (*str*) –

**Return type** *list(Part)*

**with\_title** (*title*)

A list of all parts whose *Part.title* is *title*.

**Parameters** **title** (*str*) –

**Return type** *list(Part)*

**with\_tag** (*tag*)

A list of all parts whose *Part.tag* is *tag*.

**Parameters** **tag** (*str*) –

**Return type** *list(Part)*

**with\_module** (*module\_name*)

A list of all parts that contain a *Module* whose *Module.name* is *module\_name*.

**Parameters** **module\_name** (*str*) –

**Return type** *list(Part)*

**in\_stage** (*stage*)

A list of all parts that are activated in the given *stage*.

**Parameters** **stage** (*int*) –

**Return type** *list(Part)*

---

**Note:** See the discussion on *Staging*.

---

**in\_decouple\_stage** (*stage*)

A list of all parts that are decoupled in the given *stage*.

**Parameters** *stage* (*int*) –

**Return type** list(*Part*)

---

**Note:** See the discussion on *Staging*.

---

**modules\_with\_name** (*module\_name*)

A list of modules (combined across all parts in the vessel) whose *Module.name* is *module\_name*.

**Parameters** *module\_name* (*str*) –

**Return type** list(*Module*)

**antennas**

A list of all antennas in the vessel.

**Attribute** Read-only, cannot be set

**Return type** list(*Antenna*)

**cargo\_bays**

A list of all cargo bays in the vessel.

**Attribute** Read-only, cannot be set

**Return type** list(*CargoBay*)

**control\_surfaces**

A list of all control surfaces in the vessel.

**Attribute** Read-only, cannot be set

**Return type** list(*ControlSurface*)

**decouplers**

A list of all decouplers in the vessel.

**Attribute** Read-only, cannot be set

**Return type** list(*Decoupler*)

**docking\_ports**

A list of all docking ports in the vessel.

**Attribute** Read-only, cannot be set

**Return type** list(*DockingPort*)

**engines**

A list of all engines in the vessel.

**Attribute** Read-only, cannot be set

**Return type** list(*Engine*)

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters, jet engines and RCS thrusters.

---

**experiments**

A list of all science experiments in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(Experiment)`

#### **fairings**

A list of all fairings in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(Fairing)`

#### **intakes**

A list of all intakes in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(Intake)`

#### **legs**

A list of all landing legs attached to the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(Leg)`

#### **launch\_clamps**

A list of all launch clamps attached to the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(LaunchClamp)`

#### **lights**

A list of all lights in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(Light)`

#### **parachutes**

A list of all parachutes in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(Parachute)`

#### **radiators**

A list of all radiators in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(Radiator)`

#### **rcs**

A list of all RCS blocks/thrusters in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(RCS)`

#### **reaction\_wheels**

A list of all reaction wheels in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(ReactionWheel)`

**resource\_converters**

A list of all resource converters in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(ResourceConverter)`

**resource\_harvesters**

A list of all resource harvesters in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(ResourceHarvester)`

**sensors**

A list of all sensors in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(Sensor)`

**solar\_panels**

A list of all solar panels in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(SolarPanel)`

**wheels**

A list of all wheels in the vessel.

**Attribute** Read-only, cannot be set

**Return type** `list(Wheel)`

## Part

**class Part**

Represents an individual part. Vessels are made up of multiple parts. Instances of this class can be obtained by several methods in *Parts*.

**name**

Internal name of the part, as used in [part cfg files](#). For example “Mark1-2Pod”.

**Attribute** Read-only, cannot be set

**Return type** `str`

**title**

Title of the part, as shown when the part is right clicked in-game. For example “Mk1-2 Command Pod”.

**Attribute** Read-only, cannot be set

**Return type** `str`

**tag**

The name tag for the part. Can be set to a custom string using the in-game user interface.

**Attribute** Can be read or written

**Return type** `str`

---

**Note:** This requires either the [NameTag](#) or [kOS](#) mod to be installed.

---

**highlighted**

Whether the part is highlighted.

**Attribute** Can be read or written

**Return type** bool

**highlight\_color**

The color used to highlight the part, as an RGB triple.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

**cost**

The cost of the part, in units of funds.

**Attribute** Read-only, cannot be set

**Return type** float

**vessel**

The vessel that contains this part.

**Attribute** Read-only, cannot be set

**Return type** *Vessel*

**parent**

The parts parent. Returns *None* if the part does not have a parent. This, in combination with *Part.children*, can be used to traverse the vessels parts tree.

**Attribute** Read-only, cannot be set

**Return type** *Part*

---

**Note:** See the discussion on *Trees of Parts*.

---

**children**

The parts children. Returns an empty list if the part has no children. This, in combination with *Part.parent*, can be used to traverse the vessels parts tree.

**Attribute** Read-only, cannot be set

**Return type** list(*Part*)

---

**Note:** See the discussion on *Trees of Parts*.

---

**axially\_attached**

Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns *False*.

**Attribute** Read-only, cannot be set

**Return type** bool

---

**Note:** See the discussion on *Attachment Modes*.

---

**radially\_attached**

Whether the part is radially attached to its parent, i.e. on the side of its parent. If the part has no parent, returns `False`.

**Attribute** Read-only, cannot be set

**Return type** `bool`

---

**Note:** See the discussion on *Attachment Modes*.

---

**stage**

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

**Attribute** Read-only, cannot be set

**Return type** `int`

---

**Note:** See the discussion on *Staging*.

---

**decouple\_stage**

The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

**Attribute** Read-only, cannot be set

**Return type** `int`

---

**Note:** See the discussion on *Staging*.

---

**massless**

Whether the part is `massless`.

**Attribute** Read-only, cannot be set

**Return type** `bool`

**mass**

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

**Attribute** Read-only, cannot be set

**Return type** `float`

**dry\_mass**

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

**Attribute** Read-only, cannot be set

**Return type** `float`

**shielded**

Whether the part is shielded from the exterior of the vessel, for example by a fairing.

**Attribute** Read-only, cannot be set

**Return type** `bool`

**dynamic\_pressure**

The dynamic pressure acting on the part, in Pascals.



**Attribute** Read-only, cannot be set

**Return type** float

**impact\_tolerance**

The impact tolerance of the part, in meters per second.

**Attribute** Read-only, cannot be set

**Return type** float

**temperature**

Temperature of the part, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**skin\_temperature**

Temperature of the skin of the part, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**max\_temperature**

Maximum temperature that the part can survive, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**max\_skin\_temperature**

Maximum temperature that the skin of the part can survive, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**thermal\_mass**

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**thermal\_skin\_mass**

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**thermal\_resource\_mass**

A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**thermal\_conduction\_flux**

The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Attribute** Read-only, cannot be set

**Return type** float

**thermal\_convection\_flux**

The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Attribute** Read-only, cannot be set

**Return type** float

**thermal\_radiation\_flux**

The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Attribute** Read-only, cannot be set

**Return type** float

**thermal\_internal\_flux**

The rate at which heat energy is begin generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

**Attribute** Read-only, cannot be set

**Return type** float

**thermal\_skin\_to\_internal\_flux**

The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

**Attribute** Read-only, cannot be set

**Return type** float

**resources**

A *Resources* object for the part.

**Attribute** Read-only, cannot be set

**Return type** *Resources*

**crossfeed**

Whether this part is crossfeed capable.

**Attribute** Read-only, cannot be set

**Return type** bool

**is\_fuel\_line**

Whether this part is a fuel line.

**Attribute** Read-only, cannot be set

**Return type** bool

**fuel\_lines\_from**

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

**Attribute** Read-only, cannot be set

**Return type** list(*Part*)

---

**Note:** See the discussion on *Fuel Lines*.

---

#### **fuel\_lines\_to**

The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

**Attribute** Read-only, cannot be set

**Return type** list(*Part*)

---

**Note:** See the discussion on *Fuel Lines*.

---

#### **modules**

The modules for this part.

**Attribute** Read-only, cannot be set

**Return type** list(*Module*)

#### **antenna**

A *Antenna* if the part is an antenna, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Antenna*

#### **cargo\_bay**

A *CargoBay* if the part is a cargo bay, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *CargoBay*

#### **control\_surface**

A *ControlSurface* if the part is an aerodynamic control surface, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *ControlSurface*

#### **decoupler**

A *Decoupler* if the part is a decoupler, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Decoupler*

#### **docking\_port**

A *DockingPort* if the part is a docking port, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *DockingPort*

#### **engine**

An *Engine* if the part is an engine, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Engine*

**experiment**

An *Experiment* if the part is a science experiment, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Experiment*

**fairing**

A *Fairing* if the part is a fairing, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Fairing*

**intake**

An *Intake* if the part is an intake, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Intake*

---

**Note:** This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters and jet engines. For RCS thrusters see *RCS*.

---

**leg**

A *Leg* if the part is a landing leg, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Leg*

**launch\_clamp**

A *LaunchClamp* if the part is a launch clamp, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *LaunchClamp*

**light**

A *Light* if the part is a light, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Light*

**parachute**

A *Parachute* if the part is a parachute, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Parachute*

**radiator**

A *Radiator* if the part is a radiator, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *Radiator*

**rcs**

A *RCS* if the part is an RCS block/thruster, otherwise *None*.

**Attribute** Read-only, cannot be set

**Return type** *RCS*

**reaction\_wheel**

A *ReactionWheel* if the part is a reaction wheel, otherwise None.

**Attribute** Read-only, cannot be set

**Return type** *ReactionWheel*

**resource\_converter**

A *ResourceConverter* if the part is a resource converter, otherwise None.

**Attribute** Read-only, cannot be set

**Return type** *ResourceConverter*

**resource\_harvester**

A *ResourceHarvester* if the part is a resource harvester, otherwise None.

**Attribute** Read-only, cannot be set

**Return type** *ResourceHarvester*

**sensor**

A *Sensor* if the part is a sensor, otherwise None.

**Attribute** Read-only, cannot be set

**Return type** *Sensor*

**solar\_panel**

A *SolarPanel* if the part is a solar panel, otherwise None.

**Attribute** Read-only, cannot be set

**Return type** *SolarPanel*

**wheel**

A *Wheel* if the part is a wheel, otherwise None.

**Attribute** Read-only, cannot be set

**Return type** *Wheel*

**position** (*reference\_frame*)

The position of the part in the given reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** tuple(float, float, float)

---

**Note:** This is a fixed position in the part, defined by the parts model. It is not necessarily the same as the parts center of mass. Use *Part.center\_of\_mass()* to get the parts center of mass.

---

**center\_of\_mass** (*reference\_frame*)

The position of the parts center of mass in the given reference frame. If the part is physicsless, this is equivalent to *Part.position()*.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** tuple(float, float, float)

**bounding\_box** (*reference\_frame*)

The axis-aligned bounding box of the part in the given reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned position vectors are in.

**Returns** The positions of the minimum and maximum vertices of the box, as position vectors.

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

---

**Note:** This is computed from the collision mesh of the part. If the part is not collidable, the box has zero volume and is centered on the *Part.position()* of the part.

---

**direction** (*reference\_frame*)

The direction the part points in, in the given reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

**velocity** (*reference\_frame*)

The linear velocity of the part in the given reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned velocity vector is in.

**Returns** The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

**Return type** tuple(float, float, float)

**rotation** (*reference\_frame*)

The rotation of the part, in the given reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** tuple(float, float, float, float)

**moment\_of\_inertia**

The moment of inertia of the part in  $kg.m^2$  around its center of mass in the parts reference frame (*ReferenceFrame*).

**Attribute** Read-only, cannot be set

**Return type** tuple(float, float, float)

**inertia\_tensor**

The inertia tensor of the part in the parts reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

**Attribute** Read-only, cannot be set

**Return type** list(float)

**reference\_frame**

The reference frame that is fixed relative to this part, and centered on a fixed position within the part, defined by the parts model.

- The origin is at the position of the part, as returned by `Part.position()`.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

---

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by `DockingPort.reference_frame`.

---

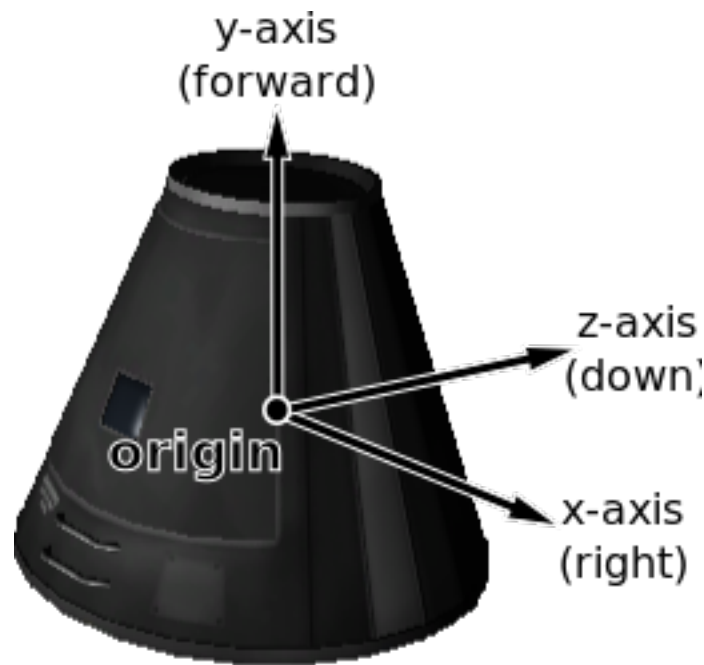


Fig. 7.7: Mk1 Command Pod reference frame origin and axes

#### **center\_of\_mass\_reference\_frame**

The reference frame that is fixed relative to this part, and centered on its center of mass.

- The origin is at the center of mass of the part, as returned by `Part.center_of_mass()`.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

---

**Note:** For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by `DockingPort.reference_frame`.

---

#### **add\_force** (*force, position, reference\_frame*)

Exert a constant force on the part, acting at the given position.

**Parameters**

- **force** (*tuple*) – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** (*tuple*) – The position at which the force acts, as a vector.
- **reference\_frame** (*ReferenceFrame*) – The reference frame that the force and position are in.

**Returns** An object that can be used to remove or modify the force.

**Return type** *Force*

**instantaneous\_force** (*force, position, reference\_frame*)

Exert an instantaneous force on the part, acting at the given position.

**Parameters**

- **force** (*tuple*) – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** (*tuple*) – The position at which the force acts, as a vector.
- **reference\_frame** (*ReferenceFrame*) – The reference frame that the force and position are in.

---

**Note:** The force is applied instantaneously in a single physics update.

---

**class Force**

Obtained by calling *Part.add\_force()*.

**part**

The part that this force is applied to.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**force\_vector**

The force vector, in Newtons.

**Attribute** Can be read or written

**Returns** A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

**Return type** *tuple(float, float, float)*

**position**

The position at which the force acts, in reference frame *ReferenceFrame*.

**Attribute** Can be read or written

**Returns** The position as a vector.

**Return type** *tuple(float, float, float)*

**reference\_frame**

The reference frame of the force vector and position.

**Attribute** Can be read or written

**Return type** *ReferenceFrame*



**remove()**  
Remove the force.

## Module

### class Module

This can be used to interact with a specific part module. This includes part modules in stock KSP, and those added by mods.

In KSP, each part has zero or more [PartModules](#) associated with it. Each one contains some of the functionality of the part. For example, an engine has a “ModuleEngines” part module that contains all the functionality of an engine.

**name**  
Name of the PartModule. For example, “ModuleEngines”.

**Attribute** Read-only, cannot be set

**Return type** str

**part**  
The part that contains this module.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**fields**  
The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

**Attribute** Read-only, cannot be set

**Return type** dict(str, str)

**has\_field(name)**  
Returns `True` if the module has a field with the given name.

**Parameters** **name** (*str*) – Name of the field.

**Return type** bool

**get\_field(name)**  
Returns the value of a field.

**Parameters** **name** (*str*) – Name of the field.

**Return type** str

**set\_field\_int(name, value)**  
Set the value of a field to the given integer number.

**Parameters**

- **name** (*str*) – Name of the field.
- **value** (*int*) – Value to set.

**set\_field\_float(name, value)**  
Set the value of a field to the given floating point number.

**Parameters**

- **name** (*str*) – Name of the field.

- **value** (*float*) – Value to set.

**set\_field\_string** (*name*, *value*)

Set the value of a field to the given string.

**Parameters**

- **name** (*str*) – Name of the field.
- **value** (*str*) – Value to set.

**reset\_field** (*name*)

Set the value of a field to its original value.

**Parameters** **name** (*str*) – Name of the field.

**events**

A list of the names of all of the modules events. Events are the clickable buttons visible in the right-click menu of the part.

**Attribute** Read-only, cannot be set

**Return type** list(str)

**has\_event** (*name*)

True if the module has an event with the given name.

**Parameters** **name** (*str*) –

**Return type** bool

**trigger\_event** (*name*)

Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

**Parameters** **name** (*str*) –

**actions**

A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

**Attribute** Read-only, cannot be set

**Return type** list(str)

**has\_action** (*name*)

True if the part has an action with the given name.

**Parameters** **name** (*str*) –

**Return type** bool

**set\_action** (*name*[, *value* = *True*])

Set the value of an action with the given name.

**Parameters**

- **name** (*str*) –
- **value** (*bool*) –

## Specific Types of Part

The following classes provide functionality for specific types of part.

- *Antenna*
- *Cargo Bay*
- *Control Surface*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Experiment*
- *Fairing*
- *Intake*
- *Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *RCS*
- *Sensor*
- *Solar Panel*
- *Thruster*
- *Wheel*

## Antenna

### **class** *Antenna*

An antenna. Obtained by calling *Part.antenna*.

#### **part**

The part object for this antenna.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### **state**

The current state of the antenna.

**Attribute** Read-only, cannot be set

**Return type** *AntennaState*

#### **deployable**

Whether the antenna is deployable.

**Attribute** Read-only, cannot be set

**Return type** bool

**deployed**

Whether the antenna is deployed.

**Attribute** Can be read or written

**Return type** bool

---

**Note:** Fixed antennas are always deployed. Returns an error if you try to deploy a fixed antenna.

---

**can\_transmit**

Whether data can be transmitted by this antenna.

**Attribute** Read-only, cannot be set

**Return type** bool

**transmit ()**

Transmit data.

**cancel ()**

Cancel current transmission of data.

**allow\_partial**

Whether partial data transmission is permitted.

**Attribute** Can be read or written

**Return type** bool

**power**

The power of the antenna.

**Attribute** Read-only, cannot be set

**Return type** float

**combinable**

Whether the antenna can be combined with other antennae on the vessel to boost the power.

**Attribute** Read-only, cannot be set

**Return type** bool

**combinable\_exponent**

Exponent used to calculate the combined power of multiple antennae on a vessel.

**Attribute** Read-only, cannot be set

**Return type** float

**packet\_interval**

Interval between sending packets in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**packet\_size**

Amount of data sent per packet in Mits.

**Attribute** Read-only, cannot be set

**Return type** float

**packet\_resource\_cost**

Units of electric charge consumed per packet sent.

**Attribute** Read-only, cannot be set

**Return type** float

**class AntennaState**

The state of an antenna. See *Antenna.state*.

**deployed**

Antenna is fully deployed.

**retracted**

Antenna is fully retracted.

**deploying**

Antenna is being deployed.

**retracting**

Antenna is being retracted.

**broken**

Antenna is broken.

## Cargo Bay

**class CargoBay**

A cargo bay. Obtained by calling *Part.cargo\_bay*.

**part**

The part object for this cargo bay.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**state**

The state of the cargo bay.

**Attribute** Read-only, cannot be set

**Return type** *CargoBayState*

**open**

Whether the cargo bay is open.

**Attribute** Can be read or written

**Return type** bool

**class CargoBayState**

The state of a cargo bay. See *CargoBay.state*.

**open**

Cargo bay is fully open.

**closed**

Cargo bay closed and locked.

**opening**

Cargo bay is opening.

**closing**

Cargo bay is closing.

**Control Surface****class ControlSurface**

An aerodynamic control surface. Obtained by calling *Part.control\_surface*.

**part**

The part object for this control surface.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**pitch\_enabled**

Whether the control surface has pitch control enabled.

**Attribute** Can be read or written

**Return type** bool

**yaw\_enabled**

Whether the control surface has yaw control enabled.

**Attribute** Can be read or written

**Return type** bool

**roll\_enabled**

Whether the control surface has roll control enabled.

**Attribute** Can be read or written

**Return type** bool

**authority\_limiter**

The authority limiter for the control surface, which controls how far the control surface will move.

**Attribute** Can be read or written

**Return type** float

**inverted**

Whether the control surface movement is inverted.

**Attribute** Can be read or written

**Return type** bool

**deployed**

Whether the control surface has been fully deployed.

**Attribute** Can be read or written

**Return type** bool

**surface\_area**

Surface area of the control surface in  $m^2$ .

**Attribute** Read-only, cannot be set

**Return type** float

**available\_torque**

The available torque, in Newton meters, that can be produced by this control surface, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.reference\_frame*.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**Decoupler****class Decoupler**

A decoupler. Obtained by calling *Part.decoupler*

**part**

The part object for this decoupler.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**decouple ()**

Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

**Return type** *Vessel*

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *active\_vessel* no longer refer to the active vessel.

---

**decoupled**

Whether the decoupler has fired.

**Attribute** Read-only, cannot be set

**Return type** bool

**staged**

Whether the decoupler is enabled in the staging sequence.

**Attribute** Read-only, cannot be set

**Return type** bool

**impulse**

The impulse that the decoupler imparts when it is fired, in Newton seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**Docking Port****class DockingPort**

A docking port. Obtained by calling *Part.docking\_port*

**part**

The part object for this docking port.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**state**

The current state of the docking port.

**Attribute** Read-only, cannot be set

**Return type** *DockingPortState*

**docked\_part**

The part that this docking port is docked to. Returns *None* if this docking port is not docked to anything.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**undock()**

Undocks the docking port and returns the new *Vessel* that is created. This method can be called for either docking port in a docked pair. Throws an exception if the docking port is not docked to anything.

**Return type** *Vessel*

---

**Note:** When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *active\_vessel* no longer refer to the active vessel.

