kRPC

Release 0.4.8

Oct 28, 2018

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 KSPs 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*.

CHAPTER

ONE

GETTING STARTED

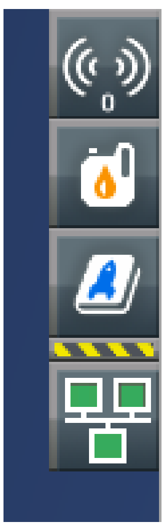
This short guide explains the basics for getting the kRPC server set up and running, and writing a basic Python script to interact with the game.

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.

CHAPTER TWO

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:

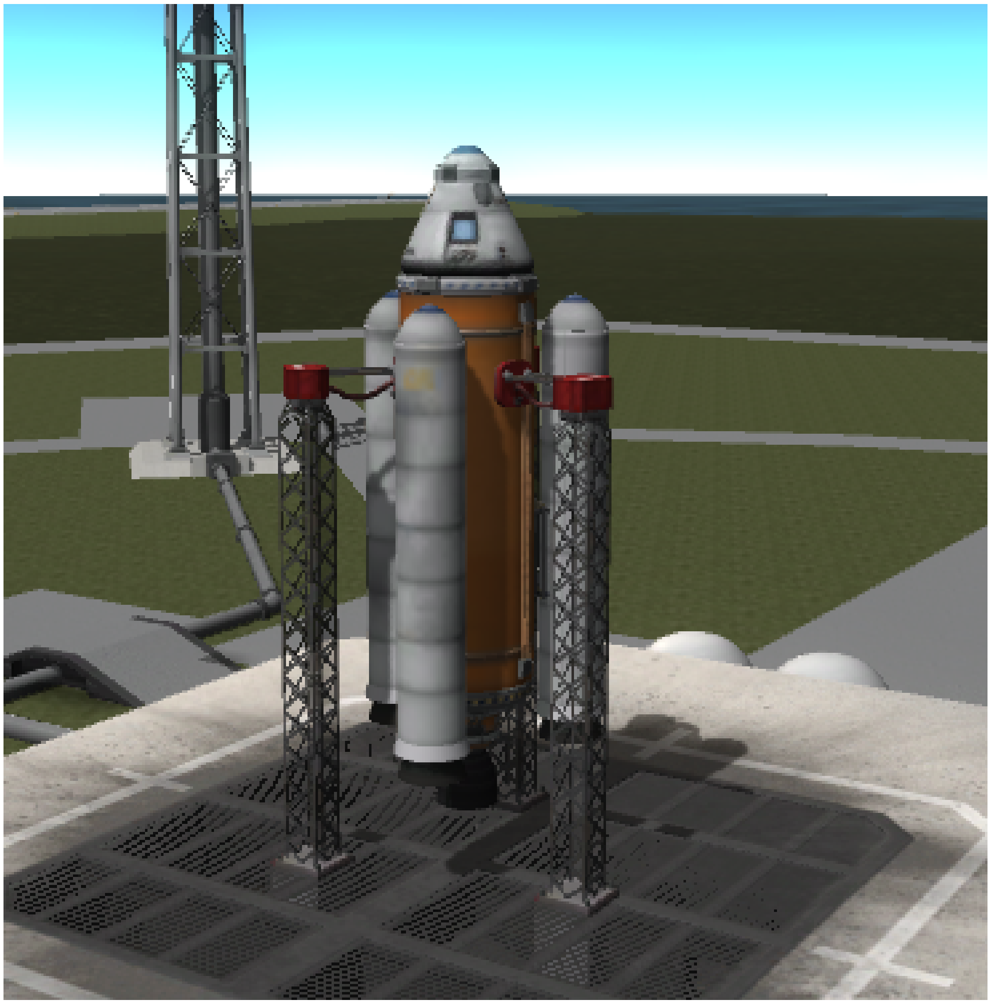
- Controlling a rocket (activating stages, setting the throttle)
- Using the auto pilot to point the vessel in a specific direction
- Using events to wait for things to happen in game
- Tracking the amount of resources in the vessel
- Tracking flight and orbital data (such as altitude and apoapsis altitude)

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

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

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

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

```
local krpc = require 'krpc'
local platform = require 'krpc.platform'
local conn = krpc.connect('Sub-orbital flight')
```

```
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

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

```
  auto vessel = space_center.active_vessel();
```

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

```
local vessel = conn.space_center.active_vessel
```

```
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

```
        vessel.AutoPilot.TargetPitchAndHeading (90, 90);
        vessel.AutoPilot.Engage ();
        vessel.Control.Throttle = 1;
        System.Threading.Thread.Sleep (1000);
```

```
  vessel.auto_pilot().target_pitch_and_heading(90, 90);
  vessel.auto_pilot().engage();
  vessel.control().set_throttle(1);
  std::this_thread::sleep_for(std::chrono::seconds(1));
```

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

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

```
vessel.auto_pilot.target_pitch_and_heading(90, 90)
vessel.auto_pilot.engage()
vessel.control.throttle = 1
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

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

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

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

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

```
print('Launch!')
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

```
            var solidFuel = Connection.GetCall(() => vessel.Resources.Amount("SolidFuel"));
            var expr = Expression.LessThan(
                conn, Expression.Call(conn, solidFuel), Expression.ConstantFloat(conn, 0.1f));
            var evnt = conn.KRPC().AddEvent(expr);
            lock (evnt.Condition) {
                evnt.Wait();
            }
```

```
    auto solid_fuel = vessel.resources().amount_call("SolidFuel");
    auto expr = Expr::less_than(
      conn, Expr::call(conn, solid_fuel), Expr::constant_float(conn, 0.1));
    auto event = krpc.add_event(expr);
    event.acquire();
    event.wait();
    event.release();
```

```
    {
      ProcedureCall solidFuel = connection.getCall(vessel.getResources(), "amount", "SolidFuel");
      Expression expr = Expression.lessThan(
        connection,
        Expression.call(connection, solidFuel),
        Expression.constantFloat(connection, 0.1f));
      Event event = krpc.addEvent(expr);
      synchronized (event.getCondition()) {
        event.waitFor();
      }
    }

    System.out.println("Booster separation");
    vessel.getControl().activateNextStage();
```

```
while vessel.resources:amount('SolidFuel') > 0.1 do
    platform.sleep(1)
end
print('Booster separation')
vessel.control:activate_next_stage()
```

```
fuel_amount = conn.get_call(vessel.resources.amount, 'SolidFuel')
expr = conn.krpc.Expression.less_than(
    conn.krpc.Expression.call(fuel_amount),
    conn.krpc.Expression.constant_float(0.1))
event = conn.krpc.add_event(expr)
with event.condition:
    event.wait()
print('Booster separation')
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

```
            var meanAltitude = Connection.GetCall(() => vessel.Flight(null).
↪MeanAltitude);
            var expr = Expression.GreaterThan(
                conn, Expression.Call(conn, meanAltitude), Expression.
↪ConstantDouble(conn, 10000));
            var evnt = conn.KRPC().AddEvent(expr);
            lock (evnt.Condition) {
                evnt.Wait();
            }
```

```
    auto mean_altitude = vessel.flight().mean_altitude_call();
    auto expr = Expr::greater_than(
      conn, Expr::call(conn, mean_altitude), Expr::constant_double(conn, 10000));
    auto event = krpc.add_event(expr);
    event.acquire();
    event.wait();
    event.release();
```

```
      ProcedureCall meanAltitude = connection.getCall(vessel.flight(null),
↪"getMeanAltitude");
      Expression expr = Expression.greaterThan(
        connection,
        Expression.call(connection, meanAltitude),
        Expression.constantDouble(connection, 10000));
      Event event = krpc.addEvent(expr);
      synchronized (event.getCondition()) {
        event.waitFor();
      }
```

```
while vessel:flight().mean_altitude < 10000 do
    platform.sleep(1)
end
```

```
mean_altitude = conn.get_call(getattr, vessel.flight(), 'mean_altitude')
expr = conn.krpc.Expression.greater_than(
    conn.krpc.Expression.call(mean_altitude),
    conn.krpc.Expression.constant_double(10000))
event = conn.krpc.add_event(expr)
with event.condition:
    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

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

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

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

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

```
print('Gravity turn')
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

```
        {
            var apoapsisAltitude = Connection.GetCall(() => vessel.Orbit.
↪ApoapsisAltitude);
            var expr = Expression.GreaterThan(
                conn, Expression.Call(conn, apoapsisAltitude), Expression.
↪ConstantDouble(conn, 100000));
            var evnt = conn.KRPC().AddEvent(expr);
            lock (evnt.Condition) {
                evnt.Wait();
            }
        }

        Console.WriteLine ("Launch stage separation");
        vessel.Control.Throttle = 0;
        System.Threading.Thread.Sleep (1000);
        vessel.Control.ActivateNextStage ();
        vessel.AutoPilot.Disengage ();
```

```cpp
  {
    auto apoapsis_altitude = vessel.orbit().apoapsis_altitude_call();
    auto expr = Expr::greater_than(
      conn, Expr::call(conn, apoapsis_altitude), Expr::constant_double(conn, 100000));
    auto event = krpc.add_event(expr);
    event.acquire();
    event.wait();
    event.release();
  }

  std::cout << "Launch stage separation" << std::endl;
  vessel.control().set_throttle(0);
  std::this_thread::sleep_for(std::chrono::seconds(1));
  vessel.control().activate_next_stage();
  vessel.auto_pilot().disengage();
```

```java
    {
      ProcedureCall apoapsisAltitude = connection.getCall(
        vessel.getOrbit(), "getApoapsisAltitude");
      Expression expr = Expression.greaterThan(
        connection,
        Expression.call(connection, apoapsisAltitude),
        Expression.constantDouble(connection, 100000));
      Event event = krpc.addEvent(expr);
      synchronized (event.getCondition()) {
        event.waitFor();
      }
    }

    System.out.println("Launch stage separation");
    vessel.getControl().setThrottle(0);
    Thread.sleep(1000);
    vessel.getControl().activateNextStage();
    vessel.getAutoPilot().disengage();
```

```lua
while vessel.orbit.apoapsis_altitude < 100000 do
    platform.sleep(1)
end
print('Launch stage separation')
vessel.control.throttle = 0
platform.sleep(1)
vessel.control:activate_next_stage()
vessel.auto_pilot:disengage()
```

```python
apoapsis_altitude = conn.get_call(getattr, vessel.orbit, 'apoapsis_altitude')
expr = conn.krpc.Expression.greater_than(
    conn.krpc.Expression.call(apoapsis_altitude),
    conn.krpc.Expression.constant_double(100000))
event = conn.krpc.add_event(expr)
with event.condition:
    event.wait()

print('Launch stage separation')
vessel.control.throttle = 0
time.sleep(1)
vessel.control.activate_next_stage()
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

```
        {
            var srfAltitude = Connection.GetCall(() => vessel.Flight(null).SurfaceAltitude);
            var expr = Expression.LessThan(
                conn, Expression.Call(conn, srfAltitude), Expression.ConstantDouble(conn, 1000));
            var evnt = conn.KRPC().AddEvent(expr);
            lock (evnt.Condition) {
                evnt.Wait();
            }
        }

        vessel.Control.ActivateNextStage ();
```

```
  {
    auto srf_altitude = vessel.flight().surface_altitude_call();
    auto expr = Expr::less_than(
      conn, Expr::call(conn, srf_altitude), Expr::constant_double(conn, 1000));
    auto event = krpc.add_event(expr);
    event.acquire();
    event.wait();
    event.release();
  }

  vessel.control().activate_next_stage();
```

```
    {
      ProcedureCall srfAltitude = connection.getCall(
        vessel.flight(null), "getSurfaceAltitude");
      Expression expr = Expression.lessThan(
        connection,
        Expression.call(connection, srfAltitude),
        Expression.constantDouble(connection, 1000));
      Event event = krpc.addEvent(expr);
      synchronized (event.getCondition()) {
        event.waitFor();
      }
    }
```

(continues on next page)

(continued from previous page)

```

    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();
```

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

```python
while vessel.flight(vessel.orbit.body.reference_frame).vertical_speed < -0.1:
    print('Altitude = %.1f meters' % vessel.flight().surface_altitude)
    time.sleep(1)
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

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:

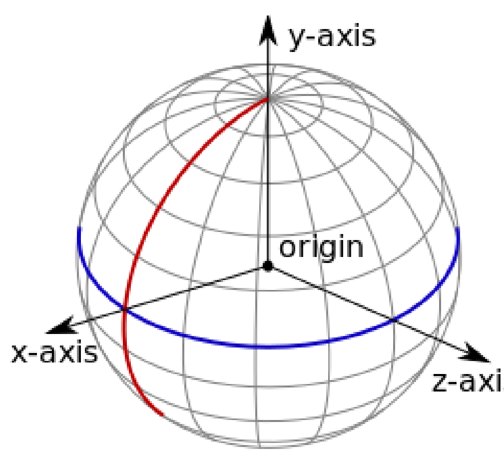


Fig. 1: The reference frame for a celestial body, such as Kerbin. The equator is shown in blue, and the prime meridian in red. The black arrows show the coordinate axes, and the origin is at the center of the planet.

C#

C++

C

Java

Lua

Python

```
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;
}
```

```
#include <unistd.h>
#include <math.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Vessel position");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_Orbit_t orbit;
  krpc_SpaceCenter_Vessel_Orbit(conn, &orbit, vessel);
  krpc_SpaceCenter_CelestialBody_t body;
  krpc_SpaceCenter_Orbit_Body(conn, &body, orbit);
  krpc_SpaceCenter_ReferenceFrame_t body_frame;
  krpc_SpaceCenter_CelestialBody_ReferenceFrame(conn, &body_frame, body);

  krpc_tuple_double_double_double_t position;
  krpc_SpaceCenter_Vessel_Position(conn, &position, vessel, body_frame);
  printf("%.2f, %.2f, %.2f\n", position.e0, position.e1, position.e2);
}
```

```java
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();
    }
}
```

```lua
local krpc = require 'krpc'
local conn = krpc.connect()
local vessel = conn.space_center.active_vessel
print(vessel:position(vessel.orbit.body.reference_frame))
```

```python
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,

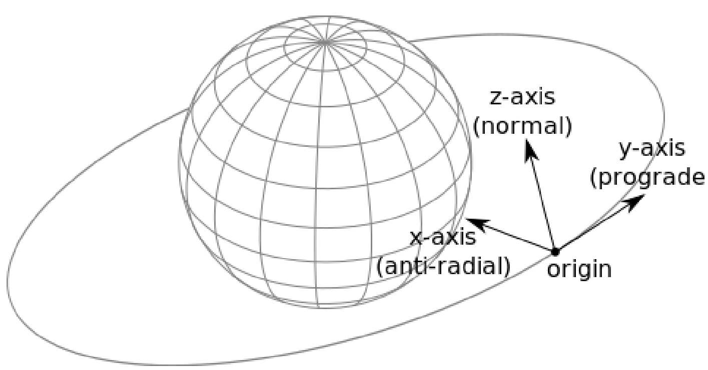


Fig. 2: The orbital reference frame for a vessel.

- 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.

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.

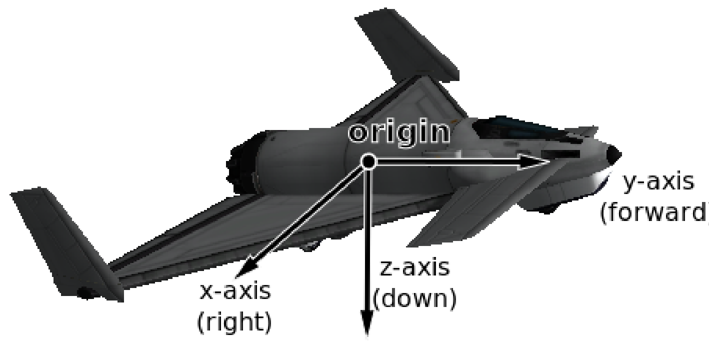


Fig. 3: The reference frame for an aircraft.

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++

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()*
- *krpc_SpaceCenter_Vessel_ReferenceFrame()*
- *krpc_SpaceCenter_Vessel_OrbitalReferenceFrame()*
- *krpc_SpaceCenter_Vessel_SurfaceReferenceFrame()*
- *krpc_SpaceCenter_Vessel_SurfaceVelocityReferenceFrame()*
- *krpc_SpaceCenter_CelestialBody_ReferenceFrame()*
- *krpc_SpaceCenter_CelestialBody_NonRotatingReferenceFrame()*
- *krpc_SpaceCenter_CelestialBody_OrbitalReferenceFrame()*
- *krpc_SpaceCenter_Node_ReferenceFrame()*
- *krpc_SpaceCenter_Node_OrbitalReferenceFrame()*
- *krpc_SpaceCenter_Part_ReferenceFrame()*
- *krpc_SpaceCenter_Part_CenterOfMassReferenceFrame()*

- *krpc_SpaceCenter_DockingPort_ReferenceFrame()*
- *krpc_SpaceCenter_Thruster_ThrustReferenceFrame()*
- *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++

C

Java

Lua

Python

- *SpaceCenter.TransformPosition*
- *SpaceCenter.TransformDirection*
- *SpaceCenter.TransformRotation*
- *SpaceCenter.TransformVelocity*
- *SpaceCenter::transform_position()*
- *SpaceCenter::transform_direction()*

- *SpaceCenter::transform_rotation()*
- *SpaceCenter::transform_velocity()*
- *krpc_SpaceCenter_TransformPosition()*
- *krpc_SpaceCenter_TransformDirection()*
- *krpc_SpaceCenter_TransformRotation()*
- *krpc_SpaceCenter_TransformVelocity()*
- *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++

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 ()
```

(continues on next page)

(continued from previous page)

```
    {
        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) {
  }
}
```

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>
#include <krpc_cnano/services/ui.h>
#include <krpc_cnano/services/drawing.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Visual debugging");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);

  krpc_SpaceCenter_ReferenceFrame_t ref_frame;
  krpc_SpaceCenter_Vessel_SurfaceVelocityReferenceFrame(conn, &ref_frame, vessel);
  krpc_tuple_double_double_double_t direction = { 0, 1, 0 };
  krpc_Drawing_AddDirection(conn, NULL, &direction, ref_frame, 10, true);
  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;
```

(continues on next page)

(continued from previous page)

```java
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) {
        }
    }
}
```

```lua
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
```

```python
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++

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));
```

(continues on next page)

(continued from previous page)

```
  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();
}
```

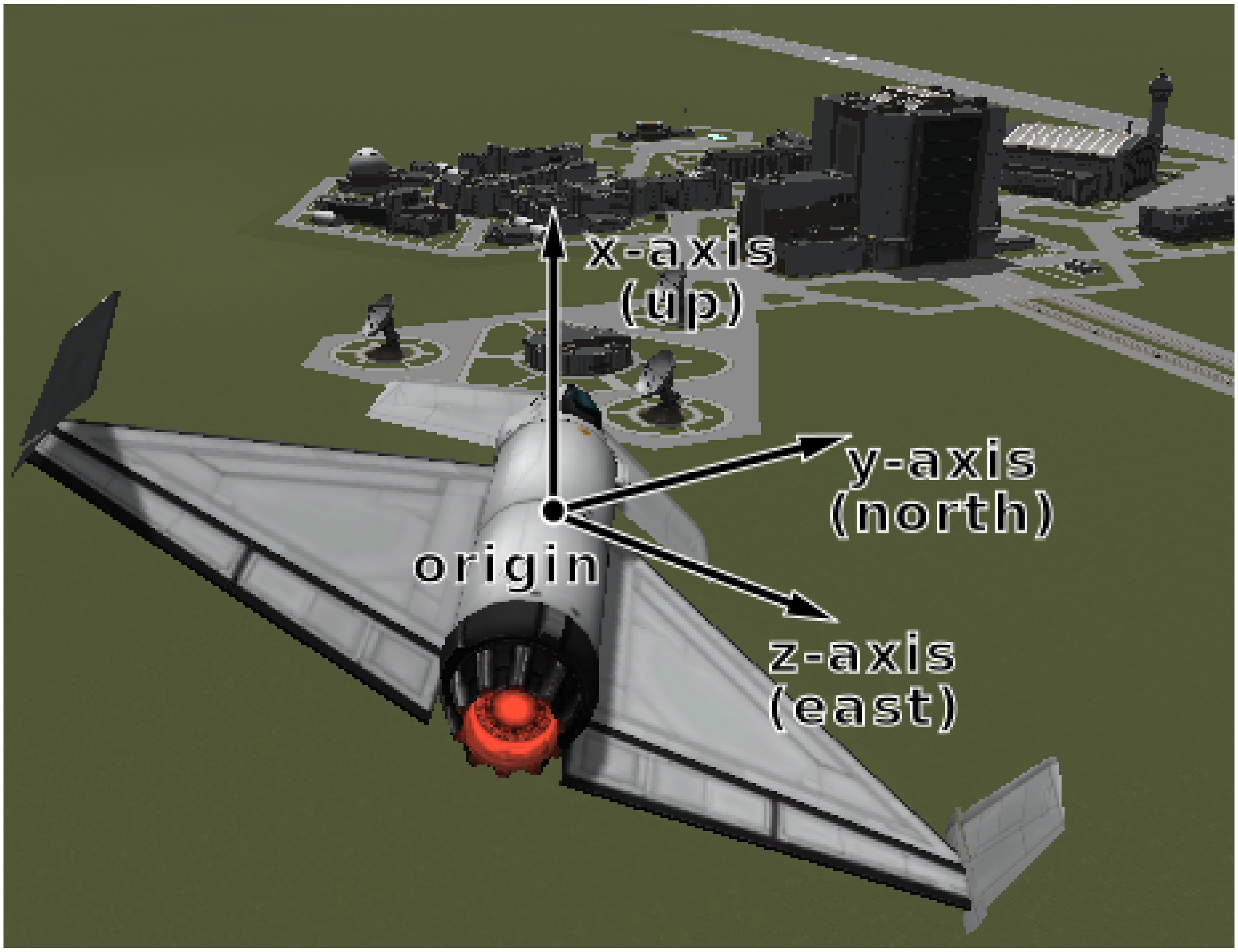
```
#include <unistd.h>
#include <math.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

typedef krpc_tuple_double_double_double_t vector3;

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Navball directions");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_ReferenceFrame_t vessel_srf_ref;
  krpc_SpaceCenter_Vessel_SurfaceReferenceFrame(conn, &vessel_srf_ref, vessel);

  krpc_SpaceCenter_AutoPilot_t ap;
  krpc_SpaceCenter_Vessel_AutoPilot(conn, &ap, vessel);
  krpc_SpaceCenter_AutoPilot_set_ReferenceFrame(conn, ap, vessel_srf_ref);
  krpc_SpaceCenter_AutoPilot_Engage(conn, ap);

  // Point the vessel north on the navball, with a pitch of 0 degrees
  {
    krpc_tuple_double_double_double_t direction = { 0, 1, 0 };
    krpc_SpaceCenter_AutoPilot_set_TargetDirection(conn, ap, &direction);
    krpc_SpaceCenter_AutoPilot_Wait(conn, ap);
  }

  // Point the vessel vertically upwards on the navball
  {
    krpc_tuple_double_double_double_t direction = { 1, 0, 0 };
    krpc_SpaceCenter_AutoPilot_set_TargetDirection(conn, ap, &direction);
    krpc_SpaceCenter_AutoPilot_Wait(conn, ap);
  }

  // Point the vessel west (heading of 270 degrees), with a pitch of 0 degrees
  {
    krpc_tuple_double_double_double_t direction = { 0, 0, -1 };
    krpc_SpaceCenter_AutoPilot_set_TargetDirection(conn, ap, &direction);
    krpc_SpaceCenter_AutoPilot_Wait(conn, ap);
  }

  krpc_SpaceCenter_AutoPilot_Disengage(conn, ap);
}
```

```java
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();
    }
}
```

```lua
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()
```

```python
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++

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>

```

(continues on next page)

(continued from previous page)

```
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();
}
```

```
#include <unistd.h>
#include <math.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

typedef krpc_tuple_double_double_double_t vector3;

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Orbital directions");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_ReferenceFrame_t vessel_obt_ref;
  krpc_SpaceCenter_Vessel_OrbitalReferenceFrame(conn, &vessel_obt_ref, vessel);

  krpc_SpaceCenter_AutoPilot_t ap;
  krpc_SpaceCenter_Vessel_AutoPilot(conn, &ap, vessel);
  krpc_SpaceCenter_AutoPilot_set_ReferenceFrame(conn, ap, vessel_obt_ref);
  krpc_SpaceCenter_AutoPilot_Engage(conn, ap);

  // Point the vessel in the prograde direction
  {
    krpc_tuple_double_double_double_t direction = { 0, 1, 0 };
    krpc_SpaceCenter_AutoPilot_set_TargetDirection(conn, ap, &direction);
    krpc_SpaceCenter_AutoPilot_Wait(conn, ap);
  }

  // Point the vessel in the orbit normal direction
  {
    krpc_tuple_double_double_double_t direction = { 0, 0, 1 };
    krpc_SpaceCenter_AutoPilot_set_TargetDirection(conn, ap, &direction);
```

(continues on next page)

(continued from previous page)

```
    krpc_SpaceCenter_AutoPilot_Wait(conn, ap);
  }

  // Point the vessel in the orbit radial direction
  {
    krpc_tuple_double_double_double_t direction = { -1, 0, 0 };
    krpc_SpaceCenter_AutoPilot_set_TargetDirection(conn, ap, &direction);
    krpc_SpaceCenter_AutoPilot_Wait(conn, ap);
  }

  krpc_SpaceCenter_AutoPilot_Disengage(conn, ap);
}
```

```
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()
```

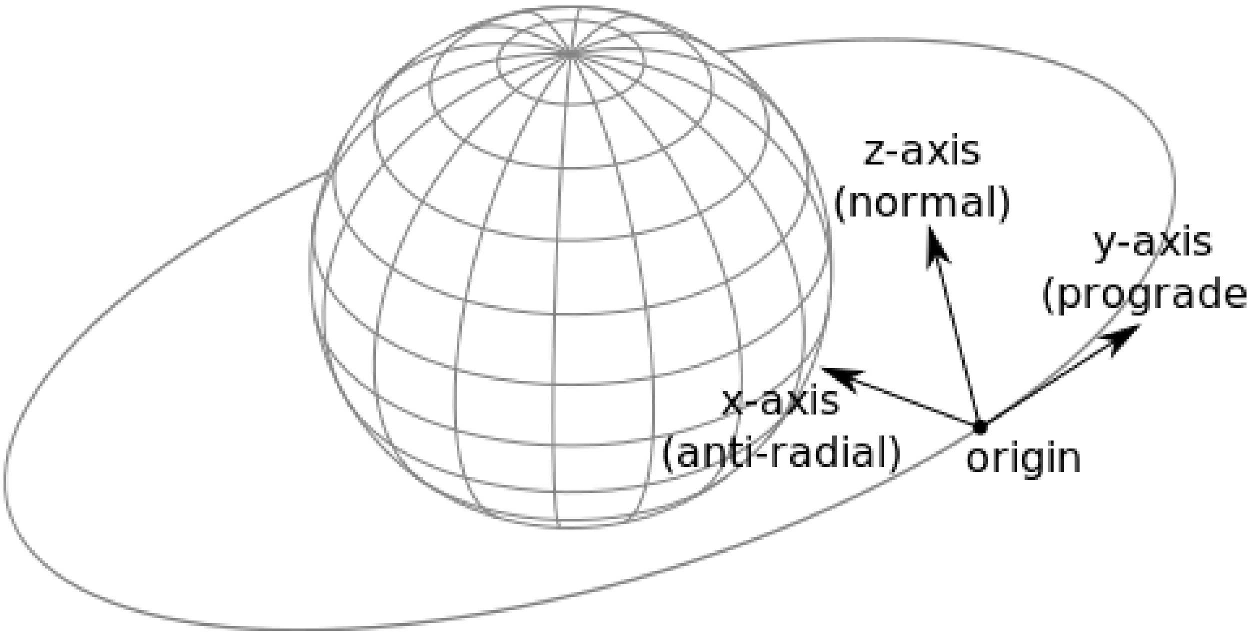
(continues on next page)

(continued from previous page)

```
-- 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++

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 ();
        }
    }
}
```

```cpp
#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();
}
```

```c
#include <unistd.h>
#include <math.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

typedef krpc_tuple_double_double_double_t vector3;

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Surface prograde");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_ReferenceFrame_t vessel_srf_vel_ref;
  krpc_SpaceCenter_Vessel_SurfaceVelocityReferenceFrame(conn, &vessel_srf_vel_ref,␣
↪vessel);

  krpc_SpaceCenter_AutoPilot_t ap;
  krpc_SpaceCenter_Vessel_AutoPilot(conn, &ap, vessel);

  krpc_SpaceCenter_AutoPilot_set_ReferenceFrame(conn, ap, vessel_srf_vel_ref);
  krpc_tuple_double_double_double_t direction = { 0, 1, 0 };
  krpc_SpaceCenter_AutoPilot_set_TargetDirection(conn, ap, &direction);
  krpc_SpaceCenter_AutoPilot_Engage(conn, ap);
  krpc_SpaceCenter_AutoPilot_Wait(conn, ap);
  krpc_SpaceCenter_AutoPilot_Disengage(conn, ap);
}
```

```java
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 {
```

(continues on next page)

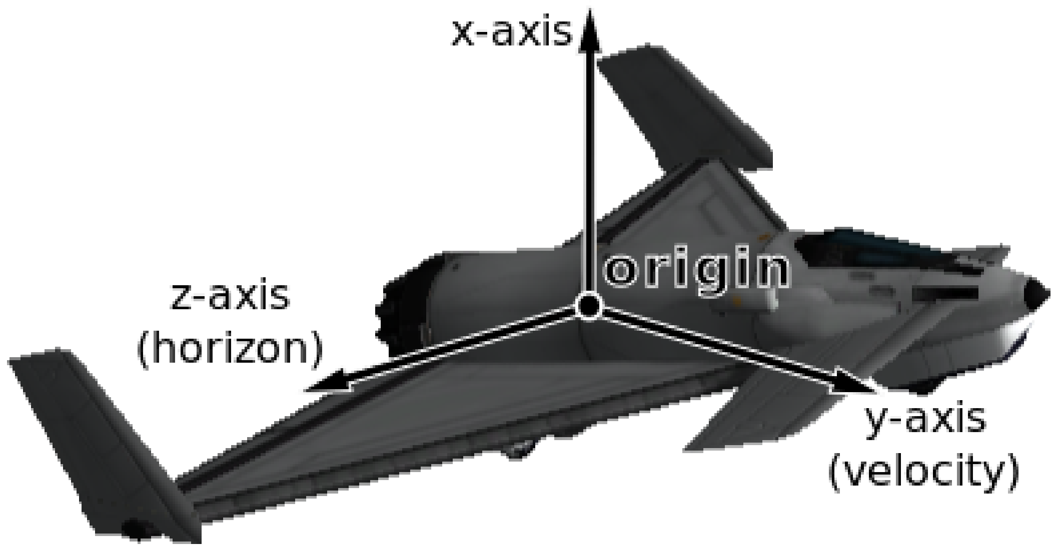
(continued from previous page)

```
    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++

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 (
```

(continues on next page)

(continued from previous page)

```
                "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));
  }
}
```

```
#include <unistd.h>
#include <math.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

typedef krpc_tuple_double_double_double_t vector3;

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Vessel speed");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_Orbit_t orbit;
  krpc_SpaceCenter_Vessel_Orbit(conn, &orbit, vessel);
  krpc_SpaceCenter_CelestialBody_t body;
  krpc_SpaceCenter_Orbit_Body(conn, &body, orbit);

  krpc_SpaceCenter_ReferenceFrame_t obt_frame;
  krpc_SpaceCenter_ReferenceFrame_t srf_frame;
  krpc_SpaceCenter_CelestialBody_NonRotatingReferenceFrame(conn, &obt_frame, body);
  krpc_SpaceCenter_CelestialBody_ReferenceFrame(conn, &srf_frame, body);

  while (true) {
    double obt_speed;
```

(continues on next page)

(continued from previous page)

```
    double srf_speed;
    krpc_SpaceCenter_Flight_t flight;
    krpc_SpaceCenter_Vessel_Flight(conn, &flight, vessel, obt_frame);
    krpc_SpaceCenter_Flight_Speed(conn, &obt_speed, flight);
    krpc_SpaceCenter_Vessel_Flight(conn, &flight, vessel, srf_frame);
    krpc_SpaceCenter_Flight_Speed(conn, &srf_speed, flight);
    printf("Orbital speed = %.1f m/s, Surface speed = %.1f m/s\n", obt_speed, srf_
↪speed);
    sleep(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)
```

(continues on next page)

(continued from previous page)

```
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++

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) {
```

(continues on next page)

(continued from previous page)

```
            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));
  }
}
```

```
#include <unistd.h>
#include <math.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

typedef krpc_tuple_double_double_double_t vector3;

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Vessel velocity");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_Orbit_t orbit;
  krpc_SpaceCenter_Vessel_Orbit(conn, &orbit, vessel);
  krpc_SpaceCenter_CelestialBody_t body;
  krpc_SpaceCenter_Orbit_Body(conn, &body, orbit);
```

(continues on next page)

(continued from previous page)

```
  krpc_SpaceCenter_ReferenceFrame_t vessel_srf_frame;
  krpc_SpaceCenter_Vessel_SurfaceReferenceFrame(conn, &vessel_srf_frame, vessel);
  krpc_SpaceCenter_ReferenceFrame_t body_frame;
  krpc_SpaceCenter_CelestialBody_ReferenceFrame(conn, &body_frame, body);

  krpc_SpaceCenter_ReferenceFrame_t ref_frame;
  krpc_SpaceCenter_ReferenceFrame_CreateHybrid(
    conn, &ref_frame, body_frame, vessel_srf_frame, KRPC_NULL, KRPC_NULL);

  while (true) {
    krpc_tuple_double_double_double_t velocity;
    krpc_SpaceCenter_Flight_t flight;
    krpc_SpaceCenter_Vessel_Flight(conn, &flight, vessel, ref_frame);
    krpc_SpaceCenter_Flight_Velocity(conn, &velocity, flight);
    printf("Surface velocity = %.1f, %.1f, %.1f\n", velocity.e0, velocity.e1,␣
↪velocity.e2);
    sleep(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'
```

(continues on next page)

(continued from previous page)

```
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++

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);
```

(continues on next page)

(continued from previous page)

```
            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
```

(continues on next page)

(continued from previous page)

```
    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));
  }
}
```

```
#include <unistd.h>
#include <math.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

typedef krpc_tuple_double_double_double_t vector3;

static double pi = 3.1415926535897;

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Angle of attack");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_Orbit_t orbit;
  krpc_SpaceCenter_Vessel_Orbit(conn, &orbit, vessel);
  krpc_SpaceCenter_CelestialBody_t body;
  krpc_SpaceCenter_Orbit_Body(conn, &body, orbit);
  krpc_SpaceCenter_ReferenceFrame_t body_ref;
  krpc_SpaceCenter_CelestialBody_ReferenceFrame(conn, &body_ref, body);

  while (true) {
    vector3 d;
    vector3 v;
    krpc_SpaceCenter_Vessel_Direction(conn, &d, vessel, body_ref);
    krpc_SpaceCenter_Vessel_Velocity(conn, &v, vessel, body_ref);

    // Compute the dot product of d and v
    double dotProd = d.e0*v.e0 + d.e1*v.e1 + d.e2*v.e2;

    // Compute the magnitude of v
    double vMag = sqrt(v.e0*v.e0 + v.e1*v.e1 + v.e2*v.e2);
    // 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));

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

    sleep(1);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

```
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')
```

(continues on next page)

(continued from previous page)

```
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

    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++

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(
```

(continues on next page)

(continued from previous page)

```
                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)
    );
```

(continues on next page)

(continued from previous page)

```
  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));
}
```

```
#include <unistd.h>
#include <math.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>
#include <krpc_cnano/services/ui.h>
#include <krpc_cnano/services/drawing.h>

typedef krpc_tuple_double_double_double_t vector3;
typedef krpc_tuple_double_double_double_double_t quaternion;

static double pi = 3.1415926535897;

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Landing site");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_Orbit_t orbit;
  krpc_SpaceCenter_Vessel_Orbit(conn, &orbit, vessel);
  krpc_SpaceCenter_CelestialBody_t body;
  krpc_SpaceCenter_Orbit_Body(conn, &body, orbit);
  krpc_SpaceCenter_ReferenceFrame_t body_ref;
```

(continues on next page)

(continued from previous page)

```
  krpc_SpaceCenter_CelestialBody_ReferenceFrame(conn, &body_ref, 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)
  vector3 landing_position;
  krpc_SpaceCenter_CelestialBody_SurfacePosition(
    conn, &landing_position, body, landing_latitude, landing_longitude, body_ref);
  quaternion q_long = {
    0.0,
    sin(-landing_longitude * 0.5 * pi / 180.0),
    0.0,
    cos(-landing_longitude * 0.5 * pi / 180.0)
  };
  quaternion q_lat = {
    0.0,
    0.0,
    sin(landing_latitude * 0.5 * pi / 180.0),
    cos(landing_latitude * 0.5 * pi / 180.0)
  };

  krpc_SpaceCenter_ReferenceFrame_t landing_reference_frame;
  {
    vector3 zero = {0, 0, 0};
    quaternion q_zero = {0, 0, 0, 1};
    krpc_SpaceCenter_ReferenceFrame_t parent_ref;
    krpc_SpaceCenter_ReferenceFrame_CreateRelative(
      conn, &parent_ref, body_ref, &landing_position, &q_long, &zero, &zero);
    krpc_SpaceCenter_ReferenceFrame_CreateRelative(
      conn, &parent_ref, parent_ref, &zero, &q_lat, &zero, &zero);
    vector3 position = { landing_altitude, 0, 0 };
    krpc_SpaceCenter_ReferenceFrame_CreateRelative(
      conn, &landing_reference_frame, parent_ref, &position, &q_zero, &zero, &zero);
  }

  // Draw axes
  vector3 zero = {0, 0, 0};
  vector3 x_axis = {1, 0, 0};
  vector3 y_axis = {0, 1, 0};
  vector3 z_axis = {0, 0, 1};
  krpc_Drawing_AddLine(conn, NULL, &zero, &x_axis, landing_reference_frame, true);
  krpc_Drawing_AddLine(conn, NULL, &zero, &y_axis, landing_reference_frame, true);
  krpc_Drawing_AddLine(conn, NULL, &zero, &z_axis, landing_reference_frame, true);

  while (true)
    sleep(1);
}
```

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

(continues on next page)

(continued from previous page)

```
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),
```

(continues on next page)

(continued from previous page)

```
            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
```

(continues on next page)

(continued from previous page)

```
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

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

class LaunchIntoOrbit
{
    public static void Main ()
    {
        var conn = new Connection ("Launch into orbit");
        var vessel = conn.SpaceCenter ().ActiveVessel;

        float turnStartAltitude = 250;
        float turnEndAltitude = 45000;
        float targetAltitude = 150000;