---

**reengage\_distance**

The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

**Attribute** Read-only, cannot be set

**Return type** *float*

**has\_shield**

Whether the docking port has a shield.

**Attribute** Read-only, cannot be set

**Return type** *bool*

**shielded**

The state of the docking ports shield, if it has one.

Returns *True* if the docking port has a shield, and the shield is closed. Otherwise returns *False*. When set to *True*, the shield is closed, and when set to *False* the shield is opened. If the docking port does not have a shield, setting this attribute has no effect.

**Attribute** Can be read or written

**Return type** *bool*

**position(reference\_frame)**

The position of the docking port, in the given reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** *tuple(float, float, float)*



**direction** (*reference\_frame*)

The direction that docking port points in, in the given reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

**rotation** (*reference\_frame*)

The rotation of the docking port, in the given reference frame.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned rotation is in.

**Returns** The rotation as a quaternion of the form  $(x, y, z, w)$ .

**Return type** tuple(float, float, float, float)

**reference\_frame**

The reference frame that is fixed relative to this docking port, and oriented with the port.

- The origin is at the position of the docking port.
- The axes rotate with the docking port.
- The x-axis points out to the right side of the docking port.
- The y-axis points in the direction the docking port is facing.
- The z-axis points out of the bottom off the docking port.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

---

**Note:** This reference frame is not necessarily equivalent to the reference frame for the part, returned by *Part.reference\_frame*.

---

**class DockingPortState**

The state of a docking port. See *DockingPort.state*.

**ready**

The docking port is ready to dock to another docking port.

**docked**

The docking port is docked to another docking port, or docked to another part (from the VAB/SPH).

**docking**

The docking port is very close to another docking port, but has not docked. It is using magnetic force to acquire a solid dock.

**undocking**

The docking port has just been undocked from another docking port, and is disabled until it moves away by a sufficient distance (*DockingPort.reengage\_distance*).

**shielded**

The docking port has a shield, and the shield is closed.

**moving**

The docking ports shield is currently opening/closing.

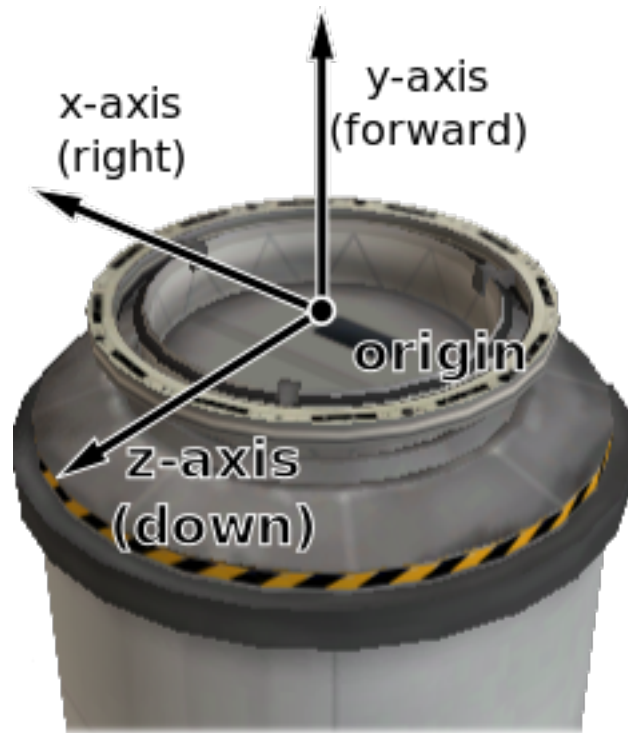


Fig. 7.8: Docking port reference frame origin and axes

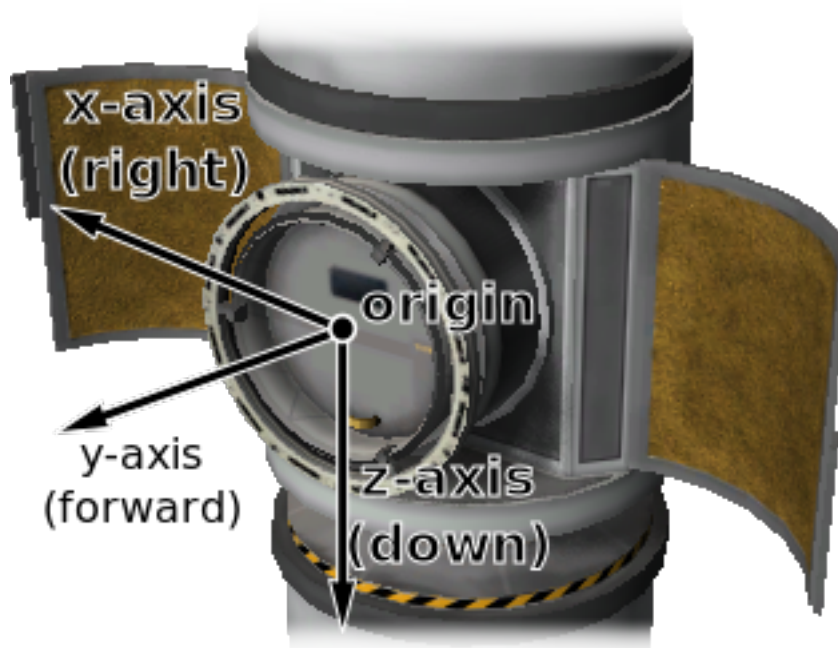


Fig. 7.9: Inline docking port reference frame origin and axes

## Engine

### class Engine

An engine, including ones of various types. For example liquid fuelled gimballed engines, solid rocket boosters and jet engines. Obtained by calling *Part.engine*.

---

**Note:** For RCS thrusters *Part.rcs*.

---

#### part

The part object for this engine.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### active

Whether the engine is active. Setting this attribute may have no effect, depending on *Engine.can\_shutdown* and *Engine.can\_restart*.

**Attribute** Can be read or written

**Return type** bool

#### thrust

The current amount of thrust being produced by the engine, in Newtons.

**Attribute** Read-only, cannot be set

**Return type** float

#### available\_thrust

The amount of thrust, in Newtons, that would be produced by the engine when activated and with its throttle set to 100%. Returns zero if the engine does not have any fuel. Takes the engine's current *Engine.thrust\_limit* and atmospheric conditions into account.

**Attribute** Read-only, cannot be set

**Return type** float

#### max\_thrust

The amount of thrust, in Newtons, that would be produced by the engine when activated and fueled, with its throttle and throttle limiter set to 100%.

**Attribute** Read-only, cannot be set

**Return type** float

#### max\_vacuum\_thrust

The maximum amount of thrust that can be produced by the engine in a vacuum, in Newtons. This is the amount of thrust produced by the engine when activated, *Engine.thrust\_limit* is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

**Attribute** Read-only, cannot be set

**Return type** float

#### thrust\_limit

The thrust limiter of the engine. A value between 0 and 1. Setting this attribute may have no effect, for example the thrust limit for a solid rocket booster cannot be changed in flight.

**Attribute** Can be read or written

**Return type** float

**thrusters**

The components of the engine that generate thrust.

**Attribute** Read-only, cannot be set

**Return type** list(*Thruster*)

---

**Note:** For example, this corresponds to the rocket nozzle on a solid rocket booster, or the individual nozzles on a RAPIER engine. The overall thrust produced by the engine, as reported by *Engine.available\_thrust*, *Engine.max\_thrust* and others, is the sum of the thrust generated by each thruster.

---

**specific\_impulse**

The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

**Attribute** Read-only, cannot be set

**Return type** float

**vacuum\_specific\_impulse**

The vacuum specific impulse of the engine, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**kerbin\_sea\_level\_specific\_impulse**

The specific impulse of the engine at sea level on Kerbin, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**propellant\_names**

The names of the propellants that the engine consumes.

**Attribute** Read-only, cannot be set

**Return type** list(str)

**propellant\_ratios**

The ratio of resources that the engine consumes. A dictionary mapping resource names to the ratio at which they are consumed by the engine.

**Attribute** Read-only, cannot be set

**Return type** dict(str, float)

---

**Note:** For example, if the ratios are 0.6 for LiquidFuel and 0.4 for Oxidizer, then for every 0.6 units of LiquidFuel that the engine burns, it will burn 0.4 units of Oxidizer.

---

**propellants**

The propellants that the engine consumes.

**Attribute** Read-only, cannot be set

**Return type** list(*Propellant*)

**has\_fuel**

Whether the engine has any fuel available.

**Attribute** Read-only, cannot be set

**Return type** bool

---

**Note:** The engine must be activated for this property to update correctly.

---

#### **throttle**

The current throttle setting for the engine. A value between 0 and 1. This is not necessarily the same as the vessel's main throttle setting, as some engines take time to adjust their throttle (such as jet engines).

**Attribute** Read-only, cannot be set

**Return type** float

#### **throttle\_locked**

Whether the `Control.throttle` affects the engine. For example, this is `True` for liquid fueled rockets, and `False` for solid rocket boosters.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **can\_restart**

Whether the engine can be restarted once shutdown. If the engine cannot be shutdown, returns `False`. For example, this is `True` for liquid fueled rockets and `False` for solid rocket boosters.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **can\_shutdown**

Whether the engine can be shutdown once activated. For example, this is `True` for liquid fueled rockets and `False` for solid rocket boosters.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **has\_modes**

Whether the engine has multiple modes of operation.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **mode**

The name of the current engine mode.

**Attribute** Can be read or written

**Return type** str

#### **modes**

The available modes for the engine. A dictionary mapping mode names to *Engine* objects.

**Attribute** Read-only, cannot be set

**Return type** dict(str, *Engine*)

#### **toggle\_mode()**

Toggle the current engine mode.

#### **auto\_mode\_switch**

Whether the engine will automatically switch modes.

**Attribute** Can be read or written

**Return type** bool

**gimballed**

Whether the engine is gimballed.

**Attribute** Read-only, cannot be set

**Return type** bool

**gimbal\_range**

The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

**Attribute** Read-only, cannot be set

**Return type** float

**gimbal\_locked**

Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

**Attribute** Can be read or written

**Return type** bool

**gimbal\_limit**

The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

**Attribute** Can be read or written

**Return type** float

**available\_torque**

The available torque, in Newton meters, that can be produced by this engine, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.reference\_frame*. Returns zero if the engine is inactive, or not gimballed.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**class Propellant**

A propellant for an engine. Obtains by calling *Engine.propellants*.

**name**

The name of the propellant.

**Attribute** Read-only, cannot be set

**Return type** str

**current\_amount**

The current amount of propellant.

**Attribute** Read-only, cannot be set

**Return type** float

**current\_requirement**

The required amount of propellant.

**Attribute** Read-only, cannot be set

**Return type** float

**total\_resource\_available**

The total amount of the underlying resource currently reachable given resource flow rules.

**Attribute** Read-only, cannot be set

**Return type** float

#### **total\_resource\_capacity**

The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

**Attribute** Read-only, cannot be set

**Return type** float

#### **ignore\_for\_isp**

If this propellant should be ignored when calculating required mass flow given specific impulse.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **ignore\_for\_thrust\_curve**

If this propellant should be ignored for thrust curve calculations.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **draw\_stack\_gauge**

If this propellant has a stack gauge or not.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **is\_deprived**

If this propellant is deprived.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **ratio**

The propellant ratio.

**Attribute** Read-only, cannot be set

**Return type** float

## Experiment

### **class Experiment**

Obtained by calling *Part.experiment*.

#### **part**

The part object for this experiment.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### **run()**

Run the experiment.

#### **transmit()**

Transmit all experimental data contained by this part.

**dump()**

Dump the experimental data contained by the experiment.

**reset()**

Reset the experiment.

**deployed**

Whether the experiment has been deployed.

**Attribute** Read-only, cannot be set

**Return type** bool

**rerunnable**

Whether the experiment can be re-run.

**Attribute** Read-only, cannot be set

**Return type** bool

**inoperable**

Whether the experiment is inoperable.

**Attribute** Read-only, cannot be set

**Return type** bool

**has\_data**

Whether the experiment contains data.

**Attribute** Read-only, cannot be set

**Return type** bool

**data**

The data contained in this experiment.

**Attribute** Read-only, cannot be set

**Return type** list(*ScienceData*)

**biome**

The name of the biome the experiment is currently in.

**Attribute** Read-only, cannot be set

**Return type** str

**available**

Determines if the experiment is available given the current conditions.

**Attribute** Read-only, cannot be set

**Return type** bool

**science\_subject**

Containing information on the corresponding specific science result for the current conditions. Returns *None* if the experiment is unavailable.

**Attribute** Read-only, cannot be set

**Return type** *ScienceSubject*

**class ScienceData**

Obtained by calling *Experiment.data*.



**data\_amount**

Data amount.

**Attribute** Read-only, cannot be set**Return type** float**science\_value**

Science value.

**Attribute** Read-only, cannot be set**Return type** float**transmit\_value**

Transmit value.

**Attribute** Read-only, cannot be set**Return type** float**class ScienceSubject**Obtained by calling *Experiment.science\_subject*.**title**

Title of science subject, displayed in science archives

**Attribute** Read-only, cannot be set**Return type** str**is\_complete**

Whether the experiment has been completed.

**Attribute** Read-only, cannot be set**Return type** bool**science**

Amount of science already earned from this subject, not updated until after transmission/recovery.

**Attribute** Read-only, cannot be set**Return type** float**science\_cap**

Total science allowable for this subject.

**Attribute** Read-only, cannot be set**Return type** float**data\_scale**

Multiply science value by this to determine data amount in mits.

**Attribute** Read-only, cannot be set**Return type** float**subject\_value**

Multiplier for specific Celestial Body/Experiment Situation combination.

**Attribute** Read-only, cannot be set**Return type** float**scientific\_value**

Diminishing value multiplier for decreasing the science value returned from repeated experiments.

**Attribute** Read-only, cannot be set

**Return type** float

## Fairing

### **class Fairing**

A fairing. Obtained by calling *Part.fairing*.

#### **part**

The part object for this fairing.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### **jettison()**

Jettison the fairing. Has no effect if it has already been jettisoned.

#### **jettisoned**

Whether the fairing has been jettisoned.

**Attribute** Read-only, cannot be set

**Return type** bool

## Intake

### **class Intake**

An air intake. Obtained by calling *Part.intake*.

#### **part**

The part object for this intake.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### **open**

Whether the intake is open.

**Attribute** Can be read or written

**Return type** bool

#### **speed**

Speed of the flow into the intake, in *m/s*.

**Attribute** Read-only, cannot be set

**Return type** float

#### **flow**

The rate of flow into the intake, in units of resource per second.

**Attribute** Read-only, cannot be set

**Return type** float

#### **area**

The area of the intake's opening, in square meters.

**Attribute** Read-only, cannot be set

**Return type** float

## Leg

### class Leg

A landing leg. Obtained by calling *Part.leg*.

#### part

The part object for this landing leg.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### state

The current state of the landing leg.

**Attribute** Read-only, cannot be set

**Return type** *LegState*

#### deployable

Whether the leg is deployable.

**Attribute** Read-only, cannot be set

**Return type** bool

#### deployed

Whether the landing leg is deployed.

**Attribute** Can be read or written

**Return type** bool

---

**Note:** Fixed landing legs are always deployed. Returns an error if you try to deploy fixed landing gear.

---

#### is\_grounded

Returns whether the leg is touching the ground.

**Attribute** Read-only, cannot be set

**Return type** bool

### class LegState

The state of a landing leg. See *Leg.state*.

#### deployed

Landing leg is fully deployed.

#### retracted

Landing leg is fully retracted.

#### deploying

Landing leg is being deployed.

#### retracting

Landing leg is being retracted.

#### broken

Landing leg is broken.

## Launch Clamp

### **class LaunchClamp**

A launch clamp. Obtained by calling *Part.launch\_clamp*.

#### **part**

The part object for this launch clamp.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### **release()**

Releases the docking clamp. Has no effect if the clamp has already been released.

## Light

### **class Light**

A light. Obtained by calling *Part.light*.

#### **part**

The part object for this light.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### **active**

Whether the light is switched on.

**Attribute** Can be read or written

**Return type** bool

#### **color**

The color of the light, as an RGB triple.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

#### **power\_usage**

The current power usage, in units of charge per second.

**Attribute** Read-only, cannot be set

**Return type** float

## Parachute

### **class Parachute**

A parachute. Obtained by calling *Part.parachute*.

#### **part**

The part object for this parachute.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### **deploy()**

Deploys the parachute. This has no effect if the parachute has already been deployed.

**deployed**

Whether the parachute has been deployed.

**Attribute** Read-only, cannot be set

**Return type** bool

**arm()**

Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

**armed**

Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

**Attribute** Read-only, cannot be set

**Return type** bool

**state**

The current state of the parachute.

**Attribute** Read-only, cannot be set

**Return type** *ParachuteState*

**deploy\_altitude**

The altitude at which the parachute will full deploy, in meters. Only applicable to stock parachutes.

**Attribute** Can be read or written

**Return type** float

**deploy\_min\_pressure**

The minimum pressure at which the parachute will semi-deploy, in atmospheres. Only applicable to stock parachutes.

**Attribute** Can be read or written

**Return type** float

**class ParachuteState**

The state of a parachute. See *Parachute.state*.

**stowed**

The parachute is safely tucked away inside its housing.

**armed**

The parachute is armed for deployment. (RealChutes only)

**active**

The parachute is still stowed, but ready to semi-deploy. (Stock parachutes only)

**semi\_deployed**

The parachute has been deployed and is providing some drag, but is not fully deployed yet. (Stock parachutes only)

**deployed**

The parachute is fully deployed.

**cut**

The parachute has been cut.

## Radiator

### **class Radiator**

A radiator. Obtained by calling *Part.radiator*.

#### **part**

The part object for this radiator.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### **deployable**

Whether the radiator is deployable.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **deployed**

For a deployable radiator, True if the radiator is extended. If the radiator is not deployable, this is always True.

**Attribute** Can be read or written

**Return type** bool

#### **state**

The current state of the radiator.

**Attribute** Read-only, cannot be set

**Return type** *RadiatorState*

---

**Note:** A fixed radiator is always *RadiatorState.extended*.

---

### **class RadiatorState**

The state of a radiator. *RadiatorState*

#### **extended**

Radiator is fully extended.

#### **retracted**

Radiator is fully retracted.

#### **extending**

Radiator is being extended.

#### **retracting**

Radiator is being retracted.

#### **broken**

Radiator is being broken.

## Resource Converter

### **class ResourceConverter**

A resource converter. Obtained by calling *Part.resource\_converter*.

#### **part**

The part object for this converter.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**count**

The number of converters in the part.

**Attribute** Read-only, cannot be set

**Return type** *int*

**name** (*index*)

The name of the specified converter.

**Parameters** **index** (*int*) – Index of the converter.

**Return type** *str*

**active** (*index*)

True if the specified converter is active.

**Parameters** **index** (*int*) – Index of the converter.

**Return type** *bool*

**start** (*index*)

Start the specified converter.

**Parameters** **index** (*int*) – Index of the converter.

**stop** (*index*)

Stop the specified converter.

**Parameters** **index** (*int*) – Index of the converter.

**state** (*index*)

The state of the specified converter.

**Parameters** **index** (*int*) – Index of the converter.

**Return type** *ResourceConverterState*

**status\_info** (*index*)

Status information for the specified converter. This is the full status message shown in the in-game UI.

**Parameters** **index** (*int*) – Index of the converter.

**Return type** *str*

**inputs** (*index*)

List of the names of resources consumed by the specified converter.

**Parameters** **index** (*int*) – Index of the converter.

**Return type** *list(str)*

**outputs** (*index*)

List of the names of resources produced by the specified converter.

**Parameters** **index** (*int*) – Index of the converter.

**Return type** *list(str)*

**class ResourceConverterState**

The state of a resource converter. See *ResourceConverter.state()*.

**running**

Converter is running.

**idle**  
Converter is idle.

**missing\_resource**  
Converter is missing a required resource.

**storage\_full**  
No available storage for output resource.

**capacity**  
At preset resource capacity.

**unknown**  
Unknown state. Possible with modified resource converters. In this case, check *ResourceConverter.status\_info()* for more information.

## Resource Harvester

**class ResourceHarvester**  
A resource harvester (drill). Obtained by calling *Part.resource\_harvester*.

**part**  
The part object for this harvester.  
**Attribute** Read-only, cannot be set  
**Return type** *Part*

**state**  
The state of the harvester.  
**Attribute** Read-only, cannot be set  
**Return type** *ResourceHarvesterState*

**deployed**  
Whether the harvester is deployed.  
**Attribute** Can be read or written  
**Return type** bool

**active**  
Whether the harvester is actively drilling.  
**Attribute** Can be read or written  
**Return type** bool

**extraction\_rate**  
The rate at which the drill is extracting ore, in units per second.  
**Attribute** Read-only, cannot be set  
**Return type** float

**thermal\_efficiency**  
The thermal efficiency of the drill, as a percentage of its maximum.  
**Attribute** Read-only, cannot be set  
**Return type** float



**core\_temperature**

The core temperature of the drill, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**optimum\_core\_temperature**

The core temperature at which the drill will operate with peak efficiency, in Kelvin.

**Attribute** Read-only, cannot be set

**Return type** float

**class ResourceHarvesterState**

The state of a resource harvester. See *ResourceHarvester.state*.

**deploying**

The drill is deploying.

**deployed**

The drill is deployed and ready.

**retracting**

The drill is retracting.

**retracted**

The drill is retracted.

**active**

The drill is running.

**Reaction Wheel****class ReactionWheel**

A reaction wheel. Obtained by calling *Part.reaction\_wheel*.

**part**

The part object for this reaction wheel.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**active**

Whether the reaction wheel is active.

**Attribute** Can be read or written

**Return type** bool

**broken**

Whether the reaction wheel is broken.

**Attribute** Read-only, cannot be set

**Return type** bool

**available\_torque**

The available torque, in Newton meters, that can be produced by this reaction wheel, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.reference\_frame*. Returns zero if the reaction wheel is inactive or broken.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**max\_torque**

The maximum torque, in Newton meters, that can be produced by this reaction wheel, when it is active, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.reference\_frame*.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

## RCS

**class RCS**

An RCS block or thruster. Obtained by calling *Part.rcs*.

**part**

The part object for this RCS.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**active**

Whether the RCS thrusters are active. An RCS thruster is inactive if the RCS action group is disabled (*Control.rcs*), the RCS thruster itself is not enabled (*RCS.enabled*) or it is covered by a fairing (*Part.shielded*).

**Attribute** Read-only, cannot be set

**Return type** bool

**enabled**

Whether the RCS thrusters are enabled.

**Attribute** Can be read or written

**Return type** bool

**pitch\_enabled**

Whether the RCS thruster will fire when pitch control input is given.

**Attribute** Can be read or written

**Return type** bool

**yaw\_enabled**

Whether the RCS thruster will fire when yaw control input is given.

**Attribute** Can be read or written

**Return type** bool

**roll\_enabled**

Whether the RCS thruster will fire when roll control input is given.

**Attribute** Can be read or written

**Return type** bool

**forward\_enabled**

Whether the RCS thruster will fire when pitch control input is given.

**Attribute** Can be read or written

**Return type** bool

**up\_enabled**

Whether the RCS thruster will fire when yaw control input is given.

**Attribute** Can be read or written

**Return type** bool

**right\_enabled**

Whether the RCS thruster will fire when roll control input is given.

**Attribute** Can be read or written

**Return type** bool

**available\_torque**

The available torque, in Newton meters, that can be produced by this RCS, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.reference\_frame*. Returns zero if RCS is disable.

**Attribute** Read-only, cannot be set

**Return type** tuple(tuple(float, float, float), tuple(float, float, float))

**max\_thrust**

The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

**Attribute** Read-only, cannot be set

**Return type** float

**max\_vacuum\_thrust**

The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

**Attribute** Read-only, cannot be set

**Return type** float

**thrusters**

A list of thrusters, one of each nozzle in the RCS part.

**Attribute** Read-only, cannot be set

**Return type** list(*Thruster*)

**specific\_impulse**

The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

**Attribute** Read-only, cannot be set

**Return type** float

**vacuum\_specific\_impulse**

The vacuum specific impulse of the RCS, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**kerbin\_sea\_level\_specific\_impulse**

The specific impulse of the RCS at sea level on Kerbin, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**propellants**

The names of resources that the RCS consumes.

**Attribute** Read-only, cannot be set

**Return type** list(str)

**propellant\_ratios**

The ratios of resources that the RCS consumes. A dictionary mapping resource names to the ratios at which they are consumed by the RCS.

**Attribute** Read-only, cannot be set

**Return type** dict(str, float)

**has\_fuel**

Whether the RCS has fuel available.

**Attribute** Read-only, cannot be set

**Return type** bool

---

**Note:** The RCS thruster must be activated for this property to update correctly.

---

## Sensor

**class Sensor**

A sensor, such as a thermometer. Obtained by calling *Part.sensor*.

**part**

The part object for this sensor.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**active**

Whether the sensor is active.

**Attribute** Can be read or written

**Return type** bool

**value**

The current value of the sensor.

**Attribute** Read-only, cannot be set

**Return type** str

## Solar Panel

**class SolarPanel**

A solar panel. Obtained by calling *Part.solar\_panel*.

**part**

The part object for this solar panel.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**deployable**

Whether the solar panel is deployable.

**Attribute** Read-only, cannot be set

**Return type** bool

**deployed**

Whether the solar panel is extended.

**Attribute** Can be read or written

**Return type** bool

**state**

The current state of the solar panel.

**Attribute** Read-only, cannot be set

**Return type** *SolarPanelState*

**energy\_flow**

The current amount of energy being generated by the solar panel, in units of charge per second.

**Attribute** Read-only, cannot be set

**Return type** float

**sun\_exposure**

The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

**Attribute** Read-only, cannot be set

**Return type** float

**class SolarPanelState**

The state of a solar panel. See *SolarPanel.state*.

**extended**

Solar panel is fully extended.

**retracted**

Solar panel is fully retracted.

**extending**

Solar panel is being extended.

**retracting**

Solar panel is being retracted.

**broken**

Solar panel is broken.

## Thruster

**class Thruster**

The component of an *Engine* or *RCS* part that generates thrust. Can obtained by calling *Engine.thrusters* or *RCS.thrusters*.

---

**Note:** Engines can consist of multiple thrusters. For example, the S3 KS-25x4 “Mammoth” has four rocket nozzels, and so consists of four thrusters.

---

**part**

The *Part* that contains this thruster.

**Attribute** Read-only, cannot be set

**Return type** *Part*

**thrust\_position** (*reference\_frame*)

The position at which the thruster generates thrust, in the given reference frame. For gimballed engines, this takes into account the current rotation of the gimbal.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** tuple(float, float, float)

**thrust\_direction** (*reference\_frame*)

The direction of the force generated by the thruster, in the given reference frame. This is opposite to the direction in which the thruster expels propellant. For gimballed engines, this takes into account the current rotation of the gimbal.

**Parameters** **reference\_frame** (*ReferenceFrame*) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

**thrust\_reference\_frame**

A reference frame that is fixed relative to the thruster and orientated with its thrust direction (*Thruster.thrust\_direction()*). For gimballed engines, this takes into account the current rotation of the gimbal.

- The origin is at the position of thrust for this thruster (*Thruster.thrust\_position()*).
- The axes rotate with the thrust direction. This is the direction in which the thruster expels propellant, including any gimbaling.
- The y-axis points along the thrust direction.
- The x-axis and z-axis are perpendicular to the thrust direction.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

**gimballed**

Whether the thruster is gimballed.

**Attribute** Read-only, cannot be set

**Return type** bool

**gimbal\_position** (*reference\_frame*)

Position around which the gimbal pivots.

**Parameters** `reference_frame` (`ReferenceFrame`) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** `tuple(float, float, float)`

#### **`gimbal_angle`**

The current gimbal angle in the pitch, roll and yaw axes, in degrees.

**Attribute** Read-only, cannot be set

**Return type** `tuple(float, float, float)`

#### **`initial_thrust_position`** (*reference\_frame*)

The position at which the thruster generates thrust, when the engine is in its initial position (no gimbaling), in the given reference frame.

**Parameters** `reference_frame` (`ReferenceFrame`) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** `tuple(float, float, float)`

---

**Note:** This position can move when the gimbal rotates. This is because the thrust position and gimbal position are not necessarily the same.

---

#### **`initial_thrust_direction`** (*reference\_frame*)

The direction of the force generated by the thruster, when the engine is in its initial position (no gimbaling), in the given reference frame. This is opposite to the direction in which the thruster expels propellant.

**Parameters** `reference_frame` (`ReferenceFrame`) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** `tuple(float, float, float)`

## Wheel

### **`class Wheel`**

A wheel. Includes landing gear and rover wheels. Obtained by calling `Part.wheel`. Can be used to control the motors, steering and deployment of wheels, among other things.

#### **`part`**

The part object for this wheel.

**Attribute** Read-only, cannot be set

**Return type** `Part`

#### **`state`**

The current state of the wheel.

**Attribute** Read-only, cannot be set

**Return type** `WheelState`

**radius**

Radius of the wheel, in meters.

**Attribute** Read-only, cannot be set

**Return type** float

**grounded**

Whether the wheel is touching the ground.

**Attribute** Read-only, cannot be set

**Return type** bool

**has\_brakes**

Whether the wheel has brakes.

**Attribute** Read-only, cannot be set

**Return type** bool

**brakes**

The braking force, as a percentage of maximum, when the brakes are applied.

**Attribute** Can be read or written

**Return type** float

**auto\_friction\_control**

Whether automatic friction control is enabled.

**Attribute** Can be read or written

**Return type** bool

**manual\_friction\_control**

Manual friction control value. Only has an effect if automatic friction control is disabled. A value between 0 and 5 inclusive.

**Attribute** Can be read or written

**Return type** float

**deployable**

Whether the wheel is deployable.

**Attribute** Read-only, cannot be set

**Return type** bool

**deployed**

Whether the wheel is deployed.

**Attribute** Can be read or written

**Return type** bool

**powered**

Whether the wheel is powered by a motor.

**Attribute** Read-only, cannot be set

**Return type** bool

**motor\_enabled**

Whether the motor is enabled.

**Attribute** Can be read or written



**Return type** bool

**motor\_inverted**

Whether the direction of the motor is inverted.

**Attribute** Can be read or written

**Return type** bool

**motor\_state**

Whether the direction of the motor is inverted.

**Attribute** Read-only, cannot be set

**Return type** *MotorState*

**motor\_output**

The output of the motor. This is the torque currently being generated, in Newton meters.

**Attribute** Read-only, cannot be set

**Return type** float

**traction\_control\_enabled**

Whether automatic traction control is enabled. A wheel only has traction control if it is powered.

**Attribute** Can be read or written

**Return type** bool

**traction\_control**

Setting for the traction control. Only takes effect if the wheel has automatic traction control enabled. A value between 0 and 5 inclusive.

**Attribute** Can be read or written

**Return type** float

**drive\_limiter**

Manual setting for the motor limiter. Only takes effect if the wheel has automatic traction control disabled. A value between 0 and 100 inclusive.

**Attribute** Can be read or written

**Return type** float

**steerable**

Whether the wheel has steering.

**Attribute** Read-only, cannot be set

**Return type** bool

**steering\_enabled**

Whether the wheel steering is enabled.

**Attribute** Can be read or written

**Return type** bool

**steering\_inverted**

Whether the wheel steering is inverted.

**Attribute** Can be read or written

**Return type** bool

**has\_suspension**

Whether the wheel has suspension.

**Attribute** Read-only, cannot be set

**Return type** bool

**suspension\_spring\_strength**

Suspension spring strength, as set in the editor.

**Attribute** Read-only, cannot be set

**Return type** float

**suspension\_damper\_strength**

Suspension damper strength, as set in the editor.

**Attribute** Read-only, cannot be set

**Return type** float

**broken**

Whether the wheel is broken.

**Attribute** Read-only, cannot be set

**Return type** bool

**repairable**

Whether the wheel is repairable.

**Attribute** Read-only, cannot be set

**Return type** bool

**stress**

Current stress on the wheel.

**Attribute** Read-only, cannot be set

**Return type** float

**stress\_tolerance**

Stress tolerance of the wheel.

**Attribute** Read-only, cannot be set

**Return type** float

**stress\_percentage**

Current stress on the wheel as a percentage of its stress tolerance.

**Attribute** Read-only, cannot be set

**Return type** float

**deflection**

Current deflection of the wheel.

**Attribute** Read-only, cannot be set

**Return type** float

**slip**

Current slip of the wheel.

**Attribute** Read-only, cannot be set

**Return type** float

#### **class WheelState**

The state of a wheel. See *Wheel.state*.

##### **deployed**

Wheel is fully deployed.

##### **retracted**

Wheel is fully retracted.

##### **deploying**

Wheel is being deployed.

##### **retracting**

Wheel is being retracted.

##### **broken**

Wheel is broken.

#### **class MotorState**

The state of the motor on a powered wheel. See *Wheel.motor\_state*.

##### **idle**

The motor is idle.

##### **running**

The motor is running.

##### **disabled**

The motor is disabled.

##### **inoperable**

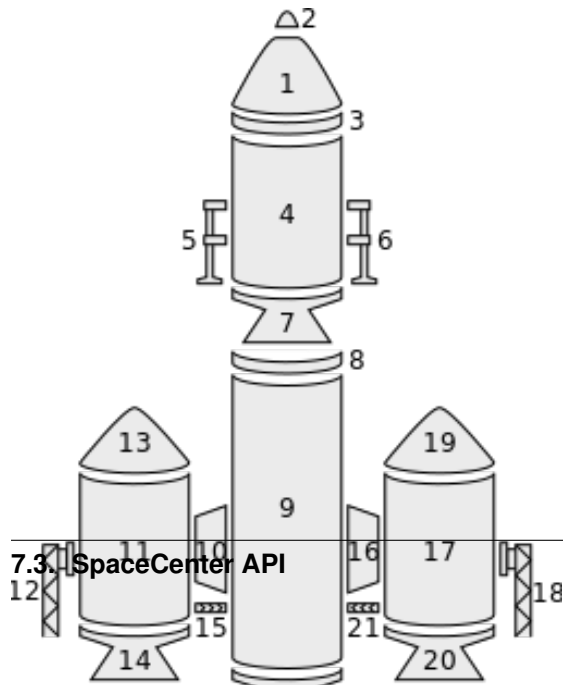
The motor is inoperable.

##### **not\_enough\_resources**

The motor does not have enough resources to run.

## Trees of Parts

Vessels in KSP are comprised of a number of parts, connected to one another in a *tree* structure. An example vessel is shown in Figure 1, and the corresponding tree of parts in Figure 2. The craft file for this example can also be downloaded [here](#).



## Traversing the Tree

The tree of parts can be traversed using the attributes *Parts.root*, *Part.parent* and *Part.children*.

The root of the tree is the same as the vessels *root part* (part number 1 in the example above) and can be obtained by calling *Parts.root*. A parts children can be obtained by calling *Part.children*. If the part does not have any children, *Part.children* returns an empty list. A parts parent can be obtained by calling *Part.parent*. If the part does not have a parent (as is the case for the root part), *Part.parent* returns *None*.

The following Python example uses these attributes to perform a depth-first traversal over all of the parts in a vessel:

```
import krpc
conn = krpc.connect()
vessel = conn.space

root = vessel.parts[0]
stack = [(root, 0)]
while stack:
    part, depth = stack.pop()
    print(' ' * depth, part.name)
    for child in part.children:
        stack.append((child, depth + 1))
```

When this code is executed using the craft file for the example vessel pictured above, the following is printed out:

```
Command Pod Mk1
TR-18A Stack Decoupler
FL-T400 Fuel Tank
LV-909 Liquid Fuel Tank
TR-18A Stack Decoupler
FL-T800 Fuel Tank
LV-909 Liquid Fuel Tank
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launcher
FTX-2 External Tank
LV-909 Liquid Fuel Tank
Aerodynamic Surface
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launcher
FTX-2 External Tank
LV-909 Liquid Fuel Tank
Aerodynamic Surface
LT-1 Landing Struts
LT-1 Landing Struts
Mk16 Parachute
```

## Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

The root part of a vessel (for example the command pod – part 1) does not have a parent part, so does not have an attachment mode. However, the part is considered to be *axially* attached to nothing.

The following Python example does a depth-first traversal as before, but also prints out the attachment mode used by the part:

```
import krpc
conn = krpc.connect()
vessel = conn.space_center.active_vessel

root = vessel.parts.root
stack = [(root, 0)]
while stack:
    part, depth = stack.pop()
    if part.axially_attached:
        attach_mode = 'axial'
    else: # radially_attached
        attach_mode = 'radial'
    print(
        '↳ ' * depth, part.title, '-', attach_mode)
    for child in part.children:
        stack.append((child, depth+1))
```

When this code is execute using the craft file for the example vessel pictured above, the following is printed out:

```
Command Pod Mk1 - axial
TR-18A Stack Decoupler - axial
FL-T400 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TR-18A Stack Decoupler - axial
FL-T800 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial

↳ TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial

↳ TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
LT-1 Landing Struts - radial
LT-1 Landing Struts - radial
Mk16 Parachute - axial
```

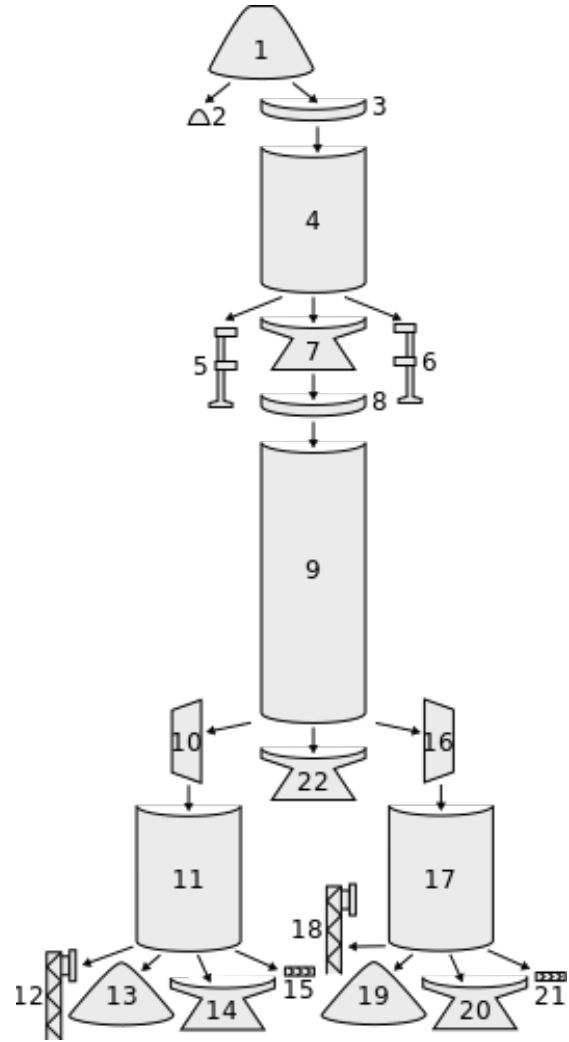


Fig. 7.11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the child part.

## Fuel Lines

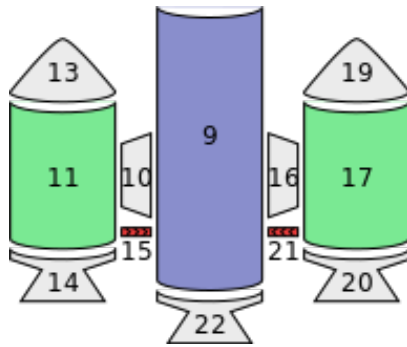


Fig. 7.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.

The attributes `Part.fuel_lines_from` and `Part.fuel_lines_to` can be used to discover these connections. In the example in Figure 5, when `Part.fuel_lines_to` is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When `Part.fuel_lines_from` is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).

## Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using `Part.stage` and `Part.decouple_stage` respectively. For parts that are not activated by staging, `Part.stage` returns -1. For parts that are never decoupled, `Part.decouple_stage` returns a value of -1.

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *activated*. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

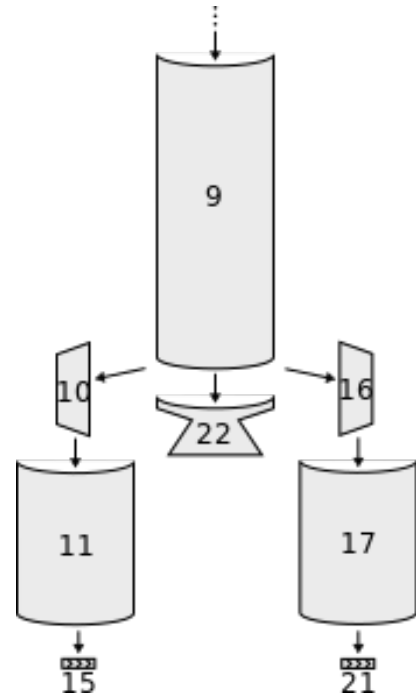


Fig. 7.13: **Figure 4** – A subset of the parts tree from Figure 2 above.

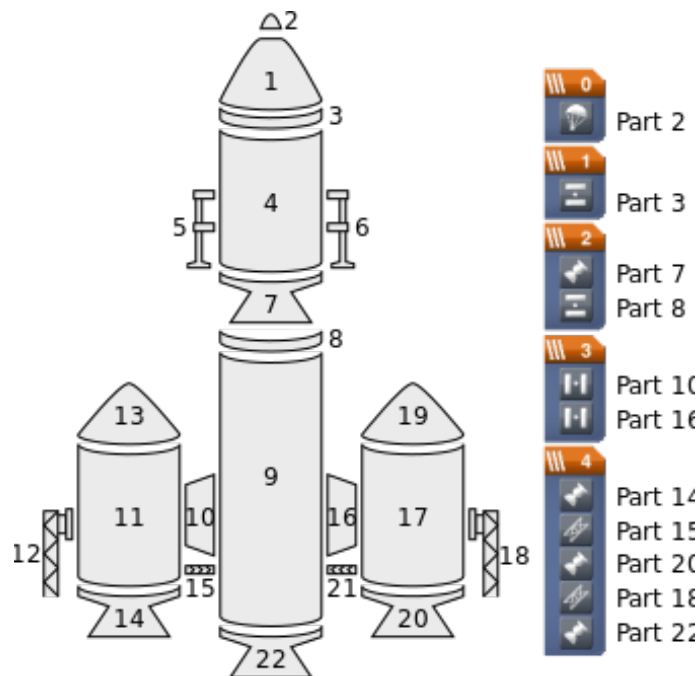


Fig. 7.14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

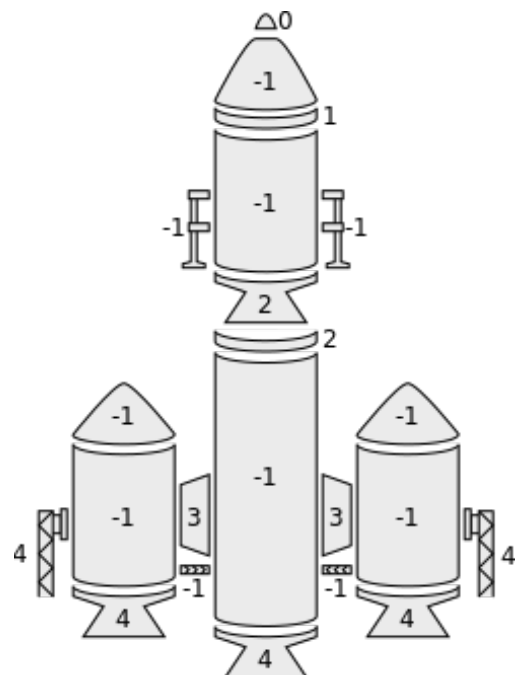
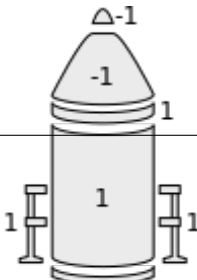


Fig. 7.15: **Figure 7** – The stage in which each part is *activated*.

### 7.3.9 Resources

**class Resources**  
Represents the collection of resources stored in a vessel,



stage or part. Created by calling `Vessel.resources`, `Vessel.resources_in_decouple_stage()` or `Part.resources`.