        // Set up streams for telemetry
        var ut = conn.AddStream (() => conn.SpaceCenter ().UT);
        var flight = vessel.Flight ();
        var altitude = conn.AddStream (() => flight.MeanAltitude);
        var apoapsis = conn.AddStream (() => vessel.Orbit.ApoapsisAltitude);
        var stage2Resources =
            vessel.ResourcesInDecoupleStage (stage: 2, cumulative: false);
        var srbFuel = conn.AddStream(() => stage2Resources.Amount("SolidFuel"));

        // Pre-launch setup
        vessel.Control.SAS = false;
        vessel.Control.RCS = false;
        vessel.Control.Throttle = 1;

        // Countdown...
        Console.WriteLine ("3...");
        System.Threading.Thread.Sleep (1000);
        Console.WriteLine ("2...");
        System.Threading.Thread.Sleep (1000);
        Console.WriteLine ("1...");
```

(continues on next page)

(continued from previous page)

```
        System.Threading.Thread.Sleep (1000);
        Console.WriteLine ("Launch!");
```

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

int main() {
  krpc::Client conn = krpc::connect("Launch into orbit");
  krpc::services::SpaceCenter space_center(&conn);
  auto vessel = space_center.active_vessel();

  float turn_start_altitude = 250;
  float turn_end_altitude = 45000;
  float target_altitude = 150000;

  // Set up streams for telemetry
  auto ut = space_center.ut_stream();
  auto altitude = vessel.flight().mean_altitude_stream();
  auto apoapsis = vessel.orbit().apoapsis_altitude_stream();
  auto stage_2_resources = vessel.resources_in_decouple_stage(2, false);
  auto srb_fuel = stage_2_resources.amount_stream("SolidFuel");

  // Pre-launch setup
  vessel.control().set_sas(false);
  vessel.control().set_rcs(false);
  vessel.control().set_throttle(1);

  // Countdown...
  std::cout << "3..." << std::endl;
  std::this_thread::sleep_for(std::chrono::seconds(1));
  std::cout << "2..." << std::endl;
  std::this_thread::sleep_for(std::chrono::seconds(1));
  std::cout << "1..." << std::endl;
  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.Flight;
import krpc.client.services.SpaceCenter.Node;
import krpc.client.services.SpaceCenter.ReferenceFrame;
import krpc.client.services.SpaceCenter.Resources;

import org.javatuples.Triplet;

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

public class LaunchIntoOrbit {
    public static void main(String[] args)
```

(continues on next page)

(continued from previous page)

```java
        throws IOException, RPCException, InterruptedException, StreamException {
        Connection connection = Connection.newInstance("Launch into orbit");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        SpaceCenter.Vessel vessel = spaceCenter.getActiveVessel();

        float turnStartAltitude = 250;
        float turnEndAltitude = 45000;
        float targetAltitude = 150000;

        // Set up streams for telemetry
        spaceCenter.getUT();
        Stream<Double> ut = connection.addStream(SpaceCenter.class, "getUT");
        ReferenceFrame refFrame = vessel.getSurfaceReferenceFrame();
        Flight flight = vessel.flight(refFrame);
        Stream<Double> altitude = connection.addStream(flight, "getMeanAltitude");
        Stream<Double> apoapsis =
            connection.addStream(vessel.getOrbit(), "getApoapsisAltitude");
        Resources stage2Resources = vessel.resourcesInDecoupleStage(2, false);
        Stream<Float> srbFuel =
            connection.addStream(stage2Resources, "amount", "SolidFuel");

        // Pre-launch setup
        vessel.getControl().setSAS(false);
        vessel.getControl().setRCS(false);
        vessel.getControl().setThrottle(1);

        // Countdown...
        System.out.println("3...");
        Thread.sleep(1000);
        System.out.println("2...");
        Thread.sleep(1000);
        System.out.println("1...");
        Thread.sleep(1000);
```

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

local turn_start_altitude = 250
local turn_end_altitude = 45000
local target_altitude = 150000

local conn = krpc.connect('Launch into orbit')
local vessel = conn.space_center.active_vessel

flight = vessel:flight()
stage_2_resources = vessel:resources_in_decouple_stage(2, False)

-- Pre-launch setup
vessel.control.sas = false
vessel.control.rcs = false
vessel.control.throttle = 1

-- Countdown...
print('3...')
```

(continues on next page)

(continued from previous page)

```
platform.sleep(1)
print('2...')
platform.sleep(1)
print('1...')
platform.sleep(1)
print('Launch!')
```

```
import math
import time
import krpc

turn_start_altitude = 250
turn_end_altitude = 45000
target_altitude = 150000

conn = krpc.connect(name='Launch into orbit')
vessel = conn.space_center.active_vessel

# Set up streams for telemetry
ut = conn.add_stream(getattr, conn.space_center, 'ut')
altitude = conn.add_stream(getattr, vessel.flight(), 'mean_altitude')
apoapsis = conn.add_stream(getattr, vessel.orbit, 'apoapsis_altitude')
stage_2_resources = vessel.resources_in_decouple_stage(stage=2, cumulative=False)
srb_fuel = conn.add_stream(stage_2_resources.amount, 'SolidFuel')

# Pre-launch setup
vessel.control.sas = False
vessel.control.rcs = False
vessel.control.throttle = 1.0

# Countdown...
print('3...')
time.sleep(1)
print('2...')
time.sleep(1)
print('1...')
time.sleep(1)
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

```
        // Activate the first stage
        vessel.Control.ActivateNextStage ();
        vessel.AutoPilot.Engage ();
        vessel.AutoPilot.TargetPitchAndHeading (90, 90);

```

(continues on next page)

(continued from previous page)

```
        // Main ascent loop
        bool srbsSeparated = false;
        double turnAngle = 0;
        while (true) {

            // Gravity turn
            if (altitude.Get () > turnStartAltitude &&
                altitude.Get () < turnEndAltitude) {
                double frac = (altitude.Get () - turnStartAltitude)
                              / (turnEndAltitude - turnStartAltitude);
                double newTurnAngle = frac * 90.0;
                if (Math.Abs (newTurnAngle - turnAngle) > 0.5) {
                    turnAngle = newTurnAngle;
                    vessel.AutoPilot.TargetPitchAndHeading (
                        (float)(90 - turnAngle), 90);
                }
            }

            // Separate SRBs when finished
            if (!srbsSeparated) {
                if (srbFuel.Get () < 0.1) {
                    vessel.Control.ActivateNextStage ();
                    srbsSeparated = true;
                    Console.WriteLine ("SRBs separated");
                }
            }

            // Decrease throttle when approaching target apoapsis
            if (apoapsis.Get () > targetAltitude * 0.9) {
                Console.WriteLine ("Approaching target apoapsis");
                break;
            }
        }
```

```
  // Activate the first stage
  vessel.control().activate_next_stage();
  vessel.auto_pilot().engage();
  vessel.auto_pilot().target_pitch_and_heading(90, 90);

  // Main ascent loop
  bool srbs_separated = false;
  double turn_angle = 0;
  while (true) {
    // Gravity turn
    if (altitude() > turn_start_altitude && altitude() < turn_end_altitude) {
      double frac = (altitude() - turn_start_altitude)
                    / (turn_end_altitude - turn_start_altitude);
      double new_turn_angle = frac * 90.0;
      if (std::abs(new_turn_angle - turn_angle) > 0.5) {
        turn_angle = new_turn_angle;
        vessel.auto_pilot().target_pitch_and_heading(90.0 - turn_angle, 90.0);
      }
    }

    // Separate SRBs when finished
    if (!srbs_separated) {
```

(continues on next page)

(continued from previous page)

```
      if (srb_fuel() < 0.1) {
        vessel.control().activate_next_stage();
        srbs_separated = true;
        std::cout << "SRBs separated" << std::endl;
      }
    }

    // Decrease throttle when approaching target apoapsis
    if (apoapsis() > target_altitude * 0.9) {
      std::cout << "Approaching target apoapsis" << std::endl;
      break;
    }
```

```
        // Activate the first stage
        vessel.getControl().activateNextStage();
        vessel.getAutoPilot().engage();
        vessel.getAutoPilot().targetPitchAndHeading(90, 90);

        // Main ascent loop
        boolean srbsSeparated = false;
        double turnAngle = 0;
        while (true) {

            // Gravity turn
            if (altitude.get() > turnStartAltitude &&
                altitude.get() < turnEndAltitude) {
                double frac = (altitude.get() - turnStartAltitude)
                              / (turnEndAltitude - turnStartAltitude);
                double newTurnAngle = frac * 90.0;
                if (Math.abs(newTurnAngle - turnAngle) > 0.5) {
                    turnAngle = newTurnAngle;
                    vessel.getAutoPilot().targetPitchAndHeading(
                        (float)(90 - turnAngle), 90);
                }
            }

            // Separate SRBs when finished
            if (!srbsSeparated) {
              if (srbFuel.get() < 0.1) {
                    vessel.getControl().activateNextStage();
                    srbsSeparated = true;
                    System.out.println("SRBs separated");
                }
            }

            // Decrease throttle when approaching target apoapsis
            if (apoapsis.get() > targetAltitude * 0.9) {
                System.out.println("Approaching target apoapsis");
                break;
            }
```

```
-- Activate the first stage
vessel.control:activate_next_stage()
vessel.auto_pilot:engage()
vessel.auto_pilot:target_pitch_and_heading(90, 90)

```

(continues on next page)

(continued from previous page)

```lua
-- Main ascent loop
local srbs_separated = false
local turn_angle = 0
while true do

    -- Gravity turn
    if flight.mean_altitude > turn_start_altitude and flight.mean_altitude < turn_end_
↪altitude then
        frac = (flight.mean_altitude - turn_start_altitude) / (turn_end_altitude -␣
↪turn_start_altitude)
        new_turn_angle = frac * 90
        if math.abs(new_turn_angle - turn_angle) > 0.5 then
            turn_angle = new_turn_angle
            vessel.auto_pilot:target_pitch_and_heading(90-turn_angle, 90)
        end
    end

    -- Separate SRBs when finished
    if not srbs_separated then
        if stage_2_resources:amount('SolidFuel') < 0.1 then
            vessel.control:activate_next_stage()
            srbs_separated = true
            print('SRBs separated')
        end
    end

    -- Decrease throttle when approaching target apoapsis
    if vessel.orbit.apoapsis_altitude > target_altitude*0.9 then
        print('Approaching target apoapsis')
        break
    end
end
```

```python
# Activate the first stage
vessel.control.activate_next_stage()
vessel.auto_pilot.engage()
vessel.auto_pilot.target_pitch_and_heading(90, 90)

# Main ascent loop
srbs_separated = False
turn_angle = 0
while True:

    # Gravity turn
    if altitude() > turn_start_altitude and altitude() < turn_end_altitude:
        frac = ((altitude() - turn_start_altitude) /
                (turn_end_altitude - turn_start_altitude))
        new_turn_angle = frac * 90
        if abs(new_turn_angle - turn_angle) > 0.5:
            turn_angle = new_turn_angle
            vessel.auto_pilot.target_pitch_and_heading(90-turn_angle, 90)

    # Separate SRBs when finished
    if not srbs_separated:
        if srb_fuel() < 0.1:
            vessel.control.activate_next_stage()
```

(continues on next page)

(continued from previous page)

```
            srbs_separated = True
            print('SRBs separated')

    # Decrease throttle when approaching target apoapsis
    if apoapsis() > target_altitude*0.9:
        print('Approaching target apoapsis')
        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

```
        // Disable engines when target apoapsis is reached
        vessel.Control.Throttle = 0.25f;
        while (apoapsis.Get () < targetAltitude) {
        }
        Console.WriteLine ("Target apoapsis reached");
        vessel.Control.Throttle = 0;

        // Wait until out of atmosphere
        Console.WriteLine ("Coasting out of atmosphere");
        while (altitude.Get () < 70500) {
        }
```

```
  // Disable engines when target apoapsis is reached
  vessel.control().set_throttle(0.25);
  while (apoapsis() < target_altitude) {
  }
  std::cout << "Target apoapsis reached" << std::endl;
  vessel.control().set_throttle(0);

  // Wait until out of atmosphere
  std::cout << "Coasting out of atmosphere" << std::endl;
  while (altitude() < 70500) {
```

```
        // Disable engines when target apoapsis is reached
        vessel.getControl().setThrottle(0.25f);
        while (apoapsis.get() < targetAltitude) {
        }
        System.out.println("Target apoapsis reached");
        vessel.getControl().setThrottle(0);

        // Wait until out of atmosphere
        System.out.println("Coasting out of atmosphere");
        while (altitude.get() < 70500) {
```

```
-- Disable engines when target apoapsis is reached
vessel.control.throttle = 0.25
while vessel.orbit.apoapsis_altitude < target_altitude do
```

(continues on next page)

(continued from previous page)

```
end
print('Target apoapsis reached')
vessel.control.throttle = 0

-- Wait until out of atmosphere
print('Coasting out of atmosphere')
while flight.mean_altitude < 70500 do
end
```

```
# Disable engines when target apoapsis is reached
vessel.control.throttle = 0.25
while apoapsis() < target_altitude:
    pass
print('Target apoapsis reached')
vessel.control.throttle = 0.0

# Wait until out of atmosphere
print('Coasting out of atmosphere')
while altitude() < 70500:
    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

```
        // Plan circularization burn (using vis-viva equation)
        Console.WriteLine ("Planning circularization burn");
        double mu = vessel.Orbit.Body.GravitationalParameter;
        double r = vessel.Orbit.Apoapsis;
        double a1 = vessel.Orbit.SemiMajorAxis;
        double a2 = r;
        double v1 = Math.Sqrt (mu * ((2.0 / r) - (1.0 / a1)));
        double v2 = Math.Sqrt (mu * ((2.0 / r) - (1.0 / a2)));
        double deltaV = v2 - v1;
        var node = vessel.Control.AddNode (
            ut.Get () + vessel.Orbit.TimeToApoapsis, prograde: (float)deltaV);

        // Calculate burn time (using rocket equation)
        double F = vessel.AvailableThrust;
        double Isp = vessel.SpecificImpulse * 9.82;
        double m0 = vessel.Mass;
        double m1 = m0 / Math.Exp (deltaV / Isp);
        double flowRate = F / Isp;
        double burnTime = (m0 - m1) / flowRate;
```

```
  // Plan circularization burn (using vis-viva equation)
  std::cout << "Planning circularization burn" << std::endl;
```

(continues on next page)

(continued from previous page)

```
  double mu = vessel.orbit().body().gravitational_parameter();
  double r = vessel.orbit().apoapsis();
  double a1 = vessel.orbit().semi_major_axis();
  double a2 = r;
  double v1 = std::sqrt(mu * ((2.0 / r) - (1.0 / a1)));
  double v2 = std::sqrt(mu * ((2.0 / r) - (1.0 / a2)));
  double delta_v = v2 - v1;
  auto node = vessel.control().add_node(
    ut() + vessel.orbit().time_to_apoapsis(), delta_v);

  // Calculate burn time (using rocket equation)
  double F = vessel.available_thrust();
  double Isp = vessel.specific_impulse() * 9.82;
  double m0 = vessel.mass();
  double m1 = m0 / std::exp(delta_v / Isp);
  double flow_rate = F / Isp;
```

```
        // Plan circularization burn (using vis-viva equation)
        System.out.println("Planning circularization burn");
        double mu = vessel.getOrbit().getBody().getGravitationalParameter();
        double r = vessel.getOrbit().getApoapsis();
        double a1 = vessel.getOrbit().getSemiMajorAxis();
        double a2 = r;
        double v1 = Math.sqrt(mu * ((2.0 / r) - (1.0 / a1)));
        double v2 = Math.sqrt(mu * ((2.0 / r) - (1.0 / a2)));
        double deltaV = v2 - v1;
        Node node = vessel.getControl().addNode(
          ut.get() + vessel.getOrbit().getTimeToApoapsis(), (float)deltaV, 0, 0);

        // Calculate burn time (using rocket equation)
        double force = vessel.getAvailableThrust();
        double isp = vessel.getSpecificImpulse() * 9.82;
        double m0 = vessel.getMass();
        double m1 = m0 / Math.exp(deltaV / isp);
        double flowRate = force / isp;
```

```
---- Plan circularization burn (using vis-viva equation)
print('Planning circularization burn')
local mu = vessel.orbit.body.gravitational_parameter
local r = vessel.orbit.apoapsis
local a1 = vessel.orbit.semi_major_axis
local a2 = r
local v1 = math.sqrt(mu*((2./r)-(1./a1)))
local v2 = math.sqrt(mu*((2./r)-(1./a2)))
local delta_v = v2 - v1
local node = vessel.control:add_node(conn.space_center.ut + vessel.orbit.time_to_apoapsis, delta_v, 0, 0)

---- Calculate burn time (using rocket equation)
local F = vessel.available_thrust
local Isp = vessel.specific_impulse * 9.82
local m0 = vessel.mass
local m1 = m0 / math.exp(delta_v/Isp)
local flow_rate = F / Isp
local burn_time = (m0 - m1) / flow_rate
```

```
# Plan circularization burn (using vis-viva equation)
print('Planning circularization burn')
mu = vessel.orbit.body.gravitational_parameter
r = vessel.orbit.apoapsis
a1 = vessel.orbit.semi_major_axis
a2 = r
v1 = math.sqrt(mu*((2./r)-(1./a1)))
v2 = math.sqrt(mu*((2./r)-(1./a2)))
delta_v = v2 - v1
node = vessel.control.add_node(
    ut() + vessel.orbit.time_to_apoapsis, prograde=delta_v)

# Calculate burn time (using rocket equation)
F = vessel.available_thrust
Isp = vessel.specific_impulse * 9.82
m0 = vessel.mass
m1 = m0 / math.exp(delta_v/Isp)
flow_rate = F / Isp
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

```
        // Orientate ship
        Console.WriteLine ("Orientating ship for circularization burn");
        vessel.AutoPilot.ReferenceFrame = node.ReferenceFrame;
        vessel.AutoPilot.TargetDirection = Tuple.Create (0.0, 1.0, 0.0);
        vessel.AutoPilot.Wait ();

        // Wait until burn
        Console.WriteLine ("Waiting until circularization burn");
        double burnUT = ut.Get () + vessel.Orbit.TimeToApoapsis - (burnTime / 2.0);
        double leadTime = 5;
        conn.SpaceCenter ().WarpTo (burnUT - leadTime);
```

```
  // Orientate ship
  std::cout << "Orientating ship for circularization burn" << std::endl;
  vessel.auto_pilot().set_reference_frame(node.reference_frame());
  vessel.auto_pilot().set_target_direction(std::make_tuple(0.0, 1.0, 0.0));
  vessel.auto_pilot().wait();

  // Wait until burn
  std::cout << "Waiting until circularization burn" << std::endl;
  double burn_ut = ut() + vessel.orbit().time_to_apoapsis() - (burn_time / 2.0);
  double lead_time = 5;
```

```
        // Orientate ship
        System.out.println("Orientating ship for circularization burn");
```

(continues on next page)

(continued from previous page)

```
        vessel.getAutoPilot().setReferenceFrame(node.getReferenceFrame());
        vessel.getAutoPilot().setTargetDirection(
          new Triplet<Double,Double,Double>(0.0, 1.0, 0.0));
        vessel.getAutoPilot().wait_();

        // Wait until burn
        System.out.println("Waiting until circularization burn");
        double burnUt =
          ut.get() + vessel.getOrbit().getTimeToApoapsis() - (burnTime / 2.0);
        double leadTime = 5;
```

```
-- Orientate ship
print('Orientating ship for circularization burn')
vessel.auto_pilot.reference_frame = node.reference_frame
vessel.auto_pilot.target_direction = List{0, 1, 0}
vessel.auto_pilot:wait()

-- Wait until burn
print('Waiting until circularization burn')
local burn_ut = conn.space_center.ut + vessel.orbit.time_to_apoapsis - (burn_time/2.)
local lead_time = 5
conn.space_center.warp_to(burn_ut - lead_time)
```

```
# Orientate ship
print('Orientating ship for circularization burn')
vessel.auto_pilot.reference_frame = node.reference_frame
vessel.auto_pilot.target_direction = (0, 1, 0)
vessel.auto_pilot.wait()

# Wait until burn
print('Waiting until circularization burn')
burn_ut = ut() + vessel.orbit.time_to_apoapsis - (burn_time/2.)
lead_time = 5
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

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

(continues on next page)

(continued from previous page)

```
        Console.WriteLine ("Fine tuning");
        vessel.Control.Throttle = 0.05f;
        var remainingBurn = conn.AddStream (
            () => node.RemainingBurnVector (node.ReferenceFrame));
        while (remainingBurn.Get ().Item1 > 0) {
        }
        vessel.Control.Throttle = 0;
        node.Remove ();

        Console.WriteLine ("Launch complete");
        conn.Dispose();
    }
}
```

```
  // Execute burn
  std::cout << "Ready to execute burn" << std::endl;
  auto time_to_apoapsis = vessel.orbit().time_to_apoapsis_stream();
  while (time_to_apoapsis() - (burn_time / 2.0) > 0) {
  }
  std::cout << "Executing burn" << std::endl;
  vessel.control().set_throttle(1);
  std::this_thread::sleep_for(
    std::chrono::milliseconds(static_cast<int>((burn_time - 0.1) * 1000)));
  std::cout << "Fine tuning" << std::endl;
  vessel.control().set_throttle(0.05);
  auto remaining_burn = node.remaining_burn_vector_stream(node.reference_frame());
  while (std::get<0>(remaining_burn()) > 0) {
  }
  vessel.control().set_throttle(0);
  node.remove();

  std::cout << "Launch complete" << std::endl;
}
```

```
        // Execute burn
        System.out.println("Ready to execute burn");
        Stream<Double> timeToApoapsis =
          connection.addStream(vessel.getOrbit(), "getTimeToApoapsis");
        while (timeToApoapsis.get() - (burnTime / 2.0) > 0) {
        }
        System.out.println("Executing burn");
        vessel.getControl().setThrottle(1);
        Thread.sleep((int)((burnTime - 0.1) * 1000));
        System.out.println("Fine tuning");
        vessel.getControl().setThrottle(0.05f);
        Stream<Triplet<Double,Double,Double>> remainingBurn =
          connection.addStream(
            node, "remainingBurnVector", node.getReferenceFrame());
        while (remainingBurn.get().getValue1() > 0) {
        }
        vessel.getControl().setThrottle(0);
        node.remove();

        System.out.println("Launch complete");
        connection.close();
    }
```

(continues on next page)

(continued from previous page)

```
}
```

```
-- Execute burn
print('Ready to execute burn')
while vessel.orbit.time_to_apoapsis - (burn_time/2.) > 0 do
end
print('Executing burn')
vessel.control.throttle = 1
platform.sleep(burn_time - 0.1)
print('Fine tuning')
vessel.control.throttle = 0.05
while node:remaining_burn_vector(node.reference_frame)[2] > 0 do
end
vessel.control.throttle = 0
node:remove()

print('Launch complete')
```

```
# Execute burn
print('Ready to execute burn')
time_to_apoapsis = conn.add_stream(getattr, vessel.orbit, 'time_to_apoapsis')
while time_to_apoapsis() - (burn_time/2.) > 0:
    pass
print('Executing burn')
vessel.control.throttle = 1.0
time.sleep(burn_time - 0.1)
print('Fine tuning')
vessel.control.throttle = 0.05
remaining_burn = conn.add_stream(node.remaining_burn_vector, node.reference_frame)
while remaining_burn()[1] > 0:
    pass
vessel.control.throttle = 0.0
node.remove()

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++

C

Java

Lua

Python

```
using System;
using KRPC.Client;
```

(continues on next page)

(continued from previous page)

```
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);
```

(continues on next page)

(continued from previous page)

```
            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 <cmath>
#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);
```

(continues on next page)

(continued from previous page)

```
}

double magnitude(const vector3& v) {
  return std::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 std::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
```

(continues on next page)

(continued from previous page)

```
    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));
  }
}
```

```
#include <math.h>
#include <unistd.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

static double pi = 3.1415926535897;
typedef krpc_tuple_double_double_double_t vector3;

vector3 cross_product(vector3 u, vector3 v) {
  vector3 result = {
    u.e1*v.e2 - u.e2*v.e1,
    u.e2*v.e0 - u.e0*v.e2,
    u.e0*v.e1 - u.e1*v.e0
  };
  return result;
}

double dot_product(vector3 u, vector3 v) {
  return u.e0*v.e0 + u.e1*v.e1 + u.e2*v.e2;
}

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

// Compute the angle between vector u and v
double angle_between_vectors(vector3 u, 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_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "");
```

(continues on next page)

(continued from previous page)

```
  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_ReferenceFrame_t srf_ref;
  krpc_SpaceCenter_Vessel_SurfaceReferenceFrame(conn, &srf_ref, vessel);
  krpc_SpaceCenter_ReferenceFrame_t vessel_ref;
  krpc_SpaceCenter_Vessel_ReferenceFrame(conn, &vessel_ref, vessel);

  while (true) {
    vector3 vessel_direction;
    krpc_SpaceCenter_Vessel_Direction(conn, &vessel_direction, vessel, srf_ref);

    // Get the direction of the vessel in the horizon plane
    vector3 horizon_direction = {
      0, vessel_direction.e1, vessel_direction.e2
    };

    // 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 (vessel_direction.e0 < 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 (horizon_direction.e2 < 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;
    vector3 tmp = { 0, 0, -1 };
    krpc_SpaceCenter_TransformDirection(conn, &vessel_up, &tmp, vessel_ref, srf_ref);
    // 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 (vessel_up.e0 > 0)
        roll *= -1;
    else if (roll < 0)
        roll += 180;
    else
        roll -= 180;

    printf("pitch = %.1f, heading = %.1f, roll = %.1f\n", pitch, heading, roll);
    sleep(1);
  }
}
```

```
import krpc.client.Connection;
```

(continues on next page)

(continued from previous page)

```
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());
```

(continues on next page)

(continued from previous page)

```
            // 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],
```

(continues on next page)

(continued from previous page)

```
                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
```

(continues on next page)

(continued from previous page)

```
    -- 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 = %1.f, 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
```

(continues on next page)

(continued from previous page)

```
    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++

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*
- *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++

C

Java

Lua

Python

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

class DeployParachutes
{
    public static void Main ()
    {
```

(continues on next page)

(continued from previous page)

```
        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();
}
```

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "DeployParachutes");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);

  krpc_SpaceCenter_Parts_t parts;
  krpc_SpaceCenter_Vessel_Parts(conn, &parts, vessel);

  krpc_list_object_t parachutes;
  krpc_SpaceCenter_Parts_Parachutes(conn, &parachutes, parts);
  for (size_t i = 0; i < parachutes.size; i++) {
    krpc_SpaceCenter_Parachute_t parachute = parachutes.items[i];
    krpc_SpaceCenter_Parachute_Deploy(conn, parachute);
  }
}
```

```
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()) {
```

(continues on next page)

(continued from previous page)

```
            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++

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>
```

(continues on next page)

(continued from previous page)

```
#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);
}
```

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "ControlFromHere");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);

  krpc_SpaceCenter_Parts_t parts;
  krpc_SpaceCenter_Vessel_Parts(conn, &parts, vessel);

  krpc_list_object_t docking_port_parts;
  krpc_SpaceCenter_Parts_WithTitle(conn, &docking_port_parts, parts, "Clamp-O-Tron␣
↪Docking Port");
  krpc_object_t part = docking_port_parts.items[0];
  krpc_SpaceCenter_Parts_set_Controlling(conn, parts, 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
```

```python
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 for a description of the maths: https://wiki.kerbalspaceprogram.com/wiki/Specific_impulse#Multiple_engines

C#

C++

C

Java

Lua

Python

```csharp
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);
        }
    }
}
```

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

```

(continues on next page)

(continued from previous page)

```
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;
}
```

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "CombinedISP");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);

  krpc_SpaceCenter_Parts_t parts;
  krpc_SpaceCenter_Vessel_Parts(conn, &parts, vessel);

  krpc_list_object_t engines = KRPC_NULL_LIST;
  krpc_SpaceCenter_Parts_Engines(conn, &engines, parts);

  krpc_list_object_t active_engines = KRPC_NULL_LIST;
  active_engines.size = 0;
  active_engines.items = krpc_calloc(engines.size, sizeof(krpc_object_t));

  for (size_t i = 0; i < engines.size; i++) {
    krpc_SpaceCenter_Engine_t engine = engines.items[i];
    bool active;
    bool has_fuel;
    krpc_SpaceCenter_Engine_Active(conn, &active, engine);
    krpc_SpaceCenter_Engine_HasFuel(conn, &has_fuel, engine);
```

(continues on next page)

(continued from previous page)

```
    if (active && has_fuel) {
      active_engines.items[active_engines.size] = engine;
      active_engines.size++;
    }
  }

  printf("Active engines:\n");
  for (size_t i = 0; i < active_engines.size; i++) {
    krpc_SpaceCenter_Engine_t engine = active_engines.items[i];
    krpc_SpaceCenter_Part_t part;
    krpc_SpaceCenter_Engine_Part(conn, &part, engine);
    char * title = NULL;
    int stage;
    krpc_SpaceCenter_Part_Title(conn, &title, part);
    krpc_SpaceCenter_Part_Stage(conn, &stage, part);
    printf("   %s in stage %d\n", title, stage);
  }

  double thrust = 0;
  double fuel_consumption = 0;
  for (size_t i = 0; i < active_engines.size; i++) {
    krpc_SpaceCenter_Engine_t engine = active_engines.items[i];
    float engine_thrust;
    float engine_isp;
    krpc_SpaceCenter_Engine_Thrust(conn, &engine_thrust, engine);
    krpc_SpaceCenter_Engine_SpecificImpulse(conn, &engine_isp, engine);
    thrust += engine_thrust;
    fuel_consumption += engine_thrust / engine_isp;
  }
  double isp = thrust / fuel_consumption;
  printf("Combined vacuum Isp = %.2f seconds\n", isp);

  KRPC_FREE_LIST(engines);
  KRPC_FREE_LIST(active_engines);
}
```

```
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);
```

(continues on next page)

(continued from previous page)

```
            }
        }

        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();
    }
}
```

```lua
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))
```

```python
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:
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
            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))
```

(continues on next page)

(continued from previous page)

```
            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
```

(continues on next page)

(continued from previous page)

```
        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);
```

(continues on next page)

(continued from previous page)

```
  // 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(
```

(continues on next page)

(continued from previous page)

```
            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}
```

(continues on next page)

(continued from previous page)

```
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)
```

(continues on next page)

(continued from previous page)

```
    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

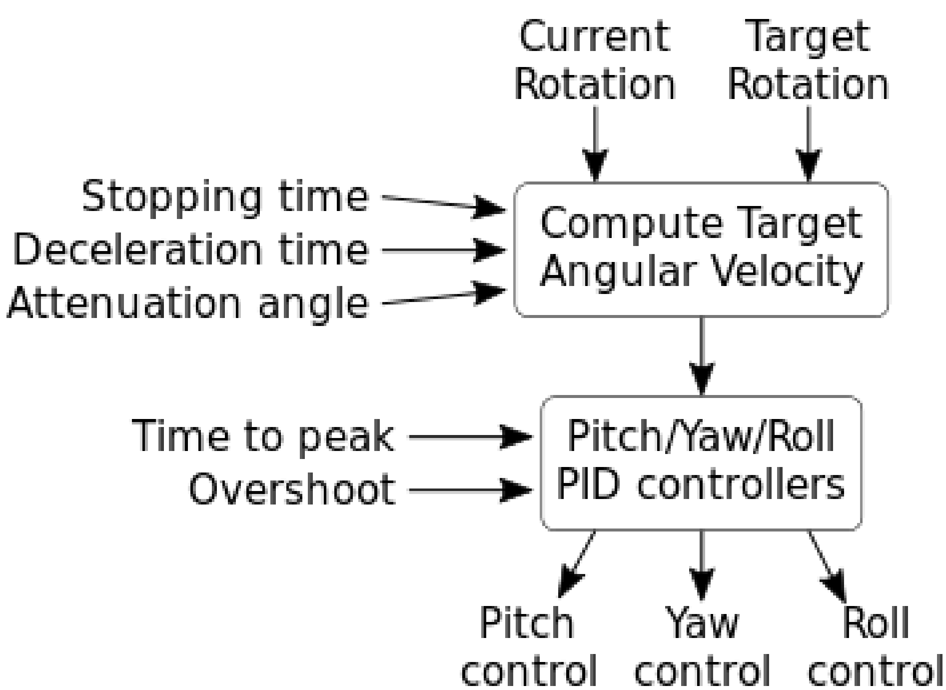
The inputs to the autopilot are:

- 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

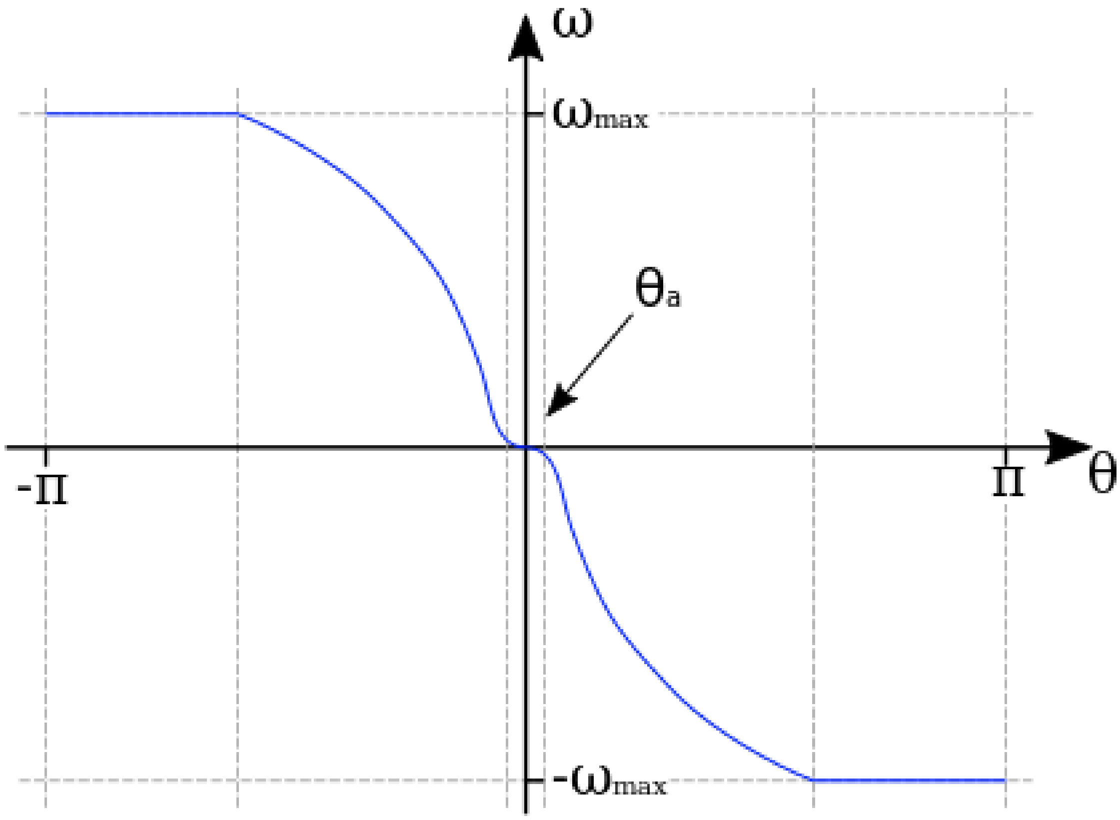
There are several parameters that affect the behavior of the autopilot. The default values for these should suffice in most cases, but they can be adjusted to fit your needs.

- 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 ω in a given axis is computed from the angular error θ using the following function:



The equation for this function is:

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

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

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

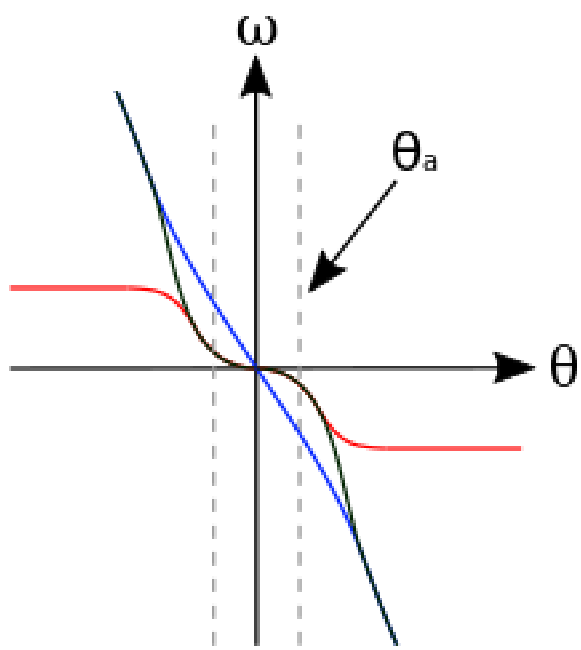
  where τ_{max} is the maximum torque the vessel can generate, and I is its moment of inertia.

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

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

- 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 θ.

  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:

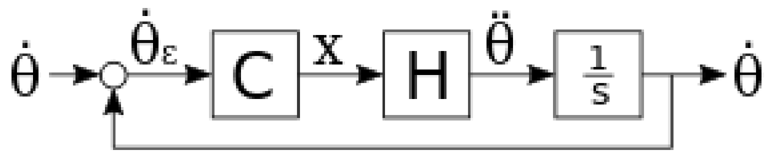
  $$f_a(\theta) = \frac{1}{1 + e^{-6/\theta_a(|\theta| - \theta_a)}}$$

  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 ω. The error in the angular speed ω_ϵ 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 τ_{max} that is being applied to the vessel. Call this the current torque, denoted τ. 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:

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

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 ζ is the damping ratio and ω_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 ζ and ω_0:

$$K_P = \frac{2\zeta\omega_0 I}{\tau_{max}}$$
$$K_I = \frac{I\omega_0^2}{\tau_{max}}$$

We now need to choose some performance requirements to place on the system, which will allow us to determine the values of ζ and ω_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 ζ and ω_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.

CHAPTER

THREE

C-NANO

3.1 C-nano Client

This client provides a C API for interacting with a kRPC server. It is intended for use on embedded systems with tight resource constraints, hence the "nano" in its name.

3.1.1 Installing the Library

Manually include the source in your project

The source files can be included in your project manually, by downloading and extracting the source archive. The header files can be found in the `include` directory and the source files are in `src`.

Arduino Library Manager

If you are writing an Arduino sketch, the library can be installed using the Arduino Library Manager by searching for and installing "kRPC".

Note: The source files installed by the Arduino Library Manager are renamed to end with `.cpp` so that they are built using the C++ compiler. This allows the library to use the C++ only `HardwareSerial` class for communication.

Using the configure script

You can build and install the 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
```

Compilation Options

The following options control how the library operates. They must be specified at compile time as an argument to the compiler.

- Error handling
  - `KRPC_ERROR_CHECK_RETURN` (the default) – when a remote procedure call gets an error, it returns the error code.
  - `KRPC_ERROR_CHECK_EXIT` – terminates the program (by calling `exit()`) when an error occurs in a remote procedure call.
  - `KRPC_ERROR_CHECK_ASSERT` – fails a debug assertion (by calling `assert()`) when an error occurs in a remote procedure call.
  - `KRPC_ERROR_CHECK_FN` – specifies the `krpc_error_handler` function should be called when an error occurs in a remote procedure call. This should be set to a pointer to a function that takes a single parameter of type `krpc_error_t`.
  - `KRPC_PRINT_ERRORS_TO_STDERR` – enables printing of a descriptive error message to stderr when an error occurs
  - `PB_NO_ERRMSG` – disables error messages in the nanopb library, which kRPC uses to communicate with the server. Enabled by default on in the Arduino version of the library.
- Communication
  - `KRPC_COMMUNICATION_POSIX` – Specifies that the library should be built to communicate over a serial port using POSIX read/write functions communication mechanisms. This is the default, unless the a different platform is detected.
  - `KRPC_COMMUNICATION_ARDUINO` – Specifies that the library should be built using Arduino serial communication mechanisms. The Arduino platform will be auto-detected so you do not need to specify this manually.
  - `KRPC_COMMUNICATION_CUSTOM` – Allows you to provide your own implementation for the communication mechanism.
- Memory allocation
  - `KRPC_ALLOC_BLOCK_SIZE` – The size of collections (lists, sets, etc.) are not know ahead of time, so when they are received from the server they are decoded into dynamically allocated memory on the heap. This option controls how many items to increase the capacity of the collection by when its space is exhausted. Setting this to 1 will consume the least amount of heap memory, but will require one heap allocation call per item. Setting this to a higher value will consume more memory, but require fewer allocations.
  - `KRPC_CUSTOM_MEMORY_ALLOC` – Disables the default implementation of memory allocation functions `krpc_malloc`, `krpc_calloc`, `krpc_recalloc` and krpc_free so that you can provide your own implementation.

Note: On embedded systems you probably want to define KRPC_NO_PRINT_ERROR and PB_NO_ERRMSG to minimize the memory footprint of kRPC.

3.1.2 Configuring the Server

The C-nano client library communicates with the server over a serial port using protobuf messages. The kRPC server, which runs in the game, needs to be configured to use the serial port protocol (instead of the default TCP/IP protocol). This can be done from the in-game server configuration window, which also allows settings such as the port name and baud rate to be configured.

3.1.3 Using the Library on a POSIX System

On POSIX systems (such as Linux) the following example program connects to the server, queries it for its version and prints it out:

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/krpc.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Basic example");
  krpc_schema_Status status;
  krpc_KRPC_GetStatus(conn, &status);
  printf("Connected to kRPC server version %s\n", status.version);
}
```

To compile this program using GCC, save the source as `main.c` and run the following:

```
gcc main.c -lkrpc_cnano
```

The *krpc_connect()* function is used to open a connection to a server. It takes as its first argument a connection object into which the connection information is written. This is passed to subsequent calls to interact with the server. The second argument is a name for the connection (displayed in game) and the third is the name of the serial port to connect over.

3.1.4 Using the Library on from an Arduino

The following example demonstrates how to connect to the server from an Arduino, through its serial port interface:

```
#include <krpc.h>
#include <krpc/services/krpc.h>

HardwareSerial * conn;

void setup() {
  pinMode(LED_BUILTIN, OUTPUT);
  digitalWrite(LED_BUILTIN, LOW);

  conn = &Serial;
  // Open the serial port connection
```

(continues on next page)

(continued from previous page)

```
  krpc_open(&conn, NULL);
  // Set up communication with the server
  krpc_connect(conn, "Arduino Example");

  // Indicate succesful connection by lighting the on-board LED
  digitalWrite(LED_BUILTIN, HIGH);
}

void loop() {
}
```

Note: The main include file and include directory are named `krpc` instead of `krpc_cnano` in the Arduino version of the library.

3.1.5 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.h`.

The following example demonstrates how to invoke remote procedures using the Cnano client. It calls *`krpc_SpaceCenter_ActiveVessel()`* to get a handle to the active vessel (of type *`krpc_SpaceCenter_Vessel_t`*). It sets the name of the vessel and then prints out its altitude:

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "Remote Procedures example");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);

  krpc_SpaceCenter_Vessel_set_Name(conn, vessel, "My Vessel");

  // Get a handle to a Flight object for the vessel
  krpc_SpaceCenter_Flight_t flight;
  krpc_SpaceCenter_Vessel_Flight(conn, &flight, vessel, KRPC_NULL);
  // Get the altiude
  double altitude;
  krpc_SpaceCenter_Flight_MeanAltitude(conn, &altitude, flight);
  printf("%.2f\n", altitude);
}
```

3.1.6 Streams and Events

These features are not yet supported by this client.

3.1.7 Client API Reference

krpc_error_t **krpc_open** (krpc_connection_t * *connection*, const krpc_connection_config_t * *arg*)

Create a communication handle over which the client can talk to a server.

When the library is built using `KRPC_COMMUNICATION_POSIX` (which is defined by default) calling this function opens a serial port using the port name passed as *arg*, using a call to `open(arg, ...)`. In this case the type of the *arg* parameter is `const char *`. For example:

```
krpc_connection_t conn;
krpc_open(&conn, "COM0");
```

When the library is built using `KRPC_COMMUNICATION_ARDUINO`, *connection* must be a pointer to a `HardwareSerial` object. *arg* is optionally used to pass additional configuration options used to initialize the connection, including baud rate for the serial port.

If *arg* is set to `NULL` the connection is initialized with a baud rate of 9600 and defaults `SERIAL_8N1` for data, parity and stop bits. For example:

```
krpc_connection_t conn;
krpc_open(&conn, NULL);
```

When *arg* set to a pointer to a structure of type `krpc_connection_config_t`, the baud rate, and data, parity and stop bits in the structure are used to initialize the connection. For example:

```
krpc_connection_t conn;
krpc_connection_config_t config;
config.speed = 115200;
config.config = SERIAL_5N1;
krpc_open(&conn, &config);
```

krpc_error_t **krpc_connect** (krpc_connection_t *connection*, const char * *name*)

Connect to a kRPC server.

Parameters

- **connection** (*krpc_connection_t*) – A connection handle, created using a call to *krpc_open()*.
- **name** (*const char**) – A descriptive name for the connection. This is passed to the server and appears in the in-game server window.

krpc_error_t **krpc_close** (krpc_connection_t *connection*)

Closes the communication handle.

krpc_error_t

All kRPC functions return error codes of this type.

KRPC_OK

The function completed successfully and no error occurred.

KRPC_ERROR_IO

An input/output error occurred when communicating with the server.

KRPC_ERROR_EOF

End of file was received from the server.

KRPC_ERROR_CONNECTION_FAILED

Failed to establish a connection to the server.

KRPC_ERROR_NO_RESULTS
The remote procedure call did not return a result.

KRPC_ERROR_RPC_FAILED
The remote procedure call threw an exception.

KRPC_ERROR_ENCODING_FAILED
The encoder failed to construct the remote procedure call.

KRPC_ERROR_DECODING_FAILED
The decoder failed to interpret a result sent by the server.

const char * **krpc_get_error** (*krpc_error_t error*)
Returns a descriptive string for the given error code.

3.2 KRPC API

3.2.1 KRPC

None None None None None

Service KRPC

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

krpc_error_t **krpc_KRPC_GetClientID** (krpc_connection_t *connection*, krpc_bytes_t * *result*)
Returns the identifier for the current client.

Game Scenes All

krpc_error_t **krpc_KRPC_GetClientName** (krpc_connection_t *connection*, char * * *result*)
Returns the name of the current client. This is an empty string if the client has no name.

Game Scenes All

krpc_error_t **krpc_KRPC_Clients** (krpc_connection_t *connection*, krpc_list_tuple_bytes_string_string_t * *result*)
A list of RPC clients that are currently connected to the server. Each entry in the list is a clients identifier, name and address.

Game Scenes All

krpc_error_t **krpc_KRPC_GetStatus** (krpc_connection_t *connection*, krpc_schema_Status * *result*)
Returns some information about the server, such as the version.

Game Scenes All

krpc_error_t **krpc_KRPC_GetServices** (krpc_connection_t *connection*, krpc_schema_Services * *result*)
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.

Game Scenes All

krpc_error_t **krpc_KRPC_CurrentGameScene** (krpc_connection_t *connection*, *krpc_KRPC_GameScene_t * result*)
Get the current game scene.

Game Scenes All

krpc_error_t **krpc_KRPC_Paused** (krpc_connection_t *connection*, bool * *result*)

void **krpc_KRPC_set_Paused** (bool *value*)

Whether the game is paused.

Game Scenes All

krpc_KRPC_GameScene_t

The game scene. See *krpc_KRPC_CurrentGameScene()*.

KRPC_KRPC_GAMESCENE_SPACECENTER

The game scene showing the Kerbal Space Center buildings.

KRPC_KRPC_GAMESCENE_FLIGHT

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

KRPC_KRPC_GAMESCENE_TRACKINGSTATION

The tracking station.

KRPC_KRPC_GAMESCENE_EDITORVAB

The Vehicle Assembly Building.

KRPC_KRPC_GAMESCENE_EDITORSPH

The Space Plane Hangar.

Exception class InvalidOperationException

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

Exception class ArgumentException

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

Exception class ArgumentNullException

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

Exception class ArgumentOutOfRangeException

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

3.2.2 Expressions

krpc_KRPC_Expression_t

A server side expression.

krpc_error_t **krpc_KRPC_Expression_ConstantDouble** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, double *value*)

A constant value of double precision floating point type.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_ConstantFloat** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, float *value*)

A constant value of single precision floating point type.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_ConstantInt** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, int32_t *value*)

A constant value of integer type.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_ConstantBool** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, bool *value*)

A constant value of boolean type.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_ConstantString** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, const char * *value*)

A constant value of string type.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Call** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, const krpc_schema_ProcedureCall * *call*)

An RPC call.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Equal** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t* *arg0*, *krpc_KRPC_Expression_t* *arg1*)

Equality comparison.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_NotEqual** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t* *arg0*, *krpc_KRPC_Expression_t* *arg1*)

Inequality comparison.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_GreaterThan** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t* *arg0*, *krpc_KRPC_Expression_t* *arg1*)

Greater than numerical comparison.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_GreaterThanOrEqual** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Greater than or equal numerical comparison.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_LessThan** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Less than numerical comparison.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_LessThanOrEqual** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Less than or equal numerical comparison.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_And** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Boolean and operator.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Or** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Boolean or operator.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_ExclusiveOr** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Boolean exclusive-or operator.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Not** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg*)

Boolean negation operator.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Add** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Numerical addition.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Subtract** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Numerical subtraction.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Multiply** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Numerical multiplication.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Divide** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Numerical division.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Modulo** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg0*, *krpc_KRPC_Expression_t arg1*)

Numerical modulo operator.

Parameters

Returns The remainder of arg0 divided by arg1

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Power** (krpc_connection_t *connection*, krpc_KRPC_Expression_t * *result*, krpc_KRPC_Expression_t *arg0*, krpc_KRPC_Expression_t *arg1*)

Numerical power operator.

Parameters

Returns arg0 raised to the power of arg1, with type of arg0

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_LeftShift** (krpc_connection_t *connection*, krpc_KRPC_Expression_t * *result*, krpc_KRPC_Expression_t *arg0*, krpc_KRPC_Expression_t *arg1*)

Bitwise left shift.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_RightShift** (krpc_connection_t *connection*, krpc_KRPC_Expression_t * *result*, krpc_KRPC_Expression_t *arg0*, krpc_KRPC_Expression_t *arg1*)

Bitwise right shift.

Parameters

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Cast** (krpc_connection_t *connection*, krpc_KRPC_Expression_t * *result*, krpc_KRPC_Expression_t *arg*, krpc_KRPC_Type_t *type*)

Perform a cast to the given type.

Parameters

- **type** – Type to cast the argument to.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Parameter** (krpc_connection_t *connection*, krpc_KRPC_Expression_t * *result*, const char * *name*, krpc_KRPC_Type_t *type*)

A named parameter of type double.

Parameters

- **name** – The name of the parameter.
- **type** – The type of the parameter.

Returns A named parameter.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Function** (krpc_connection_t *connection*, krpc_KRPC_Expression_t * *result*, const krpc_list_object_t * *parameters*, krpc_KRPC_Expression_t *body*)

A function.

Parameters

- **parameters** – The parameters of the function.
- **body** – The body of the function.

Returns A function.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Invoke** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t* *function*, const krpc_dictionary_string_object_t * *args*)

A function call.

Parameters

- **function** – The function to call.
- **args** – The arguments to call the function with.

Returns A function call.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_CreateTuple** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, const krpc_list_object_t * *elements*)

Construct a tuple.

Parameters

- **elements** – The elements.

Returns The tuple.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_CreateList** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, const krpc_list_object_t * *values*)

Construct a list.

Parameters

- **values** – The value. Should all be of the same type.

Returns The list.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_CreateSet** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, const krpc_set_object_t * *values*)

Construct a set.

Parameters

- **values** – The values. Should all be of the same type.

Returns The set.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_CreateDictionary** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, const krpc_list_object_t * *keys*, const krpc_list_object_t * *values*)

Construct a dictionary, from a list of corresponding keys and values.

Parameters

- **keys** – The keys. Should all be of the same type.
- **values** – The values. Should all be of the same type.

Returns The dictionary.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_ToList** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg*)

Convert a collection to a list.

Parameters

- **arg** – The collection.

Returns The collection as a list.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_ToSet** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg*)

Convert a collection to a set.

Parameters

- **arg** – The collection.

Returns The collection as a set.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Get** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg*, *krpc_KRPC_Expression_t index*)

Access an element in a tuple, list or dictionary.

Parameters

- **arg** – The tuple, list or dictionary.
- **index** – The index of the element to access. A zero indexed integer for a tuple or list, or a key for a dictionary.

Returns The element.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Count** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t * result*, *krpc_KRPC_Expression_t arg*)

Number of elements in a collection.

Parameters

- **arg** – The list, set or dictionary.

Returns The number of elements in the collection.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Sum** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg*)

Sum all elements of a collection.

Parameters

- **arg** – The list or set.

Returns The sum of the elements in the collection.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Max** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg*)

Maximum of all elements in a collection.

Parameters

- **arg** – The list or set.

Returns The maximum elements in the collection.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Min** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg*)

Minimum of all elements in a collection.

Parameters

- **arg** – The list or set.

Returns The minimum elements in the collection.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Average** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg*)

Minimum of all elements in a collection.

Parameters

- **arg** – The list or set.

Returns The minimum elements in the collection.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Select** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg*, *krpc_KRPC_Expression_t func*)

Run a function on every element in the collection.

Parameters

- **arg** – The list or set.
- **func** – The function.

Returns The modified collection.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Where** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t* *arg*, *krpc_KRPC_Expression_t* *func*)

Run a function on every element in the collection.

Parameters

- **arg** – The list or set.
- **func** – The function.

Returns The modified collection.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Contains** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t* *arg*, *krpc_KRPC_Expression_t* *value*)

Determine if a collection contains a value.

Parameters

- **arg** – The collection.
- **value** – The value to look for.

Returns Whether the collection contains a value.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Aggregate** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t* *arg*, *krpc_KRPC_Expression_t* *func*)

Applies an accumulator function over a sequence.

Parameters

- **arg** – The collection.
- **func** – The accumulator function.

Returns The accumulated value.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_AggregateWithSeed** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t* *arg*, *krpc_KRPC_Expression_t* *seed*, *krpc_KRPC_Expression_t* *func*)

Applies an accumulator function over a sequence, with a given seed.

Parameters

- **arg** – The collection.
- **seed** – The seed value.
- **func** – The accumulator function.

Returns The accumulated value.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Concat** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg1*, *krpc_KRPC_Expression_t arg2*)

Concatenate two sequences.

Parameters

- **arg1** – The first sequence.
- **arg2** – The second sequence.

Returns The first sequence followed by the second sequence.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_OrderBy** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg*, *krpc_KRPC_Expression_t key*)

Order a collection using a key function.

Parameters

- **arg** – The collection to order.
- **key** – A function that takes a value from the collection and generates a key to sort on.

Returns The ordered collection.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_All** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg*, *krpc_KRPC_Expression_t predicate*)

Determine whether all items in a collection satisfy a boolean predicate.

Parameters

- **arg** – The collection.
- **predicate** – The predicate function.

Returns Whether all items satisfy the predicate.

Game Scenes All

krpc_error_t **krpc_KRPC_Expression_Any** (krpc_connection_t *connection*, *krpc_KRPC_Expression_t* * *result*, *krpc_KRPC_Expression_t arg*, *krpc_KRPC_Expression_t predicate*)

Determine whether any item in a collection satisfies a boolean predicate.

Parameters

- **arg** – The collection.
- **predicate** – The predicate function.

Returns Whether any item satisfies the predicate.

Game Scenes All

krpc_KRPC_Type_t

A server side expression.

krpc_error_t **krpc_KRPC_Type_Double** (krpc_connection_t *connection*, *krpc_KRPC_Type_t * result*)

Double type.

Game Scenes All

krpc_error_t **krpc_KRPC_Type_Float** (krpc_connection_t *connection*, *krpc_KRPC_Type_t * result*)

Float type.

Game Scenes All

krpc_error_t **krpc_KRPC_Type_Int** (krpc_connection_t *connection*, *krpc_KRPC_Type_t * result*)

Int type.

Game Scenes All

krpc_error_t **krpc_KRPC_Type_Bool** (krpc_connection_t *connection*, *krpc_KRPC_Type_t * result*)

Bool type.

Game Scenes All

krpc_error_t **krpc_KRPC_Type_String** (krpc_connection_t *connection*, *krpc_KRPC_Type_t * result*)

String type.

Game Scenes All

3.3 SpaceCenter API

3.3.1 SpaceCenter

Service SpaceCenter

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

krpc_error_t **krpc_SpaceCenter_Science** (krpc_connection_t *connection*, float * *result*)

The current amount of science.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Funds** (krpc_connection_t *connection*, double * *result*)

The current amount of funds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Reputation** (krpc_connection_t *connection*, float * *result*)

The current amount of reputation.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ActiveVessel** (krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t * result*)

void **krpc_SpaceCenter_set_ActiveVessel** (*krpc_SpaceCenter_Vessel_t value*)

The currently active vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessels** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all the vessels in the game.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Bodies** (krpc_connection_t *connection*, krpc_dictionary_string_object_t * *result*)

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

Game Scenes All

krpc_error_t **krpc_SpaceCenter_TargetBody** (krpc_connection_t *connection*, *krpc_SpaceCenter_CelestialBody_t* * *result*)

void **krpc_SpaceCenter_set_TargetBody** (*krpc_SpaceCenter_CelestialBody_t value*)

The currently targeted celestial body.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_TargetVessel** (krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t* * *result*)

void **krpc_SpaceCenter_set_TargetVessel** (*krpc_SpaceCenter_Vessel_t value*)

The currently targeted vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_TargetDockingPort** (krpc_connection_t *connection*, *krpc_SpaceCenter_DockingPort_t* * *result*)

void **krpc_SpaceCenter_set_TargetDockingPort** (*krpc_SpaceCenter_DockingPort_t value*)

The currently targeted docking port.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_ClearTarget** (krpc_connection_t *connection*)

Clears the current target.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_LaunchableVessels** (krpc_connection_t *connection*, krpc_list_string_t * *result*, const char * *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"`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_LaunchVessel** (krpc_connection_t *connection*, const char * *craftDirectory*, const char * *name*, const char * *launchSite*, bool *recover*)

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"`.
- **recover** – If true and there is a vessel on the launch site, recover it before launching.

Game Scenes All

Note: Throws an exception if any of the games pre-flight checks fail.

krpc_error_t **krpc_SpaceCenter_LaunchVesselFromVAB** (krpc_connection_t *connection*, const char * *name*, bool *recover*)

Launch a new vessel from the VAB onto the launchpad.

Parameters

- **name** – Name of the vessel to launch.
- **recover** – If true and there is a vessel on the launch pad, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *krpc_SpaceCenter_LaunchVessel()* with the craft directory set to "VAB" and the launch site set to "LaunchPad". Throws an exception if any of the games pre-flight checks fail.

krpc_error_t **krpc_SpaceCenter_LaunchVesselFromSPH** (krpc_connection_t *connection*, const char * *name*, bool *recover*)

Launch a new vessel from the SPH onto the runway.

Parameters

- **name** – Name of the vessel to launch.
- **recover** – If true and there is a vessel on the runway, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *krpc_SpaceCenter_LaunchVessel()* with the craft directory set to "SPH" and the launch site set to "Runway". Throws an exception if any of the games pre-flight checks fail.

krpc_error_t **krpc_SpaceCenter_Save** (krpc_connection_t *connection*, const char * *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

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Load** (krpc_connection_t *connection*, const char * *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

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Quicksave** (krpc_connection_t *connection*)
Save a quicksave.

Game Scenes All

Note: This is the same as calling *krpc_SpaceCenter_Save()* with the name "quicksave".

krpc_error_t **krpc_SpaceCenter_Quickload** (krpc_connection_t *connection*)
Load a quicksave.

Game Scenes All

Note: This is the same as calling *krpc_SpaceCenter_Load()* with the name "quicksave".

krpc_error_t **krpc_SpaceCenter_UIVisible** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_set_UIVisible** (bool *value*)
Whether the UI is visible.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Navball** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_set_Navball** (bool *value*)
Whether the navball is visible.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_UT** (krpc_connection_t *connection*, double * *result*)
The current universal time in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_G** (krpc_connection_t *connection*, double * *result*)
The value of the gravitational constant G in $N(m/kg)^2$.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_WarpRate** (krpc_connection_t *connection*, float * *result*)
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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_WarpFactor** (krpc_connection_t *connection*, float * *result*)
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 *krpc_SpaceCenter_RailsWarpFactor()*, and in physics time warp, this is equal to *krpc_SpaceCenter_PhysicsWarpFactor()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_RailsWarpFactor** (krpc_connection_t *connection*, int32_t * *result*) void **krpc_SpaceCenter_set_RailsWarpFactor** (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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_PhysicsWarpFactor** (krpc_connection_t *connection*, int32_t * *result*)

void **krpc_SpaceCenter_set_PhysicsWarpFactor** (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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_CanRailsWarpAt** (krpc_connection_t *connection*, bool * *result*, int32_t *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** – The warp factor to check.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_MaximumRailsWarpFactor** (krpc_connection_t *connection*, int32_t * *result*)

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_WarpTo** (krpc_connection_t *connection*, 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** – 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_TransformPosition** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, const krpc_tuple_double_double_double_t * *position*, *krpc_SpaceCenter_ReferenceFrame_t from*, *krpc_SpaceCenter_ReferenceFrame_t 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 covert the position to.

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

Game Scenes All

krpc_error_t **krpc_SpaceCenter_TransformDirection**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, const krpc_tuple_double_double_double_t * *direction*, *krpc_SpaceCenter_ReferenceFrame_t from*, *krpc_SpaceCenter_ReferenceFrame_t 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 covert the direction to.

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

Game Scenes All

krpc_error_t **krpc_SpaceCenter_TransformRotation**(krpc_connection_t *connection*, krpc_tuple_double_double_double_double_t * *result*, const krpc_tuple_double_double_double_double_t * *rotation*, *krpc_SpaceCenter_ReferenceFrame_t from*, *krpc_SpaceCenter_ReferenceFrame_t 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 covert the rotation to.

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

Game Scenes All

krpc_error_t **krpc_SpaceCenter_TransformVelocity** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, const krpc_tuple_double_double_double_t * *position*, const krpc_tuple_double_double_double_t * *velocity*, *krpc_SpaceCenter_ReferenceFrame_t from*, *krpc_SpaceCenter_ReferenceFrame_t 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 covert the velocity to.

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

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RaycastDistance** (krpc_connection_t *connection*, double * *result*, const krpc_tuple_double_double_double_t * *position*, const krpc_tuple_double_double_double_t * *direction*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*)

Cast a ray from a given position in a given direction, and return the distance to the hit point. If no hit occurs, returns infinity.

Parameters

- **position** – Position, as a vector, of the origin of the ray.
- **direction** – Direction of the ray, as a unit vector.
- **referenceFrame** – The reference frame that the position and direction are in.

Returns The distance to the hit, in meters, or infinity if there was no hit.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RaycastPart** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*, const krpc_tuple_double_double_double_t * *position*, const krpc_tuple_double_double_double_t * *direction*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*)

Cast a ray from a given position in a given direction, and return the part that it hits. If no hit occurs, returns `nullptr`.

Parameters

- **position** – Position, as a vector, of the origin of the ray.
- **direction** – Direction of the ray, as a unit vector.
- **referenceFrame** – The reference frame that the position and direction are in.

Returns The part that was hit or `nullptr` if there was no hit.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_FARAvailable** (krpc_connection_t *connection*, bool * *result*)

Whether Ferram Aerospace Research is installed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_GameMode** (krpc_connection_t *connection*, *krpc_SpaceCenter_GameMode_t* * *result*)

The current mode the game is in.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_WarpMode** (krpc_connection_t *connection*, *krpc_SpaceCenter_WarpMode_t* * *result*)

The current time warp mode. Returns *KRPC_SPACECENTER_WARPMODE_NONE* if time warp is not active, *KRPC_SPACECENTER_WARPMODE_RAILS* if regular "on-rails" time warp is active, or *KRPC_SPACECENTER_WARPMODE_PHYSICS* if physical time warp is active.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera** (krpc_connection_t *connection*, *krpc_SpaceCenter_Camera_t* * *result*)

An object that can be used to control the camera.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_WaypointManager** (krpc_connection_t *connection*, *krpc_SpaceCenter_WaypointManager_t* * *result*)

The waypoint manager.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_ContractManager** (krpc_connection_t *connection*, *krpc_SpaceCenter_ContractManager_t* * *result*)

The contract manager.

Game Scenes All

krpc_SpaceCenter_GameMode_t

The game mode. Returned by *krpc_SpaceCenter_GameMode_t*

KRPC_SPACECENTER_GAMEMODE_SANDBOX

Sandbox mode.

KRPC_SPACECENTER_GAMEMODE_CAREER

Career mode.

KRPC_SPACECENTER_GAMEMODE_SCIENCE

Science career mode.

KRPC_SPACECENTER_GAMEMODE_SCIENCESANDBOX

Science sandbox mode.

KRPC_SPACECENTER_GAMEMODE_MISSION
Mission mode.

KRPC_SPACECENTER_GAMEMODE_MISSIONBUILDER
Mission builder mode.

KRPC_SPACECENTER_GAMEMODE_SCENARIO
Scenario mode.

KRPC_SPACECENTER_GAMEMODE_SCENARIONONRESUMABLE
Scenario mode that cannot be resumed.

krpc_SpaceCenter_WarpMode_t
The time warp mode. Returned by *krpc_SpaceCenter_WarpMode_t*

KRPC_SPACECENTER_WARPMODE_RAILS
Time warp is active, and in regular "on-rails" mode.

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

KRPC_SPACECENTER_WARPMODE_NONE
Time warp is not active.

3.3.2 Vessel

krpc_SpaceCenter_Vessel_t
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 *krpc_SpaceCenter_ActiveVessel()* or *krpc_SpaceCenter_Vessels()*.

krpc_error_t **krpc_SpaceCenter_Vessel_Name** (krpc_connection_t *connection*, char * * *result*)

void **krpc_SpaceCenter_Vessel_set_Name** (const char * *value*)
The name of the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_Type** (krpc_connection_t *connection*, *krpc_SpaceCenter_VesselType_t* * *result*)

void **krpc_SpaceCenter_Vessel_set_Type** (*krpc_SpaceCenter_VesselType_t value*)
The type of the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_Situation** (krpc_connection_t *connection*, *krpc_SpaceCenter_VesselSituation_t* * *result*)
The situation the vessel is in.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_Recoverable** (krpc_connection_t *connection*, bool * *result*)
Whether the vessel is recoverable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_Recover** (krpc_connection_t *connection*)
Recover the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_MET** (krpc_connection_t *connection*, double * *result*)
The mission elapsed time in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_Biome** (krpc_connection_t *connection*, char * * *result*)
The name of the biome the vessel is currently in.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_Flight** (krpc_connection_t *connection*, *krpc_SpaceCenter_Flight_t* * *result*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*)
Returns a `krpc_SpaceCenter_Flight_t` 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 (`krpc_SpaceCenter_Vessel_SurfaceReferenceFrame()`).

Game Scenes 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*.

---

krpc_error_t **krpc_SpaceCenter_Vessel_Orbit** (krpc_connection_t *connection*, *krpc_SpaceCenter_Orbit_t* * *result*)
The current orbit of the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_Control** (krpc_connection_t *connection*, *krpc_SpaceCenter_Control_t* * *result*)
Returns a `krpc_SpaceCenter_Control_t` object that can be used to manipulate the vessel's control inputs. For example, its pitch/yaw/roll controls, RCS and thrust.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_Comms** (krpc_connection_t *connection*, *krpc_SpaceCenter_Comms_t* * *result*)
Returns a `krpc_SpaceCenter_Comms_t` object that can be used to interact with CommNet for this vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_AutoPilot** (krpc_connection_t *connection*, *krpc_SpaceCenter_AutoPilot_t* * *result*)
An `krpc_SpaceCenter_AutoPilot_t` object, that can be used to perform simple auto-piloting of the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_CrewCapacity** (krpc_connection_t *connection*, int32_t * *result*)
The number of crew that can occupy the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_CrewCount** (krpc_connection_t *connection*, int32_t * *result*)

The number of crew that are occupying the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_Crew** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The crew in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_Resources** (krpc_connection_t *connection*, *krpc_SpaceCenter_Resources_t* * *result*)

A `krpc_SpaceCenter_Resources_t` object, that can used to get information about resources stored in the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_ResourcesInDecoupleStage** (krpc_connection_t *connection*, *krpc_SpaceCenter_Resources_t* * *result*, int32_t *stage*, bool *cumulative*)

Returns a `krpc_SpaceCenter_Resources_t` object, that can 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.

Game Scenes Flight

Note: For details on stage numbering, see the discussion on *Staging*.

krpc_error_t **krpc_SpaceCenter_Vessel_Parts** (krpc_connection_t *connection*, *krpc_SpaceCenter_Parts_t* * *result*)

A `krpc_SpaceCenter_Parts_t` object, that can used to interact with the parts that make up this vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_Mass** (krpc_connection_t *connection*, float * *result*)

The total mass of the vessel, including resources, in kg.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_DryMass** (krpc_connection_t *connection*, float * *result*)

The total mass of the vessel, excluding resources, in kg.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_Thrust** (krpc_connection_t *connection*, float * *result*)

The total thrust currently being produced by the vessel's engines, in Newtons. This is computed by summing `krpc_SpaceCenter_Engine_Thrust()` for every engine in the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_AvailableThrust** (krpc_connection_t *connection*, float * *result*)

Gets the total available thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *krpc_SpaceCenter_Engine_AvailableThrust()* for every active engine in the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_MaxThrust** (krpc_connection_t *connection*, float * *result*)

The total maximum thrust that can be produced by the vessel's active engines, in Newtons. This is computed by summing *krpc_SpaceCenter_Engine_MaxThrust()* for every active engine.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_MaxVacuumThrust** (krpc_connection_t *connection*, float * *result*)

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 *krpc_SpaceCenter_Engine_MaxVacuumThrust()* for every active engine.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_SpecificImpulse** (krpc_connection_t *connection*, float * *result*)

The combined specific impulse of all active engines, in seconds. This is computed using the formula described here.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_VacuumSpecificImpulse** (krpc_connection_t *connection*, float * *result*)

The combined vacuum specific impulse of all active engines, in seconds. This is computed using the formula described here.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_KerbinSeaLevelSpecificImpulse** (krpc_connection_t *connection*, float * *result*)

The combined specific impulse of all active engines at sea level on Kerbin, in seconds. This is computed using the formula described here.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_MomentOfInertia** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

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 (*krpc_SpaceCenter_ReferenceFrame_t*).

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_InertiaTensor** (krpc_connection_t *connection*, krpc_list_double_t * *result*)

The inertia tensor of the vessel around its center of mass, in the vessels reference frame (*krpc_SpaceCenter_ReferenceFrame_t*). Returns the 3x3 matrix as a list of elements, in row-major order.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Vessel_AvailableTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tuple_double_ * *result*)

The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gimballed engines and aerodynamic control surfaces. Returns the torques in $N.m$ around each of the coordinate axes of the vessels reference frame (`krpc_SpaceCenter_ReferenceFrame_t`). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_AvailableReactionWheelTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_do * *result*)

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 (`krpc_SpaceCenter_ReferenceFrame_t`). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_AvailableRCSTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tuple_do * *result*)

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 (`krpc_SpaceCenter_ReferenceFrame_t`). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_AvailableEngineTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tuple * *result*)

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 (`krpc_SpaceCenter_ReferenceFrame_t`). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_AvailableControlSurfaceTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_d * *result*)

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 (`krpc_SpaceCenter_ReferenceFrame_t`). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_AvailableOtherTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tuple_ * *result*)

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 (*krpc_SpaceCenter_ReferenceFrame_t*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_ReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t * result*)

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.

Game Scenes Flight

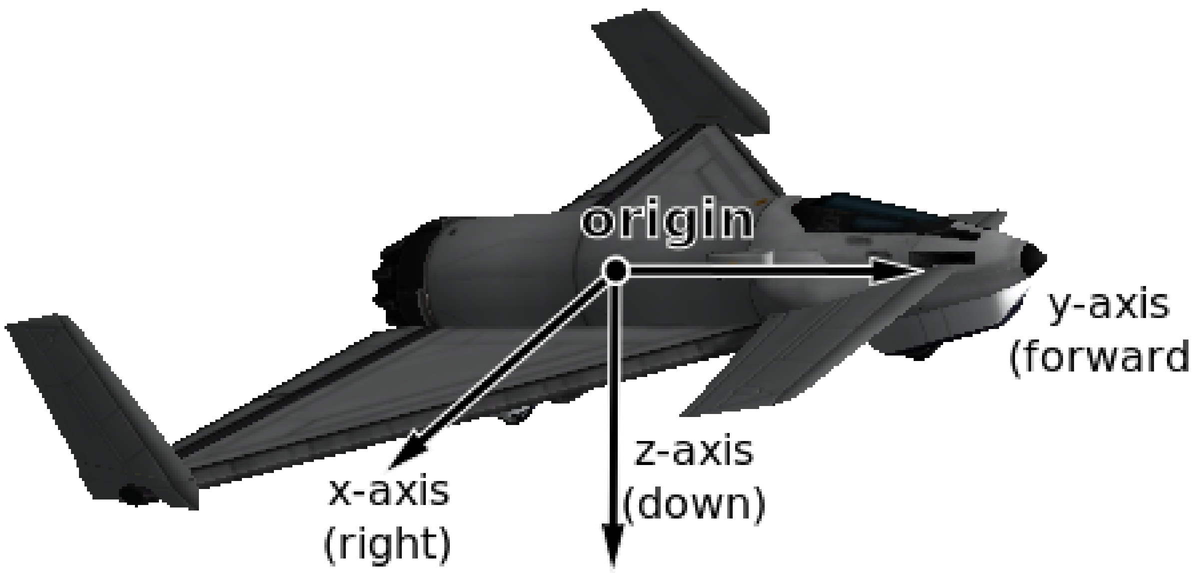


Fig. 1: Vessel reference frame origin and axes for the Aeris 3A aircraft

krpc_error_t **krpc_SpaceCenter_Vessel_OrbitalReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t * result*)

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.

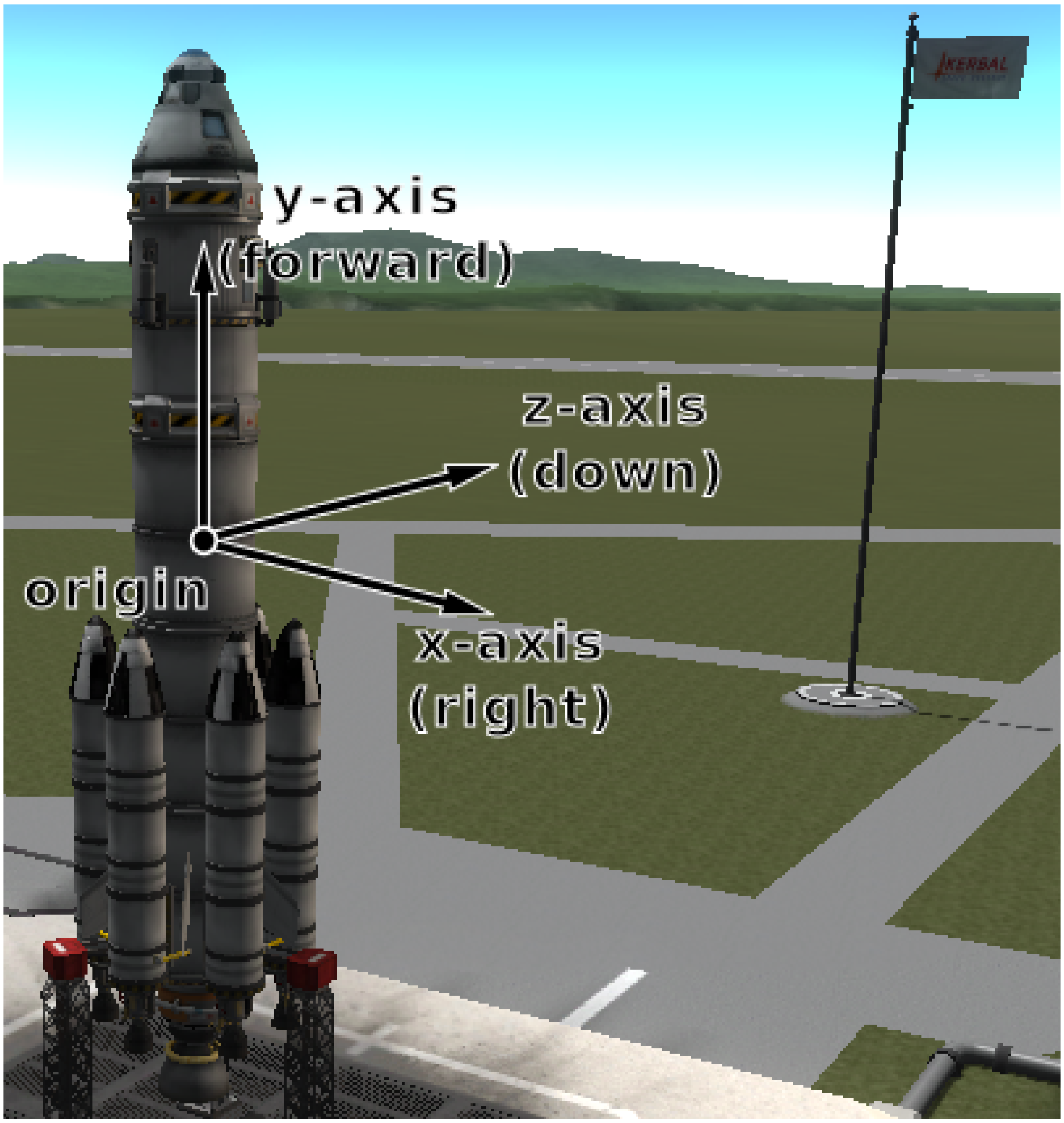


Fig. 2: Vessel reference frame origin and axes for the Kerbal-X rocket

Game Scenes Flight

Note: Be careful not to confuse this with 'orbit' mode on the navball.