**all**

All the individual resources that can be stored.

**Attribute** Read-only, cannot be set

**Return type** `list(Resource)`

**with\_resource** (*name*)

All the individual resources with the given name that can be stored.

**Parameters** **name** (*str*) –

**Return type** `list(Resource)`

**names**

A list of resource names that can be stored.

**Attribute** Read-only, cannot be set

**Return type** `list(str)`

**has\_resource** (*name*)

Check whether the named resource can be stored.

**Parameters** **name** (*str*) – The name of the resource.

**Return type** `bool`

**amount** (*name*)

Returns the amount of a resource that is currently stored.

**Parameters** **name** (*str*) – The name of the resource.

**Return type** `float`

**max** (*name*)

Returns the amount of a resource that can be stored.

**Parameters** **name** (*str*) – The name of the resource.

**Return type** `float`

**static\_density** (*name*)

Returns the density of a resource, in *kg/l*.

**Parameters** **name** (*str*) – The name of the resource.

**Return type** `float`

**static\_flow\_mode** (*name*)

Returns the flow mode of a resource.

**Parameters** **name** (*str*) – The name of the resource.

**Return type** `ResourceFlowMode`

**enabled**

Whether use of all the resources are enabled.

**Attribute** Can be read or written

**Return type** `bool`



---

**Note:** This is `True` if all of the resources are enabled. If any of the resources are not enabled, this is `False`.

---

### **class Resource**

An individual resource stored within a part. Created using methods in the *Resources* class.

#### **name**

The name of the resource.

**Attribute** Read-only, cannot be set

**Return type** `str`

#### **part**

The part containing the resource.

**Attribute** Read-only, cannot be set

**Return type** *Part*

#### **amount**

The amount of the resource that is currently stored in the part.

**Attribute** Read-only, cannot be set

**Return type** `float`

#### **max**

The total amount of the resource that can be stored in the part.

**Attribute** Read-only, cannot be set

**Return type** `float`

#### **density**

The density of the resource, in *kg/l*.

**Attribute** Read-only, cannot be set

**Return type** `float`

#### **flow\_mode**

The flow mode of the resource.

**Attribute** Read-only, cannot be set

**Return type** *ResourceFlowMode*

#### **enabled**

Whether use of this resource is enabled.

**Attribute** Can be read or written

**Return type** `bool`

### **class ResourceTransfer**

Transfer resources between parts.

#### **static start** (*from\_part, to\_part, resource, max\_amount*)

Start transferring a resource transfer between a pair of parts. The transfer will move at most *max\_amount* units of the resource, depending on how much of the resource is available in the source part and how much storage is available in the destination part. Use

*ResourceTransfer.complete* to check if the transfer is complete. Use *ResourceTransfer.amount* to see how much of the resource has been transferred.

**Parameters**

- **from\_part** (*Part*) – The part to transfer to.
- **to\_part** (*Part*) – The part to transfer from.
- **resource** (*str*) – The name of the resource to transfer.
- **max\_amount** (*float*) – The maximum amount of resource to transfer.

**Return type** *ResourceTransfer*

**amount**

The amount of the resource that has been transferred.

**Attribute** Read-only, cannot be set

**Return type** float

**complete**

Whether the transfer has completed.

**Attribute** Read-only, cannot be set

**Return type** bool

**class ResourceFlowMode**

The way in which a resource flows between parts. See *Resources.flow\_mode()*.

**vessel**

The resource flows to any part in the vessel. For example, electric charge.

**stage**

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

**adjacent**

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

**none**

The resource does not flow. For example, solid fuel.

## 7.3.10 Node

**class Node**

Represents a maneuver node. Can be created using *Control.add\_node()*.

**prograde**

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

**Attribute** Can be read or written

**Return type** float

**normal**

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

**Attribute** Can be read or written

**Return type** float

**radial**

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

**Attribute** Can be read or written

**Return type** float

**delta\_v**

The delta-v of the maneuver node, in meters per second.

**Attribute** Can be read or written

**Return type** float

---

**Note:** Does not change when executing the maneuver node. See *Node.remaining\_delta\_v*.

---

**remaining\_delta\_v**

Gets the remaining delta-v of the maneuver node, in meters per second. Changes as the node is executed. This is equivalent to the delta-v reported in-game.

**Attribute** Read-only, cannot be set

**Return type** float

**burn\_vector** ([*reference\_frame* = None])

Returns the burn vector for the maneuver node.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame that the returned vector is in. Defaults to *Vessel.orbital\_reference\_frame*.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

**Return type** tuple(float, float, float)

---

**Note:** Does not change when executing the maneuver node. See *Node.remaining\_burn\_vector()*.

---

**remaining\_burn\_vector** ([*reference\_frame* = None])

Returns the remaining burn vector for the maneuver node.

**Parameters** **reference\_frame** (ReferenceFrame) – The reference frame that the returned vector is in. Defaults to *Vessel.orbital\_reference\_frame*.

**Returns** A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

**Return type** tuple(float, float, float)

---

**Note:** Changes as the maneuver node is executed. See *Node*.  
*burn\_vector()*.

---

**ut**

The universal time at which the maneuver will occur, in seconds.

**Attribute** Can be read or written

**Return type** float

**time\_to**

The time until the maneuver node will be encountered, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**orbit**

The orbit that results from executing the maneuver node.

**Attribute** Read-only, cannot be set

**Return type** *Orbit*

**remove()**

Removes the maneuver node.

**reference\_frame**

The reference frame that is fixed relative to the maneuver node's burn.

- The origin is at the position of the maneuver node.
- The y-axis points in the direction of the burn.
- The x-axis and z-axis point in arbitrary but fixed directions.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

**orbital\_reference\_frame**

The reference frame that is fixed relative to the maneuver node, and orientated with the orbital prograde/normal/radial directions of the original orbit at the maneuver node's position.

- The origin is at the position of the maneuver node.
- The x-axis points in the orbital anti-radial direction of the original orbit, at the position of the maneuver node.
- The y-axis points in the orbital prograde direction of the original orbit, at the position of the maneuver node.
- The z-axis points in the orbital normal direction of the original orbit, at the position of the maneuver node.

**Attribute** Read-only, cannot be set

**Return type** *ReferenceFrame*

**position** (*reference\_frame*)

The position vector of the maneuver node in the given reference frame.

**Parameters** **reference\_frame** (`ReferenceFrame`) – The reference frame that the returned position vector is in.

**Returns** The position as a vector.

**Return type** tuple(float, float, float)

**direction** (*reference\_frame*)

The direction of the maneuver nodes burn.

**Parameters** **reference\_frame** (`ReferenceFrame`) – The reference frame that the returned direction is in.

**Returns** The direction as a unit vector.

**Return type** tuple(float, float, float)

### 7.3.11 ReferenceFrame

**class ReferenceFrame**

Represents a reference frame for positions, rotations and velocities. Contains:

- The position of the origin.
- The directions of the x, y and z axes.
- The linear velocity of the frame.
- The angular velocity of the frame.

---

**Note:** This class does not contain any properties or methods. It is only used as a parameter to other functions.

---

**static create\_relative** (*reference\_frame* [, *position* = (0.0, 0.0, 0.0) ] [, *rotation* = (0.0, 0.0, 0.0, 1.0) ] [, *velocity* = (0.0, 0.0, 0.0) ] [, *angular\_velocity* = (0.0, 0.0, 0.0) ] )

Create a relative reference frame. This is a custom reference frame whose components offset the components of a parent reference frame.

**Parameters**

- **reference\_frame** (`ReferenceFrame`) – The parent reference frame on which to base this reference frame.
- **position** (*tuple*) – The offset of the position of the origin, as a position vector. Defaults to (0, 0, 0)
- **rotation** (*tuple*) – The rotation to apply to the parent frames rotation, as a quaternion of the form (*x*, *y*, *z*, *w*). Defaults to (0, 0, 0, 1) (i.e. no rotation)
- **velocity** (*tuple*) – The linear velocity to offset the parent frame by, as a vector pointing in the direction of travel, whose magnitude is the speed in meters per second. Defaults to (0, 0, 0).

- **angular\_velocity** (*tuple*) – The angular velocity to offset the parent frame by, as a vector. This vector points in the direction of the axis of rotation, and its magnitude is the speed of the rotation in radians per second. Defaults to (0, 0, 0).

**Return type** *ReferenceFrame*

**static create\_hybrid**(*position*[, *rotation* = None][, *velocity* = None][, *angular\_velocity* = None])

Create a hybrid reference frame. This is a custom reference frame whose components inherited from other reference frames.

**Parameters**

- **position** (*ReferenceFrame*) – The reference frame providing the position of the origin.
- **rotation** (*ReferenceFrame*) – The reference frame providing the rotation of the frame.
- **velocity** (*ReferenceFrame*) – The reference frame providing the linear velocity of the frame.
- **angular\_velocity** (*ReferenceFrame*) – The reference frame providing the angular velocity of the frame.

**Return type** *ReferenceFrame*

---

**Note:** The *position* reference frame is required but all other reference frames are optional. If omitted, they are set to the *position* reference frame.

---

## 7.3.12 AutoPilot

**class AutoPilot**

Provides basic auto-piloting utilities for a vessel. Created by calling *Vessel.auto\_pilot*.

---

**Note:** If a client engages the auto-pilot and then closes its connection to the server, the auto-pilot will be disengaged and its target reference frame, direction and roll reset to default.

---

**engage**()

Engage the auto-pilot.

**disengage**()

Disengage the auto-pilot.

**wait**()

Blocks until the vessel is pointing in the target direction and has the target roll (if set). Throws an exception if the auto-pilot has not been engaged.

**error**

The error, in degrees, between the direction the ship has been asked to point in and the direction it is pointing in. Throws an exception

if the auto-pilot has not been engaged and SAS is not enabled or is in stability assist mode.

**Attribute** Read-only, cannot be set

**Return type** float

#### **pitch\_error**

The error, in degrees, between the vessels current and target pitch.  
Throws an exception if the auto-pilot has not been engaged.

**Attribute** Read-only, cannot be set

**Return type** float

#### **heading\_error**

The error, in degrees, between the vessels current and target heading.  
Throws an exception if the auto-pilot has not been engaged.

**Attribute** Read-only, cannot be set

**Return type** float

#### **roll\_error**

The error, in degrees, between the vessels current and target roll.  
Throws an exception if the auto-pilot has not been engaged or no target roll is set.

**Attribute** Read-only, cannot be set

**Return type** float

#### **reference\_frame**

The reference frame for the target direction (*AutoPilot.target\_direction*).

**Attribute** Can be read or written

**Return type** *ReferenceFrame*

---

**Note:** An error will be thrown if this property is set to a reference frame that rotates with the vessel being controlled, as it is impossible to rotate the vessel in such a reference frame.

---

#### **target\_pitch**

The target pitch, in degrees, between -90° and +90°.

**Attribute** Can be read or written

**Return type** float

#### **target\_heading**

The target heading, in degrees, between 0° and 360°.

**Attribute** Can be read or written

**Return type** float

#### **target\_roll**

The target roll, in degrees. NaN if no target roll is set.

**Attribute** Can be read or written

**Return type** float

**target\_direction**

Direction vector corresponding to the target pitch and heading. This is in the reference frame specified by *ReferenceFrame*.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

**target\_pitch\_and\_heading** (*pitch, heading*)

Set target pitch and heading angles.

**Parameters**

- **pitch** (*float*) – Target pitch angle, in degrees between -90° and +90°.
- **heading** (*float*) – Target heading angle, in degrees between 0° and 360°.

**sas**

The state of SAS.

**Attribute** Can be read or written

**Return type** bool

---

**Note:** Equivalent to *Control.sas*

---

**sas\_mode**

The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

**Attribute** Can be read or written

**Return type** *SASMode*

---

**Note:** Equivalent to *Control.sas\_mode*

---

**roll\_threshold**

The threshold at which the autopilot will try to match the target roll angle, if any. Defaults to 5 degrees.

**Attribute** Can be read or written

**Return type** float

**stopping\_time**

The maximum amount of time that the vessel should need to come to a complete stop. This determines the maximum angular velocity of the vessel. A vector of three stopping times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 0.5 seconds for each axis.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

**deceleration\_time**

The time the vessel should take to come to a stop pointing in the target direction. This determines the angular acceleration used to decelerate the vessel. A vector of three times, in seconds, one for



each of the pitch, roll and yaw axes. Defaults to 5 seconds for each axis.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

#### **attenuation\_angle**

The angle at which the autopilot considers the vessel to be pointing close to the target. This determines the midpoint of the target velocity attenuation function. A vector of three angles, in degrees, one for each of the pitch, roll and yaw axes. Defaults to 1° for each axis.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

#### **auto\_tune**

Whether the rotation rate controllers PID parameters should be automatically tuned using the vessels moment of inertia and available torque. Defaults to True. See *AutoPilot.time\_to\_peak* and *AutoPilot.overshoot*.

**Attribute** Can be read or written

**Return type** bool

#### **time\_to\_peak**

The target time to peak used to autotune the PID controllers. A vector of three times, in seconds, for each of the pitch, roll and yaw axes. Defaults to 3 seconds for each axis.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

#### **overshoot**

The target overshoot percentage used to autotune the PID controllers. A vector of three values, between 0 and 1, for each of the pitch, roll and yaw axes. Defaults to 0.01 for each axis.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

#### **pitch\_pid\_gains**

Gains for the pitch PID controller.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

---

**Note:** When *AutoPilot.auto\_tune* is true, these values are updated automatically, which will overwrite any manual changes.

---

#### **roll\_pid\_gains**

Gains for the roll PID controller.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

---

**Note:** When *AutoPilot.auto\_tune* is true, these values are updated automatically, which will overwrite any manual changes.

---

**yaw\_pid\_gains**

Gains for the yaw PID controller.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

---

**Note:** When *AutoPilot.auto\_tune* is true, these values are updated automatically, which will overwrite any manual changes.

---

### 7.3.13 Camera

**class Camera**

Controls the game's camera. Obtained by calling *camera*.

**mode**

The current mode of the camera.

**Attribute** Can be read or written

**Return type** *CameraMode*

**pitch**

The pitch of the camera, in degrees. A value between *Camera.min\_pitch* and *Camera.max\_pitch*

**Attribute** Can be read or written

**Return type** float

**heading**

The heading of the camera, in degrees.

**Attribute** Can be read or written

**Return type** float

**distance**

The distance from the camera to the subject, in meters. A value between *Camera.min\_distance* and *Camera.max\_distance*.

**Attribute** Can be read or written

**Return type** float

**min\_pitch**

The minimum pitch of the camera.

**Attribute** Read-only, cannot be set

**Return type** float

**max\_pitch**

The maximum pitch of the camera.

**Attribute** Read-only, cannot be set

**Return type** float

**min\_distance**

Minimum distance from the camera to the subject, in meters.

**Attribute** Read-only, cannot be set

**Return type** float

**max\_distance**

Maximum distance from the camera to the subject, in meters.

**Attribute** Read-only, cannot be set

**Return type** float

**default\_distance**

Default distance from the camera to the subject, in meters.

**Attribute** Read-only, cannot be set

**Return type** float

**focussed\_body**

In map mode, the celestial body that the camera is focussed on.

Returns *None* if the camera is not focussed on a celestial body.

Returns an error if the camera is not in map mode.

**Attribute** Can be read or written

**Return type** *CelestialBody*

**focussed\_vessel**

In map mode, the vessel that the camera is focussed on. Returns

*None* if the camera is not focussed on a vessel. Returns an error if the camera is not in map mode.

**Attribute** Can be read or written

**Return type** *Vessel*

**focussed\_node**

In map mode, the maneuver node that the camera is focussed on.

Returns *None* if the camera is not focussed on a maneuver node.

Returns an error if the camera is not in map mode.

**Attribute** Can be read or written

**Return type** *Node*

**class CameraMode**

See *Camera.mode*.

**automatic**

The camera is showing the active vessel, in “auto” mode.

**free**

The camera is showing the active vessel, in “free” mode.

**chase**

The camera is showing the active vessel, in “chase” mode.

**locked**

The camera is showing the active vessel, in “locked” mode.

**orbital**

The camera is showing the active vessel, in “orbital” mode.

**iva**

The Intra-Vehicular Activity view is being shown.

**map**

The map view is being shown.

## 7.3.14 Waypoints

**class WaypointManager**

Waypoints are the location markers you can see on the map view showing you where contracts are targeted for. With this structure, you can obtain coordinate data for the locations of these waypoints. Obtained by calling *waypoint\_manager*.

**waypoints**

A list of all existing waypoints.

**Attribute** Read-only, cannot be set

**Return type** *list(Waypoint)*

**add\_waypoint** (*latitude, longitude, body, name*)

Creates a waypoint at the given position at ground level, and returns a *Waypoint* object that can be used to modify it.

**Parameters**

- **latitude** (*float*) – Latitude of the waypoint.
- **longitude** (*float*) – Longitude of the waypoint.
- **body** (*CelestialBody*) – Celestial body the waypoint is attached to.
- **name** (*str*) – Name of the waypoint.

**Return type** *Waypoint*

**add\_waypoint\_at\_altitude** (*latitude, longitude, altitude, body, name*)

Creates a waypoint at the given position and altitude, and returns a *Waypoint* object that can be used to modify it.

**Parameters**

- **latitude** (*float*) – Latitude of the waypoint.
- **longitude** (*float*) – Longitude of the waypoint.
- **altitude** (*float*) – Altitude (above sea level) of the waypoint.
- **body** (*CelestialBody*) – Celestial body the waypoint is attached to.
- **name** (*str*) – Name of the waypoint.

**Return type** *Waypoint*

**colors**

An example map of known color - seed pairs. Any other integers may be used as seed.

**Attribute** Read-only, cannot be set

**Return type** *dict(str, int)*

**icons**

Returns all available icons (from “Game-Data/Squad/Contracts/Icons”).

**Attribute** Read-only, cannot be set

**Return type** list(str)

**class Waypoint**

Represents a waypoint. Can be created using *WaypointManager.add\_waypoint()*.

**body**

The celestial body the waypoint is attached to.

**Attribute** Can be read or written

**Return type** *CelestialBody*

**name**

The name of the waypoint as it appears on the map and the contract.

**Attribute** Can be read or written

**Return type** str

**color**

The seed of the icon color. See *WaypointManager.colors* for example colors.

**Attribute** Can be read or written

**Return type** int

**icon**

The icon of the waypoint.

**Attribute** Can be read or written

**Return type** str

**latitude**

The latitude of the waypoint.

**Attribute** Can be read or written

**Return type** float

**longitude**

The longitude of the waypoint.

**Attribute** Can be read or written

**Return type** float

**mean\_altitude**

The altitude of the waypoint above sea level, in meters.

**Attribute** Can be read or written

**Return type** float

**surface\_altitude**

The altitude of the waypoint above the surface of the body or sea level, whichever is closer, in meters.

**Attribute** Can be read or written

**Return type** float

**bedrock\_altitude**

The altitude of the waypoint above the surface of the body, in meters.  
When over water, this is the altitude above the sea floor.

**Attribute** Can be read or written

**Return type** float

**near\_surface**

True if the waypoint is near to the surface of a body.

**Attribute** Read-only, cannot be set

**Return type** bool

**grounded**

True if the waypoint is attached to the ground.

**Attribute** Read-only, cannot be set

**Return type** bool

**index**

The integer index of this waypoint within its cluster of sibling waypoints. In other words, when you have a cluster of waypoints called “Somewhere Alpha”, “Somewhere Beta” and “Somewhere Gamma”, the alpha site has index 0, the beta site has index 1 and the gamma site has index 2. When *Waypoint.clustering* is *False*, this is zero.

**Attribute** Read-only, cannot be set

**Return type** int

**clustering**

True if this waypoint is part of a set of clustered waypoints with greek letter names appended (Alpha, Beta, Gamma, etc). If *True*, there is a one-to-one correspondence with the greek letter name and the *Waypoint.index*.

**Attribute** Read-only, cannot be set

**Return type** bool

**has\_contract**

Whether the waypoint belongs to a contract.

**Attribute** Read-only, cannot be set

**Return type** bool

**contract**

The associated contract.

**Attribute** Read-only, cannot be set

**Return type** *Contract*

**remove()**

Removes the waypoint.

### 7.3.15 Contracts

#### **class ContractManager**

Contracts manager. Obtained by calling *waypoint\_manager*.

#### **types**

A list of all contract types.

**Attribute** Read-only, cannot be set

**Return type** set(str)

#### **all\_contracts**

A list of all contracts.

**Attribute** Read-only, cannot be set

**Return type** list(*Contract*)

#### **active\_contracts**

A list of all active contracts.

**Attribute** Read-only, cannot be set

**Return type** list(*Contract*)

#### **offered\_contracts**

A list of all offered, but unaccepted, contracts.

**Attribute** Read-only, cannot be set

**Return type** list(*Contract*)

#### **completed\_contracts**

A list of all completed contracts.

**Attribute** Read-only, cannot be set

**Return type** list(*Contract*)

#### **failed\_contracts**

A list of all failed contracts.

**Attribute** Read-only, cannot be set

**Return type** list(*Contract*)

#### **class Contract**

A contract. Can be accessed using *contract\_manager*.

#### **type**

Type of the contract.

**Attribute** Read-only, cannot be set

**Return type** str

#### **title**

Title of the contract.

**Attribute** Read-only, cannot be set

**Return type** str

#### **description**

Description of the contract.

**Attribute** Read-only, cannot be set

**Return type** str

**notes**

Notes for the contract.

**Attribute** Read-only, cannot be set

**Return type** str

**synopsis**

Synopsis for the contract.

**Attribute** Read-only, cannot be set

**Return type** str

**keywords**

Keywords for the contract.

**Attribute** Read-only, cannot be set

**Return type** list(str)

**state**

State of the contract.

**Attribute** Read-only, cannot be set

**Return type** *ContractState*

**seen**

Whether the contract has been seen.

**Attribute** Read-only, cannot be set

**Return type** bool

**read**

Whether the contract has been read.

**Attribute** Read-only, cannot be set

**Return type** bool

**active**

Whether the contract is active.