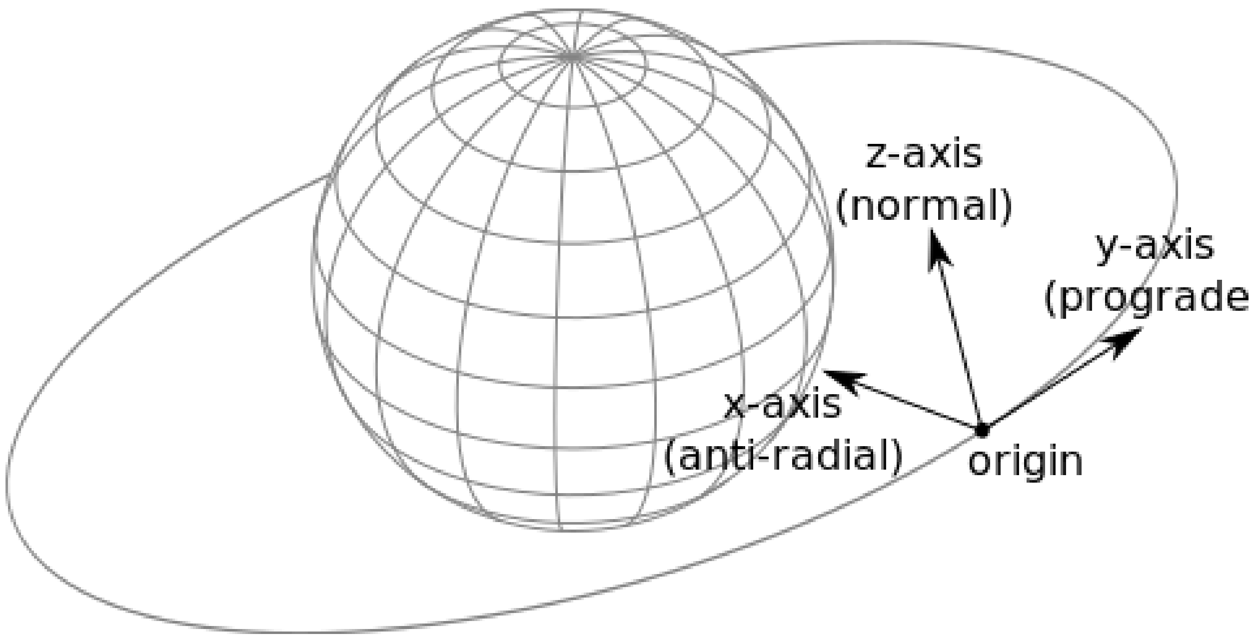


Fig. 3: Vessel orbital reference frame origin and axes

krpc_error_t **krpc_SpaceCenter_Vessel_SurfaceReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t * result*)

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).

Game Scenes Flight

Note: Be careful not to confuse this with 'surface' mode on the navball.

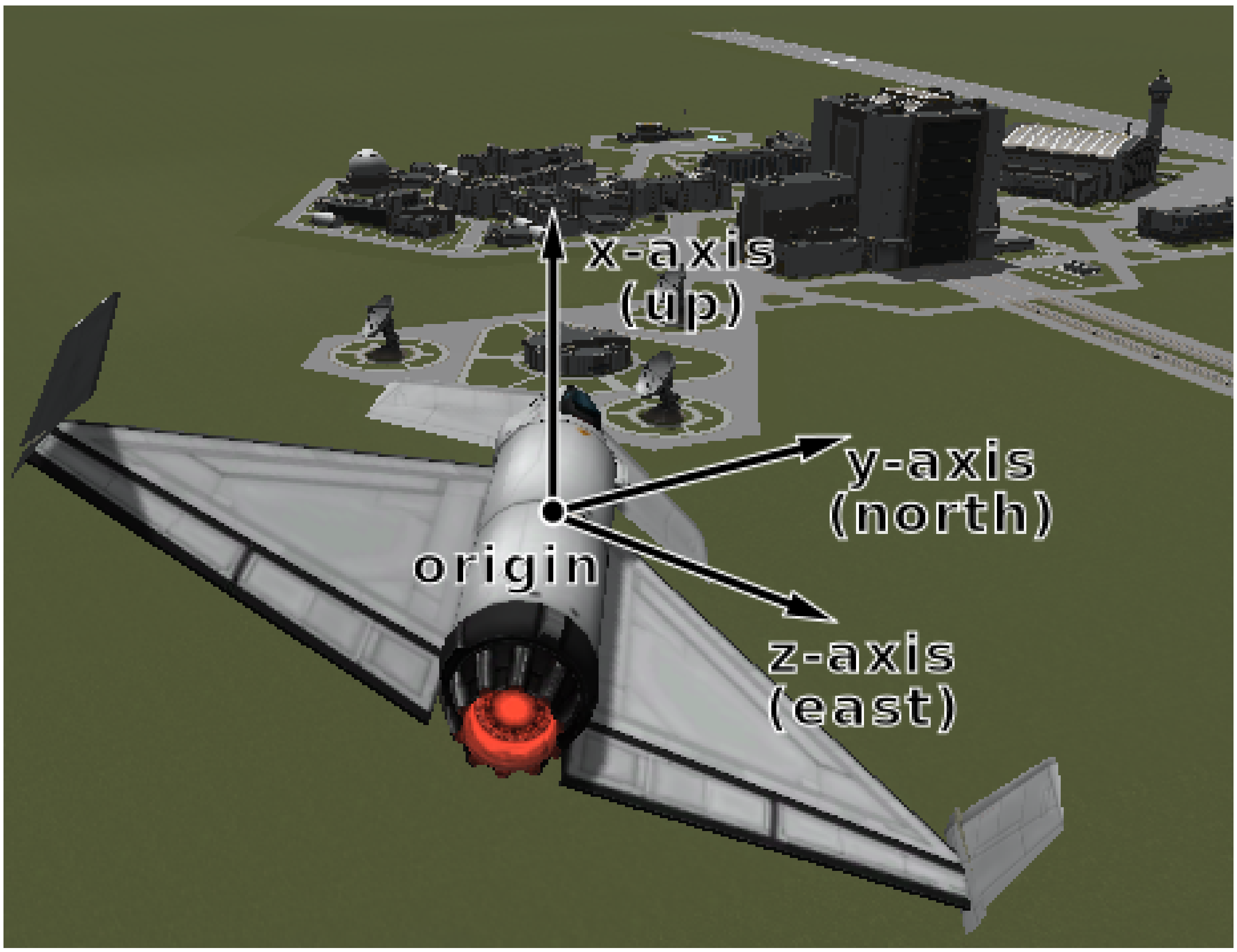


Fig. 4: Vessel surface reference frame origin and axes

krpc_error_t **krpc_SpaceCenter_Vessel_SurfaceVelocityReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFram* * *result*)

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.

Game Scenes Flight

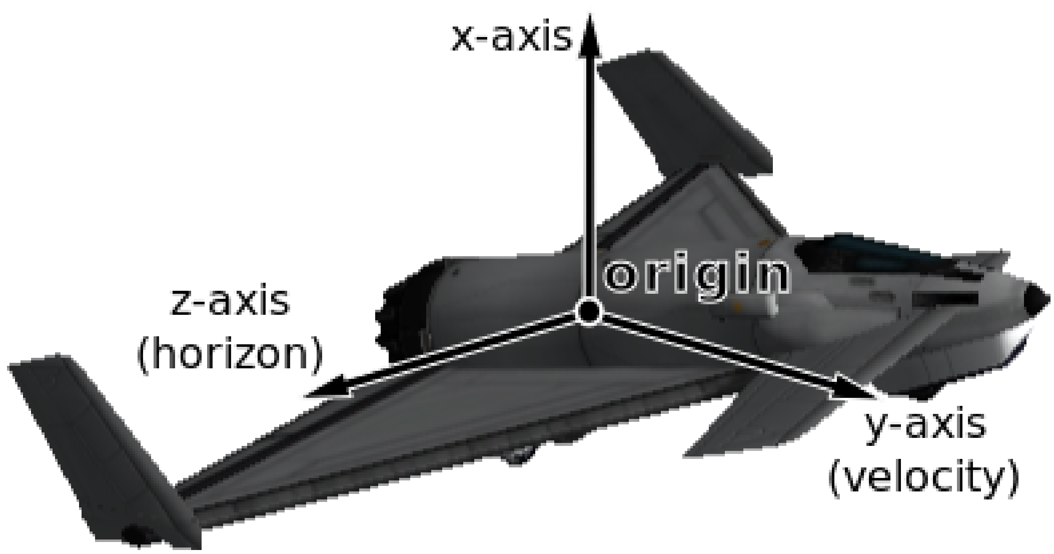


Fig. 5: Vessel surface velocity reference frame origin and axes

krpc_error_t **krpc_SpaceCenter_Vessel_Position** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_BoundingBox** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tuple_double_doub * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_Velocity** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_Rotation** (krpc_connection_t *connection*, krpc_tuple_double_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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).

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_Direction** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Vessel_AngularVelocity** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes Flight

krpc_SpaceCenter_VesselType_t

The type of a vessel. See *krpc_SpaceCenter_Vessel_Type()*.

KRPC_SPACECENTER_VESSELTYPE_BASE
Base.

KRPC_SPACECENTER_VESSELTYPE_DEBRIS
Debris.

KRPC_SPACECENTER_VESSELTYPE_LANDER
Lander.

KRPC_SPACECENTER_VESSELTYPE_PLANE
Plane.

KRPC_SPACECENTER_VESSELTYPE_PROBE
Probe.

KRPC_SPACECENTER_VESSELTYPE_RELAY
Relay.

KRPC_SPACECENTER_VESSELTYPE_ROVER
Rover.

KRPC_SPACECENTER_VESSELTYPE_SHIP
Ship.

KRPC_SPACECENTER_VESSELTYPE_STATION
Station.

krpc_SpaceCenter_VesselSituation_t

The situation a vessel is in. See *krpc_SpaceCenter_Vessel_Situation()*.

KRPC_SPACECENTER_VESSELSITUATION_DOCKED
Vessel is docked to another.

KRPC_SPACECENTER_VESSELSITUATION_ESCAPING
Escaping.

KRPC_SPACECENTER_VESSELSITUATION_FLYING
Vessel is flying through an atmosphere.

KRPC_SPACECENTER_VESSELSITUATION_LANDED
Vessel is landed on the surface of a body.

KRPC_SPACECENTER_VESSELSITUATION_ORBITING
Vessel is orbiting a body.

KRPC_SPACECENTER_VESSELSITUATION_PRELAUNCH
Vessel is awaiting launch.

KRPC_SPACECENTER_VESSELSITUATION_SPLASHED
Vessel has splashed down in an ocean.

KRPC_SPACECENTER_VESSELSITUATION_SUBORBITAL
Vessel is on a sub-orbital trajectory.

krpc_SpaceCenter_CrewMember_t

Represents crew in a vessel. Can be obtained using `krpc_SpaceCenter_Vessel_Crew()`.

krpc_error_t **krpc_SpaceCenter_CrewMember_Name** (krpc_connection_t *connection*, char * * *result*)

void **krpc_SpaceCenter_CrewMember_set_Name** (const char * *value*)

The crew members name.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CrewMember_Type** (krpc_connection_t *connection*, *krpc_SpaceCenter_CrewMemberType_t* * *result*)

The type of crew member.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CrewMember_OnMission** (krpc_connection_t *connection*, bool * *result*)

Whether the crew member is on a mission.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CrewMember_Courage** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_CrewMember_set_Courage** (float *value*)

The crew members courage.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CrewMember_Stupidity** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_CrewMember_set_Stupidity** (float *value*)

The crew members stupidity.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CrewMember_Experience** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_CrewMember_set_Experience** (float *value*)

The crew members experience.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CrewMember_Badass** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_CrewMember_set_Badass** (bool *value*)

Whether the crew member is a badass.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CrewMember_Veteran** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_CrewMember_set_Veteran** (bool *value*)

Whether the crew member is a veteran.

Game Scenes All

krpc_SpaceCenter_CrewMemberType_t

The type of a crew member. See `krpc_SpaceCenter_CrewMember_Type()`.

KRPC_SPACECENTER_CREWMEMBERTYPE_APPLICANT
An applicant for crew.

KRPC_SPACECENTER_CREWMEMBERTYPE_CREW
Rocket crew.

KRPC_SPACECENTER_CREWMEMBERTYPE_TOURIST
A tourist.

KRPC_SPACECENTER_CREWMEMBERTYPE_UNOWNED
An unowned crew member.

3.3.3 CelestialBody

krpc_SpaceCenter_CelestialBody_t
Represents a celestial body (such as a planet or moon). See `krpc_SpaceCenter_Bodies()`.

krpc_error_t **krpc_SpaceCenter_CelestialBody_Name** (krpc_connection_t *connection*, char * * *result*)
The name of the body.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_Satellites** (krpc_connection_t *connection*, krpc_list_object_t * *result*)
A list of celestial bodies that are in orbit around this celestial body.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_Orbit** (krpc_connection_t *connection*, *krpc_SpaceCenter_Orbit_t* * *result*)
The orbit of the body.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_Mass** (krpc_connection_t *connection*, float * *result*)
The mass of the body, in kilograms.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_GravitationalParameter** (krpc_connection_t *connection*, float * *result*)
The standard gravitational parameter of the body in m^3s^{-2}.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_SurfaceGravity** (krpc_connection_t *connection*, float * *result*)
The acceleration due to gravity at sea level (mean altitude) on the body, in m/s^2.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_RotationalPeriod** (krpc_connection_t *connection*, float * *result*)
The sidereal rotational period of the body, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_RotationalSpeed**(krpc_connection_t *connection*, float * *result*)

The rotational speed of the body, in radians per second.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_RotationAngle**(krpc_connection_t *connection*, double * *result*)

The current rotation angle of the body, in radians. A value between 0 and 2π

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_InitialRotation**(krpc_connection_t *connection*, double * *result*)

The initial rotation angle of the body (at UT 0), in radians. A value between 0 and 2π

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_EquatorialRadius**(krpc_connection_t *connection*, float * *result*)

The equatorial radius of the body, in meters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_SurfaceHeight**(krpc_connection_t *connection*, double * *result*, 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_BedrockHeight**(krpc_connection_t *connection*, double * *result*, 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_MSLPosition**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, double *latitude*, double *longitude*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_SurfacePosition**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, double *latitude*, double *longitude*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_BedrockPosition**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, double *latitude*, double *longitude*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_PositionAtAltitude**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, double *latitude*, double *longitude*, double *altitude*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_AltitudeAtPosition**(krpc_connection_t *connection*, double * *result*, const krpc_tuple_double_double_double_t * *position*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_LatitudeAtPosition**(krpc_connection_t *connection*, double * *result*, const krpc_tuple_double_double_double_t * *position*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_LongitudeAtPosition** (krpc_connection_t *connection*, double * *result*, const krpc_tuple_double_double_double_t * *position*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_SphereOfInfluence** (krpc_connection_t *connection*, float * *result*)

The radius of the sphere of influence of the body, in meters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_HasAtmosphere** (krpc_connection_t *connection*, bool * *result*)

`true` if the body has an atmosphere.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_AtmosphereDepth** (krpc_connection_t *connection*, float * *result*)

The depth of the atmosphere, in meters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_AtmosphericDensityAtPosition** (krpc_connection_t *connection*, double * *result*, const krpc_tuple_double_double_double_t * *position*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_HasAtmosphericOxygen** (krpc_connection_t *connection*, bool * *result*)

`true` if there is oxygen in the atmosphere, required for air-breathing engines.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_TemperatureAt** (krpc_connection_t *connection*, double * *result*, const krpc_tuple_double_double_double_t * *position*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

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.

krpc_error_t **krpc_SpaceCenter_CelestialBody_DensityAt** (krpc_connection_t *connection*, double * *result*, double *altitude*)

Gets the air density, in kg/m^3, for the specified altitude above sea level, in meters.

Parameters

Game Scenes All

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.

krpc_error_t **krpc_SpaceCenter_CelestialBody_PressureAt** (krpc_connection_t *connection*, double * *result*, double *altitude*)

Gets the air pressure, in Pascals, for the specified altitude above sea level, in meters.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_Biomes** (krpc_connection_t *connection*, krpc_set_string_t * *result*)

The biomes present on this body.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_BiomeAt** (krpc_connection_t *connection*, char * * *result*, double *latitude*, double *longitude*)

The biome at the given latitude and longitude, in degrees.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_FlyingHighAltitudeThreshold**(krpc_connection_t *connection*, float * *result*)

The altitude, in meters, above which a vessel is considered to be flying "high" when doing science.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_SpaceHighAltitudeThreshold**(krpc_connection_t *connection*, float * *result*)

The altitude, in meters, above which a vessel is considered to be in "high" space when doing science.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_ReferenceFrame**(krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t* * *result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_NonRotatingReferenceFrame**(krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceI* * *result*)

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.

Game Scenes All

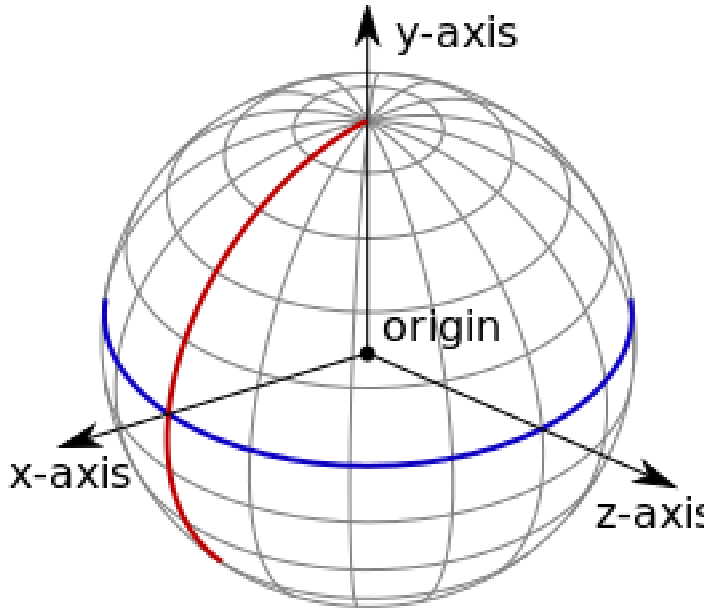


Fig. 6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

krpc_error_t **krpc_SpaceCenter_CelestialBody_OrbitalReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t * result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_Position** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, krpc_SpaceCenter_ReferenceFrame_t *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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_Velocity**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_Rotation**(krpc_connection_t *connection*, krpc_tuple_double_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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).

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_Direction**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CelestialBody_AngularVelocity**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

3.3.4 Flight

krpc_SpaceCenter_Flight_t

Used to get flight telemetry for a vessel, by calling *krpc_SpaceCenter_Vessel_Flight()*. All of the information returned by this class is given in the reference frame passed to that method. Obtained by calling *krpc_SpaceCenter_Vessel_Flight()*.

Note: To get orbital information, such as the apoapsis or inclination, see *krpc_SpaceCenter_Orbit_t*.

krpc_error_t **krpc_SpaceCenter_Flight_GForce**(krpc_connection_t *connection*, float * *result*)

The current G force acting on the vessel in g.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_MeanAltitude**(krpc_connection_t *connection*, double * *result*)

The altitude above sea level, in meters. Measured from the center of mass of the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_SurfaceAltitude**(krpc_connection_t *connection*, double * *result*)

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_BedrockAltitude**(krpc_connection_t *connection*, double * *result*)

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Elevation**(krpc_connection_t *connection*, double * *result*)

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Latitude**(krpc_connection_t *connection*, double * *result*)

The latitude of the vessel for the body being orbited, in degrees.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Longitude**(krpc_connection_t *connection*, double * *result*)

The longitude of the vessel for the body being orbited, in degrees.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Velocity**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The velocity of the vessel, in the reference frame *krpc_SpaceCenter_ReferenceFrame_t*.

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Speed**(krpc_connection_t *connection*, double * *result*)
The speed of the vessel in meters per second, in the reference frame *krpc_SpaceCenter_ReferenceFrame_t*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_HorizontalSpeed**(krpc_connection_t *connection*, double * *result*)
The horizontal speed of the vessel in meters per second, in the reference frame *krpc_SpaceCenter_ReferenceFrame_t*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_VerticalSpeed**(krpc_connection_t *connection*, double * *result*)
The vertical speed of the vessel in meters per second, in the reference frame *krpc_SpaceCenter_ReferenceFrame_t*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_CenterOfMass**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)
The position of the center of mass of the vessel, in the reference frame *krpc_SpaceCenter_ReferenceFrame_t*

Returns The position as a vector.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Rotation**(krpc_connection_t *connection*, krpc_tuple_double_double_double_double_t * *result*)
The rotation of the vessel, in the reference frame *krpc_SpaceCenter_ReferenceFrame_t*

Returns The rotation as a quaternion of the form (x, y, z, w).

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Direction**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)
The direction that the vessel is pointing in, in the reference frame *krpc_SpaceCenter_ReferenceFrame_t*.

Returns The direction as a unit vector.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Pitch**(krpc_connection_t *connection*, float * *result*)
The pitch of the vessel relative to the horizon, in degrees. A value between -90° and +90°.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Heading**(krpc_connection_t *connection*, float * *result*)
The heading of the vessel (its angle relative to north), in degrees. A value between 0° and 360°.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Roll**(krpc_connection_t *connection*, float * *result*)
The roll of the vessel relative to the horizon, in degrees. A value between -180° and +180°.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Prograde** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The prograde direction of the vessels orbit, in the reference frame `krpc_SpaceCenter_ReferenceFrame_t`.

Returns The direction as a unit vector.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Retrograde** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The retrograde direction of the vessels orbit, in the reference frame `krpc_SpaceCenter_ReferenceFrame_t`.

Returns The direction as a unit vector.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Normal** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The direction normal to the vessels orbit, in the reference frame `krpc_SpaceCenter_ReferenceFrame_t`.

Returns The direction as a unit vector.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_AntiNormal** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The direction opposite to the normal of the vessels orbit, in the reference frame `krpc_SpaceCenter_ReferenceFrame_t`.

Returns The direction as a unit vector.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Radial** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The radial direction of the vessels orbit, in the reference frame `krpc_SpaceCenter_ReferenceFrame_t`.

Returns The direction as a unit vector.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_AntiRadial** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The direction opposite to the radial direction of the vessels orbit, in the reference frame `krpc_SpaceCenter_ReferenceFrame_t`.

Returns The direction as a unit vector.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_AtmosphereDensity** (krpc_connection_t *connection*, float * *result*)

The current density of the atmosphere around the vessel, in kg/m^3.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_DynamicPressure**(krpc_connection_t *connection*, float * *result*)

The dynamic pressure acting on the vessel, in Pascals. This is a measure of the strength of the aerodynamic forces. It is equal to $\frac{1}{2}$.air density.velocity^2. It is commonly denoted Q.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_StaticPressure**(krpc_connection_t *connection*, float * *result*)

The static atmospheric pressure acting on the vessel, in Pascals.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_StaticPressureAtMSL**(krpc_connection_t *connection*, float * *result*)

The static atmospheric pressure at mean sea level, in Pascals.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_AerodynamicForce**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The total aerodynamic forces acting on the vessel, in reference frame *krpc_SpaceCenter_ReferenceFrame_t*.

Returns A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_SimulateAerodynamicForceAt**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_CelestialBody_t body*, const krpc_tuple_double_double_double_t * *position*, const krpc_tuple_double_double_double_t * *velocity*)

Simulate and return the total aerodynamic forces acting on the vessel, if it where 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Lift**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Drag**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_SpeedOfSound**(krpc_connection_t *connection*, float * *result*)

The speed of sound, in the atmosphere around the vessel, in m/s.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_Mach**(krpc_connection_t *connection*, float * *result*)

The speed of the vessel, in multiples of the speed of sound.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_ReynoldsNumber**(krpc_connection_t *connection*, float * *result*)

The vessels Reynolds number.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

---

krpc_error_t **krpc_SpaceCenter_Flight_TrueAirSpeed**(krpc_connection_t *connection*, float * *result*)

The true air speed of the vessel, in meters per second.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_EquivalentAirSpeed**(krpc_connection_t *connection*, float * *result*)

The equivalent air speed of the vessel, in meters per second.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_TerminalVelocity**(krpc_connection_t *connection*, float * *result*)

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_AngleOfAttack**(krpc_connection_t *connection*, float * *result*)

The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_SideslipAngle**(krpc_connection_t *connection*, float * *result*)

The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_TotalAirTemperature**(krpc_connection_t *connection*, float * *result*)

The total air temperature of the atmosphere around the vessel, in Kelvin. This includes the `krpc_SpaceCenter_Flight_StaticAirTemperature()` and the vessel's kinetic energy.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_StaticAirTemperature**(krpc_connection_t *connection*, float * *result*)

The static (ambient) temperature of the atmosphere around the vessel, in Kelvin.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Flight_StallFraction** (krpc_connection_t *connection*, float * *result*)

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.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

krpc_error_t **krpc_SpaceCenter_Flight_DragCoefficient** (krpc_connection_t *connection*, float * *result*)

The coefficient of drag. This is the amount of drag produced by the vessel. It depends on air speed, air density and wing area.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

krpc_error_t **krpc_SpaceCenter_Flight_LiftCoefficient** (krpc_connection_t *connection*, float * *result*)

The coefficient of lift. This is the amount of lift produced by the vessel, and depends on air speed, air density and wing area.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

krpc_error_t **krpc_SpaceCenter_Flight_BallisticCoefficient** (krpc_connection_t *connection*, float * *result*)

The ballistic coefficient.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

krpc_error_t **krpc_SpaceCenter_Flight_ThrustSpecificFuelConsumption** (krpc_connection_t *connection*, float * *result*)

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.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

3.3.5 Orbit

krpc_SpaceCenter_Orbit_t

Describes an orbit. For example, the orbit of a vessel, obtained by calling *krpc_SpaceCenter_Vessel_Orbit()*, or a celestial body, obtained by calling *krpc_SpaceCenter_CelestialBody_Orbit()*.

krpc_error_t **krpc_SpaceCenter_Orbit_Body** (krpc_connection_t *connection*, *krpc_SpaceCenter_CelestialBody_t * result*)

The celestial body (e.g. planet or moon) around which the object is orbiting.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_Apoapsis** (krpc_connection_t *connection*, double * *result*)

Gets the apoapsis of the orbit, in meters, from the center of mass of the body being orbited.

Game Scenes All

Note: For the apoapsis altitude reported on the in-game map view, use *krpc_SpaceCenter_Orbit_ApoapsisAltitude()*.

krpc_error_t **krpc_SpaceCenter_Orbit_Periapsis** (krpc_connection_t *connection*, double * *result*)

The periapsis of the orbit, in meters, from the center of mass of the body being orbited.

Game Scenes All

Note: For the periapsis altitude reported on the in-game map view, use *krpc_SpaceCenter_Orbit_PeriapsisAltitude()*.

krpc_error_t **krpc_SpaceCenter_Orbit_ApoapsisAltitude** (krpc_connection_t *connection*, double * *result*)

The apoapsis of the orbit, in meters, above the sea level of the body being orbited.

Game Scenes All

Note: This is equal to *krpc_SpaceCenter_Orbit_Apoapsis()* minus the equatorial radius of the body.

krpc_error_t **krpc_SpaceCenter_Orbit_PeriapsisAltitude** (krpc_connection_t *connection*, double * *result*)

The periapsis of the orbit, in meters, above the sea level of the body being orbited.

Game Scenes All

Note: This is equal to *krpc_SpaceCenter_Orbit_Periapsis()* minus the equatorial radius of the body.

krpc_error_t **krpc_SpaceCenter_Orbit_SemiMajorAxis** (krpc_connection_t *connection*, double * *result*)

The semi-major axis of the orbit, in meters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_SemiMinorAxis** (krpc_connection_t *connection*, double * *result*)

The semi-minor axis of the orbit, in meters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_Radius** (krpc_connection_t *connection*, double * *result*)

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.

Game Scenes All

Note: This value will change over time if the orbit is elliptical.

krpc_error_t **krpc_SpaceCenter_Orbit_RadiusAt** (krpc_connection_t *connection*, double * *result*, double *ut*)

The orbital radius at the given time, in meters.

Parameters

- **ut** – The universal time to measure the radius at.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_PositionAt** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, double *ut*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_Speed** (krpc_connection_t *connection*, double * *result*)

The current orbital speed of the object in meters per second.

Game Scenes All

Note: This value will change over time if the orbit is elliptical.

krpc_error_t **krpc_SpaceCenter_Orbit_Period** (krpc_connection_t *connection*, double * *result*)

The orbital period, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_TimeToApoapsis** (krpc_connection_t *connection*, double * *result*)

The time until the object reaches apoapsis, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_TimeToPeriapsis** (krpc_connection_t *connection*, double * *result*)

The time until the object reaches periapsis, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_Eccentricity** (krpc_connection_t *connection*, double * *result*)

The eccentricity of the orbit.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_Inclination** (krpc_connection_t *connection*, double * *result*)

The inclination of the orbit, in radians.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_LongitudeOfAscendingNode** (krpc_connection_t *connection*, double * *result*)

The longitude of the ascending node, in radians.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_ArgumentOfPeriapsis** (krpc_connection_t *connection*, double * *result*)

The argument of periapsis, in radians.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_MeanAnomalyAtEpoch** (krpc_connection_t *connection*, double * *result*)

The mean anomaly at epoch.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_Epoch** (krpc_connection_t *connection*, double * *result*)

The time since the epoch (the point at which the mean anomaly at epoch was measured, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_MeanAnomaly** (krpc_connection_t *connection*, double * *result*)

The mean anomaly.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_MeanAnomalyAtUT** (krpc_connection_t *connection*, double * *result*, double *ut*)

The mean anomaly at the given time.

Parameters

- **ut** – The universal time in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_EccentricAnomaly** (krpc_connection_t *connection*, double * *result*)

The eccentric anomaly.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_EccentricAnomalyAtUT** (krpc_connection_t *connection*, double * *result*, double *ut*)

The eccentric anomaly at the given universal time.

Parameters

- **ut** – The universal time, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_TrueAnomaly** (krpc_connection_t *connection*, double * *result*)

The true anomaly.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_TrueAnomalyAtUT** (krpc_connection_t *connection*, double * *result*, double *ut*)

The true anomaly at the given time.

Parameters

- **ut** – The universal time in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_TrueAnomalyAtRadius** (krpc_connection_t *connection*, double * *result*, double *radius*)

The true anomaly at the given orbital radius.

Parameters

- **radius** – The orbital radius in meters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_UTAtTrueAnomaly** (krpc_connection_t *connection*, double * *result*, double *trueAnomaly*)

The universal time, in seconds, corresponding to the given true anomaly.

Parameters

- **trueAnomaly** – True anomaly.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_RadiusAtTrueAnomaly** (krpc_connection_t *connection*, double * *result*, double *trueAnomaly*)

The orbital radius at the point in the orbit given by the true anomaly.

Parameters

- **trueAnomaly** – The true anomaly.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_TrueAnomalyAtAN** (krpc_connection_t *connection*, double * *result*, *krpc_SpaceCenter_Orbit_t target*)

The true anomaly of the ascending node with the given target orbit.

Parameters

- **target** – Target orbit.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_TrueAnomalyAtDN** (krpc_connection_t *connection*, double * *result*, krpc_SpaceCenter_Orbit_t *target*)

The true anomaly of the descending node with the given target orbit.

Parameters

- **target** – Target orbit.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_OrbitalSpeed** (krpc_connection_t *connection*, double * *result*)

The current orbital speed in meters per second.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_OrbitalSpeedAt** (krpc_connection_t *connection*, double * *result*, double *time*)

The orbital speed at the given time, in meters per second.

Parameters

- **time** – Time from now, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_ReferencePlaneNormal** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, krpc_SpaceCenter_ReferenceFrame_t *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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_ReferencePlaneDirection** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, krpc_SpaceCenter_ReferenceFrame_t *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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_RelativeInclination** (krpc_connection_t *connection*, double * *result*, *krpc_SpaceCenter_Orbit_t target*)

Relative inclination of this orbit and the target orbit, in radians.

Parameters

- **target** – Target orbit.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_TimeToSOIChange** (krpc_connection_t *connection*, double * *result*)

The time until the object changes sphere of influence, in seconds. Returns `NaN` if the object is not going to change sphere of influence.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_NextOrbit** (krpc_connection_t *connection*, *krpc_SpaceCenter_Orbit_t* * *result*)

If the object is going to change sphere of influence in the future, returns the new orbit after the change. Otherwise returns `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_TimeOfClosestApproach** (krpc_connection_t *connection*, double * *result*, *krpc_SpaceCenter_Orbit_t target*)

Estimates and returns the time at closest approach to a target orbit.

Parameters

- **target** – Target orbit.

Returns The universal time at closest approach, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_DistanceAtClosestApproach** (krpc_connection_t *connection*, double * *result*, *krpc_SpaceCenter_Orbit_t target*)

Estimates and returns the distance at closest approach to a target orbit, in meters.

Parameters

- **target** – Target orbit.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Orbit_ListClosestApproaches** (krpc_connection_t *connection*, krpc_list_list_double_t * *result*, *krpc_SpaceCenter_Orbit_t target*, int32_t *orbits*)

Returns the times at closest approach and corresponding distances, to a target orbit.

Parameters

- **target** – Target orbit.

- **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.

Game Scenes All

3.3.6 Control

krpc_SpaceCenter_Control_t

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 *krpc_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.

krpc_error_t **krpc_SpaceCenter_Control_Source** (krpc_connection_t *connection*, *krpc_SpaceCenter_ControlSource_t* * *result*)

The source of the vessels control, for example by a kerbal or a probe core.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_ControlState_t* * *result*)

The control state of the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_SAS** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_SAS** (bool *value*)

The state of SAS.

Game Scenes Flight

Note: Equivalent to *krpc_SpaceCenter_AutoPilot_SAS()*

krpc_error_t **krpc_SpaceCenter_Control_SASMode** (krpc_connection_t *connection*, *krpc_SpaceCenter_SASMode_t* * *result*)

void **krpc_SpaceCenter_Control_set_SASMode** (*krpc_SpaceCenter_SASMode_t value*)

The current *krpc_SpaceCenter_SASMode_t*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

Game Scenes Flight

Note: Equivalent to *krpc_SpaceCenter_AutoPilot_SASMode()*

krpc_error_t **krpc_SpaceCenter_Control_SpeedMode** (krpc_connection_t *connection*, *krpc_SpaceCenter_SpeedMode_t* * *result*)

void **krpc_SpaceCenter_Control_set_SpeedMode** (*krpc_SpaceCenter_SpeedMode_t value*)
The current `krpc_SpaceCenter_SpeedMode_t` of the navball. This is the mode displayed next to the speed at the top of the navball.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_RCS** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_RCS** (bool *value*)
The state of RCS.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_ReactionWheels** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_ReactionWheels** (bool *value*)
Returns whether all reactive wheels on the vessel are active, and sets the active state of all reaction wheels. See `krpc_SpaceCenter_ReactionWheel_Active()`.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Gear** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_Gear** (bool *value*)
The state of the landing gear/legs.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Legs** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_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 `krpc_SpaceCenter_Leg_Deployed()`.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Wheels** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_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 `krpc_SpaceCenter_Wheel_Deployed()`.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Lights** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_Lights** (bool *value*)
The state of the lights.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Brakes** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_Brakes** (bool *value*)
The state of the wheel brakes.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Antennas** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_Antennas** (bool *value*)
Returns whether all antennas on the vessel are deployed, and sets the deployment state of all antennas. See *krpc_SpaceCenter_Antenna_Deployed()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_CargoBays** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_CargoBays** (bool *value*)
Returns whether any of the cargo bays on the vessel are open, and sets the open state of all cargo bays. See *krpc_SpaceCenter_CargoBay_Open()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Intakes** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_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 *krpc_SpaceCenter_Intake_Open()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Parachutes** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_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 *krpc_SpaceCenter_Parachute_Deployed()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Radiators** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_Radiators** (bool *value*)
Returns whether all radiators on the vessel are deployed, and sets the deployment state of all radiators. See *krpc_SpaceCenter_Radiator_Deployed()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_ResourceHarvesters** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_ResourceHarvesters** (bool *value*)
Returns whether all of the resource harvesters on the vessel are deployed, and sets the deployment state of all resource harvesters. See *krpc_SpaceCenter_ResourceHarvester_Deployed()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_ResourceHarvestersActive** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_ResourceHarvestersActive** (bool *value*)
Returns whether any of the resource harvesters on the vessel are active, and sets the active state of all resource harvesters. See *krpc_SpaceCenter_ResourceHarvester_Active()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_SolarPanels** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_SolarPanels** (bool *value*)
Returns whether all solar panels on the vessel are deployed, and sets the deployment state of all solar panels. See *krpc_SpaceCenter_SolarPanel_Deployed()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Abort** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Control_set_Abort** (bool *value*)
The state of the abort action group.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Throttle** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Control_set_Throttle** (float *value*)
The state of the throttle. A value between 0 and 1.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_InputMode** (krpc_connection_t *connection*, *krpc_SpaceCenter_ControlInputMode_t* * *result*)

void **krpc_SpaceCenter_Control_set_InputMode** (*krpc_SpaceCenter_ControlInputMode_t 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Pitch** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Control_set_Pitch** (float *value*)
The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Yaw** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Control_set_Yaw** (float *value*)
The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Roll** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Control_set_Roll** (float *value*)
The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Forward** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Control_set_Forward** (float *value*)
The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Up** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Control_set_Up** (float *value*)
The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Right** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Control_set_Right** (float *value*)
The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_WheelThrottle** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Control_set_WheelThrottle** (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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_WheelSteering** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Control_set_WheelSteering** (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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_CurrentStage** (krpc_connection_t *connection*, int32_t * *result*)
The current stage of the vessel. Corresponds to the stage number in the in-game UI.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_ActivateNextStage** (krpc_connection_t *connection*, krpc_list_object_t * *result*)
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.

Game Scenes Flight

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 `krpc_SpaceCenter_ActiveVessel()` no longer refer to the active vessel.