**Attribute** Read-only, cannot be set

**Return type** bool

**failed**

Whether the contract has been failed.

**Attribute** Read-only, cannot be set

**Return type** bool

**can\_be\_canceled**

Whether the contract can be canceled.

**Attribute** Read-only, cannot be set

**Return type** bool



**can\_be\_declined**

Whether the contract can be declined.

**Attribute** Read-only, cannot be set

**Return type** bool

**can\_be\_failed**

Whether the contract can be failed.

**Attribute** Read-only, cannot be set

**Return type** bool

**accept ()**

Accept an offered contract.

**cancel ()**

Cancel an active contract.

**decline ()**

Decline an offered contract.

**funds\_advance**

Funds received when accepting the contract.

**Attribute** Read-only, cannot be set

**Return type** float

**funds\_completion**

Funds received on completion of the contract.

**Attribute** Read-only, cannot be set

**Return type** float

**funds\_failure**

Funds lost if the contract is failed.

**Attribute** Read-only, cannot be set

**Return type** float

**reputation\_completion**

Reputation gained on completion of the contract.

**Attribute** Read-only, cannot be set

**Return type** float

**reputation\_failure**

Reputation lost if the contract is failed.

**Attribute** Read-only, cannot be set

**Return type** float

**science\_completion**

Science gained on completion of the contract.

**Attribute** Read-only, cannot be set

**Return type** float

**parameters**

Parameters for the contract.

**Attribute** Read-only, cannot be set

**Return type** list(*ContractParameter*)

**class ContractState**

The state of a contract. See *Contract.state*.

**active**

The contract is active.

**canceled**

The contract has been canceled.

**completed**

The contract has been completed.

**deadline\_expired**

The deadline for the contract has expired.

**declined**

The contract has been declined.

**failed**

The contract has been failed.

**generated**

The contract has been generated.

**offered**

The contract has been offered to the player.

**offer\_expired**

The contract was offered to the player, but the offer expired.

**withdrawn**

The contract has been withdrawn.

**class ContractParameter**

A contract parameter. See *Contract.parameters*.

**title**

Title of the parameter.

**Attribute** Read-only, cannot be set

**Return type** str

**notes**

Notes for the parameter.

**Attribute** Read-only, cannot be set

**Return type** str

**children**

Child contract parameters.

**Attribute** Read-only, cannot be set

**Return type** list(*ContractParameter*)

**completed**

Whether the parameter has been completed.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **failed**

Whether the parameter has been failed.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **optional**

Whether the contract parameter is optional.

**Attribute** Read-only, cannot be set

**Return type** bool

#### **funds\_completion**

Funds received on completion of the contract parameter.

**Attribute** Read-only, cannot be set

**Return type** float

#### **funds\_failure**

Funds lost if the contract parameter is failed.

**Attribute** Read-only, cannot be set

**Return type** float

#### **reputation\_completion**

Reputation gained on completion of the contract parameter.

**Attribute** Read-only, cannot be set

**Return type** float

#### **reputation\_failure**

Reputation lost if the contract parameter is failed.

**Attribute** Read-only, cannot be set

**Return type** float

#### **science\_completion**

Science gained on completion of the contract parameter.

**Attribute** Read-only, cannot be set

**Return type** float

## 7.3.16 Geometry Types

### Vectors

3-dimensional vectors are represented as a 3-tuple. For example:

```
import krpc
conn = krpc.connect()
v = conn.
↪space_center.active_vessel.flight().prograde
print(v[0], v[1], v[2])
```

## Quaternions

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

```
import krpc
conn = krpc.connect()
q = conn.
    ↳space_center.active_vessel.flight().rotation
print(q[0], q[1], q[2], q[3])
```

## 7.4 Drawing API

### 7.4.1 Drawing

Provides functionality for drawing objects in the flight scene.

**static add\_line** (*start*, *end*, *reference\_frame* [, *visible = True* ])

Draw a line in the scene.

#### Parameters

- **start** (*tuple*) – Position of the start of the line.
- **end** (*tuple*) – Position of the end of the line.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the positions are in.
- **visible** (*bool*) – Whether the line is visible.

**Return type** *Line*

**static add\_direction** (*direction*, *reference\_frame* [, *length = 10.0* ] [, *visible = True* ])

Draw a direction vector in the scene, from the center of mass of the active vessel.

#### Parameters

- **direction** (*tuple*) – Direction to draw the line in.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the direction is in.
- **length** (*float*) – The length of the line.
- **visible** (*bool*) – Whether the line is visible.

**Return type** *Line*

**static add\_polygon** (*vertices*, *reference\_frame* [, *visible = True* ])

Draw a polygon in the scene, defined by a list of vertices.

#### Parameters

- **vertices** (*list*) – Vertices of the polygon.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the vertices are in.
- **visible** (*bool*) – Whether the polygon is visible.

**Return type** *Polygon*

**static add\_text** (*text*, *reference\_frame*, *position*, *rotation*[, *visible* = *True*])  
 Draw text in the scene.

**Parameters**

- **text** (*str*) – The string to draw.
- **reference\_frame** (*SpaceCenter.ReferenceFrame*) – Reference frame that the text position is in.
- **position** (*tuple*) – Position of the text.
- **rotation** (*tuple*) – Rotation of the text, as a quaternion.
- **visible** (*bool*) – Whether the text is visible.

**Return type** *Text*

**static clear** ([*client\_only* = *False*])  
 Remove all objects being drawn.

**Parameters** **client\_only** (*bool*) – If true, only remove objects created by the calling client.

## 7.4.2 Line

**class Line**

A line. Created using *add\_line()*.

**start**

Start position of the line.

**Attribute** Can be read or written

**Return type** *tuple(float, float, float)*

**end**

End position of the line.

**Attribute** Can be read or written

**Return type** *tuple(float, float, float)*

**reference\_frame**

Reference frame for the positions of the object.

**Attribute** Can be read or written

**Return type** *SpaceCenter.ReferenceFrame*

**visible**

Whether the object is visible.

**Attribute** Can be read or written

**Return type** *bool*

**color**

Set the color

**Attribute** Can be read or written

**Return type** *tuple(float, float, float)*

**material**

Material used to render the object. Creates the material from a shader with the given name.

**Attribute** Can be read or written

**Return type** str

**thickness**

Set the thickness

**Attribute** Can be read or written

**Return type** float

**remove ()**

Remove the object.

### 7.4.3 Polygon

**class Polygon**

A polygon. Created using *add\_polygon()*.

**vertices**

Vertices for the polygon.

**Attribute** Can be read or written

**Return type** list(tuple(float, float, float))

**reference\_frame**

Reference frame for the positions of the object.

**Attribute** Can be read or written

**Return type** *SpaceCenter.ReferenceFrame*

**visible**

Whether the object is visible.

**Attribute** Can be read or written

**Return type** bool

**remove ()**

Remove the object.

**color**

Set the color

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

**material**

Material used to render the object. Creates the material from a shader with the given name.

**Attribute** Can be read or written

**Return type** str

**thickness**

Set the thickness

**Attribute** Can be read or written

**Return type** float

## 7.4.4 Text

### **class Text**

Text. Created using *add\_text()*.

#### **position**

Position of the text.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

#### **rotation**

Rotation of the text as a quaternion.

**Attribute** Can be read or written

**Return type** tuple(float, float, float, float)

#### **reference\_frame**

Reference frame for the positions of the object.

**Attribute** Can be read or written

**Return type** *SpaceCenter.ReferenceFrame*

#### **visible**

Whether the object is visible.

**Attribute** Can be read or written

**Return type** bool

#### **remove()**

Remove the object.

#### **content**

The text string

**Attribute** Can be read or written

**Return type** str

#### **font**

Name of the font

**Attribute** Can be read or written

**Return type** str

#### **static available\_fonts()**

A list of all available fonts.

**Return type** list(str)

#### **size**

Font size.

**Attribute** Can be read or written

**Return type** int

**character\_size**

Character size.

**Attribute** Can be read or written

**Return type** float

**style**

Font style.

**Attribute** Can be read or written

**Return type** *UI.FontStyle*

**color**

Set the color

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

**material**

Material used to render the object. Creates the material from a shader with the given name.

**Attribute** Can be read or written

**Return type** str

**alignment**

Alignment.

**Attribute** Can be read or written

**Return type** *UI.TextAlignment*

**line\_spacing**

Line spacing.

**Attribute** Can be read or written

**Return type** float

**anchor**

Anchor.

**Attribute** Can be read or written

**Return type** *UI.TextAnchor*

## 7.5 InfernalRobotics API

Provides RPCs to interact with the [InfernalRobotics](#) mod. Provides the following classes:

### 7.5.1 InfernalRobotics

This service provides functionality to interact with [InfernalRobotics](#).

**available**

Whether Infernal Robotics is installed.



**Attribute** Read-only, cannot be set

**Return type** bool

**static servo\_groups** (*vessel*)

A list of all the servo groups in the given *vessel*.

**Parameters** **vessel** (`SpaceCenter.Vessel`) –

**Return type** list(*ServoGroup*)

**static servo\_group\_with\_name** (*vessel*, *name*)

Returns the servo group in the given *vessel* with the given *name*, or `None` if none exists. If multiple servo groups have the same name, only one of them is returned.

**Parameters**

- **vessel** (`SpaceCenter.Vessel`) – Vessel to check.
- **name** (*str*) – Name of servo group to find.

**Return type** *ServoGroup*

**static servo\_with\_name** (*vessel*, *name*)

Returns the servo in the given *vessel* with the given *name* or `None` if none exists. If multiple servos have the same name, only one of them is returned.

**Parameters**

- **vessel** (`SpaceCenter.Vessel`) – Vessel to check.
- **name** (*str*) – Name of the servo to find.

**Return type** *Servo*

## 7.5.2 ServoGroup

**class ServoGroup**

A group of servos, obtained by calling *servo\_groups()* or *servo\_group\_with\_name()*. Represents the “Servo Groups” in the InfernalRobotics UI.

**name**

The name of the group.

**Attribute** Can be read or written

**Return type** str

**forward\_key**

The key assigned to be the “forward” key for the group.

**Attribute** Can be read or written

**Return type** str

**reverse\_key**

The key assigned to be the “reverse” key for the group.

**Attribute** Can be read or written

**Return type** str

**speed**

The speed multiplier for the group.

**Attribute** Can be read or written

**Return type** float

**expanded**

Whether the group is expanded in the InfernalRobotics UI.

**Attribute** Can be read or written

**Return type** bool

**servos**

The servos that are in the group.

**Attribute** Read-only, cannot be set

**Return type** list(*Servo*)

**servo\_with\_name** (*name*)

Returns the servo with the given *name* from this group, or None if none exists.

**Parameters** **name** (*str*) – Name of servo to find.

**Return type** *Servo*

**parts**

The parts containing the servos in the group.

**Attribute** Read-only, cannot be set

**Return type** list(*SpaceCenter.Part*)

**move\_right** ()

Moves all of the servos in the group to the right.

**move\_left** ()

Moves all of the servos in the group to the left.

**move\_center** ()

Moves all of the servos in the group to the center.

**move\_next\_preset** ()

Moves all of the servos in the group to the next preset.

**move\_prev\_preset** ()

Moves all of the servos in the group to the previous preset.

**stop** ()

Stops the servos in the group.

### 7.5.3 Servo

**class Servo**

Represents a servo. Obtained using *ServoGroup.servos*, *ServoGroup.servo\_with\_name()* or *servo\_with\_name()*.

**name**

The name of the servo.

**Attribute** Can be read or written

**Return type** str

#### **part**

The part containing the servo.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

#### **highlight**

Whether the servo should be highlighted in-game.

**Attribute** Write-only, cannot be read

**Return type** bool

#### **position**

The position of the servo.

**Attribute** Read-only, cannot be set

**Return type** float

#### **min\_config\_position**

The minimum position of the servo, specified by the part configuration.

**Attribute** Read-only, cannot be set

**Return type** float

#### **max\_config\_position**

The maximum position of the servo, specified by the part configuration.

**Attribute** Read-only, cannot be set

**Return type** float

#### **min\_position**

The minimum position of the servo, specified by the in-game tweak menu.

**Attribute** Can be read or written

**Return type** float

#### **max\_position**

The maximum position of the servo, specified by the in-game tweak menu.

**Attribute** Can be read or written

**Return type** float

#### **config\_speed**

The speed multiplier of the servo, specified by the part configuration.

**Attribute** Read-only, cannot be set

**Return type** float

#### **speed**

The speed multiplier of the servo, specified by the in-game tweak menu.

**Attribute** Can be read or written

**Return type** float

**current\_speed**

The current speed at which the servo is moving.

**Attribute** Can be read or written

**Return type** float

**acceleration**

The current speed multiplier set in the UI.

**Attribute** Can be read or written

**Return type** float

**is\_moving**

Whether the servo is moving.

**Attribute** Read-only, cannot be set

**Return type** bool

**is\_free\_moving**

Whether the servo is freely moving.

**Attribute** Read-only, cannot be set

**Return type** bool

**is\_locked**

Whether the servo is locked.

**Attribute** Can be read or written

**Return type** bool

**is\_axis\_inverted**

Whether the servos axis is inverted.

**Attribute** Can be read or written

**Return type** bool

**move\_right()**

Moves the servo to the right.

**move\_left()**

Moves the servo to the left.

**move\_center()**

Moves the servo to the center.

**move\_next\_preset()**

Moves the servo to the next preset.

**move\_prev\_preset()**

Moves the servo to the previous preset.

**move\_to(position, speed)**

Moves the servo to *position* and sets the speed multiplier to *speed*.

**Parameters**

- **position** (*float*) – The position to move the servo to.

- **speed** (*float*) – Speed multiplier for the movement.

**stop()**

Stops the servo.

### 7.5.4 Example

The following example gets the control group named “MyGroup”, prints out the names and positions of all of the servos in the group, then moves all of the servos to the right for 1 second.

```
import time
import krpc

conn_
↳= krpc.connect(name='InfernalRobotics Example')
vessel = conn.space_center.active_vessel

group = conn.infernal_robotics.
↳servo_group_with_name(vessel, 'MyGroup')
if group is None:
    print('Group not found')
    exit(1)

for servo in group.servos:
    print(servo.name, servo.position)

group.move_right()
time.sleep(1)
group.stop()
```

## 7.6 Kerbal Alarm Clock API

Provides RPCs to interact with the [Kerbal Alarm Clock](#) mod. Provides the following classes:

### 7.6.1 KerbalAlarmClock

This service provides functionality to interact with [Kerbal Alarm Clock](#).

**available**

Whether Kerbal Alarm Clock is available.

**Attribute** Read-only, cannot be set

**Return type** bool

**alarms**

A list of all the alarms.

**Attribute** Read-only, cannot be set

**Return type** list(*Alarm*)

**static alarm\_with\_name** (*name*)

Get the alarm with the given *name*, or `None` if no alarms have that name. If more than one alarm has the name, only returns one of them.

**Parameters** **name** (*str*) – Name of the alarm to search for.

**Return type** *Alarm*

**static alarms\_with\_type** (*type*)

Get a list of alarms of the specified *type*.

**Parameters** **type** (*AlarmType*) – Type of alarm to return.

**Return type** `list(Alarm)`

**static create\_alarm** (*type, name, ut*)

Create a new alarm and return it.

**Parameters**

- **type** (*AlarmType*) – Type of the new alarm.
- **name** (*str*) – Name of the new alarm.
- **ut** (*float*) – Time at which the new alarm should trigger.

**Return type** *Alarm*

## 7.6.2 Alarm

**class Alarm**

Represents an alarm. Obtained by calling *alarms*, *alarm\_with\_name()* or *alarms\_with\_type()*.

**action**

The action that the alarm triggers.

**Attribute** Can be read or written

**Return type** *AlarmAction*

**margin**

The number of seconds before the event that the alarm will fire.

**Attribute** Can be read or written

**Return type** `float`

**time**

The time at which the alarm will fire.

**Attribute** Can be read or written

**Return type** `float`

**type**

The type of the alarm.

**Attribute** Read-only, cannot be set

**Return type** *AlarmType*

**id**

The unique identifier for the alarm.

**Attribute** Read-only, cannot be set

**Return type** str

#### **name**

The short name of the alarm.

**Attribute** Can be read or written

**Return type** str

#### **notes**

The long description of the alarm.

**Attribute** Can be read or written

**Return type** str

#### **remaining**

The number of seconds until the alarm will fire.

**Attribute** Read-only, cannot be set

**Return type** float

#### **repeat**

Whether the alarm will be repeated after it has fired.

**Attribute** Can be read or written

**Return type** bool

#### **repeat\_period**

The time delay to automatically create an alarm after it has fired.

**Attribute** Can be read or written

**Return type** float

#### **vessel**

The vessel that the alarm is attached to.

**Attribute** Can be read or written

**Return type** *SpaceCenter.Vessel*

#### **xfer\_origin\_body**

The celestial body the vessel is departing from.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CelestialBody*

#### **xfer\_target\_body**

The celestial body the vessel is arriving at.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CelestialBody*

#### **remove()**

Removes the alarm.

### 7.6.3 AlarmType

**class AlarmType**

The type of an alarm.

**raw**

An alarm for a specific date/time or a specific period in the future.

**maneuver**

An alarm based on the next maneuver node on the current ships flight path. This node will be stored and can be restored when you come back to the ship.

**maneuver\_auto**

See *AlarmType.maneuver*.

**apoapsis**

An alarm for furthest part of the orbit from the planet.

**periapsis**

An alarm for nearest part of the orbit from the planet.

**ascending\_node**

Ascending node for the targeted object, or equatorial ascending node.

**descending\_node**

Descending node for the targeted object, or equatorial descending node.

**closest**

An alarm based on the closest approach of this vessel to the targeted vessel, some number of orbits into the future.

**contract**

An alarm based on the expiry or deadline of contracts in career modes.

**contract\_auto**

See *AlarmType.contract*.

**crew**

An alarm that is attached to a crew member.

**distance**

An alarm that is triggered when a selected target comes within a chosen distance.

**earth\_time**

An alarm based on the time in the “Earth” alternative Universe (aka the Real World).

**launch\_rendevous**

An alarm that fires as your landed craft passes under the orbit of your target.

**soi\_change**

An alarm manually based on when the next SOI point is on the flight path or set to continually monitor the active flight path and add alarms as it detects SOI changes.



**soi\_change\_auto**

See *AlarmType.soi\_change*.

**transfer**

An alarm based on Interplanetary Transfer Phase Angles, i.e. when should I launch to planet X? Based on Kosmo Not's post and used in Olex's Calculator.

**transfer\_modelled**

See *AlarmType.transfer*.

## 7.6.4 AlarmAction

**class AlarmAction**

The action performed by an alarm when it fires.

**do\_nothing**

Don't do anything at all...

**do\_nothing\_delete\_when\_passed**

Don't do anything, and delete the alarm.

**kill\_warp**

Drop out of time warp.

**kill\_warp\_only**

Drop out of time warp.

**message\_only**

Display a message.

**pause\_game**

Pause the game.

## 7.6.5 Example

The following example creates a new alarm for the active vessel. The alarm is set to trigger after 10 seconds have passed, and display a message.

```

import krpc
conn = _
↳krpc.connect(name='Kerbal Alarm Clock Example')

alarm = conn.kerbal_alarm_clock.create_alarm(
    conn.kerbal_alarm_clock.AlarmType.raw,
    'My New Alarm',
    conn.space_center.ut+10)

alarm.notes = '10 seconds_
↳have now passed since the alarm was created.'
alarm.action = _
↳conn.kerbal_alarm_clock.AlarmAction.message_only

```

## 7.7 RemoteTech API

Provides RPCs to interact with the `RemoteTech` mod. Provides the following classes:

### 7.7.1 RemoteTech

This service provides functionality to interact with `RemoteTech`.

**available**

Whether RemoteTech is installed.

**Attribute** Read-only, cannot be set

**Return type** `bool`

**ground\_stations**

The names of the ground stations.

**Attribute** Read-only, cannot be set

**Return type** `list(str)`

**static antenna** (*part*)

Get the antenna object for a particular part.

**Parameters** **part** (`SpaceCenter.Part`) –

**Return type** `Antenna`

**static comms** (*vessel*)

Get a communications object, representing the communication capability of a particular vessel.

**Parameters** **vessel** (`SpaceCenter.Vessel`) –

**Return type** `Comms`

### 7.7.2 Comms

**class Comms**

Communications for a vessel.

**vessel**

Get the vessel.

**Attribute** Read-only, cannot be set

**Return type** `SpaceCenter.Vessel`

**has\_local\_control**

Whether the vessel can be controlled locally.

**Attribute** Read-only, cannot be set

**Return type** `bool`

**has\_flight\_computer**

Whether the vessel has a flight computer on board.

**Attribute** Read-only, cannot be set

**Return type** bool

**has\_connection**

Whether the vessel has any connection.

**Attribute** Read-only, cannot be set

**Return type** bool

**has\_connection\_to\_ground\_station**

Whether the vessel has a connection to a ground station.

**Attribute** Read-only, cannot be set

**Return type** bool

**signal\_delay**

The shortest signal delay to the vessel, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**signal\_delay\_to\_ground\_station**

The signal delay between the vessel and the closest ground station, in seconds.

**Attribute** Read-only, cannot be set

**Return type** float

**signal\_delay\_to\_vessel** (*other*)

The signal delay between the this vessel and another vessel, in seconds.

**Parameters** *other* (*SpaceCenter.Vessel*) –

**Return type** float

**antennas**

The antennas for this vessel.

**Attribute** Read-only, cannot be set

**Return type** list(*Antenna*)

### 7.7.3 Antenna

**class Antenna**

A RemoteTech antenna. Obtained by calling *Comms.antennas* or *antenna()*.

**part**

Get the part containing this antenna.

**Attribute** Read-only, cannot be set

**Return type** *SpaceCenter.Part*

**has\_connection**

Whether the antenna has a connection.

**Attribute** Read-only, cannot be set

**Return type** bool

**target**

The object that the antenna is targetting. This property can be used to set the target to *Target.none* or *Target.active\_vessel*. To set the target to a celestial body, ground station or vessel see *Antenna.target\_body*, *Antenna.target\_ground\_station* and *Antenna.target\_vessel*.