---

krpc_error_t **krpc_SpaceCenter_Control_GetActionGroup** (krpc_connection_t *connection*, bool * *result*, 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_SetActionGroup** (krpc_connection_t *connection*, 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_ToggleActionGroup** (krpc_connection_t *connection*, 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_AddNode** (krpc_connection_t *connection*, *krpc_SpaceCenter_Node_t* * *result*, double *ut*, float *prograde*, float *normal*, float *radial*)

Creates a maneuver node at the given universal time, and returns a `krpc_SpaceCenter_Node_t` 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_Nodes** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

Returns a list of all existing maneuver nodes, ordered by time from first to last.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Control_RemoveNodes** (krpc_connection_t *connection*)

Remove all maneuver nodes.

Game Scenes Flight

krpc_SpaceCenter_ControlState_t

The control state of a vessel. See `krpc_SpaceCenter_Control_State()`.

KRPC_SPACECENTER_CONTROLSTATE_FULL

Full controllable.

KRPC_SPACECENTER_CONTROLSTATE_PARTIAL

Partially controllable.

KRPC_SPACECENTER_CONTROLSTATE_NONE

Not controllable.

krpc_SpaceCenter_ControlSource_t

The control source of a vessel. See `krpc_SpaceCenter_Control_Source()`.

KRPC_SPACECENTER_CONTROLSOURCE_KERBAL

Vessel is controlled by a Kerbal.

KRPC_SPACECENTER_CONTROLSOURCE_PROBE
Vessel is controlled by a probe core.

KRPC_SPACECENTER_CONTROLSOURCE_NONE
Vessel is not controlled.

krpc_SpaceCenter_SASMode_t
The behavior of the SAS auto-pilot. See *krpc_SpaceCenter_AutoPilot_SASMode()*.

KRPC_SPACECENTER_SASMODE_STABILITYASSIST
Stability assist mode. Dampen out any rotation.

KRPC_SPACECENTER_SASMODE_MANEUVER
Point in the burn direction of the next maneuver node.

KRPC_SPACECENTER_SASMODE_PROGRADE
Point in the prograde direction.

KRPC_SPACECENTER_SASMODE_RETROGRADE
Point in the retrograde direction.

KRPC_SPACECENTER_SASMODE_NORMAL
Point in the orbit normal direction.

KRPC_SPACECENTER_SASMODE_ANTINORMAL
Point in the orbit anti-normal direction.

KRPC_SPACECENTER_SASMODE_RADIAL
Point in the orbit radial direction.

KRPC_SPACECENTER_SASMODE_ANTIRADIAL
Point in the orbit anti-radial direction.

KRPC_SPACECENTER_SASMODE_TARGET
Point in the direction of the current target.

KRPC_SPACECENTER_SASMODE_ANTITARGET
Point away from the current target.

krpc_SpaceCenter_SpeedMode_t
The mode of the speed reported in the navball. See *krpc_SpaceCenter_Control_SpeedMode()*.

KRPC_SPACECENTER_SPEEDMODE_ORBIT
Speed is relative to the vessel's orbit.

KRPC_SPACECENTER_SPEEDMODE_SURFACE
Speed is relative to the surface of the body being orbited.

KRPC_SPACECENTER_SPEEDMODE_TARGET
Speed is relative to the current target.

krpc_SpaceCenter_ControlInputMode_t
See *krpc_SpaceCenter_Control_InputMode()*.

KRPC_SPACECENTER_CONTROLINPUTMODE_ADDITIVE
Control inputs are added to the vessels current control inputs.

KRPC_SPACECENTER_CONTROLINPUTMODE_OVERRIDE
Control inputs (when they are non-zero) override the vessels current control inputs.

3.3.7 Communications

krpc_SpaceCenter_Comms_t

Used to interact with CommNet for a given vessel. Obtained by calling *krpc_SpaceCenter_Vessel_Comms()*.

krpc_error_t **krpc_SpaceCenter_Comms_CanCommunicate** (krpc_connection_t *connection*, bool * *result*)

Whether the vessel can communicate with KSC.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Comms_CanTransmitScience** (krpc_connection_t *connection*, bool * *result*)

Whether the vessel can transmit science data to KSC.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Comms_SignalStrength** (krpc_connection_t *connection*, double * *result*)

Signal strength to KSC.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Comms_SignalDelay** (krpc_connection_t *connection*, double * *result*)

Signal delay to KSC in seconds.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Comms_Power** (krpc_connection_t *connection*, double * *result*)

The combined power of all active antennae on the vessel.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Comms_ControlPath** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The communication path used to control the vessel.

Game Scenes Flight

krpc_SpaceCenter_CommLink_t

Represents a communication node in the network. For example, a vessel or the KSC.

krpc_error_t **krpc_SpaceCenter_CommLink_Type** (krpc_connection_t *connection*, *krpc_SpaceCenter_CommLinkType_t* * *result*)

The type of link.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CommLink_SignalStrength** (krpc_connection_t *connection*, double * *result*)

Signal strength of the link.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CommLink_Start** (krpc_connection_t *connection*, *krpc_SpaceCenter_CommNode_t* * *result*)

Start point of the link.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CommLink_End** (krpc_connection_t *connection*, *krpc_SpaceCenter_CommNode_t * result*)

Start point of the link.

Game Scenes All

krpc_SpaceCenter_CommLinkType_t

The type of a communication link. See *krpc_SpaceCenter_CommLink_Type()*.

KRPC_SPACECENTER_COMMLINKTYPE_HOME

Link is to a base station on Kerbin.

KRPC_SPACECENTER_COMMLINKTYPE_CONTROL

Link is to a control source, for example a manned spacecraft.

KRPC_SPACECENTER_COMMLINKTYPE_RELAY

Link is to a relay satellite.

krpc_SpaceCenter_CommNode_t

Represents a communication node in the network. For example, a vessel or the KSC.

krpc_error_t **krpc_SpaceCenter_CommNode_Name** (krpc_connection_t *connection*, char * * *result*)

Name of the communication node.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CommNode_IsHome** (krpc_connection_t *connection*, bool * *result*)

Whether the communication node is on Kerbin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CommNode_IsControlPoint** (krpc_connection_t *connection*, bool * *result*)

Whether the communication node is a control point, for example a manned vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CommNode_IsVessel** (krpc_connection_t *connection*, bool * *result*)

Whether the communication node is a vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CommNode_Vessel** (krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t * result*)

The vessel for this communication node.

Game Scenes All

3.3.8 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*

Parts

`krpc_SpaceCenter_Parts_t`

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling `krpc_SpaceCenter_Vessel_Parts()`.

krpc_error_t **`krpc_SpaceCenter_Parts_All`** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all of the vessels parts.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Root** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The vessels root part.

Game Scenes All

Note: See the discussion on *Trees of Parts*.

krpc_error_t **krpc_SpaceCenter_Parts_Controlling** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

void **krpc_SpaceCenter_Parts_set_Controlling** (*krpc_SpaceCenter_Part_t value*)

The part from which the vessel is controlled.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_WithName** (krpc_connection_t *connection*, krpc_list_object_t * *result*, const char * *name*)

A list of parts whose `krpc_SpaceCenter_Part_Name()` is *name*.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_WithTitle** (krpc_connection_t *connection*, krpc_list_object_t * *result*, const char * *title*)

A list of all parts whose `krpc_SpaceCenter_Part_Title()` is *title*.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_WithTag** (krpc_connection_t *connection*, krpc_list_object_t * *result*, const char * *tag*)

A list of all parts whose `krpc_SpaceCenter_Part_Tag()` is *tag*.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_WithModule** (krpc_connection_t *connection*, krpc_list_object_t * *result*, const char * *moduleName*)

A list of all parts that contain a `krpc_SpaceCenter_Module_t` whose `krpc_SpaceCenter_Module_Name()` is *moduleName*.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_InStage** (krpc_connection_t *connection*, krpc_list_object_t * *result*, int32_t *stage*)

A list of all parts that are activated in the given *stage*.

Parameters

Game Scenes All

Note: See the discussion on *Staging*.

krpc_error_t **krpc_SpaceCenter_Parts_InDecoupleStage** (krpc_connection_t *connection*, krpc_list_object_t * *result*, int32_t *stage*)

A list of all parts that are decoupled in the given *stage*.

Parameters

Game Scenes All

Note: See the discussion on *Staging*.

krpc_error_t **krpc_SpaceCenter_Parts_ModulesWithName** (krpc_connection_t *connection*, krpc_list_object_t * *result*, const char * *moduleName*)

A list of modules (combined across all parts in the vessel) whose `krpc_SpaceCenter_Module_Name()` is *moduleName*.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Antennas** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all antennas in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_CargoBays** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all cargo bays in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_ControlSurfaces** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all control surfaces in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Decouplers** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all decouplers in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_DockingPorts** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all docking ports in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Engines** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all engines in the vessel.

Game Scenes All

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.

krpc_error_t **krpc_SpaceCenter_Parts_Experiments** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all science experiments in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Fairings** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all fairings in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Intakes** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all intakes in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Legs** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all landing legs attached to the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_LaunchClamps** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all launch clamps attached to the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Lights** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all lights in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Parachutes** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all parachutes in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Radiators** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all radiators in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_RCS** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all RCS blocks/thrusters in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_ReactionWheels** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all reaction wheels in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_ResourceConverters** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all resource converters in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_ResourceHarvesters** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all resource harvesters in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Sensors** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all sensors in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_SolarPanels** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all solar panels in the vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parts_Wheels** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all wheels in the vessel.

Game Scenes All

Part

krpc_SpaceCenter_Part_t

Represents an individual part. Vessels are made up of multiple parts. Instances of this class can be obtained by several methods in `krpc_SpaceCenter_Parts_t`.

krpc_error_t **krpc_SpaceCenter_Part_Name** (krpc_connection_t *connection*, char * * *result*)

Internal name of the part, as used in part cfg files. For example "Mark1-2Pod".

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Title** (krpc_connection_t *connection*, char * * *result*)

Title of the part, as shown when the part is right clicked in-game. For example "Mk1-2 Command Pod".

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Tag** (krpc_connection_t *connection*, char * * *result*)

void **krpc_SpaceCenter_Part_set_Tag** (const char * *value*)

The name tag for the part. Can be set to a custom string using the in-game user interface.

Game Scenes All

Note: This string is shared with kOS if it is installed.

krpc_error_t **krpc_SpaceCenter_Part_Highlighted** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Part_set_Highlighted** (bool *value*)

Whether the part is highlighted.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_HighlightColor** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*) void **krpc_SpaceCenter_Part_set_HighlightColor** (const krpc_tuple_double_double_double_t * *value*)

The color used to highlight the part, as an RGB triple.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Cost** (krpc_connection_t *connection*, double * *result*)

The cost of the part, in units of funds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Vessel** (krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t* * *result*)

The vessel that contains this part.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Parent** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The parts parent. Returns `nullptr` if the part does not have a parent. This, in combination with `krpc_SpaceCenter_Part_Children()`, can be used to traverse the vessels parts tree.

Game Scenes All

Note: See the discussion on *Trees of Parts*.

krpc_error_t **krpc_SpaceCenter_Part_Children** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The parts children. Returns an empty list if the part has no children. This, in combination with `krpc_SpaceCenter_Part_Parent()`, can be used to traverse the vessels parts tree.

Game Scenes All

Note: See the discussion on *Trees of Parts*.

krpc_error_t **krpc_SpaceCenter_Part_AxiallyAttached** (krpc_connection_t *connection*, bool * *result*)

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`.

Game Scenes All

Note: See the discussion on *Attachment Modes*.

krpc_error_t **krpc_SpaceCenter_Part_RadiallyAttached** (krpc_connection_t *connection*, bool * *result*)

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`.

Game Scenes All

Note: See the discussion on *Attachment Modes*.

krpc_error_t **krpc_SpaceCenter_Part_Stage** (krpc_connection_t *connection*, int32_t * *result*)

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

Game Scenes All

Note: See the discussion on *Staging*.

krpc_error_t **krpc_SpaceCenter_Part_DecoupleStage** (krpc_connection_t *connection*, int32_t * *result*)
The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

Game Scenes All

Note: See the discussion on *Staging*.

krpc_error_t **krpc_SpaceCenter_Part_Massless** (krpc_connection_t *connection*, bool * *result*)
Whether the part is massless.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Mass** (krpc_connection_t *connection*, double * *result*)
The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_DryMass** (krpc_connection_t *connection*, double * *result*)
The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Shielded** (krpc_connection_t *connection*, bool * *result*)
Whether the part is shielded from the exterior of the vessel, for example by a fairing.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_DynamicPressure** (krpc_connection_t *connection*, float * *result*)
The dynamic pressure acting on the part, in Pascals.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ImpactTolerance** (krpc_connection_t *connection*, double * *result*)
The impact tolerance of the part, in meters per second.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Temperature** (krpc_connection_t *connection*, double * *result*)
Temperature of the part, in Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_SkinTemperature** (krpc_connection_t *connection*, double * *result*)
Temperature of the skin of the part, in Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_MaxTemperature** (krpc_connection_t *connection*, double * *result*)
Maximum temperature that the part can survive, in Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_MaxSkinTemperature** (krpc_connection_t *connection*, double * *result*)

Maximum temperature that the skin of the part can survive, in Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ThermalMass** (krpc_connection_t *connection*, float * *result*)

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ThermalSkinMass** (krpc_connection_t *connection*, float * *result*)

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ThermalResourceMass** (krpc_connection_t *connection*, float * *result*)

A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ThermalConductionFlux** (krpc_connection_t *connection*, float * *result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ThermalConvectionFlux** (krpc_connection_t *connection*, float * *result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ThermalRadiationFlux** (krpc_connection_t *connection*, float * *result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ThermalInternalFlux** (krpc_connection_t *connection*, float * *result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ThermalSkinToInternalFlux** (krpc_connection_t *connection*, float * *result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Resources** (krpc_connection_t *connection*, *krpc_SpaceCenter_Resources_t * result*)

A *krpc_SpaceCenter_Resources_t* object for the part.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Crossfeed** (krpc_connection_t *connection*, bool * *result*)

Whether this part is crossfeed capable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_IsFuelLine** (krpc_connection_t *connection*, bool * *result*)

Whether this part is a fuel line.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_FuelLinesFrom** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

Game Scenes All

Note: See the discussion on *Fuel Lines*.

krpc_error_t **krpc_SpaceCenter_Part_FuelLinesTo** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

Game Scenes All

Note: See the discussion on *Fuel Lines*.

krpc_error_t **krpc_SpaceCenter_Part_Modules** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The modules for this part.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Antenna** (krpc_connection_t *connection*, *krpc_SpaceCenter_Antenna_t * result*)

A *krpc_SpaceCenter_Antenna_t* if the part is an antenna, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_CargoBay** (krpc_connection_t *connection*, *krpc_SpaceCenter_CargoBay_t * result*)

A *krpc_SpaceCenter_CargoBay_t* if the part is a cargo bay, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ControlSurface** (krpc_connection_t *connection*, *krpc_SpaceCenter_ControlSurface_t * result*)

A *krpc_SpaceCenter_ControlSurface_t* if the part is an aerodynamic control surface, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Decoupler** (krpc_connection_t *connection*, *krpc_SpaceCenter_Decoupler_t * result*)

A *krpc_SpaceCenter_Decoupler_t* if the part is a decoupler, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_DockingPort** (krpc_connection_t *connection*, *krpc_SpaceCenter_DockingPort_t * result*)

A *krpc_SpaceCenter_DockingPort_t* if the part is a docking port, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Engine** (krpc_connection_t *connection*, *krpc_SpaceCenter_Engine_t * result*)

An *krpc_SpaceCenter_Engine_t* if the part is an engine, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Experiment** (krpc_connection_t *connection*, *krpc_SpaceCenter_Experiment_t * result*)

An *krpc_SpaceCenter_Experiment_t* if the part is a science experiment, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Fairing** (krpc_connection_t *connection*, *krpc_SpaceCenter_Fairing_t * result*)

A *krpc_SpaceCenter_Fairing_t* if the part is a fairing, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Intake** (krpc_connection_t *connection*, *krpc_SpaceCenter_Intake_t * result*)

An *krpc_SpaceCenter_Intake_t* if the part is an intake, otherwise `nullptr`.

Game Scenes All

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 *krpc_SpaceCenter_RCS_t*.

---

krpc_error_t **krpc_SpaceCenter_Part_Leg** (krpc_connection_t *connection*, *krpc_SpaceCenter_Leg_t * result*)

A *krpc_SpaceCenter_Leg_t* if the part is a landing leg, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_LaunchClamp** (krpc_connection_t *connection*, *krpc_SpaceCenter_LaunchClamp_t * result*)

A *krpc_SpaceCenter_LaunchClamp_t* if the part is a launch clamp, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Light** (krpc_connection_t *connection*, *krpc_SpaceCenter_Light_t * result*)

A *krpc_SpaceCenter_Light_t* if the part is a light, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Parachute** (krpc_connection_t *connection*, *krpc_SpaceCenter_Parachute_t * result*)

A `krpc_SpaceCenter_Parachute_t` if the part is a parachute, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Radiator** (krpc_connection_t *connection*, *krpc_SpaceCenter_Radiator_t * result*)

A `krpc_SpaceCenter_Radiator_t` if the part is a radiator, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_RCS** (krpc_connection_t *connection*, *krpc_SpaceCenter_RCS_t * result*)

A `krpc_SpaceCenter_RCS_t` if the part is an RCS block/thruster, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ReactionWheel** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReactionWheel_t * result*)

A `krpc_SpaceCenter_ReactionWheel_t` if the part is a reaction wheel, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ResourceConverter** (krpc_connection_t *connection*, *krpc_SpaceCenter_ResourceConverter_t * result*)

A `krpc_SpaceCenter_ResourceConverter_t` if the part is a resource converter, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ResourceHarvester** (krpc_connection_t *connection*, *krpc_SpaceCenter_ResourceHarvester_t * result*)

A `krpc_SpaceCenter_ResourceHarvester_t` if the part is a resource harvester, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Sensor** (krpc_connection_t *connection*, *krpc_SpaceCenter_Sensor_t * result*)

A `krpc_SpaceCenter_Sensor_t` if the part is a sensor, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_SolarPanel** (krpc_connection_t *connection*, *krpc_SpaceCenter_SolarPanel_t * result*)

A `krpc_SpaceCenter_SolarPanel_t` if the part is a solar panel, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Wheel** (krpc_connection_t *connection*, *krpc_SpaceCenter_Wheel_t * result*)

A `krpc_SpaceCenter_Wheel_t` if the part is a wheel, otherwise `nullptr`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Position** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

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 *krpc_SpaceCenter_Part_CenterOfMass()* to get the parts center of mass.

krpc_error_t **krpc_SpaceCenter_Part_CenterOfMass** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*)

The position of the parts center of mass in the given reference frame. If the part is physicsless, this is equivalent to *krpc_SpaceCenter_Part_Position()*.

Parameters

- **referenceFrame** – The reference frame that the returned position vector is in.

Returns The position as a vector.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_BoundingBox** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tuple_double_double_ * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

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 *krpc_SpaceCenter_Part_Position()* of the part.

krpc_error_t **krpc_SpaceCenter_Part_Direction** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Velocity** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_Rotation** (krpc_connection_t *connection*, krpc_tuple_double_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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).

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_MomentOfInertia** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The moment of inertia of the part in $kg.m^2$ around its center of mass in the parts reference frame (*krpc_SpaceCenter_ReferenceFrame_t*).

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_InertiaTensor** (krpc_connection_t *connection*, krpc_list_double_t * *result*)

The inertia tensor of the part in the parts reference frame (*krpc_SpaceCenter_ReferenceFrame_t*). Returns the 3x3 matrix as a list of elements, in row-major order.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_ReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t* * *result*)

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 *krpc_SpaceCenter_Part_Position()*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

Game Scenes All

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *krpc_SpaceCenter_DockingPort_ReferenceFrame()*.

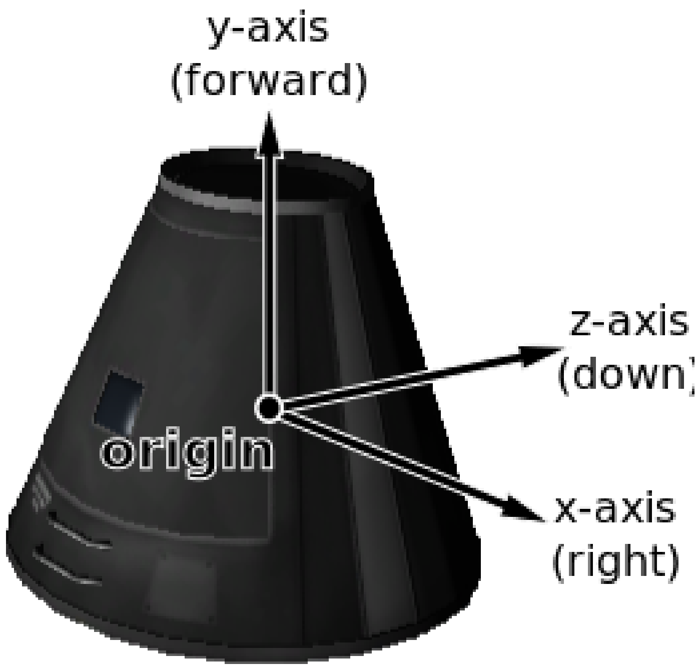


Fig. 7: Mk1 Command Pod reference frame origin and axes

krpc_error_t **krpc_SpaceCenter_Part_CenterOfMassReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t * result*)

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 *krpc_SpaceCenter_Part_CenterOfMass()*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

Game Scenes All

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *krpc_SpaceCenter_DockingPort_ReferenceFrame()*.

krpc_error_t **krpc_SpaceCenter_Part_AddForce** (krpc_connection_t *connection*, *krpc_SpaceCenter_Force_t * result*, const krpc_tuple_double_double_double_t * *force*, const krpc_tuple_double_double_double_t * *position*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Part_InstantaneousForce**(krpc_connection_t *connection*, const krpc_tuple_double_double_double_t * *force*, const krpc_tuple_double_double_double_t * *position*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

Note: The force is applied instantaneously in a single physics update.

krpc_SpaceCenter_Force_t

Obtained by calling *krpc_SpaceCenter_Part_AddForce()*.

krpc_error_t **krpc_SpaceCenter_Force_Part**(krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part that this force is applied to.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Force_ForceVector**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_SpaceCenter_Force_set_ForceVector**(const krpc_tuple_double_double_double_t * *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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Force_Position**(krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_SpaceCenter_Force_set_Position**(const krpc_tuple_double_double_double_t * *value*)

The position at which the force acts, in reference frame *krpc_SpaceCenter_ReferenceFrame_t*.

Returns The position as a vector.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Force_ReferenceFrame**(krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t * result*)

void **krpc_SpaceCenter_Force_set_ReferenceFrame**(*krpc_SpaceCenter_ReferenceFrame_t value*)

The reference frame of the force vector and position.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Force_Remove**(krpc_connection_t *connection*)

Remove the force.

Game Scenes All

Module

krpc_SpaceCenter_Module_t

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.

krpc_error_t **krpc_SpaceCenter_Module_Name**(krpc_connection_t *connection*, char * * *result*)

Name of the PartModule. For example, "ModuleEngines".

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_Part**(krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part that contains this module.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_Fields**(krpc_connection_t *connection*, krpc_dictionary_string_string_t * *result*)

The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_HasField**(krpc_connection_t *connection*, bool * *result*, const char * *name*)

Returns `true` if the module has a field with the given name.

Parameters

- **name** – Name of the field.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_GetField**(krpc_connection_t *connection*, char * * *result*, const char * *name*)

Returns the value of a field.

Parameters

- **name** – Name of the field.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_SetFieldInt**(krpc_connection_t *connection*, const char * *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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_SetFieldFloat** (krpc_connection_t *connection*, const char * *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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_SetFieldString** (krpc_connection_t *connection*, const char * *name*, const char * *value*)

Set the value of a field to the given string.

Parameters

- **name** – Name of the field.
- **value** – Value to set.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_ResetField** (krpc_connection_t *connection*, const char * *name*)

Set the value of a field to its original value.

Parameters

- **name** – Name of the field.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_Events** (krpc_connection_t *connection*, krpc_list_string_t * *result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_HasEvent** (krpc_connection_t *connection*, bool * *result*, const char * *name*)

`true` if the module has an event with the given name.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_TriggerEvent** (krpc_connection_t *connection*, const char * *name*)

Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_Actions** (krpc_connection_t *connection*, krpc_list_string_t * *result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_HasAction** (krpc_connection_t *connection*, bool * *result*, const char * *name*)

`true` if the part has an action with the given name.

Parameters

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Module_SetAction** (krpc_connection_t *connection*, const char * *name*, bool *value*)

Set the value of an action with the given name.

Parameters

Game Scenes All

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

krpc_SpaceCenter_Antenna_t

An antenna. Obtained by calling `krpc_SpaceCenter_Part_Antenna()`.

krpc_error_t **krpc_SpaceCenter_Antenna_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part object for this antenna.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_AntennaState_t * result*)

The current state of the antenna.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_Deployable** (krpc_connection_t *connection*, bool * *result*)

Whether the antenna is deployable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_Deployed** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Antenna_set_Deployed** (bool *value*)

Whether the antenna is deployed.

Game Scenes All

Note: Fixed antennas are always deployed. Returns an error if you try to deploy a fixed antenna.

krpc_error_t **krpc_SpaceCenter_Antenna_CanTransmit** (krpc_connection_t *connection*, bool * *result*)

Whether data can be transmitted by this antenna.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_Transmit** (krpc_connection_t *connection*)

Transmit data.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_Cancel** (krpc_connection_t *connection*)

Cancel current transmission of data.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_AllowPartial** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Antenna_set_AllowPartial** (bool *value*)

Whether partial data transmission is permitted.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_Power** (krpc_connection_t *connection*, double * *result*)

The power of the antenna.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_Combinable** (krpc_connection_t *connection*, bool * *result*)

Whether the antenna can be combined with other antennae on the vessel to boost the power.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_CombinableExponent** (krpc_connection_t *connection*, double * *result*)

Exponent used to calculate the combined power of multiple antennae on a vessel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_PacketInterval** (krpc_connection_t *connection*, float * *result*)

Interval between sending packets in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_PacketSize** (krpc_connection_t *connection*, float * *result*)

Amount of data sent per packet in Mits.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Antenna_PacketResourceCost** (krpc_connection_t *connection*, double * *result*)

Units of electric charge consumed per packet sent.

Game Scenes All

krpc_SpaceCenter_AntennaState_t

The state of an antenna. See `krpc_SpaceCenter_Antenna_State()`.

KRPC_SPACECENTER_ANTENNASTATE_DEPLOYED

Antenna is fully deployed.

KRPC_SPACECENTER_ANTENNASTATE_RETRACTED

Antenna is fully retracted.

KRPC_SPACECENTER_ANTENNASTATE_DEPLOYING

Antenna is being deployed.

KRPC_SPACECENTER_ANTENNASTATE_RETRACTING

Antenna is being retracted.

KRPC_SPACECENTER_ANTENNASTATE_BROKEN

Antenna is broken.

Cargo Bay

krpc_SpaceCenter_CargoBay_t

A cargo bay. Obtained by calling `krpc_SpaceCenter_Part_CargoBay()`.

krpc_error_t **krpc_SpaceCenter_CargoBay_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The part object for this cargo bay.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CargoBay_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_CargoBayState_t * result*)

The state of the cargo bay.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_CargoBay_Open** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_CargoBay_set_Open** (bool *value*)

Whether the cargo bay is open.

Game Scenes All

krpc_SpaceCenter_CargoBayState_t

The state of a cargo bay. See `krpc_SpaceCenter_CargoBay_State()`.

KRPC_SPACECENTER_CARGOBAYSTATE_OPEN

Cargo bay is fully open.

KRPC_SPACECENTER_CARGOBAYSTATE_CLOSED

Cargo bay closed and locked.

KRPC_SPACECENTER_CARGOBAYSTATE_OPENING

Cargo bay is opening.

KRPC_SPACECENTER_CARGOBAYSTATE_CLOSING

Cargo bay is closing.

Control Surface

krpc_SpaceCenter_ControlSurface_t

An aerodynamic control surface. Obtained by calling `krpc_SpaceCenter_Part_ControlSurface()`.

krpc_error_t **krpc_SpaceCenter_ControlSurface_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part object for this control surface.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ControlSurface_PitchEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_ControlSurface_set_PitchEnabled** (bool *value*)

Whether the control surface has pitch control enabled.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ControlSurface_YawEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_ControlSurface_set_YawEnabled** (bool *value*)

Whether the control surface has yaw control enabled.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ControlSurface_RollEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_ControlSurface_set_RollEnabled** (bool *value*)

Whether the control surface has roll control enabled.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ControlSurface_AuthorityLimiter** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_ControlSurface_set_AuthorityLimiter** (float *value*)

The authority limiter for the control surface, which controls how far the control surface will move.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ControlSurface_Inverted** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_ControlSurface_set_Inverted** (bool *value*)

Whether the control surface movement is inverted.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ControlSurface_Deployed** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_ControlSurface_set_Deployed** (bool *value*)

Whether the control surface has been fully deployed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ControlSurface_SurfaceArea** (krpc_connection_t *connection*, float * *result*)

Surface area of the control surface in m^2.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ControlSurface_AvailableTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tu * *result*)

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 `krpc_SpaceCenter_Vessel_ReferenceFrame()`.

Game Scenes All

Decoupler

krpc_SpaceCenter_Decoupler_t

A decoupler. Obtained by calling `krpc_SpaceCenter_Part_Decoupler()`

krpc_error_t **krpc_SpaceCenter_Decoupler_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The part object for this decoupler.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Decoupler_Decouple** (krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t* * *result*)

Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

Game Scenes All

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 `krpc_SpaceCenter_ActiveVessel()` no longer refer to the active vessel.

krpc_error_t **krpc_SpaceCenter_Decoupler_Decoupled**(krpc_connection_t *connection*, bool * *result*)

Whether the decoupler has fired.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Decoupler_Staged**(krpc_connection_t *connection*, bool * *result*)

Whether the decoupler is enabled in the staging sequence.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Decoupler_Impulse**(krpc_connection_t *connection*, float * *result*)

The impulse that the decoupler imparts when it is fired, in Newton seconds.

Game Scenes All

Docking Port

krpc_SpaceCenter_DockingPort_t

A docking port. Obtained by calling `krpc_SpaceCenter_Part_DockingPort()`

krpc_error_t **krpc_SpaceCenter_DockingPort_Part**(krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part object for this docking port.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_DockingPort_State**(krpc_connection_t *connection*, *krpc_SpaceCenter_DockingPortState_t * result*)

The current state of the docking port.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_DockingPort_DockedPart**(krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part that this docking port is docked to. Returns `nullptr` if this docking port is not docked to anything.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_DockingPort_Undock**(krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t * result*)

Undocks the docking port and returns the new `krpc_SpaceCenter_Vessel_t` 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.

Game Scenes All

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 `krpc_SpaceCenter_ActiveVessel()` no longer refer to the active vessel.

---

krpc_error_t **krpc_SpaceCenter_DockingPort_ReengageDistance** (krpc_connection_t *connection*, float * *result*)

The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_DockingPort_HasShield** (krpc_connection_t *connection*, bool * *result*)

Whether the docking port has a shield.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_DockingPort_Shielded** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_DockingPort_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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_DockingPort_Position** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_DockingPort_Direction** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_DockingPort_Rotation** (krpc_connection_t *connection*, krpc_tuple_double_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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).

Game Scenes All

krpc_error_t **`krpc_SpaceCenter_DockingPort_ReferenceFrame`** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t * result*)

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.

Game Scenes All

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by *`krpc_SpaceCenter_Part_ReferenceFrame()`*.

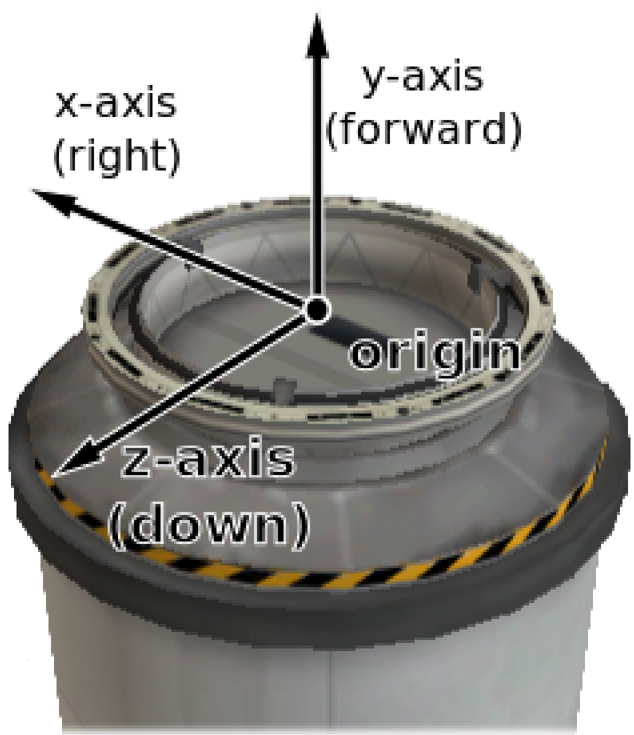


Fig. 8: Docking port reference frame origin and axes

`krpc_SpaceCenter_DockingPortState_t`

The state of a docking port. See *`krpc_SpaceCenter_DockingPort_State()`*.

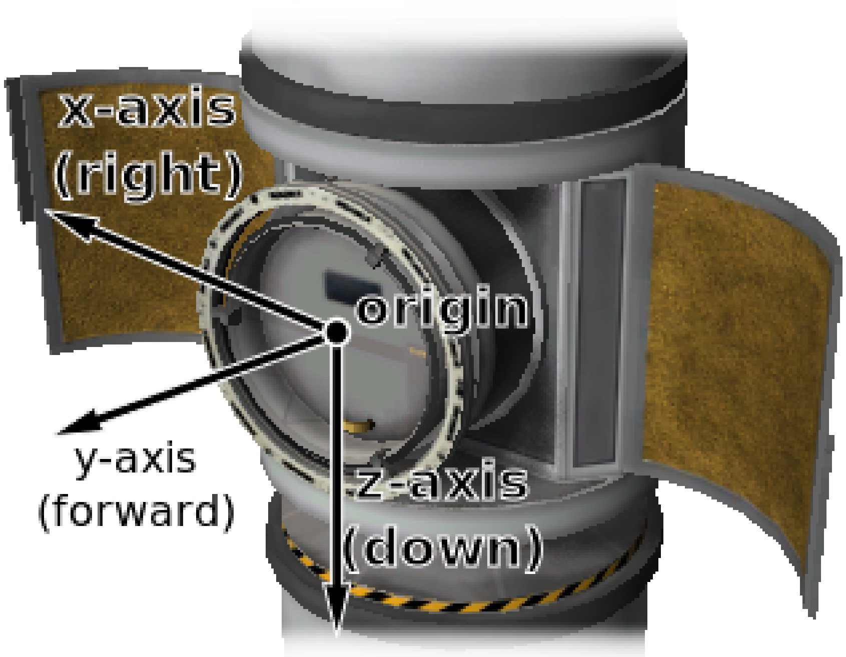


Fig. 9: Inline docking port reference frame origin and axes

KRPC_SPACECENTER_DOCKINGPORTSTATE_READY
The docking port is ready to dock to another docking port.

KRPC_SPACECENTER_DOCKINGPORTSTATE_DOCKED
The docking port is docked to another docking port, or docked to another part (from the VAB/SPH).

KRPC_SPACECENTER_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.

KRPC_SPACECENTER_DOCKINGPORTSTATE_UNDOCKING
The docking port has just been undocked from another docking port, and is disabled until it moves away by a sufficient distance (*krpc_SpaceCenter_DockingPort_ReengageDistance()*).

KRPC_SPACECENTER_DOCKINGPORTSTATE_SHIELDED
The docking port has a shield, and the shield is closed.

KRPC_SPACECENTER_DOCKINGPORTSTATE_MOVING
The docking ports shield is currently opening/closing.

Engine

krpc_SpaceCenter_Engine_t
An engine, including ones of various types. For example liquid fuelled gimballed engines, solid rocket boosters and jet engines. Obtained by calling *krpc_SpaceCenter_Part_Engine()*.

Note: For RCS thrusters *krpc_SpaceCenter_Part_RCS()*.

krpc_error_t **krpc_SpaceCenter_Engine_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part object for this engine.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_Active** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Engine_set_Active** (bool *value*)

Whether the engine is active. Setting this attribute may have no effect, depending on `krpc_SpaceCenter_Engine_CanShutdown()` and `krpc_SpaceCenter_Engine_CanRestart()`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_Thrust** (krpc_connection_t *connection*, float * *result*)

The current amount of thrust being produced by the engine, in Newtons.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_AvailableThrust** (krpc_connection_t *connection*, float * *result*)

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 `krpc_SpaceCenter_Engine_ThrustLimit()` and atmospheric conditions into account.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_MaxThrust** (krpc_connection_t *connection*, float * *result*)

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%.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_MaxVacuumThrust** (krpc_connection_t *connection*, float * *result*)

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, `krpc_SpaceCenter_Engine_ThrustLimit()` is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_ThrustLimit** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Engine_set_ThrustLimit** (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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_Thrusters** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The components of the engine that generate thrust.

Game Scenes All

Note: For example, this corresponds to the rocket nozzel on a solid rocket booster, or the individual nozzels on a RAPIER engine. The overall thrust produced by the engine, as reported by `krpc_SpaceCenter_Engine_AvailableThrust()`, `krpc_SpaceCenter_Engine_MaxThrust()` and others, is the sum of the thrust generated by each thruster.

krpc_error_t **krpc_SpaceCenter_Engine_SpecificImpulse** (krpc_connection_t *connection*, float * *result*)

The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_VacuumSpecificImpulse** (krpc_connection_t *connection*, float * *result*)

The vacuum specific impulse of the engine, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_KerbinSeaLevelSpecificImpulse** (krpc_connection_t *connection*, float * *result*)

The specific impulse of the engine at sea level on Kerbin, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_PropellantNames** (krpc_connection_t *connection*, krpc_list_string_t * *result*)

The names of the propellants that the engine consumes.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_PropellantRatios** (krpc_connection_t *connection*, krpc_dictionary_string_float_t * *result*)

The ratio of resources that the engine consumes. A dictionary mapping resource names to the ratio at which they are consumed by the engine.

Game Scenes All

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.

krpc_error_t **krpc_SpaceCenter_Engine_Propellants** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The propellants that the engine consumes.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_HasFuel** (krpc_connection_t *connection*, bool * *result*)

Whether the engine has any fuel available.

Game Scenes All

Note: The engine must be activated for this property to update correctly.

krpc_error_t **krpc_SpaceCenter_Engine_Throttle** (krpc_connection_t *connection*, float * *result*)

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).