**Attribute** Can be read or written

**Return type** *Target*

**target\_body**

The celestial body the antenna is targetting.

**Attribute** Can be read or written

**Return type** *SpaceCenter.CelestialBody*

**target\_ground\_station**

The ground station the antenna is targetting.

**Attribute** Can be read or written

**Return type** str

**target\_vessel**

The vessel the antenna is targetting.

**Attribute** Can be read or written

**Return type** *SpaceCenter.Vessel*

**class Target**

The type of object an antenna is targetting. See *Antenna.target*.

**active\_vessel**

The active vessel.

**celestial\_body**

A celestial body.

**ground\_station**

A ground station.

**vessel**

A specific vessel.

**none**

No target.

## 7.7.4 Example

The following example sets the target of a dish on the active vessel then prints out the signal delay to the active vessel.

```
import krpc
conn = krpc.connect(name='RemoteTech Example')
vessel = conn.space_center.active_vessel

# Set a dish target
```

```

part_
↳= vessel.parts.with_title('Reflectron KR-7')[0]
antenna = conn.remote_tech.antenna(part)
antenna.
↳target_body = conn.space_center.bodies['Jool']

# Get info about the vessels communications
comms = conn.remote_tech.comms(vessel)
print('Signal_
↳delay = %.4f seconds' % comms.signal_delay)

```

## 7.8 User Interface API

### 7.8.1 UI

Provides functionality for drawing and interacting with in-game user interface elements.

#### **stock\_canvas**

The stock UI canvas.

**Attribute** Read-only, cannot be set

**Return type** *Canvas*

#### **static add\_canvas()**

Add a new canvas.

**Return type** *Canvas*

---

**Note:** If you want to add UI elements to KSPs stock UI canvas, use *stock\_canvas*.

---

#### **static message**(*content*[, *duration* = 1.0][, *position* = 1])

Display a message on the screen.

#### **Parameters**

- **content** (*str*) – Message content.
- **duration** (*float*) – Duration before the message disappears, in seconds.
- **position** (*MessagePosition*) – Position to display the message.

---

**Note:** The message appears just like a stock message, for example quicksave or quickload messages.

---

#### **static clear**([*client\_only* = False])

Remove all user interface elements.

**Parameters** **client\_only** (*bool*) – If true, only remove objects created by the calling client.

#### **class MessagePosition**

Message position.

**top\_left**

Top left.

**top\_center**

Top center.

**top\_right**

Top right.

**bottom\_center**

Bottom center.

## 7.8.2 Canvas

**class Canvas**

A canvas for user interface elements. See *stock\_canvas* and *add\_canvas()*.

**rect\_transform**

The rect transform for the canvas.

**Attribute** Read-only, cannot be set

**Return type** *RectTransform*

**visible**

Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** bool

**add\_panel** ([*visible = True*])

Create a new container for user interface elements.

**Parameters** **visible** (*bool*) – Whether the panel is visible.

**Return type** *Panel*

**add\_text** (*content* [, *visible = True*])

Add text to the canvas.

**Parameters**

- **content** (*str*) – The text.
- **visible** (*bool*) – Whether the text is visible.

**Return type** *Text*

**add\_input\_field** ([*visible = True*])

Add an input field to the canvas.

**Parameters** **visible** (*bool*) – Whether the input field is visible.

**Return type** *InputField*

**add\_button** (*content* [, *visible = True*])

Add a button to the canvas.

**Parameters**

- **content** (*str*) – The label for the button.
- **visible** (*bool*) – Whether the button is visible.

**Return type** *Button*

**remove ()**

Remove the UI object.

### 7.8.3 Panel

**class Panel**

A container for user interface elements. See *Canvas*.

*add\_panel ()*.

**rect\_transform**

The rect transform for the panel.

**Attribute** Read-only, cannot be set

**Return type** *RectTransform*

**visible**

Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** bool

**add\_panel ([visible = True])**

Create a panel within this panel.

**Parameters** **visible** (*bool*) – Whether the new panel is visible.

**Return type** *Panel*

**add\_text (content[, visible = True])**

Add text to the panel.

**Parameters**

- **content** (*str*) – The text.
- **visible** (*bool*) – Whether the text is visible.

**Return type** *Text*

**add\_input\_field ([visible = True])**

Add an input field to the panel.

**Parameters** **visible** (*bool*) – Whether the input field is visible.

**Return type** *InputField*

**add\_button (content[, visible = True])**

Add a button to the panel.

**Parameters**

- **content** (*str*) – The label for the button.
- **visible** (*bool*) – Whether the button is visible.

**Return type** *Button*

**remove ()**

Remove the UI object.

## 7.8.4 Text

### **class Text**

A text label. See *Panel.add\_text()*.

#### **rect\_transform**

The rect transform for the text.

**Attribute** Read-only, cannot be set

**Return type** *RectTransform*

#### **visible**

Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** bool

#### **content**

The text string

**Attribute** Can be read or written

**Return type** str

#### **font**

Name of the font

**Attribute** Can be read or written

**Return type** str

#### **available\_fonts**

A list of all available fonts.

**Attribute** Read-only, cannot be set

**Return type** list(str)

#### **size**

Font size.

**Attribute** Can be read or written

**Return type** int

#### **style**

Font style.

**Attribute** Can be read or written

**Return type** *FontStyle*

#### **color**

Set the color

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

#### **alignment**

Alignment.

**Attribute** Can be read or written

**Return type** *TextAnchor*



**line\_spacing**

Line spacing.

**Attribute** Can be read or written

**Return type** float

**remove** ()

Remove the UI object.

**class FontStyle**

Font style.

**normal**

Normal.

**bold**

Bold.

**italic**

Italic.

**bold\_and\_italic**

Bold and italic.

**class TextAlignment**

Text alignment.

**left**

Left aligned.

**right**

Right aligned.

**center**

Center aligned.

**class TextAnchor**

Text alignment.

**lower\_center**

Lower center.

**lower\_left**

Lower left.

**lower\_right**

Lower right.

**middle\_center**

Middle center.

**middle\_left**

Middle left.

**middle\_right**

Middle right.

**upper\_center**

Upper center.

**upper\_left**

Upper left.

**upper\_right**  
Upper right.

## 7.8.5 Button

**class Button**  
A text label. See *Panel.add\_button()*.

**rect\_transform**  
The rect transform for the text.

**Attribute** Read-only, cannot be set

**Return type** *RectTransform*

**visible**  
Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** bool

**text**  
The text for the button.

**Attribute** Read-only, cannot be set

**Return type** *Text*

**clicked**  
Whether the button has been clicked.

**Attribute** Can be read or written

**Return type** bool

---

**Note:** This property is set to true when the user clicks the button.  
A client script should reset the property to false in order to detect subsequent button presses.

---

**remove()**  
Remove the UI object.

## 7.8.6 InputField

**class InputField**  
An input field. See *Panel.add\_input\_field()*.

**rect\_transform**  
The rect transform for the input field.

**Attribute** Read-only, cannot be set

**Return type** *RectTransform*

**visible**  
Whether the UI object is visible.

**Attribute** Can be read or written

**Return type** bool

**value**

The value of the input field.

**Attribute** Can be read or written

**Return type** str

**text**

The text component of the input field.

**Attribute** Read-only, cannot be set

**Return type** *Text*

---

**Note:** Use *InputField.value* to get and set the value in the field. This object can be used to alter the style of the input field's text.

---

**changed**

Whether the input field has been changed.

**Attribute** Can be read or written

**Return type** bool

---

**Note:** This property is set to true when the user modifies the value of the input field. A client script should reset the property to false in order to detect subsequent changes.

---

**remove ()**

Remove the UI object.

## 7.8.7 Rect Transform

**class RectTransform**

A Unity engine Rect Transform for a UI object. See the [Unity manual](#) for more details.

**position**

Position of the rectangles pivot point relative to the anchors.

**Attribute** Can be read or written

**Return type** tuple(float, float)

**local\_position**

Position of the rectangles pivot point relative to the anchors.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

**size**

Width and height of the rectangle.

**Attribute** Can be read or written

**Return type** tuple(float, float)

**upper\_right**

Position of the rectangles upper right corner relative to the anchors.

**Attribute** Can be read or written

**Return type** tuple(float, float)

**lower\_left**

Position of the rectangles lower left corner relative to the anchors.

**Attribute** Can be read or written

**Return type** tuple(float, float)

**anchor**

Set the minimum and maximum anchor points as a fraction of the size of the parent rectangle.

**Attribute** Write-only, cannot be read

**Return type** tuple(float, float)

**anchor\_max**

The anchor point for the lower left corner of the rectangle defined as a fraction of the size of the parent rectangle.

**Attribute** Can be read or written

**Return type** tuple(float, float)

**anchor\_min**

The anchor point for the upper right corner of the rectangle defined as a fraction of the size of the parent rectangle.

**Attribute** Can be read or written

**Return type** tuple(float, float)

**pivot**

Location of the pivot point around which the rectangle rotates, defined as a fraction of the size of the rectangle itself.

**Attribute** Can be read or written

**Return type** tuple(float, float)

**rotation**

Rotation, as a quaternion, of the object around its pivot point.

**Attribute** Can be read or written

**Return type** tuple(float, float, float, float)

**scale**

Scale factor applied to the object in the x, y and z dimensions.

**Attribute** Can be read or written

**Return type** tuple(float, float, float)

## OTHER CLIENTS, SERVICES AND SCRIPTS

This page links to clients, services, scripts, tools and other useful things for kRPC that have been made by others. If you want your own project added to this page, please feel free to ask [on the forum](#).

### 8.1 Clients

- [Ruby client](#)
- [Haskell client](#)
- [Node.js client](#) (requires the v0.4.0 pre-release version of kRPC)
- [Using the plugin in F#](#)

### 8.2 Services

- [krpcmj](#) – remote procedures to interact with MechJeb

### 8.3 Scripts/Tools/Libraries etc.

- [kIPC](#) - Inter-Process(or) Communication between kOS and kRPC
- [kautopilly](#) – an autopilot primarily intended for planes.
- [KNav](#) – a flexible platform for implementing computer-assisted navigation and control of vessels.
- [wernher](#) – a toolkit for flight control and orbit analysis.
- [A small logging script](#).



## COMPILING KRPC

### 9.1 Getting the source code

First you need to download a copy of the source code, which is available from [GitHub](https://github.com/kRPC/kRPC) or using the following on the command line:

```
git clone https://github.com/kRPC/kRPC
```

### 9.2 Install Dependencies

Next you need to install [Bazel](#). This is the build system used to compile the project.

The Bazel build scripts will automatically download most of the required dependencies for the project, but the following need to be installed manually on your system:

- [Mono C# compiler, runtime and tools](#)
- Python and virtualenv
- Autotools
- LuaRocks
- pdflatex, rsvg, libxml, libxslt and python headers (for building the documentation)

To install these dependencies via apt on Ubuntu, first follow the instructions on [Mono's website](#) to add their apt repository. Then run the following command:

```
sudo apt-get install mono-complete python-setuptools python-virtualenv \
python-dev autoconf libtool luarocks texlive-latex-base \
texlive-latex-recommended texlive-fonts-recommended texlive-latex-extra \
libxml2-dev libxslt1-dev librsvg2-bin
```

### 9.3 Set Up your Environment

Before building kRPC you need to make `lib/ksp` point to a directory containing Kerbal Space Program. For example on Linux, if your KSP directory is at `/path/to/ksp` and your kRPC source tree is at `/path/to/kRPC` you can create a symlink using `ln -s /path/to/ksp /path/to/kRPC/lib/ksp`

You may also need to modify the symlink at `lib/mono-4.5` to point to the correct location of your Mono installation.

## 9.4 Building using Bazel

To build the kRPC release archive, run `bazel build //:krpc`. The resulting archive containing the GameData directory, client libraries etc will be created at `bazel-out/krpc-<version>.zip`.

The build scripts also define targets for the different parts of the project. They can be built using `bazel build <target>`:

- `//server` builds the server plugin and associated files
- Targets for building individual clients:
  - `//client/csharp`
  - `//client/cpp`
  - `//client/java`
  - `//client/lua`
  - `//client/python`
- Targets for building individual services:
  - `//service/SpaceCenter`
  - `//service/Drawing`
  - `//service/UI`
  - `//service/InfernalRobotics`
  - `//service/KerbalAlarmClock`
  - `//service/RemoteTech`
- Targets for building protobuf definitions for individual languages:
  - `//protobuf/csharp`
  - `//protobuf/cpp`
  - `//protobuf/java`
  - `//protobuf/lua`
  - `//protobuf/python`
- `//doc:html` builds the HTML documentation
- `//doc:pdf` builds the PDF documentation

There are also several convenience scripts:

- `tools/serve-docs.sh` – builds the documentation and serves it from `http://localhost:8080`
- `tools/install.sh` – builds the plugin and the testing tools, and installs them into the GameData directory of the copy of KSP found at `lib/ksp`.

## 9.5 Building the C# projects using an IDE

A C# solution file (`krpc.sln`) is provided in the root of the project for use with MonoDevelop or a similar C# IDE.

Some of the C# source files it references are generated by the Bazel build scripts. You need to run `bazel build //:csproj` to generate these files before the solution can be built.



Alternatively, if you are unable to run Bazel to build these files, you can [download them from GitHub](#). Simply extract this archive over your copy of the source and you are good to go.

### 9.5.1 Running the Tests

kRPC contains a suite of tests for the server plugin, services, client libraries and others.

The tests, which do not require KSP to be running, can be executed using: `bazel test //:test`

kRPC also includes a suite of tests that require KSP to be running. First run `tools/install.sh` to build kRPC and a testing tools DLL, and install them into the GameData directory of the copy of KSP found at `lib/ksp`. Then run KSP, load a save game and start the server (with automatically accept client connections enabled). Then install the `krpc` python client, the `krptest` package (built by target `//tools/krptest`) and run the scripts to test a particular service, for example those found in `service/SpaceCenter/test`. These tests will automatically load a save game called `krptest`, launch a vessel and run various tests on it.



## EXTENDING KRPC

### 10.1 The kRPC Architecture

kRPC consists of two components: a server and a client. The server plugin (provided by `KRPC.dll`) runs inside KSP. It provides a collection of *procedures* that clients can run. These procedures are arranged in groups called *services* to keep things organized. It also provides an in-game user interface that can be used to start/stop the server, change settings and monitor active clients.

Clients run outside of KSP. This gives you the freedom to run scripts in whatever environment you want. A client communicates with the server to run procedures using one of the supported *Communication Protocols*. kRPC comes with several client libraries that implement one of these protocols, making it easier to write programs in these languages.

kRPC comes with a collection of standard functionality for interacting with vessels, contained in a service called `SpaceCenter`. This service provides procedures for things like getting flight/orbital data and controlling the active vessel. This service is provided by `KRPC.SpaceCenter.dll`.

### 10.2 Service API

Third party mods can add functionality to kRPC using the *Service API*. This is done by adding *attributes* to your own classes, methods and properties to make them visible through the server. When the kRPC server starts, it scans all the assemblies loaded by the game, looking for classes, methods and properties with these attributes.

The following example implements a service that can control the throttle and staging of the active vessel. To add this to the server, compile the code and place the DLL in your `GameData` directory.

```
using KRPC.Service;
using KRPC.Service.Attributes;
using KSP.UI.Screens;

namespace LaunchControl
{
    /// <summary>
    /// Service for staging vessels and controlling their throttle.
    /// </summary>
    [KRPCService (GameScene = GameScene.Flight)]
    public static class LaunchControl
    {
        /// <summary>
        /// The current throttle setting for the active vessel, between 0 and 1.
        /// </summary>
        [KRPCProperty]
        public static float Throttle {
```

```
        get { return FlightInputHandler.state.mainThrottle; }
        set { FlightInputHandler.state.mainThrottle = value; }
    }

    /// <summary>
    /// Activate the next stage in the vessel.
    /// </summary>
    [KRPCProcedure]
    public static void ActivateStage ()
    {
        StageManager.ActivateNextStage ();
    }
}
```

The following example shows how this service can then be used from a python client:

```
import krpc
conn = krpc.connect()
conn.launch_control.throttle = 1
conn.launch_control.activate_stage()
```

Some of the client libraries automatically pick up changes to the functionality provided by the server, including the Python and Lua clients. However, some clients require code to be generated from the service assembly so that they can interact with new or changed functionality. See *clientgen* for details on how to generate this code.

## 10.2.1 Attributes

The following C# attributes can be used to add functionality to the kRPC server.

**KRPCService** (*String Name, KRPC.Service.GameScene GameScene*)

### Parameters

- **Name** – Optional name for the service. If omitted, the service name is set to the name of the class this attribute is applied to.
- **GameScene** – The game scenes in which the services procedures are available.

This attribute is applied to a static class, to indicate that all methods, properties and classes declared within it are part of the the same service. The name of the service is set to the name of the class, or – if present – the Name parameter.

Multiple services with the same name can be declared, as long the classes, procedures and methods they contain have unique names. The classes will be merged to appear as a single service on the server.

The type to which this attribute is applied must satisfy the following criteria:

- The type must be a class.
- The class must be `public static`.
- The name of the class, or the Name parameter if specified, must be a valid *kRPC identifier*.
- The class must not be declared within another class that has the *KRPCService* attribute. Nesting of services is not permitted.

Services are configured to be available in specific *game scenes* via the GameScene parameter. If the GameScene parameter is not specified, the service is available in any scene. If a procedure is called when the service is not available, it will throw an exception.

## Examples

- Declare a service called EVA:

```
[KRPCService]
public static class EVA {
    ...
}
```

- Declare a service called MyEVAService (different to the name of the class):

```
[KRPCService (Name = "MyEVAService")]
public static class EVA {
    ...
}
```

- Declare a service called FlightTools that is only available during the Flight game scene:

```
[KRPCService (GameScene = GameScene.Flight)]
public static class FlightTools {
    ...
}
```

## KRPCProcedure

### Parameters

- **Nullable** – Whether the return value of the procedure can be null. Defaults to false.

This `attribute` is applied to static methods, to add them to the server as procedures.

The method to which this attribute is applied must satisfy the following criteria:

- The method must be `public static`.
- The name of the method must be a valid *kRPC identifier*.
- The method must be declared inside a class that is a *KRPCService*.
- The parameter types and return type must be *types that kRPC knows how to serialize*.
- Parameters can have default arguments.

If the procedure might return a null value, the `Nullable` parameter of the attribute must be set to true.

### Example

The following defines a service called EVA with a PlantFlag procedure that takes a name and an optional description, and returns a Flag object.

```
[KRPCService]
public static class EVA {
    [KRPCProcedure]
    public static Flag PlantFlag (string name, string description = "")
    {
        ...
    }
}
```

This can be called from a python client as follows:

```
import krpc
conn = krpc.connect()
flag = conn.eva.plant_flag('Landing Site', 'One small step for Kerbal-kind')
```

### KRPCClass (String Service)

#### Parameters

- **Service** – Optional name of the service to add this class to. If omitted, the class is added to the service that contains its definition.

This [attribute](#) is applied to non-static classes. It adds the class to the server, so that references to instances of the class can be passed between client and server.

A *KRPCClass* must be part of a service, just like a *KRPCProcedure*. However, it would be restrictive if the class had to be declared as a nested class inside a class with the *KRPCService* attribute. Therefore, a *KRPCClass* can be declared outside of any service if it has the *Service* parameter set to the name of the service that it is part of. Also, the service that the *Service* parameter refers to does not have to exist. If it does not exist, a service with the given name is created.

The class to which this attribute is applied must satisfy the following criteria:

- The class must be `public` and *not* `static`.
- The name of the class must be a valid *kRPC identifier*.
- The class must either be declared inside a class that is a *KRPCService*, or have its *Service* parameter set to the name of the service it is part of.

#### Examples

- Declare a class called `Flag` in the EVA service:

```
[KRPCService]
public static class EVA {
    [KRPCClass]
    public class Flag {
        ...
    }
}
```

- Declare a class called `Flag`, without nesting the class definition in a service class:

```
[KRPCClass (Service = "EVA")]
public class Flag {
    ...
}
```

### KRPCMethod

#### Parameters

- **Nullable** – Whether the return value of the procedure can be null. Defaults to false.

This [attribute](#) is applied to methods inside a *KRPCClass*. This allows a client to call methods on an instance, or static methods in the class.

The method to which this attribute is applied must satisfy the following criteria:

- The method must be `public`.
- The name of the method must be a valid *kRPC identifier*.

- The method must be declared in a *KRPCClass*.
- The parameter types and return type must be *types that kRPC can serialize*.
- Parameters can have default arguments.

If the method might return a null value, the `Nullable` parameter of the attribute must be set to true.

### Example

Declare a `Remove` method in the `Flag` class:

```
[KRPCClass (Service = "EVA")]
public class Flag {
    [KRPCMethod]
    void Remove()
    {
        ...
    }
}
```

## KRPCProperty

### Parameters

- **Nullable** – Whether the return value of the procedure can be null. Defaults to false.

This `attribute` is applied to class properties, and comes in two flavors:

1. Applied to static properties in a *KRPCService*. In this case, the property must satisfy the following criteria:
  - Must be `public static` and have at least one publicly accessible getter or setter.
  - The name of the property must be a valid *kRPC identifier*.
  - Must be declared inside a *KRPCService*.
2. Applied to non-static properties in a *KRPCClass*. In this case, the property must satisfy the following criteria:
  - Must be `public` and *not static*, and have at least one publicly accessible getter or setter.
  - The name of the property must be a valid *kRPC identifier*.
  - Must be declared inside a *KRPCClass*.

If the property getter might return a null value, the `Nullable` parameter of the attribute must be set to true.

### Examples

- Applied to a static property in a service:

```
[KRPCService]
public static class EVA {
    [KRPCProperty]
    public Flag LastFlag
    {
        get { ... }
    }
}
```

This property can be accessed from a python client as follows:

```
import krpc
conn = krpc.connect()
flag = conn.eva.last_flag
```

- Applied to a non-static property in a class:

```
[KRPCClass (Service = "EVA")]
public class Flag {
    [KRPCProperty]
    public void Name { get; set; }

    [KRPCProperty]
    public void Description { get; set; }
}
```

### **KRPCEnum** (*String Service*)

#### **Parameters**

- **Service** – Optional name of the service to add this enum to. If omitted, the enum is added to the service that contains its definition.

This `attribute` is applied to enumeration types. It adds the enumeration and its permissible values to the server. This attribute works similarly to `KRPCClass`, but is applied to enumeration types.

A `KRPCEnum` must be part of a service, just like a `KRPCClass`. Similarly, a `KRPCEnum` can be declared outside of a service if it has its `Service` parameter set to the name of the service that it is part of.

The enumeration type to which this attribute is applied must satisfy the following criteria:

- The enumeration must be `public`.
- The name of the enumeration must be a valid *kRPC identifier*.
- The enumeration must either be declared inside a `KRPCService`, or have its `Service` parameter set to the name of the service it is part of.
- The *underlying C# type* must be an `int`.

#### **Examples**

- Declare an enumeration type with two values:

```
[KRPCEnum (Service = "EVA")]
public enum FlagState {
    Raised,
    Lowered
}
```

This can be used from a python client as follows:

```
import krpc
conn = krpc.connect()
state = conn.eva.FlagState.lowered
```

### **KRPCException** (*String Service, Type MappedException*)

#### **Parameters**

- **Service** – Optional name of the service to add this enum to. If omitted, the enum is added to the service that contains its definition.



- **MappedException** – Optional type of an exception to map to this exception. For example, can be used to map a built-in C# exception type onto this kRPC exception type.

This **attribute** is applied to an exception class type.

A *KRPCException* must be part of a service, just like a *KRPCClass*. Similarly, a *KRPCException* can be declared outside of a service if it has its *Service* parameter set to the name of the service that it is part of.

The class type to which this attribute is applied must satisfy the following criteria:

- The class must be `public`.
- The name of the class must be a valid *kRPC identifier*.
- The class must either be declared inside a *KRPCService*, or have its *Service* parameter set to the name of the service it is part of.

**KRPCDefaultValue** (*String Name*, *Type ValueConstructor*)

#### Parameters

- **Name** – Name of the parameter to set the default value for.
- **ValueConstructor** – Type of a static class with a *Create* method that returns an instance of the default value.

This **attribute** can be applied to a kRPC method or procedure. It provides a workaround to set the default value of a parameter to a non-compile time constant. Ordinarily, C# only allows compile time constants to be used as the values of default arguments.

The *ValueConstructor* parameter is the type of a static class that contains a static method, called *Create*. When invoke, this method should return the default value.

Note: If you just want to set the default value to a compile time constant, use the C# syntax. kRPC will detect the default values and use them.

#### Examples

- Set the default value to a list:

```
public static class DefaultKerbals
{
    public static IList<string> Create ()
    {
        return new List<string> { "Jeb", "Bill", "Bob" };
    }
}

[KRPCProcedure]
[KRPCDefaultValue ("names", typeof(DefaultKerbals))]
public static void HireKerbals (IList<string> names)
{
    ...
}
```

- Set the default value to a compile time constant, which does not require the *KRPCDefaultValue* attribute:

```
[KRPCProcedure]
public static void HireKerbal (string name = "Jeb")
{
    ...
}
```

### 10.2.2 Identifiers

An identifier can only contain letters and numbers, and must start with an upper case letter. They should follow *CamelCase* capitalization conventions.

### 10.2.3 Serializable Types

A type can only be used as a parameter or return type if kRPC knows how to serialize it. The following types are serializable:

- The C# types `double`, `float`, `int`, `long`, `uint`, `ulong`, `bool`, `string` and `byte[]`
- Any type annotated with *KRPCClass*
- Any type annotated with *KRPCEnum*
- Collections of serializable types:
  - `System.Collections.Generic.IList<T>` where `T` is a serializable type
  - `System.Collections.Generic.IDictionary<K,V>` where `K` is one of `int`, `long`, `uint`, `ulong`, `bool` or `string` and `V` is a serializable type
  - `System.Collections.HashSet<V>` where `V` is a serializable type
- Return types can be `void`
- Protocol buffer message types from namespace `KRPC.Service.Messages`

### 10.2.4 Events

kRPC procedures, methods and properties can return event objects to clients. This is done using the class `KRPC.Services.Event`. This class supports two different types of event.

#### Manually Triggered Events

This type of event must be triggered by some other piece of code running somewhere in the game. It is created by calling the default constructor for type `KRPC.Services.Event`.

For example, the following example is a procedure that returns an event that triggers after a given number of milliseconds. When the event triggers it is removed.

```
[KRPCProcedure]
public static KRPC.Service.Messages.Event OnTimer(uint milliseconds) {
    // Create the event
    var evnt = new KRPC.Service.Event ();

    // Set up a timer that will trigger the event
    var timer = new System.Timers.Timer (milliseconds);
    timer.Elapsed += (s, e) => {
        evnt.Trigger ();
        evnt.Remove ();
        timer.Enabled = false;
    };
    timer.Start();

    // Return the message describing the event to the client
}
```

```

return evnt.Message;
}

```

## Actively Polled Events

This type of event contains a function that is evaluated once per game update. When the function returns true, the event is triggered. The event object is passed to the function so that it can manipulate it as desired.

For example the following creates an event that triggers when the active vessel reaches the given altitude. The event is removed the first time it triggers.

```

[KRPCProcedure]
public static KRPC.Service.Messages.Event OnAltitudeReached(uint altitude) {
    // Create the event
    var evnt = new KRPC.Service.Event ((KRPC.Service.Event e) => {
        bool result = FlightGlobals.ActiveVessel.terrainAltitude > altitude;
        if (result)
            e.Remove();
        return result;
    });

    // Return the message describing the event to the client
    return evnt.Message;
}

```

## 10.2.5 Game Scenes

Each service is configured to be available from a particular game scene, or scenes.

**enum KRPC.Service.GameScene**

### **SpaceCenter**

The game scene showing the Kerbal Space Center buildings.

### **Flight**

The game scene showing a vessel in flight (or on the launchpad/runway).

### **TrackingStation**

The tracking station.

### **EditorVAB**

The Vehicle Assembly Building.

### **EditorSPH**

The Space Plane Hangar.

### **Editor**

Either the VAB or the SPH.

### **All**

All game scenes.

## Examples

- Declare a service that is available in the `KRPC.Service.GameScene.Flight` game scene:

```
[KRPCService (GameScene = GameScene.Flight)]
public static class MyService {
    ...
}
```

- Declare a service that is available in the *KRPC.Service.GameScene.Flight* and *KRPC.Service.GameScene.Editor* game scenes:

```
[KRPCService (GameScene = (GameScene.Flight | GameScene.Editor))]]
public static class MyService {
    ...
}
```

## 10.3 Documentation

Documentation can be added using [C# XML documentation](#). For dynamic clients, such as the Python and Lua clients, the documentation will be automatically exported to clients when they connect.

## 10.4 Further Examples

See the [SpaceCenter service implementation](#) for more extensive examples.

## 10.5 Generating Service Code for Static Clients

Some of the client libraries dynamically construct the code necessary to interact with the server when they connect. This means that these libraries will automatically pick up changes to service code. Such client libraries include those for Python and Lua.

Other client libraries required code to be generated and compiled into them statically. They do not automatically pick up changes to service code. Such client libraries include those for C++ and C#.

Code for these ‘static’ libraries is generated using the *krpc-clientgen* tool. This is provided as part of the [krpctools python package](#). It can be installed using *pip*:

```
pip install krpctools
```

You can then run the script from the command line:

```
$ krpc-clientgen --help

usage: krpc-clientgen [-h] [-v] [-o OUTPUT] [--ksp KSP]
                    [--output-defs OUTPUT_DEFS]
                    {cpp,csharp,java} service input [input ...]

Generate client source code for kRPC services.

positional arguments:
  {cpp,csharp,java}    Language to generate
  service              Name of service to generate
  input                Path to service definition JSON file or assembly
                      DLL(s)
```

```

optional arguments:
  -h, --help            show this help message and exit
  -v, --version          show program's version number and exit
  -o OUTPUT, --output OUTPUT
                        Path to write source code to. If not specified, writes
                        source code to standard output.
  --ksp KSP             Path to Kerbal Space Program directory. Required when
                        reading from an assembly DLL(s)
  --output-defs OUTPUT_DEFS
                        When generating client code from a DLL, output the
                        service definitions to the given JSON file

```

Client code can be generated either directly from an assembly DLL containing the service, or from a JSON file that has previously been generated from an assembly DLL (using the `--output-defs` flag).

Generating client code from an assembly DLL requires a copy of Kerbal Space Program and a C# runtime to be available on the machine. In contrast, generating client code from a JSON file does not have these requirements and so is more portable.

### 10.5.1 Example

The following demonstrates how to generate code for the C++ and C# clients to interact with the LaunchControl service, given in an example previously.

`krpc-clientgen` expects to be passed the location of your copy of Kerbal Space Program, the name of the language to generate, the name of the service (from the *KRPCService* attribute), a path to the assembly containing the service and the path to write the generated code to.

For C++, run the following:

```
krpc-clientgen --ksp=/path/to/ksp cpp LaunchControl LaunchControl.dll
launch_control.hpp
```

To then use the LaunchControl service from C++, you need to link your code against the C++ client library, and include *launch\_control.hpp*.

For C#, run the following:

```
krpc-clientgen --ksp=/path/to/ksp csharp LaunchControl LaunchControl.dll
LaunchControl.cs
```

To then use the LaunchControl service from a C# client, you need to reference the *KRPC.Client.dll* and include *LaunchControl.cs* in your project.



## COMMUNICATION PROTOCOLS

kRPC provides two communication protocols:

- *Protocol Buffers over TCP/IP* for languages that can communicate over a TCP/IP socket.
- *Protocol Buffers over WebSockets* for web browsers.

In both protocols, clients invoke remote procedure calls by *sending and receiving protobuf messages*.

### 11.1 Protocol Buffers over TCP/IP

This communication protocol allows languages that can communication over a TCP/IP connection to interact with a kRPC server.

---

**Note:** If a client library is available for your language, you do not need to implement this protocol.

---

#### 11.1.1 Sending and Receiving Messages

Communication with the server is performed via Protocol Buffer messages, encoded according to the protobuf binary format. When sending messages to and from the server, they are prefixed with their size, in bytes, encoded as a Protocol Buffers varint.

To send a message to the server:

1. Encode the message using the Protocol Buffers format.
2. Get the length of the encoded message data (in bytes) and encode that as a Protocol Buffers varint.
3. Send the encoded length followed by the encoded message data.

To receive a message from the server, do the reverse:

1. Receive the size of the message as a Protocol Buffers encoded varint.
2. Decode the message size.
3. Receive message size bytes of message data.
4. Decode the message data.

### 11.1.2 Connecting to the RPC Server

A client invokes remote procedures by communicating with the *RPC server*. To establish a connection, a client must do the following:

1. Open a TCP socket to the server on its RPC port (which defaults to 50000).
2. Send a `ConnectionRequest` message to the server. This message is defined as:

```
message ConnectionRequest {
  Type type = 1;
  string client_name = 2;
  bytes client_identifier = 3;
  enum Type {
    RPC = 0;
    STREAM = 1;
  };
}
```

The `type` field should be set to `ConnectionRequest.RPC` and the `client_name` field can be set to the name of the client to display on the in-game UI. The `client_identifier` should be left blank.

3. Receive a `ConnectionResponse` message from the server. This message is defined as:

```
message ConnectionResponse {
  Status status = 1;
  enum Status {
    OK = 0;
    MALFORMED_MESSAGE = 1;
    TIMEOUT = 2;
    WRONG_TYPE = 3;
  }
  string message = 2;
  bytes client_identifier = 3;
}
```

If the `status` field is set to `ConnectionResponse.OK` then the connection was successful. If not, the `message` field contains a description of what went wrong.

When the connection is successful, the `client_identifier` contains a unique 16-byte identifier for the client. This is required when connecting to the stream server (described below).

### 11.1.3 Connecting to the Stream Server

Clients can receive *Streams* from the *stream server*. To establish a connection, a client must first connect to the RPC server (as above) then do the following:

1. Open a TCP socket to the server on its stream port (which defaults to 50001).
2. Send a `ConnectionRequest` message, with its `type` field set to `ConnectionRequest.STREAM` and its `client_identifier` field set to the value received in the `client_identifier` field of the `ConnectionResponse` message received when connecting to the RPC server earlier.
3. **Receive a `ConnectionResponse` message, similarly to the RPC server, and check that the value of the `status` field is `ConnectionResponse.OK`.** If not, then the connection was not successful, and the `message` field contains a description of what went wrong.

Connecting to the stream server is optional. If the client doesn't require stream functionality, there is no need to connect.



### 11.1.4 Invoking Remote Procedures

See *Messaging Protocol*.

### 11.1.5 Examples

The following Python code connects to the RPC server at address 127.0.0.1 and port 50000 using the name “Jeb”. Next, it connects to the stream server on port 50001. It then invokes the `KRPC.GetStatus` RPC, receives and decodes the result and prints out the server version number from the response.

The following python code connects to the RPC server at address 127.0.0.1 and port 50000, using the name “Jeb”. Next, it connects to the stream server on port 50001. Finally it invokes the `KRPC.GetStatus` procedure, and receives, decodes and prints the result.

To send and receive messages to the server, they need to be encoded and decoded from their binary format:

- The `encode_varint` and `decode_varint` functions convert between Python integers and Protocol Buffer varint encoded integers.
- `send_message` encodes a message, sends the length of the message to the server as a Protocol Buffer varint encoded integer, and then sends the message data.
- `recv_message` receives the size of the message, decodes it, receives the message data, and decodes it.

```
import socket
from google.protobuf.internal.encoder import _VarintEncoder
from google.protobuf.internal.decoder import _DecodeVarint
from krpc.schema import KRPC

def encode_varint(value):
    """ Encode an int as a protobuf varint """
    data = []
    _VarintEncoder()(data.append, value)
    return b''.join(data)

def decode_varint(data):
    """ Decode a protobuf varint to an int """
    return _DecodeVarint(data, 0)[0]

def send_message(conn, msg):
    """ Send a message, prefixed with its size, to a TPC/IP socket """
    data = msg.SerializeToString()
    size = encode_varint(len(data))
    conn.sendall(size + data)

def recv_message(conn, msg_type):
    """ Receive a message, prefixed with its size, from a TCP/IP socket """
    # Receive the size of the message data
    data = b''
    while True:
        try:
            data += conn.recv(1)
            size = decode_varint(data)
            break
```

```
        except IndexError:
            pass
        # Receive the message data
        data = conn.recv(size)
        # Decode the message
        msg = msg_type()
        msg.ParseFromString(data)
        return msg

# Open a TCP/IP socket to the RPC server
rpc_conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
rpc_conn.connect(('127.0.0.1', 50000))

# Send an RPC connection request
request = KRPC.ConnectionRequest()
request.type = KRPC.ConnectionRequest.RPC
request.client_name = 'Jeb'
send_message(rpc_conn, request)

# Receive the connection response
response = recv_message(rpc_conn, KRPC.ConnectionResponse)

# Check the connection was successful
if response.status != KRPC.ConnectionResponse.OK:
    raise RuntimeError('Connection failed: ' + response.message)
print('Connected to RPC server')

# Invoke the KRPC.GetStatus RPC
call = KRPC.ProcedureCall()
call.service = 'KRPC'
call.procedure = 'GetStatus'
request = KRPC.Request()
request.calls.extend([call])
send_message(rpc_conn, request)

# Receive the response
response = recv_message(rpc_conn, KRPC.Response)

# Check for an error in the response
if response.HasField('error'):
    raise RuntimeError('ERROR: ' + str(response.error))

# Check for an error in the results
assert(len(response.results) == 1)
if response.results[0].HasField('error'):
    raise RuntimeError('ERROR: ' + str(response.error))

# Decode the return value as a Status message
status = KRPC.Status()
status.ParseFromString(response.results[0].value)

# Print out the Status message
print(status)
```

The following example demonstrates how to set up and receive data from the server over a stream:

```

import binascii
import socket
from google.protobuf.internal.encoder import _VarintEncoder
from google.protobuf.internal.decoder import _DecodeVarint
from krpc.schema import KRPC

def encode_varint(value):
    """ Encode an int as a protobuf varint """
    data = []
    _VarintEncoder()(data.append, value)
    return b''.join(data)

def decode_varint(data):
    """ Decode a protobuf varint to an int """
    return _DecodeVarint(data, 0)[0]

def send_message(conn, msg):
    """ Send a message, prefixed with its size, to a TPC/IP socket """
    data = msg.SerializeToString()
    size = encode_varint(len(data))
    conn.sendall(size + data)

def recv_message(conn, msg_type):
    """ Receive a message, prefixed with its size, from a TCP/IP socket """
    # Receive the size of the message data
    data = b''
    while True:
        try:
            data += conn.recv(1)
            size = decode_varint(data)
            break
        except IndexError:
            pass
    # Receive the message data
    data = conn.recv(size)
    # Decode the message
    msg = msg_type()
    msg.ParseFromString(data)
    return msg

# Open a TCP/IP socket to the RPC server
rpc_conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
rpc_conn.connect(('127.0.0.1', 50000))

# Send an RPC connection request
request = KRPC.ConnectionRequest()
request.type = KRPC.ConnectionRequest.RPC
request.client_name = 'Jeb'
send_message(rpc_conn, request)

# Receive the connection response
response = recv_message(rpc_conn, KRPC.ConnectionResponse)

```

```
# Check the connection was successful
if response.status != KRPC.ConnectionResponse.OK:
    raise RuntimeError('Connection failed: ' + response.message)
print('Connected to RPC server')

# Print out the clients identifier
print('RPC client identifier = %s' % binascii.hexlify(
    bytearray(response.client_identifier)).decode('utf8'))

# Open a TCP/IP socket to the Stream server
stream_conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
stream_conn.connect(('127.0.0.1', 50001))

# Send a stream connection request, containing the clients identifier
request = KRPC.ConnectionRequest()
request.type = KRPC.ConnectionRequest.STREAM
request.client_identifier = response.client_identifier
send_message(stream_conn, request)

# Receive the connection response
response = recv_message(stream_conn, KRPC.ConnectionResponse)

# Check the connection was successful
if response.status != KRPC.ConnectionResponse.OK:
    raise RuntimeError("Connection failed: " + response.message)
print('Connected to stream server')

# Build a KRPC.GetStatus call to be streamed
stream_call = KRPC.ProcedureCall()
stream_call.service = 'KRPC'
stream_call.procedure = 'GetStatus'

# Call KRPC.AddStream to add the stream
call = KRPC.ProcedureCall()
call.service = 'KRPC'
call.procedure = 'AddStream'
arg = KRPC.Argument()
arg.position = 0
arg.value = stream_call.SerializeToString()
call.arguments.extend([arg])
request = KRPC.Request()
request.calls.extend([call])
send_message(rpc_conn, request)

# Receive the response
response = recv_message(rpc_conn, KRPC.Response)

# Check for an error in the response
if response.HasField('error'):
    raise RuntimeError('ERROR: ' + str(response.error))

# Check for an error in the results
assert(len(response.results) == 1)
if response.results[0].HasField('error'):
    raise RuntimeError('ERROR: ' + str(response.error))

# Decode the return value as a Stream message
stream = KRPC.Stream()
```

```
stream.ParseFromString(response.results[0].value)

# Repeatedly receive stream updates from the stream server
while True:
    update = recv_message(stream_conn, KRPC.StreamUpdate)
    assert(len(update.results) == 1)
    assert(stream.id == update.results[0].id)
    # Decode and print the return value
    status = KRPC.Status()
    status.ParseFromString(update.results[0].result.value)
    print(status)
```

## 11.2 Protocol Buffers over WebSockets

This communication protocol allows web browsers to communicate with a kRPC over a websockets connection, for example from Javascript in a web browser.

---

**Note:** If a client library is available for your language, you do not need to implement this protocol.

---

### 11.2.1 Connecting to the RPC Server

A client invokes remote procedures by communicating with the *RPC server*. To establish a connection, open a websockets connection to the server on its RPC port (which defaults to 50000). The connection URI can also contain a `name` parameter with the name of the client to display in the on the in-game UI. For example: `ws://localhost:50000/?name=Jeb`

### 11.2.2 Connecting to the Stream Server

Clients can receive *Streams* from the *stream server*. To establish a connection, a client must first connect to the RPC server (as above) then do the following:

1. Get the identifier of the client by calling the `KRPC.GetClientID` remote procedure.
2. Open a websockets connection to the server on its stream port (which defaults to 50001), with an `id` query parameter set to the client identifier encoded in base64. For example: `ws://localhost:50001/?id=Eeh8Vbj2DkWTcJZTkY1EhQ==`

Connecting to the stream server is optional. If the client doesn't require stream functionality, there is no need to connect.

### 11.2.3 Sending and Receiving Messages

Communication with the server is performed via Protocol Buffer messages, encoded according to the protobuf binary format.

To send a message to the server:

1. Encode the message using the Protocol Buffers format.
2. Send a binary websockets message to the server, with payload containing the encoded message data.

To receive a message from the server, do the reverse:

1. Receive a binary websockets message from the server.
2. Decode the messages payload using the Protocol Buffers format.

## 11.2.4 Invoking Remote Procedures

See *Messaging Protocol*.