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_ThrottleLocked** (krpc_connection_t *connection*, bool * *result*)

Whether the `krpc_SpaceCenter_Control_Throttle()` affects the engine. For example, this is `true` for liquid fueled rockets, and `false` for solid rocket boosters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_CanRestart** (krpc_connection_t *connection*, bool * *result*)

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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_CanShutdown** (krpc_connection_t *connection*, bool * *result*)

Whether the engine can be shutdown once activated. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_HasModes** (krpc_connection_t *connection*, bool * *result*)

Whether the engine has multiple modes of operation.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_Mode** (krpc_connection_t *connection*, char * * *result*)

void **krpc_SpaceCenter_Engine_set_Mode** (const char * *value*)

The name of the current engine mode.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_Modes** (krpc_connection_t *connection*, krpc_dictionary_string_object_t * *result*)

The available modes for the engine. A dictionary mapping mode names to *krpc_SpaceCenter_Engine_t* objects.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_ToggleMode** (krpc_connection_t *connection*)

Toggle the current engine mode.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_AutoModeSwitch** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Engine_set_AutoModeSwitch** (bool *value*)

Whether the engine will automatically switch modes.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_Gimballed** (krpc_connection_t *connection*, bool * *result*)

Whether the engine is gimballed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_GimbalRange** (krpc_connection_t *connection*, float * *result*)

The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_GimbalLocked** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Engine_set_GimbalLocked** (bool *value*)
Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_GimbalLimit** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Engine_set_GimbalLimit** (float *value*)
The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Engine_AvailableTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tuple_double_ * *result*)
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 *krpc_SpaceCenter_Vessel_ReferenceFrame()*. Returns zero if the engine is inactive, or not gimballed.

Game Scenes All

krpc_SpaceCenter_Propellant_t
A propellant for an engine. Obtains by calling *krpc_SpaceCenter_Engine_Propellants()*.

krpc_error_t **krpc_SpaceCenter_Propellant_Name** (krpc_connection_t *connection*, char * * *result*)
The name of the propellant.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Propellant_CurrentAmount** (krpc_connection_t *connection*, double * *result*)
The current amount of propellant.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Propellant_CurrentRequirement** (krpc_connection_t *connection*, double * *result*)
The required amount of propellant.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Propellant_TotalResourceAvailable** (krpc_connection_t *connection*, double * *result*)
The total amount of the underlying resource currently reachable given resource flow rules.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Propellant_TotalResourceCapacity** (krpc_connection_t *connection*, double * *result*)
The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Propellant_IgnoreForIsp** (krpc_connection_t *connection*, bool * *result*)
If this propellant should be ignored when calculating required mass flow given specific impulse.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Propellant_IgnoreForThrustCurve** (krpc_connection_t *connection*, bool * *result*)

If this propellant should be ignored for thrust curve calculations.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Propellant_DrawStackGauge** (krpc_connection_t *connection*, bool * *result*)

If this propellant has a stack gauge or not.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Propellant_IsDeprived** (krpc_connection_t *connection*, bool * *result*)

If this propellant is deprived.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Propellant_Ratio** (krpc_connection_t *connection*, float * *result*)

The propellant ratio.

Game Scenes All

Experiment

krpc_SpaceCenter_Experiment_t

Obtained by calling *krpc_SpaceCenter_Part_Experiment()*.

krpc_error_t **krpc_SpaceCenter_Experiment_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The part object for this experiment.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Run** (krpc_connection_t *connection*)

Run the experiment.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Transmit** (krpc_connection_t *connection*)

Transmit all experimental data contained by this part.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Dump** (krpc_connection_t *connection*)

Dump the experimental data contained by the experiment.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Reset** (krpc_connection_t *connection*)

Reset the experiment.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Deployed** (krpc_connection_t *connection*, bool * *result*)

Whether the experiment has been deployed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Rerunnable** (krpc_connection_t *connection*, bool * *result*)

Whether the experiment can be re-run.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Inoperable** (krpc_connection_t *connection*, bool * *result*)

Whether the experiment is inoperable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_HasData** (krpc_connection_t *connection*, bool * *result*)

Whether the experiment contains data.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Data** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The data contained in this experiment.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Biome** (krpc_connection_t *connection*, char * * *result*)

The name of the biome the experiment is currently in.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_Available** (krpc_connection_t *connection*, bool * *result*)

Determines if the experiment is available given the current conditions.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Experiment_ScienceSubject** (krpc_connection_t *connection*, *krpc_SpaceCenter_ScienceSubject_t* * *result*)

Containing information on the corresponding specific science result for the current conditions. Returns `nullptr` if the experiment is unavailable.

Game Scenes All

krpc_SpaceCenter_ScienceData_t

Obtained by calling *krpc_SpaceCenter_Experiment_Data()*.

krpc_error_t **krpc_SpaceCenter_ScienceData_DataAmount** (krpc_connection_t *connection*, float * *result*)

Data amount.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ScienceData_ScienceValue** (krpc_connection_t *connection*, float * *result*)

Science value.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ScienceData_TransmitValue** (krpc_connection_t *connection*, float * *result*)

Transmit value.

Game Scenes All

krpc_SpaceCenter_ScienceSubject_t

Obtained by calling *krpc_SpaceCenter_Experiment_ScienceSubject()*.

krpc_error_t **krpc_SpaceCenter_ScienceSubject_Title** (krpc_connection_t *connection*, char * * *result*)
Title of science subject, displayed in science archives

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ScienceSubject_IsComplete** (krpc_connection_t *connection*, bool * *result*)
Whether the experiment has been completed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ScienceSubject_Science** (krpc_connection_t *connection*, float * *result*)
Amount of science already earned from this subject, not updated until after transmission/recovery.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ScienceSubject_ScienceCap** (krpc_connection_t *connection*, float * *result*)
Total science allowable for this subject.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ScienceSubject_DataScale** (krpc_connection_t *connection*, float * *result*)
Multiply science value by this to determine data amount in mits.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ScienceSubject_SubjectValue** (krpc_connection_t *connection*, float * *result*)
Multiplier for specific Celestial Body/Experiment Situation combination.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ScienceSubject_ScientificValue** (krpc_connection_t *connection*, float * *result*)
Diminishing value multiplier for decreasing the science value returned from repeated experiments.

Game Scenes All

Fairing

krpc_SpaceCenter_Fairing_t
A fairing. Obtained by calling `krpc_SpaceCenter_Part_Fairing()`.

krpc_error_t **krpc_SpaceCenter_Fairing_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)
The part object for this fairing.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Fairing_Jettison** (krpc_connection_t *connection*)
Jettison the fairing. Has no effect if it has already been jettisoned.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Fairing_Jettisoned** (krpc_connection_t *connection*, bool * *result*)
Whether the fairing has been jettisoned.

Game Scenes All

Intake

krpc_SpaceCenter_Intake_t

An air intake. Obtained by calling `krpc_SpaceCenter_Part_Intake()`.

krpc_error_t **krpc_SpaceCenter_Intake_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part object for this intake.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Intake_Open** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Intake_set_Open** (bool *value*)

Whether the intake is open.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Intake_Speed** (krpc_connection_t *connection*, float * *result*)

Speed of the flow into the intake, in m/s.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Intake_Flow** (krpc_connection_t *connection*, float * *result*)

The rate of flow into the intake, in units of resource per second.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Intake_Area** (krpc_connection_t *connection*, float * *result*)

The area of the intake's opening, in square meters.

Game Scenes All

Leg

krpc_SpaceCenter_Leg_t

A landing leg. Obtained by calling `krpc_SpaceCenter_Part_Leg()`.

krpc_error_t **krpc_SpaceCenter_Leg_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part object for this landing leg.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Leg_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_LegState_t * result*)

The current state of the landing leg.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Leg_Deployable** (krpc_connection_t *connection*, bool * *result*)

Whether the leg is deployable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Leg_Deployed** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Leg_set_Deployed** (bool *value*)

Whether the landing leg is deployed.

Game Scenes All

Note: Fixed landing legs are always deployed. Returns an error if you try to deploy fixed landing gear.

krpc_error_t **krpc_SpaceCenter_Leg_IsGrounded** (krpc_connection_t *connection*, bool * *result*)
Returns whether the leg is touching the ground.

Game Scenes All

krpc_SpaceCenter_LegState_t
The state of a landing leg. See *krpc_SpaceCenter_Leg_State()*.

KRPC_SPACECENTER_LEGSTATE_DEPLOYED
Landing leg is fully deployed.

KRPC_SPACECENTER_LEGSTATE_RETRACTED
Landing leg is fully retracted.

KRPC_SPACECENTER_LEGSTATE_DEPLOYING
Landing leg is being deployed.

KRPC_SPACECENTER_LEGSTATE_RETRACTING
Landing leg is being retracted.

KRPC_SPACECENTER_LEGSTATE_BROKEN
Landing leg is broken.

Launch Clamp

krpc_SpaceCenter_LaunchClamp_t
A launch clamp. Obtained by calling *krpc_SpaceCenter_Part_LaunchClamp()*.

krpc_error_t **krpc_SpaceCenter_LaunchClamp_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)
The part object for this launch clamp.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_LaunchClamp_Release** (krpc_connection_t *connection*)
Releases the docking clamp. Has no effect if the clamp has already been released.

Game Scenes All

Light

krpc_SpaceCenter_Light_t
A light. Obtained by calling *krpc_SpaceCenter_Part_Light()*.

krpc_error_t **krpc_SpaceCenter_Light_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)
The part object for this light.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Light_Active** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Light_set_Active** (bool *value*)
Whether the light is switched on.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Light_Color** (krpc_connection_t *connection*, krpc_tuple_float_float_float_t * *result*)

void **krpc_SpaceCenter_Light_set_Color** (const krpc_tuple_float_float_float_t * *value*)

The color of the light, as an RGB triple.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Light_PowerUsage** (krpc_connection_t *connection*, float * *result*)

The current power usage, in units of charge per second.

Game Scenes All

Parachute

krpc_SpaceCenter_Parachute_t

A parachute. Obtained by calling `krpc_SpaceCenter_Part_Parachute()`.

krpc_error_t **krpc_SpaceCenter_Parachute_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The part object for this parachute.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parachute_Deploy** (krpc_connection_t *connection*)

Deploys the parachute. This has no effect if the parachute has already been deployed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parachute_Deployed** (krpc_connection_t *connection*, bool * *result*)

Whether the parachute has been deployed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parachute_Arm** (krpc_connection_t *connection*)

Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parachute_Armed** (krpc_connection_t *connection*, bool * *result*)

Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parachute_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_ParachuteState_t* * *result*)

The current state of the parachute.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parachute_DeployAltitude** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Parachute_set_DeployAltitude** (float *value*)

The altitude at which the parachute will full deploy, in meters. Only applicable to stock parachutes.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Parachute_DeployMinPressure** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Parachute_set_DeployMinPressure** (float *value*)
The minimum pressure at which the parachute will semi-deploy, in atmospheres. Only applicable to stock parachutes.

Game Scenes All

krpc_SpaceCenter_ParachuteState_t
The state of a parachute. See *krpc_SpaceCenter_Parachute_State()*.

KRPC_SPACECENTER_PARACHUTESTATE_STOWED
The parachute is safely tucked away inside its housing.

KRPC_SPACECENTER_PARACHUTESTATE_ARMED
The parachute is armed for deployment. (RealChutes only)

KRPC_SPACECENTER_PARACHUTESTATE_ACTIVE
The parachute is still stowed, but ready to semi-deploy. (Stock parachutes only)

KRPC_SPACECENTER_PARACHUTESTATE_SEMIDEPLOYED
The parachute has been deployed and is providing some drag, but is not fully deployed yet. (Stock parachutes only)

KRPC_SPACECENTER_PARACHUTESTATE_DEPLOYED
The parachute is fully deployed.

KRPC_SPACECENTER_PARACHUTESTATE_CUT
The parachute has been cut.

Radiator

krpc_SpaceCenter_Radiator_t
A radiator. Obtained by calling *krpc_SpaceCenter_Part_Radiator()*.

krpc_error_t **krpc_SpaceCenter_Radiator_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)
The part object for this radiator.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Radiator_Deployable** (krpc_connection_t *connection*, bool * *result*)
Whether the radiator is deployable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Radiator_Deployed** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Radiator_set_Deployed** (bool *value*)
For a deployable radiator, `true` if the radiator is extended. If the radiator is not deployable, this is always `true`.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Radiator_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_RadiatorState_t* * *result*)
The current state of the radiator.

Game Scenes All

Note: A fixed radiator is always *KRPC_SPACECENTER_RADIATORSTATE_EXTENDED*.

krpc_SpaceCenter_RadiatorState_t
The state of a radiator. *krpc_SpaceCenter_RadiatorState_t*

KRPC_SPACECENTER_RADIATORSTATE_EXTENDED
Radiator is fully extended.

KRPC_SPACECENTER_RADIATORSTATE_RETRACTED
Radiator is fully retracted.

KRPC_SPACECENTER_RADIATORSTATE_EXTENDING
Radiator is being extended.

KRPC_SPACECENTER_RADIATORSTATE_RETRACTING
Radiator is being retracted.

KRPC_SPACECENTER_RADIATORSTATE_BROKEN
Radiator is being broken.

Resource Converter

krpc_SpaceCenter_ResourceConverter_t
A resource converter. Obtained by calling *krpc_SpaceCenter_Part_ResourceConverter()*.

krpc_error_t **krpc_SpaceCenter_ResourceConverter_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)
The part object for this converter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_Count** (krpc_connection_t *connection*, int32_t * *result*)
The number of converters in the part.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_Name** (krpc_connection_t *connection*, char * * *result*, int32_t *index*)
The name of the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_Active** (krpc_connection_t *connection*, bool * *result*, int32_t *index*)
True if the specified converter is active.

Parameters

- **index** – Index of the converter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_Start** (krpc_connection_t *connection*, int32_t *index*)
Start the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_Stop**(krpc_connection_t *connection*, int32_t *index*)

Stop the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_State**(krpc_connection_t *connection*, *krpc_SpaceCenter_ResourceConverterState_t* * *result*, int32_t *index*)

The state of the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_StatusInfo**(krpc_connection_t *connection*, char * * *result*, 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_Inputs**(krpc_connection_t *connection*, krpc_list_string_t * *result*, int32_t *index*)

List of the names of resources consumed by the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_Outputs**(krpc_connection_t *connection*, krpc_list_string_t * *result*, int32_t *index*)

List of the names of resources produced by the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_OptimumCoreTemperature**(krpc_connection_t *connection*, float * *result*)

The core temperature at which the converter will operate with peak efficiency, in Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_CoreTemperature** (krpc_connection_t *connection*, float * *result*)

The core temperature of the converter, in Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceConverter_ThermalEfficiency** (krpc_connection_t *connection*, float * *result*)

The thermal efficiency of the converter, as a percentage of its maximum.

Game Scenes All

krpc_SpaceCenter_ResourceConverterState_t

The state of a resource converter. See `krpc_SpaceCenter_ResourceConverter_State()`.

KRPC_SPACECENTER_RESOURCECONVERTERSTATE_RUNNING

Converter is running.

KRPC_SPACECENTER_RESOURCECONVERTERSTATE_IDLE

Converter is idle.

KRPC_SPACECENTER_RESOURCECONVERTERSTATE_MISSINGRESOURCE

Converter is missing a required resource.

KRPC_SPACECENTER_RESOURCECONVERTERSTATE_STORAGEFULL

No available storage for output resource.

KRPC_SPACECENTER_RESOURCECONVERTERSTATE_CAPACITY

At preset resource capacity.

KRPC_SPACECENTER_RESOURCECONVERTERSTATE_UNKNOWN

Unknown state. Possible with modified resource converters. In this case, check `krpc_SpaceCenter_ResourceConverter_StatusInfo()` for more information.

Resource Harvester

krpc_SpaceCenter_ResourceHarvester_t

A resource harvester (drill). Obtained by calling `krpc_SpaceCenter_Part_ResourceHarvester()`.

krpc_error_t **krpc_SpaceCenter_ResourceHarvester_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The part object for this harvester.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceHarvester_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_ResourceHarvesterState_t* * *result*)

The state of the harvester.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceHarvester_Deployed** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_ResourceHarvester_set_Deployed**(bool *value*)
Whether the harvester is deployed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceHarvester_Active**(krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_ResourceHarvester_set_Active**(bool *value*)
Whether the harvester is actively drilling.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceHarvester_ExtractionRate**(krpc_connection_t *connection*, float * *result*)
The rate at which the drill is extracting ore, in units per second.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceHarvester_ThermalEfficiency**(krpc_connection_t *connection*, float * *result*)
The thermal efficiency of the drill, as a percentage of its maximum.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceHarvester_CoreTemperature**(krpc_connection_t *connection*, float * *result*)
The core temperature of the drill, in Kelvin.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceHarvester_OptimumCoreTemperature**(krpc_connection_t *connection*, float * *result*)
The core temperature at which the drill will operate with peak efficiency, in Kelvin.

Game Scenes All

krpc_SpaceCenter_ResourceHarvesterState_t
The state of a resource harvester. See `krpc_SpaceCenter_ResourceHarvester_State()`.

KRPC_SPACECENTER_RESOURCEHARVESTERSTATE_DEPLOYING
The drill is deploying.

KRPC_SPACECENTER_RESOURCEHARVESTERSTATE_DEPLOYED
The drill is deployed and ready.

KRPC_SPACECENTER_RESOURCEHARVESTERSTATE_RETRACTING
The drill is retracting.

KRPC_SPACECENTER_RESOURCEHARVESTERSTATE_RETRACTED
The drill is retracted.

KRPC_SPACECENTER_RESOURCEHARVESTERSTATE_ACTIVE
The drill is running.

Reaction Wheel

krpc_SpaceCenter_ReactionWheel_t

A reaction wheel. Obtained by calling `krpc_SpaceCenter_Part_ReactionWheel()`.

krpc_error_t **krpc_SpaceCenter_ReactionWheel_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The part object for this reaction wheel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ReactionWheel_Active** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_ReactionWheel_set_Active** (bool *value*)

Whether the reaction wheel is active.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ReactionWheel_Broken** (krpc_connection_t *connection*, bool * *result*)

Whether the reaction wheel is broken.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ReactionWheel_AvailableTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tup * *result*)

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 `krpc_SpaceCenter_Vessel_ReferenceFrame()`. Returns zero if the reaction wheel is inactive or broken.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ReactionWheel_MaxTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tuple_doubl * *result*)

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 `krpc_SpaceCenter_Vessel_ReferenceFrame()`.

Game Scenes All

RCS

krpc_SpaceCenter_RCS_t

An RCS block or thruster. Obtained by calling `krpc_SpaceCenter_Part_RCS()`.

krpc_error_t **krpc_SpaceCenter_RCS_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The part object for this RCS.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_Active** (krpc_connection_t *connection*, bool * *result*)

Whether the RCS thrusters are active. An RCS thruster is inactive if the RCS action group is disabled (`krpc_SpaceCenter_Control_RCS()`), the RCS thruster itself is not enabled (`krpc_SpaceCenter_RCS_Enabled()`) or it is covered by a fairing (`krpc_SpaceCenter_Part_Shielded()`).

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_Enabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_RCS_set_Enabled** (bool *value*)
Whether the RCS thrusters are enabled.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_PitchEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_RCS_set_PitchEnabled** (bool *value*)
Whether the RCS thruster will fire when pitch control input is given.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_YawEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_RCS_set_YawEnabled** (bool *value*)
Whether the RCS thruster will fire when yaw control input is given.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_RollEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_RCS_set_RollEnabled** (bool *value*)
Whether the RCS thruster will fire when roll control input is given.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_ForwardEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_RCS_set_ForwardEnabled** (bool *value*)
Whether the RCS thruster will fire when pitch control input is given.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_UpEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_RCS_set_UpEnabled** (bool *value*)
Whether the RCS thruster will fire when yaw control input is given.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_RightEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_RCS_set_RightEnabled** (bool *value*)
Whether the RCS thruster will fire when roll control input is given.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_AvailableTorque** (krpc_connection_t *connection*, krpc_tuple_tuple_double_double_double_tuple_double_dou * *result*)
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 `krpc_SpaceCenter_Vessel_ReferenceFrame()`. Returns zero if RCS is disable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_MaxThrust** (krpc_connection_t *connection*, float * *result*)
The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_MaxVacuumThrust** (krpc_connection_t *connection*, float * *result*)

The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_Thrusters** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of thrusters, one of each nozzel in the RCS part.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_SpecificImpulse** (krpc_connection_t *connection*, float * *result*)

The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_VacuumSpecificImpulse** (krpc_connection_t *connection*, float * *result*)

The vacuum specific impulse of the RCS, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_KerbinSeaLevelSpecificImpulse** (krpc_connection_t *connection*, float * *result*)

The specific impulse of the RCS at sea level on Kerbin, in seconds.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_Propellants** (krpc_connection_t *connection*, krpc_list_string_t * *result*)

The names of resources that the RCS consumes.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_PropellantRatios** (krpc_connection_t *connection*, krpc_dictionary_string_float_t * *result*)

The ratios of resources that the RCS consumes. A dictionary mapping resource names to the ratios at which they are consumed by the RCS.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_RCS_HasFuel** (krpc_connection_t *connection*, bool * *result*)

Whether the RCS has fuel available.

Game Scenes All

Note: The RCS thruster must be activated for this property to update correctly.

Sensor

krpc_SpaceCenter_Sensor_t

A sensor, such as a thermometer. Obtained by calling `krpc_SpaceCenter_Part_Sensor()`.

krpc_error_t **krpc_SpaceCenter_Sensor_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)
The part object for this sensor.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Sensor_Active** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Sensor_set_Active** (bool *value*)
Whether the sensor is active.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Sensor_Value** (krpc_connection_t *connection*, char * * *result*)
The current value of the sensor.

Game Scenes All

Solar Panel

krpc_SpaceCenter_SolarPanel_t
A solar panel. Obtained by calling *krpc_SpaceCenter_Part_SolarPanel()*.

krpc_error_t **krpc_SpaceCenter_SolarPanel_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)
The part object for this solar panel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_SolarPanel_Deployable** (krpc_connection_t *connection*, bool * *result*)
Whether the solar panel is deployable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_SolarPanel_Deployed** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_SolarPanel_set_Deployed** (bool *value*)
Whether the solar panel is extended.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_SolarPanel_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_SolarPanelState_t * result*)
The current state of the solar panel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_SolarPanel_EnergyFlow** (krpc_connection_t *connection*, float * *result*)
The current amount of energy being generated by the solar panel, in units of charge per second.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_SolarPanel_SunExposure** (krpc_connection_t *connection*, float * *result*)
The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

Game Scenes All

krpc_SpaceCenter_SolarPanelState_t
The state of a solar panel. See *krpc_SpaceCenter_SolarPanel_State()*.

KRPC_SPACECENTER_SOLARPANELSTATE_EXTENDED
Solar panel is fully extended.

KRPC_SPACECENTER_SOLARPANELSTATE_RETRACTED
Solar panel is fully retracted.

KRPC_SPACECENTER_SOLARPANELSTATE_EXTENDING
Solar panel is being extended.

KRPC_SPACECENTER_SOLARPANELSTATE_RETRACTING
Solar panel is being retracted.

KRPC_SPACECENTER_SOLARPANELSTATE_BROKEN
Solar panel is broken.

Thruster

krpc_SpaceCenter_Thruster_t
The component of an *krpc_SpaceCenter_Engine_t* or *krpc_SpaceCenter_RCS_t* part that generates thrust. Can obtained by calling *krpc_SpaceCenter_Engine_Thrusters()* or *krpc_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.

---

krpc_error_t **krpc_SpaceCenter_Thruster_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)
The *krpc_SpaceCenter_Part_t* that contains this thruster.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Thruster_ThrustPosition** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Thruster_ThrustDirection** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Thruster_ThrustReferenceFrame** (krpc_connection_t *connection*, krpc_SpaceCenter_ReferenceFrame_t * *result*)

A reference frame that is fixed relative to the thruster and orientated with its thrust direction (*krpc_SpaceCenter_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 (*krpc_SpaceCenter_Thruster_ThrustPosition()*).
- The axes rotate with the thrust direction. This is the direction in which the thruster expels propellant, including any gimballing.
- The y-axis points along the thrust direction.
- The x-axis and z-axis are perpendicular to the thrust direction.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Thruster_Gimballed** (krpc_connection_t *connection*, bool * *result*)

Whether the thruster is gimballed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Thruster_GimbalPosition** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Thruster_GimbalAngle** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

The current gimbal angle in the pitch, roll and yaw axes, in degrees.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Thruster_InitialThrustPosition** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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** – The reference frame that the returned position vector is in.

Returns The position as a vector.

Game Scenes All

Note: This position can move when the gimbal rotates. This is because the thrust position and gimbal position are not necessarily the same.

krpc_error_t **krpc_SpaceCenter_Thruster_InitialThrustDirection** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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** – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

Game Scenes All

Wheel

krpc_SpaceCenter_Wheel_t

A wheel. Includes landing gear and rover wheels. Obtained by calling *krpc_SpaceCenter_Part_Wheel()*. Can be used to control the motors, steering and deployment of wheels, among other things.

krpc_error_t **krpc_SpaceCenter_Wheel_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

The part object for this wheel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_WheelState_t * result*)

The current state of the wheel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Radius** (krpc_connection_t *connection*, float * *result*)

Radius of the wheel, in meters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Grounded** (krpc_connection_t *connection*, bool * *result*)

Whether the wheel is touching the ground.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_HasBrakes** (krpc_connection_t *connection*, bool * *result*)

Whether the wheel has brakes.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Brakes** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Wheel_set_Brakes** (float *value*)
The braking force, as a percentage of maximum, when the brakes are applied.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_AutoFrictionControl** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Wheel_set_AutoFrictionControl** (bool *value*)
Whether automatic friction control is enabled.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_ManualFrictionControl** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Wheel_set_ManualFrictionControl** (float *value*)
Manual friction control value. Only has an effect if automatic friction control is disabled. A value between 0 and 5 inclusive.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Deployable** (krpc_connection_t *connection*, bool * *result*)
Whether the wheel is deployable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Deployed** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Wheel_set_Deployed** (bool *value*)
Whether the wheel is deployed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Powered** (krpc_connection_t *connection*, bool * *result*)
Whether the wheel is powered by a motor.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_MotorEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Wheel_set_MotorEnabled** (bool *value*)
Whether the motor is enabled.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_MotorInverted** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Wheel_set_MotorInverted** (bool *value*)
Whether the direction of the motor is inverted.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_MotorState** (krpc_connection_t *connection*, *krpc_SpaceCenter_MotorState_t* * *result*)
Whether the direction of the motor is inverted.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_MotorOutput** (krpc_connection_t *connection*, float * *result*)

The output of the motor. This is the torque currently being generated, in Newton meters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_TractionControlEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Wheel_set_TractionControlEnabled** (bool *value*)

Whether automatic traction control is enabled. A wheel only has traction control if it is powered.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_TractionControl** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Wheel_set_TractionControl** (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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_DriveLimiter** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Wheel_set_DriveLimiter** (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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Steerable** (krpc_connection_t *connection*, bool * *result*)

Whether the wheel has steering.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_SteeringEnabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Wheel_set_SteeringEnabled** (bool *value*)

Whether the wheel steering is enabled.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_SteeringInverted** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Wheel_set_SteeringInverted** (bool *value*)

Whether the wheel steering is inverted.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_HasSuspension** (krpc_connection_t *connection*, bool * *result*)

Whether the wheel has suspension.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_SuspensionSpringStrength** (krpc_connection_t *connection*, float * *result*)

Suspension spring strength, as set in the editor.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_SuspensionDamperStrength** (krpc_connection_t *connection*, float * *result*)
Suspension damper strength, as set in the editor.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Broken** (krpc_connection_t *connection*, bool * *result*)
Whether the wheel is broken.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Repairable** (krpc_connection_t *connection*, bool * *result*)
Whether the wheel is repairable.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Stress** (krpc_connection_t *connection*, float * *result*)
Current stress on the wheel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_StressTolerance** (krpc_connection_t *connection*, float * *result*)
Stress tolerance of the wheel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_StressPercentage** (krpc_connection_t *connection*, float * *result*)
Current stress on the wheel as a percentage of its stress tolerance.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Deflection** (krpc_connection_t *connection*, float * *result*)
Current deflection of the wheel.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Wheel_Slip** (krpc_connection_t *connection*, float * *result*)
Current slip of the wheel.

Game Scenes All

krpc_SpaceCenter_WheelState_t
The state of a wheel. See `krpc_SpaceCenter_Wheel_State()`.

KRPC_SPACECENTER_WHEELSTATE_DEPLOYED
Wheel is fully deployed.

KRPC_SPACECENTER_WHEELSTATE_RETRACTED
Wheel is fully retracted.

KRPC_SPACECENTER_WHEELSTATE_DEPLOYING
Wheel is being deployed.

KRPC_SPACECENTER_WHEELSTATE_RETRACTING
Wheel is being retracted.

KRPC_SPACECENTER_WHEELSTATE_BROKEN
Wheel is broken.

krpc_SpaceCenter_MotorState_t
The state of the motor on a powered wheel. See `krpc_SpaceCenter_Wheel_MotorState()`.

KRPC_SPACECENTER_MOTORSTATE_IDLE
The motor is idle.

KRPC_SPACECENTER_MOTORSTATE_RUNNING
The motor is running.

KRPC_SPACECENTER_MOTORSTATE_DISABLED
The motor is disabled.

KRPC_SPACECENTER_MOTORSTATE_INOPERABLE
The motor is inoperable.

KRPC_SPACECENTER_MOTORSTATE_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`.

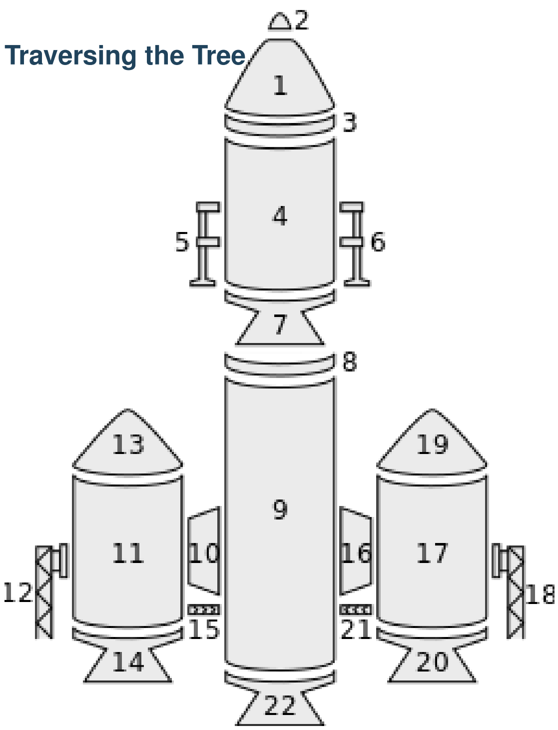


Fig. 10: **Figure 1** – Example parts making up a vessel.

Traversing the Tree

The tree of parts can be traversed using the attributes *krpc_SpaceCenter_Parts_Root()*, *krpc_SpaceCenter_Part_Parent()* and *krpc_SpaceCenter_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 *krpc_SpaceCenter_Parts_Root()*. A parts children can be obtained by calling *krpc_SpaceCenter_Part_Children()*. If the part does not have any children, *krpc_SpaceCenter_Part_Children()* returns an empty list. A parts parent can be obtained by calling *krpc_SpaceCenter_Part_Parent()*. If the part does not have a parent (as is the case for the root part), *krpc_SpaceCenter_Part_Parent()* returns `nullptr`.

The following Cnano example uses these attributes to perform a depth-first traversal over all of the parts in a vessel:

```
#include <krpc_cnano
#include <krpc_cnano

int main() {
  krpc_connection_t
  krpc_open(&conn, "
  krpc_connect(conn,
  krpc_SpaceCenter_V
  krpc_
↪SpaceCenter_Active
```

(continues on next page)

(continued from previous page)

```
krpc_SpaceCenter_P
krpc_SpaceCenter_
↪Vessel_Parts(conn,
krpc_SpaceCenter_P
krpc_
↪SpaceCenter_Parts_

typedef struct {
  krpc_SpaceCenter
  int depth;
} StackEntry;
StackEntry stack[2
int stackPtr = 0;
stack[stackPtr].pa
stack[stackPtr].de

while (stackPtr >=
  krpc_SpaceCenter
↪Part_t part = stac
  int depth = stac
  stackPtr--;  //
  char * title = N
  krpc_
↪SpaceCenter_Part_T
  for (int i = 0;
    printf(" ");
  printf("%s\n", t

  krpc_
↪list_object_t chil
  krpc_SpaceCenter
↪Part_Children(conn
 ␣
↪for (size_t i = 0;
    // Push onto t
    stackPtr++;
␣
↪   stack[stackPtr]
    stack[stackPtr
  }
}
}
```

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
```

(continues on next page)

(continued from previous page)

```
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 Cnano example does a depth-first traversal as before, but also prints out the attachment mode used by the part:

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_
↪connect(conn, "InfernalRobotics Example");

  krpc_SpaceCenter_Vessel_t vessel;
  krpc_
↪SpaceCenter_ActiveVessel(conn, &vessel);

  krpc_SpaceCenter_Parts_t parts;
  krpc_SpaceCenter_
↪Vessel_Parts(conn, &parts, vessel);
  krpc_SpaceCenter_Part_t root;
  krpc_
↪SpaceCenter_Parts_Root(conn, &root, parts);

  typedef struct {
    krpc_SpaceCenter_Part_t part;
    int depth;
  } StackEntry;
```

(continues on next page)

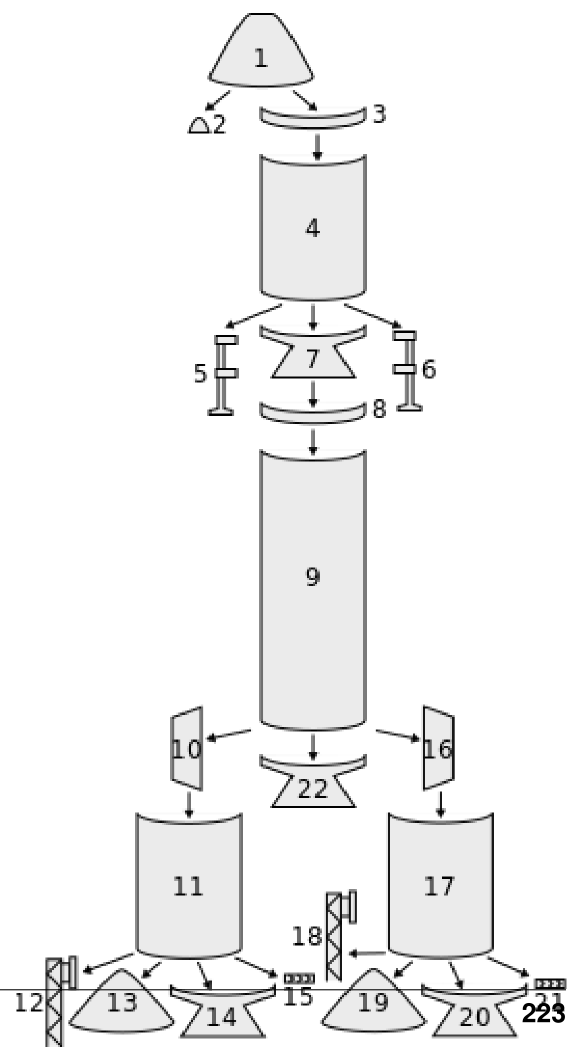


Fig. 11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the

(continued from previous page)

```
  StackEntry stack[256];
  int stackPtr = 0;
  stack[stackPtr].part = root;
  stack[stackPtr].depth = 0;

  while (stackPtr >= 0) {
    krpc_SpaceCenter_Part_t part = stack[stackPtr].part;
    int depth = stack[stackPtr].depth;
    stackPtr--;  // Pop the stack
    bool axially_attached;
    krpc_SpaceCenter_Part_AxiallyAttached(conn, &axially_attached, part);
    const char * attach_mode = axially_attached ? "axial" : "radial";
    char * title = NULL;
    krpc_SpaceCenter_Part_Title(conn, &title, part);
    for (int i = 0; i < depth; i++)
      printf(" ");
    printf("%s - %s\n", title, attach_mode);

    krpc_list_object_t children = KRPC_NULL_LIST;
    krpc_SpaceCenter_Part_Children(conn, &children, part);
    for (size_t i = 0; i < children.size; i++) {
      // Push onto the stack
      stackPtr++;
      stack[stackPtr].part = children.items[i];
      stack[stackPtr].depth = 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
```

(continues on next page)

(continued from previous page)

```
      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

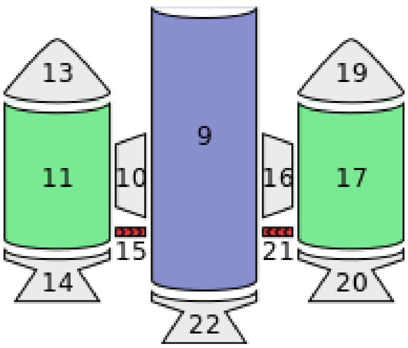


Fig. 12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

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 *krpc_SpaceCenter_Part_FuelLinesFrom()* and *krpc_SpaceCenter_Part_FuelLinesTo()* can be used to discover these connections. In the example in Figure 5, when *krpc_SpaceCenter_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 *krpc_SpaceCenter_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).

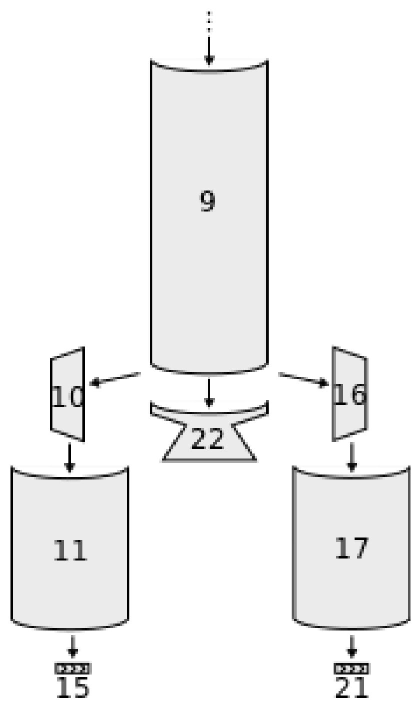


Fig. 13: **Figure 4** – A subset of the parts tree from Figure 2 above.

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 `krpc_SpaceCenter_Part_Stage()` and `krpc_SpaceCenter_Part_DecoupleStage()` respectively. For parts that are not activated by staging, `krpc_SpaceCenter_Part_Stage()` returns -1. For parts that are never decoupled, `krpc_SpaceCenter_Part_DecoupleStage()` 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*.

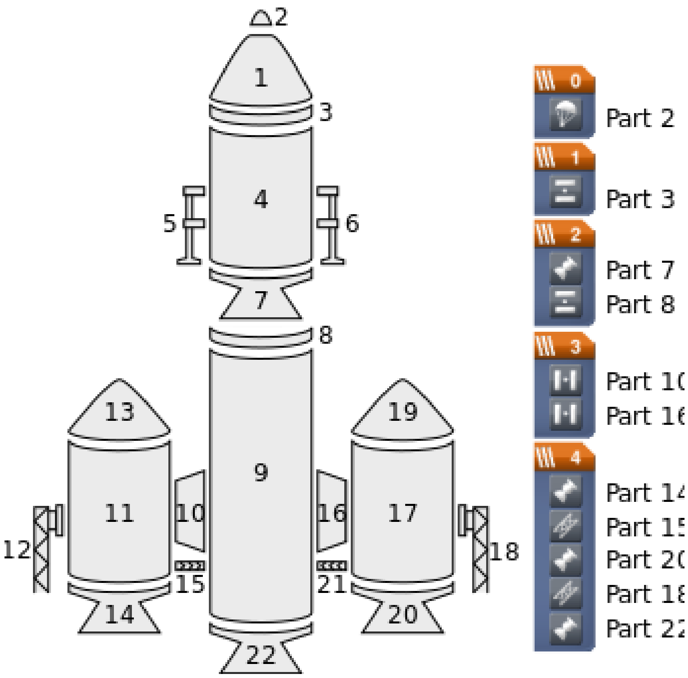


Fig. 14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

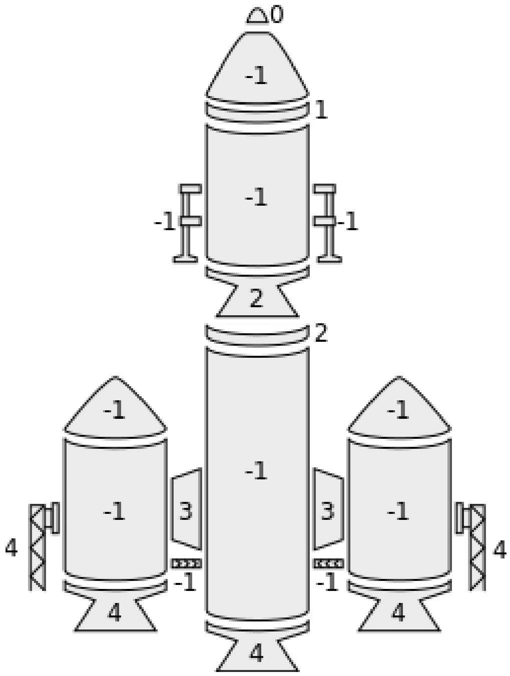


Fig. 15: **Figure 7** – The stage in which each part is *activated*.

3.3.9 Resources

`krpc_SpaceCenter_Resources_t`

Represents the collection of resources stored in a vessel, stage or part. Created by calling *`krpc_SpaceCenter_Vessel_Resources()`*, *`krpc_SpaceCenter_Vessel_ResourcesInDecoupleStage()`* or *`krpc_SpaceCenter_Part_Resources()`*.

krpc_error_t **`krpc_SpaceCenter_Resources_All`** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

All the individual resources that can be stored.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_Resources_WithResource`** (krpc_connection_t *connection*, krpc_list_object_t * *result*, const char * *name*)

All the individual resources with the given name that can be stored.

Parameters

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_Resources_Names`** (krpc_connection_t *connection*, krpc_list_string_t * *result*)

A list of resource names that can be stored.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Resources_HasResource** (krpc_connection_t *connection*, bool * *result*, const char * *name*)

Check whether the named resource can be stored.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Resources_Amount** (krpc_connection_t *connection*, float * *result*, const char * *name*)

Returns the amount of a resource that is currently stored.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Resources_Max** (krpc_connection_t *connection*, float * *result*, const char * *name*)

Returns the amount of a resource that can be stored.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Resources_Density** (krpc_connection_t *connection*, float * *result*, const char * *name*)

Returns the density of a resource, in kg/l.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Resources_FlowMode** (krpc_connection_t *connection*, *krpc_SpaceCenter_ResourceFlowMode_t* * *result*, const char * *name*)

Returns the flow mode of a resource.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Resources_Enabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Resources_set_Enabled** (bool *value*)

Whether use of all the resources are enabled.

Game Scenes Flight

Note: This is `true` if all of the resources are enabled. If any of the resources are not enabled, this is `false`.

krpc_SpaceCenter_Resource_t

An individual resource stored within a part. Created using methods in the *krpc_SpaceCenter_Resources_t* class.

krpc_error_t **krpc_SpaceCenter_Resource_Name** (krpc_connection_t *connection*, char * * *result*)

The name of the resource.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Resource_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The part containing the resource.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Resource_Amount** (krpc_connection_t *connection*, float * *result*)

The amount of the resource that is currently stored in the part.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Resource_Max** (krpc_connection_t *connection*, float * *result*)

The total amount of the resource that can be stored in the part.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Resource_Density** (krpc_connection_t *connection*, float * *result*)

The density of the resource, in kg/l.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Resource_FlowMode** (krpc_connection_t *connection*, *krpc_SpaceCenter_ResourceFlowMode_t* * *result*)

The flow mode of the resource.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Resource_Enabled** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_Resource_set_Enabled** (bool *value*)

Whether use of this resource is enabled.

Game Scenes All

krpc_SpaceCenter_ResourceTransfer_t

Transfer resources between parts.

krpc_error_t **krpc_SpaceCenter_ResourceTransfer_Start** (krpc_connection_t *connection*, *krpc_SpaceCenter_ResourceTransfer_t* * *result*, *krpc_SpaceCenter_Part_t fromPart*, *krpc_SpaceCenter_Part_t toPart*, const char * *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 *krpc_SpaceCenter_ResourceTransfer_Complete()* to check if the transfer is complete. Use *krpc_SpaceCenter_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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceTransfer_Amount** (krpc_connection_t *connection*, float * *result*)

The amount of the resource that has been transferred.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ResourceTransfer_Complete** (krpc_connection_t *connection*, bool * *result*)

Whether the transfer has completed.

Game Scenes All

krpc_SpaceCenter_ResourceFlowMode_t

The way in which a resource flows between parts. See *krpc_SpaceCenter_Resources_FlowMode()*.

KRPC_SPACECENTER_RESOURCEFLOWMODE_VESSEL

The resource flows to any part in the vessel. For example, electric charge.

KRPC_SPACECENTER_RESOURCEFLOWMODE_STAGE

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

KRPC_SPACECENTER_RESOURCEFLOWMODE_ADJACENT

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

KRPC_SPACECENTER_RESOURCEFLOWMODE_NONE

The resource does not flow. For example, solid fuel.

3.3.10 Node

krpc_SpaceCenter_Node_t

Represents a maneuver node. Can be created using *krpc_SpaceCenter_Control_AddNode()*.

krpc_error_t **krpc_SpaceCenter_Node_Prograde** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_Node_set_Prograde** (double *value*)

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Node_Normal** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_Node_set_Normal** (double *value*)

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Node_Radial** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_Node_set_Radial** (double *value*)
The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Node_DeltaV** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_Node_set_DeltaV** (double *value*)
The delta-v of the maneuver node, in meters per second.

Game Scenes Flight

Note: Does not change when executing the maneuver node. See *krpc_SpaceCenter_Node_RemainingDeltaV()*.

krpc_error_t **krpc_SpaceCenter_Node_RemainingDeltaV** (krpc_connection_t *connection*, double * *result*)
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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Node_BurnVector** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*)
Returns the burn vector for the maneuver node.

Parameters

- **referenceFrame** – The reference frame that the returned vector is in. Defaults to *krpc_SpaceCenter_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.

Game Scenes Flight

Note: Does not change when executing the maneuver node. See *krpc_SpaceCenter_Node_RemainingBurnVector()*.

krpc_error_t **krpc_SpaceCenter_Node_RemainingBurnVector** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*)
Returns the remaining burn vector for the maneuver node.

Parameters

- **referenceFrame** – The reference frame that the returned vector is in. Defaults to `krpc_SpaceCenter_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.

Game Scenes Flight

Note: Changes as the maneuver node is executed. See `krpc_SpaceCenter_Node_BurnVector()`.

krpc_error_t **`krpc_SpaceCenter_Node_UT`** (krpc_connection_t *connection*, double * *result*)

void **`krpc_SpaceCenter_Node_set_UT`** (double *value*)

The universal time at which the maneuver will occur, in seconds.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_Node_TimeTo`** (krpc_connection_t *connection*, double * *result*)

The time until the maneuver node will be encountered, in seconds.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_Node_Orbit`** (krpc_connection_t *connection*, *krpc_SpaceCenter_Orbit_t* * *result*)

The orbit that results from executing the maneuver node.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_Node_Remove`** (krpc_connection_t *connection*)

Removes the maneuver node.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_Node_ReferenceFrame`** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t* * *result*)

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.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_Node_OrbitalReferenceFrame`** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t* * *result*)

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Node_Position** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Node_Direction** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*, *krpc_SpaceCenter_ReferenceFrame_t 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.

Game Scenes Flight

3.3.11 ReferenceFrame

krpc_SpaceCenter_ReferenceFrame_t

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.

krpc_error_t **krpc_SpaceCenter_ReferenceFrame_CreateRelative**(krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t * result*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*, const krpc_tuple_double_double_double_t * *position*, const krpc_tuple_double_double_double_double_t * *rotation*, const krpc_tuple_double_double_double_t * *velocity*, const krpc_tuple_double_double_double_t * *angularVelocity*)

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 $(0, 0, 0)$.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ReferenceFrame_CreateHybrid**(krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t * result*, *krpc_SpaceCenter_ReferenceFrame_t position*, *krpc_SpaceCenter_ReferenceFrame_t rotation*, *krpc_SpaceCenter_ReferenceFrame_t velocity*, *krpc_SpaceCenter_ReferenceFrame_t angularVelocity*)

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.

Game Scenes All

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

`krpc_SpaceCenter_AutoPilot_t`

Provides basic auto-piloting utilities for a vessel. Created by calling *`krpc_SpaceCenter_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.

krpc_error_t **`krpc_SpaceCenter_AutoPilot_Engage`**(krpc_connection_t *connection*)

Engage the auto-pilot.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_AutoPilot_Disengage`**(krpc_connection_t *connection*)

Disengage the auto-pilot.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_AutoPilot_Wait`**(krpc_connection_t *connection*)

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.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_AutoPilot_Error`**(krpc_connection_t *connection*, float * *result*)

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.

Game Scenes Flight

krpc_error_t **`krpc_SpaceCenter_AutoPilot_PitchError`**(krpc_connection_t *connection*, float * *result*)

The error, in degrees, between the vessels current and target pitch. Throws an exception if the auto-pilot has not been engaged.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_HeadingError** (krpc_connection_t *connection*, float * *result*)

The error, in degrees, between the vessels current and target heading. Throws an exception if the auto-pilot has not been engaged.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_RollError** (krpc_connection_t *connection*, float * *result*)

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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_ReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t* * *result*)

void **krpc_SpaceCenter_AutoPilot_set_ReferenceFrame** (*krpc_SpaceCenter_ReferenceFrame_t value*)

The reference frame for the target direction (`krpc_SpaceCenter_AutoPilot_TargetDirection()`).

Game Scenes Flight

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.

---

krpc_error_t **krpc_SpaceCenter_AutoPilot_TargetPitch** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_AutoPilot_set_TargetPitch** (float *value*)

The target pitch, in degrees, between -90° and +90°.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_TargetHeading** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_AutoPilot_set_TargetHeading** (float *value*)

The target heading, in degrees, between 0° and 360°.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_TargetRoll** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_AutoPilot_set_TargetRoll** (float *value*)

The target roll, in degrees. `NaN` if no target roll is set.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_TargetDirection** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*) void **krpc_SpaceCenter_AutoPilot_set_TargetDirection** (const krpc_tuple_double_double_double_t * *value*)

Direction vector corresponding to the target pitch and heading. This is in the reference frame specified by *krpc_SpaceCenter_ReferenceFrame_t*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_TargetPitchAndHeading** (krpc_connection_t *connection*, 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°.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_SAS** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_AutoPilot_set_SAS** (bool *value*)

The state of SAS.

Game Scenes Flight

Note: Equivalent to *krpc_SpaceCenter_Control_SAS()*

krpc_error_t **krpc_SpaceCenter_AutoPilot_SASMode** (krpc_connection_t *connection*, *krpc_SpaceCenter_SASMode_t* * *result*)

void **krpc_SpaceCenter_AutoPilot_set_SASMode** (*krpc_SpaceCenter_SASMode_t value*)

The current *krpc_SpaceCenter_SASMode_t*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

Game Scenes Flight

Note: Equivalent to *krpc_SpaceCenter_Control_SASMode()*

krpc_error_t **krpc_SpaceCenter_AutoPilot_RollThreshold** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_AutoPilot_set_RollThreshold** (double *value*)

The threshold at which the autopilot will try to match the target roll angle, if any. Defaults to 5 degrees.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_StoppingTime** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*) void **krpc_SpaceCenter_AutoPilot_set_StoppingTime** (const krpc_tuple_double_double_double_t * *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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_DecelerationTime** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_SpaceCenter_AutoPilot_set_DecelerationTime** (const krpc_tuple_double_double_double_t * *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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_AttenuationAngle** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_SpaceCenter_AutoPilot_set_AttenuationAngle** (const krpc_tuple_double_double_double_t * *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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_AutoTune** (krpc_connection_t *connection*, bool * *result*)

void **krpc_SpaceCenter_AutoPilot_set_AutoTune** (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 *krpc_SpaceCenter_AutoPilot_TimeToPeak()* and *krpc_SpaceCenter_AutoPilot_Overshoot()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_TimeToPeak** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*) void **krpc_SpaceCenter_AutoPilot_set_TimeToPeak** (const krpc_tuple_double_double_double_t * *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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_Overshoot** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_SpaceCenter_AutoPilot_set_Overshoot** (const krpc_tuple_double_double_double_t * *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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_AutoPilot_PitchPIDGains** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_SpaceCenter_AutoPilot_set_PitchPIDGains** (const krpc_tuple_double_double_double_t * *value*)

Gains for the pitch PID controller.

Game Scenes Flight

Note: When *krpc_SpaceCenter_AutoPilot_AutoTune()* is true, these values are updated automatically, which will overwrite any manual changes.

krpc_error_t **krpc_SpaceCenter_AutoPilot_RollPIDGains** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_SpaceCenter_AutoPilot_set_RollPIDGains** (const krpc_tuple_double_double_double_t * *value*)

Gains for the roll PID controller.

Game Scenes Flight

Note: When *krpc_SpaceCenter_AutoPilot_AutoTune()* is true, these values are updated automatically, which will overwrite any manual changes.

krpc_error_t **krpc_SpaceCenter_AutoPilot_YawPIDGains** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*) void **krpc_SpaceCenter_AutoPilot_set_YawPIDGains** (const krpc_tuple_double_double_double_t * *value*)

Gains for the yaw PID controller.

Game Scenes Flight

Note: When *krpc_SpaceCenter_AutoPilot_AutoTune()* is true, these values are updated automatically, which will overwrite any manual changes.

3.3.13 Camera

krpc_SpaceCenter_Camera_t

Controls the game's camera. Obtained by calling *krpc_SpaceCenter_Camera()*.

krpc_error_t **krpc_SpaceCenter_Camera_Mode** (krpc_connection_t *connection*, *krpc_SpaceCenter_CameraMode_t* * *result*)

void **krpc_SpaceCenter_Camera_set_Mode** (*krpc_SpaceCenter_CameraMode_t value*)

The current mode of the camera.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_Pitch** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Camera_set_Pitch** (float *value*)

The pitch of the camera, in degrees. A value between *krpc_SpaceCenter_Camera_MinPitch()* and *krpc_SpaceCenter_Camera_MaxPitch()*

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_Heading** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Camera_set_Heading** (float *value*)

The heading of the camera, in degrees.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_Distance** (krpc_connection_t *connection*, float * *result*)

void **krpc_SpaceCenter_Camera_set_Distance** (float *value*)

The distance from the camera to the subject, in meters. A value between *krpc_SpaceCenter_Camera_MinDistance()* and *krpc_SpaceCenter_Camera_MaxDistance()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_MinPitch** (krpc_connection_t *connection*, float * *result*)

The minimum pitch of the camera.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_MaxPitch** (krpc_connection_t *connection*, float * *result*)
The maximum pitch of the camera.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_MinDistance** (krpc_connection_t *connection*, float * *result*)
Minimum distance from the camera to the subject, in meters.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_MaxDistance** (krpc_connection_t *connection*, float * *result*)
Maximum distance from the camera to the subject, in meters.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_DefaultDistance** (krpc_connection_t *connection*, float * *result*)
Default distance from the camera to the subject, in meters.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_FocussedBody** (krpc_connection_t *connection*, *krpc_SpaceCenter_CelestialBody_t* * *result*)

void **krpc_SpaceCenter_Camera_set_FocussedBody** (*krpc_SpaceCenter_CelestialBody_t value*)
In map mode, the celestial body that the camera is focussed on. Returns `nullptr` if the camera is not focussed on a celestial body. Returns an error is the camera is not in map mode.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_FocussedVessel** (krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t* * *result*)

void **krpc_SpaceCenter_Camera_set_FocussedVessel** (*krpc_SpaceCenter_Vessel_t value*)
In map mode, the vessel that the camera is focussed on. Returns `nullptr` if the camera is not focussed on a vessel. Returns an error is the camera is not in map mode.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Camera_FocussedNode** (krpc_connection_t *connection*, *krpc_SpaceCenter_Node_t* * *result*)

void **krpc_SpaceCenter_Camera_set_FocussedNode** (*krpc_SpaceCenter_Node_t value*)
In map mode, the maneuver node that the camera is focussed on. Returns `nullptr` if the camera is not focussed on a maneuver node. Returns an error is the camera is not in map mode.

Game Scenes Flight

krpc_SpaceCenter_CameraMode_t
See *krpc_SpaceCenter_Camera_Mode()*.

KRPC_SPACECENTER_CAMERAMODE_AUTOMATIC
The camera is showing the active vessel, in "auto" mode.

KRPC_SPACECENTER_CAMERAMODE_FREE
The camera is showing the active vessel, in "free" mode.

KRPC_SPACECENTER_CAMERAMODE_CHASE
The camera is showing the active vessel, in "chase" mode.

KRPC_SPACECENTER_CAMERAMODE_LOCKED
The camera is showing the active vessel, in "locked" mode.

KRPC_SPACECENTER_CAMERAMODE_ORBITAL
The camera is showing the active vessel, in "orbital" mode.

KRPC_SPACECENTER_CAMERAMODE_IVA
The Intra-Vehicular Activity view is being shown.

KRPC_SPACECENTER_CAMERAMODE_MAP
The map view is being shown.

3.3.14 Waypoints

krpc_SpaceCenter_WaypointManager_t
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 *krpc_SpaceCenter_WaypointManager()*.

krpc_error_t **krpc_SpaceCenter_WaypointManager_Waypoints**(krpc_connection_t *connection*, krpc_list_object_t * *result*)
A list of all existing waypoints.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_WaypointManager_AddWaypoint**(krpc_connection_t *connection*, *krpc_SpaceCenter_Waypoint_t* * *result*, double *latitude*, double *longitude*, *krpc_SpaceCenter_CelestialBody_t body*, const char * *name*)
Creates a waypoint at the given position at ground level, and returns a *krpc_SpaceCenter_Waypoint_t* 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_WaypointManager_AddWaypointAtAltitude** (krpc_connection_t *connection*, *krpc_SpaceCenter_Waypoint_t * result*, double *latitude*, double *longitude*, double *altitude*, *krpc_SpaceCenter_CelestialBody_t body*, const char * *name*)

Creates a waypoint at the given position and altitude, and returns a `krpc_SpaceCenter_Waypoint_t` 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.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_WaypointManager_Colors** (krpc_connection_t *connection*, krpc_dictionary_string_int32_t * *result*)

An example map of known color - seed pairs. Any other integers may be used as seed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_WaypointManager_Icons** (krpc_connection_t *connection*, krpc_list_string_t * *result*)

Returns all available icons (from "GameData/Squad/Contracts/Icons/").

Game Scenes All

krpc_SpaceCenter_Waypoint_t

Represents a waypoint. Can be created using `krpc_SpaceCenter_WaypointManager_AddWaypoint()`.

krpc_error_t **krpc_SpaceCenter_Waypoint_Body** (krpc_connection_t *connection*, *krpc_SpaceCenter_CelestialBody_t * result*)

void **krpc_SpaceCenter_Waypoint_set_Body** (*krpc_SpaceCenter_CelestialBody_t value*)

The celestial body the waypoint is attached to.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Name** (krpc_connection_t *connection*, char * * *result*)

void **krpc_SpaceCenter_Waypoint_set_Name** (const char * *value*)
The name of the waypoint as it appears on the map and the contract.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Color** (krpc_connection_t *connection*, int32_t * *result*)

void **krpc_SpaceCenter_Waypoint_set_Color** (int32_t *value*)
The seed of the icon color. See *krpc_SpaceCenter_WaypointManager_Colors()* for example colors.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Icon** (krpc_connection_t *connection*, char * * *result*)

void **krpc_SpaceCenter_Waypoint_set_Icon** (const char * *value*)
The icon of the waypoint.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Latitude** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_Waypoint_set_Latitude** (double *value*)
The latitude of the waypoint.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Longitude** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_Waypoint_set_Longitude** (double *value*)
The longitude of the waypoint.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_MeanAltitude** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_Waypoint_set_MeanAltitude** (double *value*)
The altitude of the waypoint above sea level, in meters.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_SurfaceAltitude** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_Waypoint_set_SurfaceAltitude** (double *value*)
The altitude of the waypoint above the surface of the body or sea level, whichever is closer, in meters.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_BedrockAltitude** (krpc_connection_t *connection*, double * *result*)

void **krpc_SpaceCenter_Waypoint_set_BedrockAltitude** (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.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_NearSurface** (krpc_connection_t *connection*, bool * *result*)
`true` if the waypoint is near to the surface of a body.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Grounded** (krpc_connection_t *connection*, bool * *result*)
`true` if the waypoint is attached to the ground.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Index** (krpc_connection_t *connection*, int32_t * *result*)
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 *krpc_SpaceCenter_Waypoint_Clustered()* is `false`, this is zero.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Clustered** (krpc_connection_t *connection*, bool * *result*)
`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 *krpc_SpaceCenter_Waypoint_Index()*.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_HasContract** (krpc_connection_t *connection*, bool * *result*)
Whether the waypoint belongs to a contract.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Contract** (krpc_connection_t *connection*, *krpc_SpaceCenter_Contract_t* * *result*)
The associated contract.

Game Scenes Flight

krpc_error_t **krpc_SpaceCenter_Waypoint_Remove** (krpc_connection_t *connection*)
Removes the waypoint.

Game Scenes Flight

3.3.15 Contracts

krpc_SpaceCenter_ContractManager_t
Contracts manager. Obtained by calling *krpc_SpaceCenter_ContractManager()*.

krpc_error_t **krpc_SpaceCenter_ContractManager_Types** (krpc_connection_t *connection*, krpc_set_string_t * *result*)

A list of all contract types.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractManager_AllContracts** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all contracts.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractManager_ActiveContracts** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all active contracts.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractManager_OfferedContracts** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all offered, but unaccepted, contracts.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractManager_CompletedContracts** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all completed contracts.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractManager_FailedContracts** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all failed contracts.

Game Scenes All

krpc_SpaceCenter_Contract_t

A contract. Can be accessed using *krpc_SpaceCenter_ContractManager()*.

krpc_error_t **krpc_SpaceCenter_Contract_Type** (krpc_connection_t *connection*, char * * *result*)

Type of the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Title** (krpc_connection_t *connection*, char * * *result*)

Title of the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Description** (krpc_connection_t *connection*, char * * *result*)

Description of the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Notes** (krpc_connection_t *connection*, char * * *result*)

Notes for the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Synopsis** (krpc_connection_t *connection*, char * * *result*)

Synopsis for the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Keywords** (krpc_connection_t *connection*, krpc_list_string_t * *result*)

Keywords for the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_State** (krpc_connection_t *connection*, *krpc_SpaceCenter_ContractState_t* * *result*)

State of the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Seen** (krpc_connection_t *connection*, bool * *result*)

Whether the contract has been seen.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Read** (krpc_connection_t *connection*, bool * *result*)

Whether the contract has been read.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Active** (krpc_connection_t *connection*, bool * *result*)

Whether the contract is active.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Failed** (krpc_connection_t *connection*, bool * *result*)

Whether the contract has been failed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_CanBeCanceled** (krpc_connection_t *connection*, bool * *result*)

Whether the contract can be canceled.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_CanBeDeclined** (krpc_connection_t *connection*, bool * *result*)

Whether the contract can be declined.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_CanBeFailed** (krpc_connection_t *connection*, bool * *result*)

Whether the contract can be failed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Accept** (krpc_connection_t *connection*)
Accept an offered contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Cancel** (krpc_connection_t *connection*)
Cancel an active contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Decline** (krpc_connection_t *connection*)
Decline an offered contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_FundsAdvance** (krpc_connection_t *connection*, double * *result*)
Funds received when accepting the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_FundsCompletion** (krpc_connection_t *connection*, double * *result*)
Funds received on completion of the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_FundsFailure** (krpc_connection_t *connection*, double * *result*)
Funds lost if the contract is failed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_ReputationCompletion** (krpc_connection_t *connection*, double * *result*)
Reputation gained on completion of the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_ReputationFailure** (krpc_connection_t *connection*, double * *result*)
Reputation lost if the contract is failed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_ScienceCompletion** (krpc_connection_t *connection*, double * *result*)
Science gained on completion of the contract.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_Contract_Parameters** (krpc_connection_t *connection*, krpc_list_object_t * *result*)
Parameters for the contract.

Game Scenes All

krpc_SpaceCenter_ContractState_t
The state of a contract. See *krpc_SpaceCenter_Contract_State()*.

KRPC_SPACECENTER_CONTRACTSTATE_ACTIVE
The contract is active.

KRPC_SPACECENTER_CONTRACTSTATE_CANCELED
The contract has been canceled.

KRPC_SPACECENTER_CONTRACTSTATE_COMPLETED
The contract has been completed.

KRPC_SPACECENTER_CONTRACTSTATE_DEADLINEEXPIRED
The deadline for the contract has expired.

KRPC_SPACECENTER_CONTRACTSTATE_DECLINED
The contract has been declined.

KRPC_SPACECENTER_CONTRACTSTATE_FAILED
The contract has been failed.

KRPC_SPACECENTER_CONTRACTSTATE_GENERATED
The contract has been generated.

KRPC_SPACECENTER_CONTRACTSTATE_OFFERED
The contract has been offered to the player.

KRPC_SPACECENTER_CONTRACTSTATE_OFFEREXPIRED
The contract was offered to the player, but the offer expired.

KRPC_SPACECENTER_CONTRACTSTATE_WITHDRAWN
The contract has been withdrawn.

krpc_SpaceCenter_ContractParameter_t
A contract parameter. See `krpc_SpaceCenter_Contract_Parameters()`.

krpc_error_t **krpc_SpaceCenter_ContractParameter_Title** (krpc_connection_t *connection*, char * * *result*)
Title of the parameter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_Notes** (krpc_connection_t *connection*, char * * *result*)
Notes for the parameter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_Children** (krpc_connection_t *connection*, krpc_list_object_t * *result*)
Child contract parameters.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_Completed** (krpc_connection_t *connection*, bool * *result*)
Whether the parameter has been completed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_Failed** (krpc_connection_t *connection*, bool * *result*)
Whether the parameter has been failed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_Optional** (krpc_connection_t *connection*, bool * *result*)
Whether the contract parameter is optional.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_FundsCompletion** (krpc_connection_t *connection*, double * *result*)

Funds received on completion of the contract parameter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_FundsFailure** (krpc_connection_t *connection*, double * *result*)

Funds lost if the contract parameter is failed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_ReputationCompletion** (krpc_connection_t *connection*, double * *result*)

Reputation gained on completion of the contract parameter.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_ReputationFailure** (krpc_connection_t *connection*, double * *result*)

Reputation lost if the contract parameter is failed.

Game Scenes All

krpc_error_t **krpc_SpaceCenter_ContractParameter_ScienceCompletion** (krpc_connection_t *connection*, double * *result*)

Science gained on completion of the contract parameter.

Game Scenes All

3.3.16 Geometry Types

Vectors

3-dimensional vectors are represented as a 3-tuple. For example:

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "");
  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_Flight_t flight;
  krpc_SpaceCenter_
↪Vessel_Flight(conn, &flight, vessel, KRPC_NULL);
  krpc_tuple_double_double_double_t v;
  krpc_
↪SpaceCenter_Flight_Prograde(conn, &v, flight);
```

(continues on next page)

(continued from previous page)

```
  printf("%.2f %.2f %.2f\n", v.e0, v.e1, v.e2);
}
```

Quaternions

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "");
  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);
  krpc_SpaceCenter_Flight_t flight;
  krpc_SpaceCenter_
↪Vessel_Flight(conn, &flight, vessel, KRPC_NULL);
  krpc_tuple_double_double_double_double_t q;
  krpc_
↪SpaceCenter_Flight_Rotation(conn, &q, flight);
  printf("
↪%.2f %.2f %.2f %.2f\n", q.e0, q.e1, q.e2, q.e3);
}
```

3.4 Drawing API

3.4.1 Drawing

Service Drawing

Provides functionality for drawing objects in the flight scene.

krpc_error_t **krpc_Drawing_AddLine** (krpc_connection_t *connection*, *krpc_Drawing_Line_t* * *result*, const krpc_tuple_double_double_double_t * *start*, const krpc_tuple_double_double_double_t * *end*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*, bool *visible*)

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.

Game Scenes Flight

krpc_error_t **krpc_Drawing_AddDirection** (krpc_connection_t *connection*, *krpc_Drawing_Line_t* * *result*, const krpc_tuple_double_double_double_t * *direction*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*, float *length*, bool *visible*)

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.

Game Scenes Flight

krpc_error_t **krpc_Drawing_AddPolygon** (krpc_connection_t *connection*, *krpc_Drawing_Polygon_t* * *result*, const krpc_list_tuple_double_double_double_t * *vertices*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*, bool *visible*)

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.

Game Scenes Flight

krpc_error_t **krpc_Drawing_AddText** (krpc_connection_t *connection*, *krpc_Drawing_Text_t* * *result*, const char * *text*, *krpc_SpaceCenter_ReferenceFrame_t referenceFrame*, const krpc_tuple_double_double_double_t * *position*, const krpc_tuple_double_double_double_double_t * *rotation*, bool *visible*)

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.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Clear** (krpc_connection_t *connection*, bool *clientOnly*)

Remove all objects being drawn.

Parameters

- **clientOnly** – If true, only remove objects created by the calling client.

Game Scenes Flight

3.4.2 Line

krpc_Drawing_Line_t
A line. Created using `krpc_Drawing_AddLine()`.

krpc_error_t **krpc_Drawing_Line_Start** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_Drawing_Line_set_Start** (const krpc_tuple_double_double_double_t * *value*)
Start position of the line.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Line_End** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_Drawing_Line_set_End** (const krpc_tuple_double_double_double_t * *value*)
End position of the line.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Line_ReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t* * *result*)

void **krpc_Drawing_Line_set_ReferenceFrame** (*krpc_SpaceCenter_ReferenceFrame_t value*)
Reference frame for the positions of the object.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Line_Visible** (krpc_connection_t *connection*, bool * *result*)

void **krpc_Drawing_Line_set_Visible** (bool *value*)
Whether the object is visible.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Line_Color** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_Drawing_Line_set_Color** (const krpc_tuple_double_double_double_t * *value*)
Set the color

Game Scenes Flight

krpc_error_t **krpc_Drawing_Line_Material** (krpc_connection_t *connection*, char * * *result*)

void **krpc_Drawing_Line_set_Material** (const char * *value*)
Material used to render the object. Creates the material from a shader with the given name.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Line_Thickness** (krpc_connection_t *connection*, float * *result*)

void **krpc_Drawing_Line_set_Thickness** (float *value*)
Set the thickness

Game Scenes Flight

krpc_error_t **krpc_Drawing_Line_Remove** (krpc_connection_t *connection*)
Remove the object.

Game Scenes Flight

3.4.3 Polygon

krpc_Drawing_Polygon_t
A polygon. Created using `krpc_Drawing_AddPolygon()`.

krpc_error_t **krpc_Drawing_Polygon_Vertices** (krpc_connection_t *connection*, krpc_list_tuple_double_double_double_t * *result*)

void **krpc_Drawing_Polygon_set_Vertices** (const krpc_list_tuple_double_double_double_t * *value*)
Vertices for the polygon.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Polygon_ReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t* * *result*)

void **krpc_Drawing_Polygon_set_ReferenceFrame** (*krpc_SpaceCenter_ReferenceFrame_t value*)
Reference frame for the positions of the object.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Polygon_Visible** (krpc_connection_t *connection*, bool * *result*)

void **krpc_Drawing_Polygon_set_Visible** (bool *value*)
Whether the object is visible.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Polygon_Remove** (krpc_connection_t *connection*)
Remove the object.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Polygon_Color** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_Drawing_Polygon_set_Color** (const krpc_tuple_double_double_double_t * *value*)
Set the color

Game Scenes Flight

krpc_error_t **krpc_Drawing_Polygon_Material** (krpc_connection_t *connection*, char * * *result*)

void **krpc_Drawing_Polygon_set_Material** (const char * *value*)
Material used to render the object. Creates the material from a shader with the given name.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Polygon_Thickness** (krpc_connection_t *connection*, float * *result*)

void **krpc_Drawing_Polygon_set_Thickness** (float *value*)
Set the thickness

Game Scenes Flight

3.4.4 Text

krpc_Drawing_Text_t
Text. Created using *krpc_Drawing_AddText()*.

krpc_error_t **krpc_Drawing_Text_Position** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_Drawing_Text_set_Position** (const krpc_tuple_double_double_double_t * *value*)
Position of the text.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Rotation** (krpc_connection_t *connection*, krpc_tuple_double_double_double_double_t * *result*)

void **krpc_Drawing_Text_set_Rotation** (const krpc_tuple_double_double_double_double_t * *value*)
Rotation of the text as a quaternion.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_ReferenceFrame** (krpc_connection_t *connection*, *krpc_SpaceCenter_ReferenceFrame_t* * *result*)

void **krpc_Drawing_Text_set_ReferenceFrame** (*krpc_SpaceCenter_ReferenceFrame_t value*)
Reference frame for the positions of the object.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Visible** (krpc_connection_t *connection*, bool * *result*)

void **krpc_Drawing_Text_set_Visible** (bool *value*)
Whether the object is visible.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Remove** (krpc_connection_t *connection*)
Remove the object.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Content** (krpc_connection_t *connection*, char * * *result*)

void **krpc_Drawing_Text_set_Content** (const char * *value*)
The text string

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Font** (krpc_connection_t *connection*, char * * *result*)

void **krpc_Drawing_Text_set_Font** (const char * *value*)
Name of the font

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_AvailableFonts** (krpc_connection_t *connection*, krpc_list_string_t * *result*)

A list of all available fonts.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Size** (krpc_connection_t *connection*, int32_t * *result*)

void **krpc_Drawing_Text_set_Size** (int32_t *value*)

Font size.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_CharacterSize** (krpc_connection_t *connection*, float * *result*)

void **krpc_Drawing_Text_set_CharacterSize** (float *value*)

Character size.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Style** (krpc_connection_t *connection*, *krpc_UI_FontStyle_t* * *result*)

void **krpc_Drawing_Text_set_Style** (*krpc_UI_FontStyle_t value*)

Font style.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Color** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_Drawing_Text_set_Color** (const krpc_tuple_double_double_double_t * *value*)

Set the color

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Material** (krpc_connection_t *connection*, char * * *result*)

void **krpc_Drawing_Text_set_Material** (const char * *value*)

Material used to render the object. Creates the material from a shader with the given name.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Alignment** (krpc_connection_t *connection*, *krpc_UI_TextAlignment_t* * *result*)

void **krpc_Drawing_Text_set_Alignment** (*krpc_UI_TextAlignment_t value*)

Alignment.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_LineSpacing** (krpc_connection_t *connection*, float * *result*)

void **krpc_Drawing_Text_set_LineSpacing** (float *value*)

Line spacing.

Game Scenes Flight

krpc_error_t **krpc_Drawing_Text_Anchor** (krpc_connection_t *connection*, *krpc_UI_TextAnchor_t* * *result*)

void **krpc_Drawing_Text_set_Anchor** (*krpc_UI_TextAnchor_t value*)
Anchor.

Game Scenes Flight

3.5 InfernalRobotics API

Provides RPCs to interact with the InfernalRobotics mod. Both the original mod and Infernal Robotics Next are supported. Provides the following classes:

3.5.1 InfernalRobotics

Service InfernalRobotics

This service provides functionality to interact with Infernal Robotics.

krpc_error_t **krpc_InfernalRobotics_Available** (krpc_connection_t *connection*, bool * *result*)
Whether Infernal Robotics is installed.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Ready** (krpc_connection_t *connection*, bool * *result*)
Whether Infernal Robotics API is ready.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroups** (krpc_connection_t *connection*, krpc_list_object_t * *result*, *krpc_SpaceCenter_Vessel_t vessel*)
A list of all the servo groups in the given *vessel*.

Parameters

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroupWithName** (krpc_connection_t *connection*, *krpc_InfernalRobotics_ServoGroup_t* * *result*, *krpc_SpaceCenter_Vessel_t vessel*, const char * *name*)
Returns the servo group in the given *vessel* with the given *name*, or `nullptr` 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.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoWithName** (krpc_connection_t *connection*, *krpc_InfernalRobotics_Servo_t* * *result*, *krpc_SpaceCenter_Vessel_t* *vessel*, const char * *name*)

Returns the servo in the given *vessel* with the given *name* or `nullptr` 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.

Game Scenes Flight

3.5.2 ServoGroup

krpc_InfernalRobotics_ServoGroup_t

A group of servos, obtained by calling *krpc_InfernalRobotics_ServoGroups()* or *krpc_InfernalRobotics_ServoGroupWithName()*. Represents the "Servo Groups" in the InfernalRobotics UI.

krpc_error_t **krpc_InfernalRobotics_ServoGroup_Name** (krpc_connection_t *connection*, char * * *result*)

void **krpc_InfernalRobotics_ServoGroup_set_Name** (const char * *value*)

The name of the group.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_ForwardKey** (krpc_connection_t *connection*, char * * *result*)

void **krpc_InfernalRobotics_ServoGroup_set_ForwardKey** (const char * *value*)

The key assigned to be the "forward" key for the group.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_ReverseKey** (krpc_connection_t *connection*, char * * *result*)

void **krpc_InfernalRobotics_ServoGroup_set_ReverseKey** (const char * *value*)

The key assigned to be the "reverse" key for the group.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_Speed** (krpc_connection_t *connection*, float * *result*)

void **krpc_InfernalRobotics_ServoGroup_set_Speed** (float *value*)

The speed multiplier for the group.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_Expanded** (krpc_connection_t *connection*, bool * *result*)

void **krpc_InfernalRobotics_ServoGroup_set_Expanded** (bool *value*)
Whether the group is expanded in the InfernalRobotics UI.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_Servos** (krpc_connection_t *connection*, krpc_list_object_t * *result*)
The servos that are in the group.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_ServoWithName** (krpc_connection_t *connection*, *krpc_InfernalRobotics_Servo_t* * *result*, const char * *name*)
Returns the servo with the given *name* from this group, or `nullptr` if none exists.