## 11.2.5 Examples

The following code connects to the RPC server at address 127.0.0.1 and port 50000 using the name “Jeb”. Next, it connects to the stream server on port 50001. It then invokes the `KRPC.GetStatus` RPC, receives and decodes the result and prints out the server version number from the response.

```
var websocket = require('ws');
var protobufjs = require('protobufjs')
var ByteBuffer = require('bytebuffer')
var proto = protobufjs.loadProtoFile('krpc.proto').build();

console.log('Connecting to RPC server')
let rpcConn = new websocket('ws://127.0.0.1:50000')
rpcConn.binaryType = 'arraybuffer'

rpcConn.onopen = (e) => {
  console.log('Successfully connected')
  let call = new proto.krpc.schema.ProcedureCall('KRPC', 'GetStatus');
  let request = new proto.krpc.schema.Request([call]);
  rpcConn.send(request.toArrayBuffer());
  rpcConn.onmessage = (e) => {
    let response = proto.krpc.schema.Response.decode(e.data)
    let status = proto.krpc.schema.Status.decode(response.results[0].value)
    console.log(status);
    process.exit(0);
  };
};
```

The following example demonstrates how to set up and receive data from the server over a stream:

```
'use strict'

var websocket = require('ws');
var protobufjs = require('protobufjs')
var proto = protobufjs.loadProtoFile('krpc.proto').build();

console.log('Connecting to RPC server')
let rpcConn = new websocket('ws://127.0.0.1:50000')
rpcConn.binaryType = 'arraybuffer'

rpcConn.onopen = (evnt) => {
  console.log('Successfully connected')
  console.log('Calling KRPC.GetClientID')
  let call = new proto.krpc.schema.ProcedureCall('KRPC', 'GetClientID');
  let request = new proto.krpc.schema.Request([call]);
  rpcConn.send(request.toArrayBuffer());
};
```

```

rpcConn.onmessage = (evnt) => {
  let response = proto.krpc.schema.Response.decode(evnt.data);
  response.results[0].value.readVarint32(); // skip size
  let client_identifier = response.results[0].value.toBase64();
  console.log('Client identifier =', client_identifier);

  console.log('Connecting to Stream server');
  let streamConn = new websocket('ws://127.0.0.1:50001?id=' + client_identifier);
  streamConn.binaryType = 'arraybuffer';

  streamConn.onopen = (evnt) => {
    console.log('Successfully connected');

    let call_to_stream = new proto.krpc.schema.ProcedureCall('KRPC', 'GetStatus');
    let arg = new proto.krpc.schema.Argument(0, call_to_stream.toArrayBuffer());
    let call = new proto.krpc.schema.ProcedureCall('KRPC', 'AddStream', [arg]);
    let request = new proto.krpc.schema.Request([call]);
    rpcConn.send(request.toArrayBuffer());
    rpcConn.onmessage = (evnt) => {
      let response = proto.krpc.schema.Response.decode(evnt.data);
      let stream = proto.krpc.schema.Stream.decode(response.results[0].value);
      console.log("added stream id =", stream.id.toString());
    };
  };

  streamConn.onmessage = (evnt) => {
    let value = new proto.krpc.schema.StreamUpdate.decode(evnt.data);
    let status = proto.krpc.schema.Status.decode(value.results[0].result.value);
    console.log(status);
  };
};

```

## 11.3 Messaging Protocol

Communication with a kRPC server is performed via Protocol Buffer messages. The kRPC download comes with a protocol buffer message definitions file ([schema/krpc.proto](#)) that defines the structure of these messages. It also contains versions of this file for C#, C++, Java, Lua and Python, compiled using [Google's protocol buffers compiler](#).

### 11.3.1 Invoking Remote Procedures

Remote procedures are arranged into groups called *services*. These act as a single-level namespacing to keep things organized. Each service has a unique name used to identify it, and within a service *each procedure has a unique name*.

Remote procedures are invoked by sending a request message to the RPC server, and waiting for a response message.

The request message contains one or more procedure calls to run, which include the names of the procedures and the values of their arguments. The response message contains the value(s) returned by the procedure(s) and any errors that were encountered.

Requests are processed in order of receipt. The next request from a client will not be processed until the previous one completes execution and its response has been received by the client. When there are multiple client connections, requests are processed in round-robin order.

Within a single request, the procedure calls are also processed in order of receipt. The results list in the response is also ordered so that the results match the calls.

## Anatomy of a Request

A request is sent to the server using a Request Protocol Buffer message with the following format:

```
message Request {
  repeated ProcedureCall calls = 1;
}

message ProcedureCall {
  string service = 1;
  string procedure = 2;
  repeated Argument arguments = 3;
}

message Argument {
  uint32 position = 1;
  bytes value = 2;
}
```

A request message contains one or more procedure calls. This allows efficient batching of multiple calls in a single message, if desired.

The fields of a procedure call are:

- `service` - The name of the service in which the remote procedure is defined.
- `procedure` - The name of the remote procedure to invoke.
- `arguments` - A sequence of `Argument` messages containing the values of the procedure's arguments. The fields of an argument message are:
  - `position` - The zero-indexed position of the of the argument in the procedure's signature.
  - `value` - The value of the argument, encoded in Protocol Buffer format.

The `Argument` messages have a `position` field to allow values for default arguments to be omitted. See *Protocol Buffer Encoding* for details on how to encode the argument values.

## Anatomy of a Response

A response is sent to the client using a Response Protocol Buffer message with the following format:

```
message Response {
  Error error = 1;
  repeated ProcedureResult results = 2;
}

message ProcedureResult {
  Error error = 1;
  bytes value = 2;
}

message Error {
  string service = 1;
  string name = 2;
  string description = 3;
  string stack_trace = 4;
}
```



A response message contains one or more results, corresponding to the procedure calls made in the associated request message.

The `value` field of a procedure result message contains the value of the return value of the remote procedure, if any, encoded in protocol buffer format. See *Protocol Buffer Encoding* for details on how to decode the return value.

If an error occurs processing a request message, the `error` field in the response message will contain a description of the error. If an individual procedure call encounters an error, the `error` field in the corresponding procedure result message will contain a description of the error.

The fields of an error message are:

- `service` - If the error was caused by an exception, this is the name of the service in which the exception type is defined.
- `name` - If the error was caused by an exception, this is the name of the exception type.
- `description` - A human readable description of the error
- `stack_trace` - If the error was caused by an exception, this is a server-side stack trace for the exception.

### Example RPC invocation

The following Python code invokes the `GetStatus` procedure from the *KRPC* service using an already established connection to the server (the `rpc_conn` variable).

The `kRPC.schema.KRPC` package contains the various Protocol Buffer message formats compiled to python code using the Protocol Buffer compiler. The `send_message` and `recv_message` are helper functions used to send/receive messages from the server.

```
call = KRPC.ProcedureCall()
call.service = 'KRPC'
call.procedure = 'GetStatus'
request = KRPC.Request()
request.calls.extend([call])
send_message(rpc_conn, request)

# Receive the response
response = recv_message(rpc_conn, KRPC.Response)

# Check for an error in the response
if response.HasField('error'):
    raise RuntimeError('ERROR: ' + str(response.error))

# Check for an error in the results
assert(len(response.results) == 1)
if response.results[0].HasField('error'):
    raise RuntimeError('ERROR: ' + str(response.error))

# Decode the return value as a Status message
status = KRPC.Status()
status.ParseFromString(response.results[0].value)

# Print out the Status message
print(status)
```

## Protocol Buffer Encoding

Values passed as arguments or received as return values are encoded using the Protocol Buffer version 3 serialization format:

- Documentation for this encoding can be found here: <https://developers.google.com/protocol-buffers/docs/encoding>
- Protocol Buffer libraries in many languages are available here: <https://github.com/google/protobuf/releases>

### 11.3.2 Streams

Streams allow the client to repeatedly execute an RPC on the server and receive its results, without needing to repeatedly call the RPC directly, avoiding the communication overhead that this would involve.

A client can create a stream on the server by calling *AddStream*. This procedure takes an optional boolean argument that controls whether the stream starts sending data to the client or not. If not, *StartStream* can be called later on to start the stream. Once the client is finished with the stream, it can remove it from the server by calling *RemoveStream*. Streams are automatically removed when the client that created it disconnects from the server. Streams are local to each client and there is no way to share a stream between clients.

The RPC for each stream is invoked every *fixed update* and the return values for all of these RPCs are collected together into a stream update message. This is then sent to the client over its stream server connection. If the value returned by a streams RPC does not change since the last update that was sent, its value is omitted from the update message in order to minimize network traffic.

#### Anatomy of a Stream Update

Stream updates are sent to the client using a *StreamUpdate* Protocol Buffer message with the following format:

```
message StreamUpdate {  
  repeated StreamResult results = 1;  
}
```

This contains a list of *StreamResult* messages, one for each stream that exists on the server for that client, and whose return value changed since the last update was sent. It has the following format:

```
message StreamResult {  
  uint64 id = 1;  
  ProcedureResult result = 2;  
}
```

The fields are:

- *id* - The identifier of the stream. This is the value returned by *AddStream* when the stream is created.
- *result* - A *ProcedureResult* message containing the result of the streams RPC. This is identical to the *ProcedureResult* message returned when calling the RPC directly. See *Anatomy of a Response* for details on the format and contents of this message.

### 11.3.3 Events

Events are a wrapper over a stream that returns a boolean value. The event is triggered when the stream returns to true. Remote procedures that return an event return an *Event* message that contains a *Stream* message describing the underlying stream for the event.

The format for an `Event` message is simply the following:

```
message Event {
  repeated Stream stream = 1;
}
```

### 11.3.4 KRPC Service

The server provides a service called KRPC containing procedures that are used to retrieve information about the server and to manage streams.

#### GetStatus

The `GetStatus` procedure returns status information about the server. It returns a Protocol Buffer message with the format:

```
message Status {
  string version = 1;
  uint64 bytes_read = 2;
  uint64 bytes_written = 3;
  float bytes_read_rate = 4;
  float bytes_written_rate = 5;
  uint64 rpcs_executed = 6;
  float rpc_rate = 7;
  bool one_rpc_per_update = 8;
  uint32 max_time_per_update = 9;
  bool adaptive_rate_control = 10;
  bool blocking_recv = 11;
  uint32 recv_timeout = 12;
  float time_per_rpc_update = 13;
  float poll_time_per_rpc_update = 14;
  float exec_time_per_rpc_update = 15;
  uint32 stream_rpcs = 16;
  uint64 stream_rpcs_executed = 17;
  float stream_rpc_rate = 18;
  float time_per_stream_update = 19;
}
```

The `version` field contains the version string of the server. The remaining fields contain performance information about the server.

#### GetServices

The `GetServices` procedure returns a Protocol Buffer message containing information about all of the services and procedures provided by the server. The format of the message is:

```
message Services {
  repeated Service services = 1;
}
```

This contains a single field, which is a list of `Service` messages with information about each service provided by the server. The content of these `Service` messages are *documented below*.

## AddStream

The `AddStream` procedure adds a new stream to the server. Its first argument is the RPC to invoke, encoded as a `ProcedureCall` message. The second argument is a boolean value indicating whether the stream should start sending data to the client immediately. The procedure returns a `Stream` message describing the stream that was added. See *Anatomy of a Request* for the format and contents of this message. See *Streams* for more information on working with streams.

## StartStream

The `StartStream` procedure starts a stream sending data to a client, if it has not yet been started. It takes a single argument – the identifier of the stream to start. This is the identifier returned when the stream was added by calling `AddStream`. See *Streams* for more information on working with streams.

## RemoveStream

The `RemoveStream` procedure removes a stream from the server. It takes a single argument – the identifier of the stream to be removed. This is the identifier returned when the stream was added by calling `AddStream`. See *Streams* for more information on working with streams.

## 11.3.5 Service Description Message

The `GetServices` procedure returns information about all of the services provided by the server. Details about a service are given by a `Service` message, with the format:

```
message Service {  
  string name = 1;  
  repeated Procedure procedures = 2;  
  repeated Class classes = 3;  
  repeated Enumeration enumerations = 4;  
  repeated Exception exceptions = 5;  
  string documentation = 6;  
}
```

The fields are:

- `name` - The name of the service.
- `procedures` - A list of `Procedure` messages, one for each procedure defined by the service.
- `classes` - A list of `Class` messages, one for each *KRPCClass* defined by the service.
- `enumerations` - A list of `Enumeration` messages, one for each *KRPCEnum* defined by the service.
- `exceptions` - A list of `Exception` messages, one for each *KRPCException* defined by the service.
- `documentation` - Documentation for the service, as [C# XML documentation](#).

---

**Note:** See the *Extending kRPC* documentation for more details about *KRPCClass*, *KRPCEnum* and *KRPCException*.

---

## Procedures

Details about a procedure are given by a `Procedure` message, with the format:

```
message Procedure {
  string name = 1;
  repeated Parameter parameters = 2;
  Type return_type = 3;
  string documentation = 4;
}

message Parameter {
  string name = 1;
  Type type = 2;
  bytes default_value = 3;
}
```

The fields are:

- `name` - The name of the procedure. See *Procedure Names* for more information.
- `parameters` - A list of `Parameter` messages containing details of the procedure's parameters, with the following fields:
  - `name` - The name of the parameter, to allow parameter passing by name.
  - `type` - The *type* of the parameter.
  - `default_value` - The value of the default value of the parameter, if any, *encoded using Protocol Buffer format*.
- `return_type` - The *return type* of the procedure. If the procedure does not return anything its type is set to `NONE`.
- `documentation` - Documentation for the procedure, as [C# XML documentation](#).

## Classes

Details about each `KRPCClass` are specified in a `Class` message, with the format:

```
message Class {
  string name = 1;
  string documentation = 2;
}
```

The fields are:

- `name` - The name of the class.
- `documentation` - Documentation for the class, as [C# XML documentation](#).

## Enumerations

Details about each `KRPCEnum` are specified in an `Enumeration` message, with the format:

```
message Enumeration {
  string name = 1;
  repeated EnumerationValue values = 2;
  string documentation = 3;
}
```

```
}  
  
message EnumerationValue {  
    string name = 1;  
    int32 value = 2;  
    string documentation = 3;  
}
```

The fields are:

- name - The name of the enumeration.
- values - A list of `EnumerationValue` messages, indicating the values that the enumeration can be assigned. The fields are:
  - name - The name associated with the value for the enumeration.
  - value - The possible value for the enumeration as a 32-bit integer.
  - documentation - Documentation for the enumeration value, as [C# XML documentation](#).
- documentation - Documentation for the enumeration, as [C# XML documentation](#).

## Exceptions

Details about each *KRPCException* are specified in an `Exception` message, with the format:

```
message Exception {  
    string name = 1;  
    string documentation = 2;  
}
```

The fields are:

- name - The name of the exception type.
- documentation - Documentation for the exception, as [C# XML documentation](#).

### 11.3.6 Procedure Names

Procedures names are CamelCase. Whether a procedure is a service procedure, class method, class property, and what class (if any) it belongs to is determined by its name:

- `ProcedureName` - a standard procedure that is just part of a service.
- `get_PropertyName` - a procedure that returns the value of a property in a service.
- `set_PropertyName` - a procedure that sets the value of a property in a service.
- `ClassName_MethodName` - a class method.
- `ClassName_static_StaticMethodName` - a static class method.
- `ClassName_get_PropertyName` - a class property getter.
- `ClassName_set_PropertyName` - a class property setter.

Only letters and numbers are permitted in class, method and property names. Underscores can therefore be used to split the name into its constituent parts.

### 11.3.7 Type

The `GetServices` procedure returns type information about parameters, return values and others as `Type` messages. The format of these messages is as follows:

```
message Type {
  TypeCode code = 1;
  string service = 2;
  string name = 3;
  repeated Type types = 4;

  enum TypeCode {
    NONE = 0;

    // Values
    DOUBLE = 1;
    FLOAT = 2;
    SINT32 = 3;
    SINT64 = 4;
    UINT32 = 5;
    UINT64 = 6;
    BOOL = 7;
    STRING = 8;
    BYTES = 9;

    // Objects
    CLASS = 100;
    ENUMERATION = 101;

    // Messages
    PROCEDURE_CALL = 200;
    STREAM = 201;
    STATUS = 202;
    SERVICES = 203;

    // Collections
    TUPLE = 300;
    LIST = 301;
    SET = 302;
    DICTIONARY = 303;
  };
}
```

The `code` field specifies the type. `NONE` is used as the return type for procedures that do not return a value.

For `CLASS` and `ENUMERATION` types the `service` and `name` fields specify the service that defines the class/enumeration and the name of the class/enumeration. For all other types these fields are empty.

For collection types the `types` repeated field will contain the sub-types:

- `TUPLE` types contain 1 or more types in the `types` field.
- `LIST` and `SET` types contain a single type in the `types` field.
- `DICTIONARY` types contain a 2 types in the `types` field - the key and value types, in that order.

For all other types the `types` field is empty.

### 11.3.8 Proxy Objects

kRPC allows procedures to create objects on the server, and pass a unique identifier for them to the client. This allows the client to create a *proxy* object for the actual object, whose methods and properties make remote procedure calls to the server. Object identifiers have type `uint64`.

When a procedure returns a proxy object or takes one as a parameter, the type code will be set to `CLASS`.



## INTERNALS OF KRPC

### 12.1 Server Performance Settings

kRPC processes its queue of remote procedures when its `FixedUpdate` method is invoked. This is called every fixed framerate frame, typically about 60 times a second. If kRPC were to only execute one RPC per `FixedUpdate`, it would only be able to execute at most 60 RPCs per second. In order to achieve a higher RPC throughput, it can execute multiple RPCs per `FixedUpdate`. However, if it is allowed to process too many RPCs per `FixedUpdate`, the game's framerate would be adversely affected. The following settings control this behavior, and the resulting tradeoff between RPC throughput and game FPS:

1. **One RPC per update.** When this is enabled, the server will execute at most one RPC per client per update. This will have minimal impact on the game's framerate, while still allowing kRPC to execute RPCs. If you don't need a high RPC throughput, this is a good option to use.
2. **Maximum time per update.** When one RPC per update is not enabled, this setting controls the maximum amount of time (in nanoseconds) that kRPC will spend executing RPCs per `FixedUpdate`. Setting this to a high value, for example 20000 ns, will allow the server to process many RPCs at the expense of the game's framerate. A low value, for example 1000 ns, won't al-

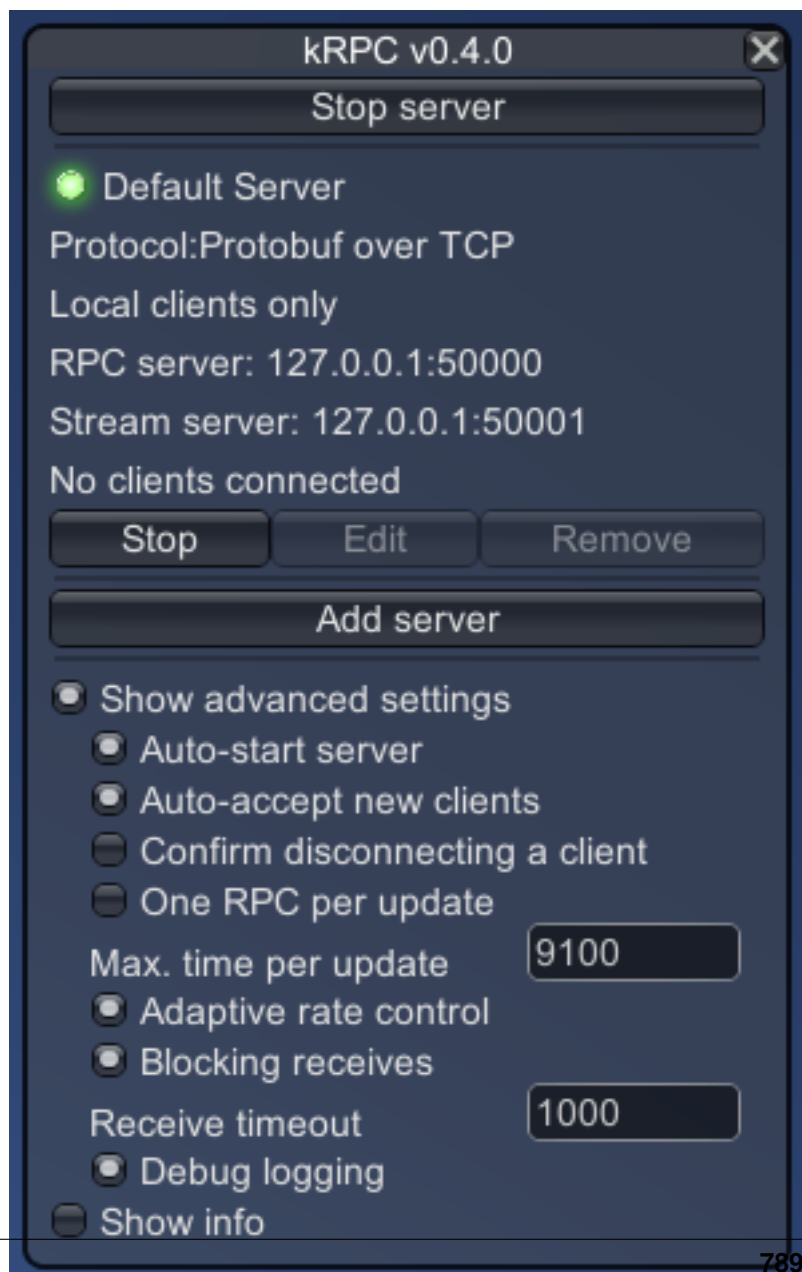


Fig. 12.1: Server window showing the advanced settings.

low the server to execute many RPCs per update, but will allow the game to run at a much higher framerate.

3. **Adaptive rate control.** When enabled, kRPC will automatically adjust the maximum time per update parameter, so that the game has a minimum framerate of 60 FPS. Enabling this setting provides a good tradeoff between RPC throughput and the game framerate.

Another consideration is the responsiveness of the server. Clients must execute RPCs in sequence, one after another, and there is usually a (short) delay between them. This means that when the server finishes executing an RPC, if it were to immediately check for a new RPC it will not find any and will return from the FixedUpdate. This means that any new RPCs will have to wait until the next FixedUpdate, and results in the server only executing a single RPC per FixedUpdate regardless of the maximum time per update setting.

Instead, higher RPC throughput can be obtained if the server waits briefly after finishing an RPC to see if any new RPCs are received. This is done in such a way that the maximum time per update setting (above) is still observed.

This behavior is enabled by the **blocking receives** option. **Receive timeout** sets the maximum amount of time the server will wait for a new RPC from a client.

## PYTHON MODULE INDEX

### d

Drawing, ??

### i

InfernalRobotics, ??

### k

KerbalAlarmClock, ??

KRPC, ??

### r

RemoteTech, ??

### s

SpaceCenter, ??

### u

UI, ??



## LUA MODULE INDEX

### d

Drawing, ??

### i

InfernalRobotics, ??

### k

KerbalAlarmClock, ??

krpc, ??

KRPC, ??

### r

RemoteTech, ??

### s

SpaceCenter, ??

### u

UI, ??