Parameters

- **name** – Name of servo to find.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_Parts** (krpc_connection_t *connection*, krpc_list_object_t * *result*)
The parts containing the servos in the group.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_MoveRight** (krpc_connection_t *connection*)
Moves all of the servos in the group to the right.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_MoveLeft** (krpc_connection_t *connection*)
Moves all of the servos in the group to the left.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_MoveCenter** (krpc_connection_t *connection*)
Moves all of the servos in the group to the center.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_MoveNextPreset** (krpc_connection_t *connection*)
Moves all of the servos in the group to the next preset.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_MovePrevPreset** (krpc_connection_t *connection*)
Moves all of the servos in the group to the previous preset.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_ServoGroup_Stop** (krpc_connection_t *connection*)
Stops the servos in the group.

Game Scenes Flight

3.5.3 Servo

krpc_InfernalRobotics_Servo_t

Represents a servo. Obtained using *krpc_InfernalRobotics_ServoGroup_Servos()*, *krpc_InfernalRobotics_ServoGroup_ServoWithName()* or *krpc_InfernalRobotics_ServoWithName()*.

krpc_error_t **krpc_InfernalRobotics_Servo_Name** (krpc_connection_t *connection*, char * * *result*)

void **krpc_InfernalRobotics_Servo_set_Name** (const char * *value*)

The name of the servo.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t* * *result*)

The part containing the servo.

Game Scenes Flight

void **krpc_InfernalRobotics_Servo_set_Highlight** (bool *value*)

Whether the servo should be highlighted in-game.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_Position** (krpc_connection_t *connection*, float * *result*)

The position of the servo.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MinConfigPosition** (krpc_connection_t *connection*, float * *result*)

The minimum position of the servo, specified by the part configuration.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MaxConfigPosition** (krpc_connection_t *connection*, float * *result*)

The maximum position of the servo, specified by the part configuration.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MinPosition** (krpc_connection_t *connection*, float * *result*)

void **krpc_InfernalRobotics_Servo_set_MinPosition** (float *value*)

The minimum position of the servo, specified by the in-game tweak menu.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MaxPosition** (krpc_connection_t *connection*, float * *result*)

void **krpc_InfernalRobotics_Servo_set_MaxPosition** (float *value*)

The maximum position of the servo, specified by the in-game tweak menu.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_ConfigSpeed**(krpc_connection_t *connection*, float * *result*)

The speed multiplier of the servo, specified by the part configuration.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_Speed**(krpc_connection_t *connection*, float * *result*)

void **krpc_InfernalRobotics_Servo_set_Speed**(float *value*)

The speed multiplier of the servo, specified by the in-game tweak menu.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_CurrentSpeed**(krpc_connection_t *connection*, float * *result*)

void **krpc_InfernalRobotics_Servo_set_CurrentSpeed**(float *value*)

The current speed at which the servo is moving.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_Acceleration**(krpc_connection_t *connection*, float * *result*)

void **krpc_InfernalRobotics_Servo_set_Acceleration**(float *value*)

The current speed multiplier set in the UI.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_IsMoving**(krpc_connection_t *connection*, bool * *result*)

Whether the servo is moving.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_IsFreeMoving**(krpc_connection_t *connection*, bool * *result*)

Whether the servo is freely moving.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_IsLocked**(krpc_connection_t *connection*, bool * *result*)

void **krpc_InfernalRobotics_Servo_set_IsLocked**(bool *value*)

Whether the servo is locked.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_IsAxisInverted**(krpc_connection_t *connection*, bool * *result*)

void **krpc_InfernalRobotics_Servo_set_IsAxisInverted**(bool *value*)

Whether the servos axis is inverted.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MoveRight**(krpc_connection_t *connection*)

Moves the servo to the right.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MoveLeft** (krpc_connection_t *connection*)

Moves the servo to the left.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MoveCenter** (krpc_connection_t *connection*)

Moves the servo to the center.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MoveNextPreset** (krpc_connection_t *connection*)

Moves the servo to the next preset.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MovePrevPreset** (krpc_connection_t *connection*)

Moves the servo to the previous preset.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_MoveTo** (krpc_connection_t *connection*, 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.

Game Scenes Flight

krpc_error_t **krpc_InfernalRobotics_Servo_Stop** (krpc_connection_t *connection*)

Stops the servo.

Game Scenes Flight

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.

```
#include <unistd.h>
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>
#include <krpc_cnano/services/infernal_robotics.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "InfernalRobotics Example");
  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);

  krpc_InfernalRobotics_ServoGroup_t group;
  krpc_InfernalRobotics_ServoGroupWithName(conn,
↪ &group, vessel, "MyGroup");
  if (!group)
    printf("Group not found\n");
```

(continues on next page)

(continued from previous page)

```
  krpc_list_object_t servos = KRPC_NULL_LIST;
  krpc_InfernalRobotics_
↪ServoGroup_Servos(conn, &servos, group);
  for (size_t i = 0; i < servos.size; i++) {
    krpc_InfernalRobotics_
↪Servo_t servo = servos.items[i];
    char * name = NULL;
    krpc_
↪InfernalRobotics_Servo_Name(conn, &name, servo);
    float position;
    krpc_InfernalRobotics_
↪Servo_Position(conn, &position, servo);
    printf("%s %.2f\n", name, position);
  }

  krpc_InfernalRobotics_
↪ServoGroup_MoveRight(conn, group);
  sleep(1);
  krpc_
↪InfernalRobotics_ServoGroup_Stop(conn, group);
}
```

3.6 Kerbal Alarm Clock API

Provides RPCs to interact with the Kerbal Alarm Clock mod. Provides the following classes:

3.6.1 KerbalAlarmClock

Service KerbalAlarmClock

This service provides functionality to interact with Kerbal Alarm Clock.

krpc_error_t **krpc_KerbalAlarmClock_Available** (krpc_connection_t *connection*, bool * *result*)

Whether Kerbal Alarm Clock is available.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarms** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

A list of all the alarms.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_AlarmWithName** (krpc_connection_t *connection*, *krpc_KerbalAlarmClock_Alarm_t* * *result*, const char * *name*)

Get the alarm with the given *name*, or `nullptr` 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.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_AlarmsWithType** (krpc_connection_t *connection*, krpc_list_object_t * *result*, *krpc_KerbalAlarmClock_AlarmType_t type*)

Get a list of alarms of the specified *type*.

Parameters

- **type** – Type of alarm to return.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_CreateAlarm** (krpc_connection_t *connection*, *krpc_KerbalAlarmClock_Alarm_t* * *result*, *krpc_KerbalAlarmClock_AlarmType_t type*, const char * *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.

Game Scenes All

3.6.2 Alarm

krpc_KerbalAlarmClock_Alarm_t

Represents an alarm. Obtained by calling `krpc_KerbalAlarmClock_Alarms()`, `krpc_KerbalAlarmClock_AlarmWithName()` or `krpc_KerbalAlarmClock_AlarmsWithType()`.

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Action** (krpc_connection_t *connection*, *krpc_KerbalAlarmClock_AlarmAction_t* * *result*)

void **krpc_KerbalAlarmClock_Alarm_set_Action** (*krpc_KerbalAlarmClock_AlarmAction_t value*)

The action that the alarm triggers.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Margin** (krpc_connection_t *connection*, double * *result*)

void **krpc_KerbalAlarmClock_Alarm_set_Margin** (double *value*)

The number of seconds before the event that the alarm will fire.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Time** (krpc_connection_t *connection*, double * *result*)

void **krpc_KerbalAlarmClock_Alarm_set_Time** (double *value*)
The time at which the alarm will fire.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Type** (krpc_connection_t *connection*, *krpc_KerbalAlarmClock_AlarmType_t * result*)
The type of the alarm.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_ID** (krpc_connection_t *connection*, char * * *result*)
The unique identifier for the alarm.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Name** (krpc_connection_t *connection*, char * * *result*)

void **krpc_KerbalAlarmClock_Alarm_set_Name** (const char * *value*)
The short name of the alarm.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Notes** (krpc_connection_t *connection*, char * * *result*)

void **krpc_KerbalAlarmClock_Alarm_set_Notes** (const char * *value*)
The long description of the alarm.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Remaining** (krpc_connection_t *connection*, double * *result*)
The number of seconds until the alarm will fire.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Repeat** (krpc_connection_t *connection*, bool * *result*)

void **krpc_KerbalAlarmClock_Alarm_set_Repeat** (bool *value*)
Whether the alarm will be repeated after it has fired.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_RepeatPeriod** (krpc_connection_t *connection*, double * *result*)

void **krpc_KerbalAlarmClock_Alarm_set_RepeatPeriod** (double *value*)
The time delay to automatically create an alarm after it has fired.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Vessel** (krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t * result*)

void **krpc_KerbalAlarmClock_Alarm_set_Vessel** (*krpc_SpaceCenter_Vessel_t value*)
The vessel that the alarm is attached to.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_XferOriginBody** (krpc_connection_t *connection*, *krpc_SpaceCenter_CelestialBody_t * result*)

void **krpc_KerbalAlarmClock_Alarm_set_XferOriginBody** (*krpc_SpaceCenter_CelestialBody_t value*)
The celestial body the vessel is departing from.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_XferTargetBody** (krpc_connection_t *connection*, *krpc_SpaceCenter_CelestialBody_t * result*)

void **krpc_KerbalAlarmClock_Alarm_set_XferTargetBody** (*krpc_SpaceCenter_CelestialBody_t value*)
The celestial body the vessel is arriving at.

Game Scenes All

krpc_error_t **krpc_KerbalAlarmClock_Alarm_Remove** (krpc_connection_t *connection*)
Removes the alarm.

Game Scenes All

3.6.3 AlarmType

krpc_KerbalAlarmClock_AlarmType_t
The type of an alarm.

KRPC_KERBALALARMCLOCK_ALARMTYPE_RAW
An alarm for a specific date/time or a specific period in the future.

KRPC_KERBALALARMCLOCK_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.

KRPC_KERBALALARMCLOCK_ALARMTYPE_MANEUVERAUTO
See *KRPC_KERBALALARMCLOCK_ALARMTYPE_MANEUVER*.

KRPC_KERBALALARMCLOCK_ALARMTYPE_APOAPSIS
An alarm for furthest part of the orbit from the planet.

KRPC_KERBALALARMCLOCK_ALARMTYPE_PERIAPSIS
An alarm for nearest part of the orbit from the planet.

KRPC_KERBALALARMCLOCK_ALARMTYPE_ASCENDINGNODE
Ascending node for the targeted object, or equatorial ascending node.

KRPC_KERBALALARMCLOCK_ALARMTYPE_DESCENDINGNODE
Descending node for the targeted object, or equatorial descending node.

KRPC_KERBALALARMCLOCK_ALARMTYPE_CLOSEST
An alarm based on the closest approach of this vessel to the targeted vessel, some number of orbits into the future.

KRPC_KERBALALARMCLOCK_ALARMTYPE_CONTRACT
An alarm based on the expiry or deadline of contracts in career modes.

KRPC_KERBALALARMCLOCK_ALARMTYPE_CONTRACTAUTO
See *KRPC_KERBALALARMCLOCK_ALARMTYPE_CONTRACT*.

KRPC_KERBALALARMCLOCK_ALARMTYPE_CREW
An alarm that is attached to a crew member.

KRPC_KERBALALARMCLOCK_ALARMTYPE_DISTANCE
An alarm that is triggered when a selected target comes within a chosen distance.

KRPC_KERBALALARMCLOCK_ALARMTYPE_EARTHTIME
An alarm based on the time in the "Earth" alternative Universe (aka the Real World).

KRPC_KERBALALARMCLOCK_ALARMTYPE_LAUNCHRENDEVOUS
An alarm that fires as your landed craft passes under the orbit of your target.

KRPC_KERBALALARMCLOCK_ALARMTYPE_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.

KRPC_KERBALALARMCLOCK_ALARMTYPE_SOICHANGEAUTO
See *KRPC_KERBALALARMCLOCK_ALARMTYPE_SOICHANGE*.

KRPC_KERBALALARMCLOCK_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.

KRPC_KERBALALARMCLOCK_ALARMTYPE_TRANSFERMODELLED
See *KRPC_KERBALALARMCLOCK_ALARMTYPE_TRANSFER*.

3.6.4 AlarmAction

krpc_KerbalAlarmClock_AlarmAction_t
The action performed by an alarm when it fires.

KRPC_KERBALALARMCLOCK_ALARMACTION_DONOTHING
Don't do anything at all...

KRPC_KERBALALARMCLOCK_ALARMACTION_DONOTHINGDELETEWHENPASSED
Don't do anything, and delete the alarm.

KRPC_KERBALALARMCLOCK_ALARMACTION_KILLWARP
Drop out of time warp.

KRPC_KERBALALARMCLOCK_ALARMACTION_KILLWARPONLY
Drop out of time warp.

KRPC_KERBALALARMCLOCK_ALARMACTION_MESSAGEONLY
Display a message.

KRPC_KERBALALARMCLOCK_ALARMACTION_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.

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>
#include␣
↪<krpc_cnano/services/kerbal_alarm_clock.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_
↪connect(conn, "Kerbal Alarm Clock Example");

  double ut;
  krpc_SpaceCenter_UT(conn, &ut);
  krpc_KerbalAlarmClock_Alarm_t alarm;
  krpc_KerbalAlarmClock_CreateAlarm(
    conn, &alarm, KRPC_KERBALALARMCLOCK_
↪ALARMTYPE_RAW, "My New Alarm", ut+10);

  krpc_KerbalAlarmClock_Alarm_set_Notes(
    conn, alarm, "10 seconds␣
↪have now passed since the alarm was created.");
  krpc_KerbalAlarmClock_Alarm_set_Action(
    conn, alarm,
↪ KRPC_KERBALALARMCLOCK_ALARMACTION_MESSAGEONLY);
}
```

3.7 RemoteTech API

Provides RPCs to interact with the RemoteTech mod. Provides the following classes:

3.7.1 RemoteTech

Service RemoteTech

This service provides functionality to interact with RemoteTech.

krpc_error_t **krpc_RemoteTech_Available** (krpc_connection_t *connection*, bool * *result*)
Whether RemoteTech is installed.

Game Scenes All

krpc_error_t **krpc_RemoteTech_GroundStations** (krpc_connection_t *connection*, krpc_list_string_t * *result*)
The names of the ground stations.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Antenna** (krpc_connection_t *connection*, *krpc_RemoteTech_Antenna_t* * *result*, *krpc_SpaceCenter_Part_t part*)

Get the antenna object for a particular part.

Parameters

Game Scenes All

krpc_error_t **krpc_RemoteTech_Comms** (krpc_connection_t *connection*, *krpc_RemoteTech_Comms_t* * *result*, *krpc_SpaceCenter_Vessel_t vessel*)

Get a communications object, representing the communication capability of a particular vessel.

Parameters

Game Scenes All

3.7.2 Comms

krpc_RemoteTech_Comms_t

Communications for a vessel.

krpc_error_t **krpc_RemoteTech_Comms_Vessel** (krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t* * *result*)

Get the vessel.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Comms_HasLocalControl** (krpc_connection_t *connection*, bool * *result*)

Whether the vessel can be controlled locally.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Comms_HasFlightComputer** (krpc_connection_t *connection*, bool * *result*)

Whether the vessel has a flight computer on board.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Comms_HasConnection** (krpc_connection_t *connection*, bool * *result*)

Whether the vessel has any connection.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Comms_HasConnectionToGroundStation** (krpc_connection_t *connection*, bool * *result*)

Whether the vessel has a connection to a ground station.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Comms_SignalDelay** (krpc_connection_t *connection*, double * *result*)

The shortest signal delay to the vessel, in seconds.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Comms_SignalDelayToGroundStation** (krpc_connection_t *connection*, double * *result*)

The signal delay between the vessel and the closest ground station, in seconds.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Comms_SignalDelayToVessel** (krpc_connection_t *connection*, double * *result*, *krpc_SpaceCenter_Vessel_t other*)

The signal delay between the this vessel and another vessel, in seconds.

Parameters

Game Scenes All

krpc_error_t **krpc_RemoteTech_Comms_Antennas** (krpc_connection_t *connection*, krpc_list_object_t * *result*)

The antennas for this vessel.

Game Scenes All

3.7.3 Antenna

krpc_RemoteTech_Antenna_t

A RemoteTech antenna. Obtained by calling *krpc_RemoteTech_Comms_Antennas()* or *krpc_RemoteTech_Antenna()*.

krpc_error_t **krpc_RemoteTech_Antenna_Part** (krpc_connection_t *connection*, *krpc_SpaceCenter_Part_t * result*)

Get the part containing this antenna.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Antenna_HasConnection** (krpc_connection_t *connection*, bool * *result*)

Whether the antenna has a connection.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Antenna_Target** (krpc_connection_t *connection*, *krpc_RemoteTech_Target_t * result*)

void **krpc_RemoteTech_Antenna_set_Target** (*krpc_RemoteTech_Target_t value*)

The object that the antenna is targetting. This property can be used to set the target to *KRPC_REMOTETECH_TARGET_NONE* or *KRPC_REMOTETECH_TARGET_ACTIVEVESSEL*. To set the target to a celestial body, ground station or vessel see *krpc_RemoteTech_Antenna_TargetBody()*, *krpc_RemoteTech_Antenna_TargetGroundStation()* and *krpc_RemoteTech_Antenna_TargetVessel()*.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Antenna_TargetBody** (krpc_connection_t *connection*, *krpc_SpaceCenter_CelestialBody_t * result*)

void **krpc_RemoteTech_Antenna_set_TargetBody** (*krpc_SpaceCenter_CelestialBody_t value*)
The celestial body the antenna is targetting.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Antenna_TargetGroundStation** (krpc_connection_t *connection*, char * * *result*)

void **krpc_RemoteTech_Antenna_set_TargetGroundStation** (const char * *value*)
The ground station the antenna is targetting.

Game Scenes All

krpc_error_t **krpc_RemoteTech_Antenna_TargetVessel** (krpc_connection_t *connection*, *krpc_SpaceCenter_Vessel_t* * *result*)

void **krpc_RemoteTech_Antenna_set_TargetVessel** (*krpc_SpaceCenter_Vessel_t value*)
The vessel the antenna is targetting.

Game Scenes All

krpc_RemoteTech_Target_t
The type of object an antenna is targetting. See *krpc_RemoteTech_Antenna_Target()*.

KRPC_REMOTETECH_TARGET_ACTIVEVESSEL
The active vessel.

KRPC_REMOTETECH_TARGET_CELESTIALBODY
A celestial body.

KRPC_REMOTETECH_TARGET_GROUNDSTATION
A ground station.

KRPC_REMOTETECH_TARGET_VESSEL
A specific vessel.

KRPC_REMOTETECH_TARGET_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.

```
#include <krpc_cnano.h>
#include <krpc_cnano/services/space_center.h>
#include <krpc_cnano/services/remote_tech.h>

int main() {
  krpc_connection_t conn;
  krpc_open(&conn, "COM0");
  krpc_connect(conn, "RemoteTech Example");
  krpc_SpaceCenter_Vessel_t vessel;
  krpc_SpaceCenter_ActiveVessel(conn, &vessel);

  // Set a dish target
  krpc_SpaceCenter_Parts_t parts;
  krpc_
↪SpaceCenter_Vessel_Parts(conn, &parts, vessel);
```

(continues on next page)

(continued from previous page)

```
  krpc_list_object_t parts_with_title;
  krpc_SpaceCenter_Parts_WithTitle(conn,
↪ &parts_with_title, parts, "Reflectron KR-7");
  krpc_SpaceCenter_
↪Part_t part = parts_with_title.items[0];

  krpc_RemoteTech_Antenna_t antenna;
  krpc_RemoteTech_Antenna(conn, &antenna, part);

  krpc_dictionary_
↪string_object_t bodies = KRPC_NULL_DICTIONARY;
  krpc_SpaceCenter_Bodies(conn, &bodies);
  krpc_SpaceCenter_CelestialBody_t jool;
  for (size_t i = 0; i < bodies.size; i++)
    if (!strcmp(bodies.entries[i].key, "Jool"))
      jool = bodies.entries[i].value;

  krpc_RemoteTech_
↪Antenna_set_TargetBody(conn, antenna, jool);

  // Get info about the vessels communications
  krpc_RemoteTech_Comms_t comms;
  krpc_RemoteTech_Comms(conn, &comms, vessel);
  double signal_delay;
  krpc_RemoteTech_
↪Comms_SignalDelay(conn, &signal_delay, comms);
  printf("Signal delay = %.2f\n", signal_delay);
}
```

3.8 User Interface API

3.8.1 UI

Service UI

Provides functionality for drawing and interacting with in-game user interface elements.

krpc_error_t **krpc_UI_StockCanvas** (krpc_connection_t *connection*, *krpc_UI_Canvas_t * result*)

The stock UI canvas.

Game Scenes All

krpc_error_t **krpc_UI_AddCanvas** (krpc_connection_t *connection*, *krpc_UI_Canvas_t * result*)

Add a new canvas.

Game Scenes All

Note: If you want to add UI elements to KSPs stock UI canvas, use `krpc_UI_StockCanvas()`.

krpc_error_t **krpc_UI_Message** (krpc_connection_t *connection*, const char * *content*, float *duration*, *krpc_UI_MessagePosition_t* *position*, const krpc_tuple_double_double_double_t * *color*, float *size*)

Display a message on the screen.

Parameters

- **content** – Message content.
- **duration** – Duration before the message disappears, in seconds.
- **position** – Position to display the message.
- **color** – The color of the message.
- **size** – Size of the message, differs per position.

Game Scenes All

Note: The message appears just like a stock message, for example quicksave or quickload messages.

krpc_error_t **krpc_UI_Clear** (krpc_connection_t *connection*, bool *clientOnly*)

Remove all user interface elements.

Parameters

- **clientOnly** – If true, only remove objects created by the calling client.

Game Scenes All

krpc_UI_MessagePosition_t

Message position.

KRPC_UI_MESSAGEPOSITION_TOPLEFT

Top left.

KRPC_UI_MESSAGEPOSITION_TOPCENTER

Top center.

KRPC_UI_MESSAGEPOSITION_TOPRIGHT

Top right.

KRPC_UI_MESSAGEPOSITION_BOTTOMCENTER

Bottom center.

3.8.2 Canvas

krpc_UI_Canvas_t

A canvas for user interface elements. See `krpc_UI_StockCanvas()` and `krpc_UI_AddCanvas()`.

krpc_error_t **krpc_UI_Canvas_RectTransform** (krpc_connection_t *connection*, *krpc_UI_RectTransform_t* * *result*)

The rect transform for the canvas.

Game Scenes All

krpc_error_t **krpc_UI_Canvas_Visible** (krpc_connection_t *connection*, bool * *result*)

void **krpc_UI_Canvas_set_Visible** (bool *value*)
Whether the UI object is visible.

Game Scenes All

krpc_error_t **krpc_UI_Canvas_AddPanel** (krpc_connection_t *connection*, *krpc_UI_Panel_t* * *result*, bool *visible*)
Create a new container for user interface elements.

Parameters

- **visible** – Whether the panel is visible.

Game Scenes All

krpc_error_t **krpc_UI_Canvas_AddText** (krpc_connection_t *connection*, *krpc_UI_Text_t* * *result*, const char * *content*, bool *visible*)
Add text to the canvas.

Parameters

- **content** – The text.
- **visible** – Whether the text is visible.

Game Scenes All

krpc_error_t **krpc_UI_Canvas_AddInputField** (krpc_connection_t *connection*, *krpc_UI_InputField_t* * *result*, bool *visible*)
Add an input field to the canvas.

Parameters

- **visible** – Whether the input field is visible.

Game Scenes All

krpc_error_t **krpc_UI_Canvas_AddButton** (krpc_connection_t *connection*, *krpc_UI_Button_t* * *result*, const char * *content*, bool *visible*)
Add a button to the canvas.

Parameters

- **content** – The label for the button.
- **visible** – Whether the button is visible.

Game Scenes All

krpc_error_t **krpc_UI_Canvas_Remove** (krpc_connection_t *connection*)
Remove the UI object.

Game Scenes All

3.8.3 Panel

krpc_UI_Panel_t
A container for user interface elements. See `krpc_UI_Canvas_AddPanel()`.

krpc_error_t **krpc_UI_Panel_RectTransform** (krpc_connection_t *connection*, *krpc_UI_RectTransform_t* * *result*)
The rect transform for the panel.

Game Scenes All

krpc_error_t **krpc_UI_Panel_Visible** (krpc_connection_t *connection*, bool * *result*)

void **krpc_UI_Panel_set_Visible** (bool *value*)
Whether the UI object is visible.

Game Scenes All

krpc_error_t **krpc_UI_Panel_AddPanel** (krpc_connection_t *connection*, *krpc_UI_Panel_t* * *result*, bool *visible*)
Create a panel within this panel.

Parameters

- **visible** – Whether the new panel is visible.

Game Scenes All

krpc_error_t **krpc_UI_Panel_AddText** (krpc_connection_t *connection*, *krpc_UI_Text_t* * *result*, const char * *content*, bool *visible*)
Add text to the panel.

Parameters

- **content** – The text.
- **visible** – Whether the text is visible.

Game Scenes All

krpc_error_t **krpc_UI_Panel_AddInputField** (krpc_connection_t *connection*, *krpc_UI_InputField_t* * *result*, bool *visible*)
Add an input field to the panel.

Parameters

- **visible** – Whether the input field is visible.

Game Scenes All

krpc_error_t **krpc_UI_Panel_AddButton** (krpc_connection_t *connection*, *krpc_UI_Button_t* * *result*, const char * *content*, bool *visible*)
Add a button to the panel.

Parameters

- **content** – The label for the button.
- **visible** – Whether the button is visible.

Game Scenes All

krpc_error_t **krpc_UI_Panel_Remove** (krpc_connection_t *connection*)
Remove the UI object.

Game Scenes All

3.8.4 Text

krpc_UI_Text_t
A text label. See `krpc_UI_Panel_AddText()`.

krpc_error_t **krpc_UI_Text_RectTransform** (krpc_connection_t *connection*, *krpc_UI_RectTransform_t* * *result*)
The rect transform for the text.

Game Scenes All

krpc_error_t **krpc_UI_Text_Visible** (krpc_connection_t *connection*, bool * *result*)

void **krpc_UI_Text_set_Visible** (bool *value*)
Whether the UI object is visible.

Game Scenes All

krpc_error_t **krpc_UI_Text_Content** (krpc_connection_t *connection*, char * * *result*)

void **krpc_UI_Text_set_Content** (const char * *value*)
The text string

Game Scenes All

krpc_error_t **krpc_UI_Text_Font** (krpc_connection_t *connection*, char * * *result*)

void **krpc_UI_Text_set_Font** (const char * *value*)
Name of the font

Game Scenes All

krpc_error_t **krpc_UI_Text_AvailableFonts** (krpc_connection_t *connection*, krpc_list_string_t * *result*)
A list of all available fonts.

Game Scenes All

krpc_error_t **krpc_UI_Text_Size** (krpc_connection_t *connection*, int32_t * *result*)

void **krpc_UI_Text_set_Size** (int32_t *value*)
Font size.

Game Scenes All

krpc_error_t **krpc_UI_Text_Style** (krpc_connection_t *connection*, *krpc_UI_FontStyle_t* * *result*)

void **krpc_UI_Text_set_Style** (*krpc_UI_FontStyle_t value*)
Font style.

Game Scenes All

krpc_error_t **krpc_UI_Text_Color** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_UI_Text_set_Color** (const krpc_tuple_double_double_double_t * *value*)
Set the color

Game Scenes All

krpc_error_t **krpc_UI_Text_Alignment** (krpc_connection_t *connection*, *krpc_UI_TextAnchor_t* * *result*)

void **krpc_UI_Text_set_Alignment** (*krpc_UI_TextAnchor_t value*)
Alignment.

Game Scenes All

krpc_error_t **krpc_UI_Text_LineSpacing** (krpc_connection_t *connection*, float * *result*)

void **krpc_UI_Text_set_LineSpacing** (float *value*)
Line spacing.

Game Scenes All

krpc_error_t **krpc_UI_Text_Remove** (krpc_connection_t *connection*)
Remove the UI object.

Game Scenes All

krpc_UI_FontStyle_t
Font style.

KRPC_UI_FONTSTYLE_NORMAL
Normal.

KRPC_UI_FONTSTYLE_BOLD
Bold.

KRPC_UI_FONTSTYLE_ITALIC
Italic.

KRPC_UI_FONTSTYLE_BOLDANDITALIC
Bold and italic.

krpc_UI_TextAlignment_t
Text alignment.

KRPC_UI_TEXTALIGNMENT_LEFT
Left aligned.

KRPC_UI_TEXTALIGNMENT_RIGHT
Right aligned.

KRPC_UI_TEXTALIGNMENT_CENTER
Center aligned.

krpc_UI_TextAnchor_t
Text alignment.

KRPC_UI_TEXTANCHOR_LOWERCENTER
Lower center.

KRPC_UI_TEXTANCHOR_LOWERLEFT
Lower left.

KRPC_UI_TEXTANCHOR_LOWERRIGHT
Lower right.

KRPC_UI_TEXTANCHOR_MIDDLECENTER
Middle center.

KRPC_UI_TEXTANCHOR_MIDDLELEFT
Middle left.

KRPC_UI_TEXTANCHOR_MIDDLERIGHT
Middle right.

KRPC_UI_TEXTANCHOR_UPPERCENTER
Upper center.

KRPC_UI_TEXTANCHOR_UPPERLEFT
Upper left.

KRPC_UI_TEXTANCHOR_UPPERRIGHT
Upper right.

3.8.5 Button

krpc_UI_Button_t
A text label. See *krpc_UI_Panel_AddButton()*.

krpc_error_t **krpc_UI_Button_RectTransform** (krpc_connection_t *connection*, *krpc_UI_RectTransform_t * result*)
The rect transform for the text.

Game Scenes All

krpc_error_t **krpc_UI_Button_Visible** (krpc_connection_t *connection*, bool * *result*)

void **krpc_UI_Button_set_Visible** (bool *value*)
Whether the UI object is visible.

Game Scenes All

krpc_error_t **krpc_UI_Button_Text** (krpc_connection_t *connection*, *krpc_UI_Text_t * result*)
The text for the button.

Game Scenes All

krpc_error_t **krpc_UI_Button_Clicked** (krpc_connection_t *connection*, bool * *result*)

void **krpc_UI_Button_set_Clicked** (bool *value*)
Whether the button has been clicked.

Game Scenes All

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.

krpc_error_t **krpc_UI_Button_Remove** (krpc_connection_t *connection*)
Remove the UI object.

Game Scenes All

3.8.6 InputField

krpc_UI_InputField_t
An input field. See *krpc_UI_Panel_AddInputField()*.

krpc_error_t **krpc_UI_InputField_RectTransform** (krpc_connection_t *connection*, *krpc_UI_RectTransform_t * result*)
The rect transform for the input field.

Game Scenes All

krpc_error_t **krpc_UI_InputField_Visible** (krpc_connection_t *connection*, bool * *result*)

void **krpc_UI_InputField_set_Visible** (bool *value*)
Whether the UI object is visible.

Game Scenes All

krpc_error_t **krpc_UI_InputField_Value** (krpc_connection_t *connection*, char * * *result*)

void **krpc_UI_InputField_set_Value** (const char * *value*)
The value of the input field.

Game Scenes All

krpc_error_t **krpc_UI_InputField_Text** (krpc_connection_t *connection*, *krpc_UI_Text_t* * *result*)
The text component of the input field.

Game Scenes All

Note: Use *krpc_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.

krpc_error_t **krpc_UI_InputField_Changed** (krpc_connection_t *connection*, bool * *result*)

void **krpc_UI_InputField_set_Changed** (bool *value*)
Whether the input field has been changed.

Game Scenes All

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.

krpc_error_t **krpc_UI_InputField_Remove** (krpc_connection_t *connection*)
Remove the UI object.

Game Scenes All

3.8.7 Rect Transform

krpc_UI_RectTransform_t
A Unity engine Rect Transform for a UI object. See the Unity manual for more details.

krpc_error_t **krpc_UI_RectTransform_Position** (krpc_connection_t *connection*, krpc_tuple_double_double_t * *result*)

void **krpc_UI_RectTransform_set_Position** (const krpc_tuple_double_double_t * *value*)
Position of the rectangles pivot point relative to the anchors.

Game Scenes All

krpc_error_t **krpc_UI_RectTransform_LocalPosition** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*) void **krpc_UI_RectTransform_set_LocalPosition** (const krpc_tuple_double_double_double_t * *value*)

Position of the rectangles pivot point relative to the anchors.

Game Scenes All

krpc_error_t **krpc_UI_RectTransform_Size** (krpc_connection_t *connection*, krpc_tuple_double_double_t * *result*)

void **krpc_UI_RectTransform_set_Size** (const krpc_tuple_double_double_t * *value*)

Width and height of the rectangle.

Game Scenes All

krpc_error_t **krpc_UI_RectTransform_UpperRight** (krpc_connection_t *connection*, krpc_tuple_double_double_t * *result*)

void **krpc_UI_RectTransform_set_UpperRight** (const krpc_tuple_double_double_t * *value*)

Position of the rectangles upper right corner relative to the anchors.

Game Scenes All

krpc_error_t **krpc_UI_RectTransform_LowerLeft** (krpc_connection_t *connection*, krpc_tuple_double_double_t * *result*)

void **krpc_UI_RectTransform_set_LowerLeft** (const krpc_tuple_double_double_t * *value*)

Position of the rectangles lower left corner relative to the anchors.

Game Scenes All

void **krpc_UI_RectTransform_set_Anchor** (const krpc_tuple_double_double_t * *value*)

Set the minimum and maximum anchor points as a fraction of the size of the parent rectangle.

Game Scenes All

krpc_error_t **krpc_UI_RectTransform_AnchorMax** (krpc_connection_t *connection*, krpc_tuple_double_double_t * *result*)

void **krpc_UI_RectTransform_set_AnchorMax** (const krpc_tuple_double_double_t * *value*)

The anchor point for the lower left corner of the rectangle defined as a fraction of the size of the parent rectangle.

Game Scenes All

krpc_error_t **krpc_UI_RectTransform_AnchorMin** (krpc_connection_t *connection*, krpc_tuple_double_double_t * *result*)

void **krpc_UI_RectTransform_set_AnchorMin** (const krpc_tuple_double_double_t * *value*)

The anchor point for the upper right corner of the rectangle defined as a fraction of the size of the parent rectangle.

Game Scenes All

krpc_error_t **krpc_UI_RectTransform_Pivot** (krpc_connection_t *connection*, krpc_tuple_double_double_t * *result*)

void **krpc_UI_RectTransform_set_Pivot** (const krpc_tuple_double_double_t * *value*)

Location of the pivot point around which the rectangle rotates, defined as a fraction of the size of the rectangle itself.

Game Scenes All

krpc_error_t **krpc_UI_RectTransform_Rotation** (krpc_connection_t *connection*, krpc_tuple_double_double_double_double_t * *result*)

void **krpc_UI_RectTransform_set_Rotation** (const krpc_tuple_double_double_double_double_t * *value*)

Rotation, as a quaternion, of the object around its pivot point.

Game Scenes All

krpc_error_t **krpc_UI_RectTransform_Scale** (krpc_connection_t *connection*, krpc_tuple_double_double_double_t * *result*)

void **krpc_UI_RectTransform_set_Scale** (const krpc_tuple_double_double_double_t * *value*)

Scale factor applied to the object in the x, y and z dimensions.

Game Scenes All

CHAPTER

FOUR

C#

4.1 C# Client

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

4.1.1 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.

4.1.2 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.

4.1.3 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);
        }
    }
}
```

4.1.4 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.

4.1.5 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;
```

(continues on next page)

(continued from previous page)

```
        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.

4.1.6 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");
        }
    }
}
```

4.1.7 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 *streamPort = 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.

Single **Rate** { **get; set;** }
The update rate of the stream in Hertz. When set to zero, the rate is unlimited.

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).

Int32 **`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. Returns a unique identifier for the callback which can be used to remove it.

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 **`RemoveCallback`** (Int32 *tag*)
Removes a callback from the stream. The tag is the identifier returned when the callback was added.

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.

Int32 **`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. Returns a unique identifier for the callback which can be used to remove it.

void **`RemoveCallback`** (Int32 *tag*)
Removes a callback from the event. The tag is the identifier returned when the callback was added.

void **`Remove`** ()
Removes the event from the server.

Stream<Boolean> **`Stream`** { **`get;`** }
Returns the underlying stream for the event.

4.2 KRPC API

4.2.1 KRPC

None 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.

Game Scenes All

String **GetClientName** ()
Returns the name of the current client. This is an empty string if the client has no name.

Game Scenes All

IList<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.

Game Scenes All

KRPC.Schema.KRPC.Status **GetStatus** ()
Returns some information about the server, such as the version.

Game Scenes All

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.

Game Scenes All

GameScene **CurrentGameScene** { **get;** }
Get the current game scene.

Game Scenes All

Boolean **Paused** { **get; set;** }
Whether the game is paused.

Game Scenes All

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.

4.2.2 Expressions

class Expression
A server side expression.

static Expression **ConstantDouble** (*IConnection connection*, Double *value*)
A constant value of double precision floating point type.

Parameters

Game Scenes All

static Expression **ConstantFloat** (*IConnection connection*, Single *value*)
A constant value of single precision floating point type.

Parameters

Game Scenes All

static Expression **ConstantInt** (*IConnection connection*, Int32 *value*)
A constant value of integer type.

Parameters

Game Scenes All

static Expression **ConstantBool** (*IConnection connection*, Boolean *value*)
A constant value of boolean type.

Parameters

Game Scenes All

static Expression **ConstantString** (*IConnection connection*, String *value*)
A constant value of string type.

Parameters

Game Scenes All

static Expression **Call** (*IConnection connection*, KRPC.Schema.KRPC.ProcedureCall *call*)
An RPC call.

Parameters

Game Scenes All

static Expression **Equal** (*IConnection connection*, *Expression arg0*, *Expression arg1*)
Equality comparison.

Parameters

Game Scenes All

static Expression **NotEqual** (*IConnection connection*, *Expression arg0*, *Expression arg1*)
Inequality comparison.

Parameters

Game Scenes All

static Expression **GreaterThan** (*IConnection connection*, *Expression arg0*, *Expression arg1*)
Greater than numerical comparison.

Parameters

Game Scenes All

static Expression **GreaterThanOrEqual** (*IConnection connection*, *Expression arg0*, *Expression arg1*)
Greater than or equal numerical comparison.

Parameters

Game Scenes All

static Expression **LessThan** (*IConnection connection*, *Expression arg0*, *Expression arg1*)
Less than numerical comparison.

Parameters

Game Scenes All

static Expression **LessThanOrEqual** (*IConnection connection*, *Expression arg0*, *Expression arg1*)
Less than or equal numerical comparison.

Parameters

Game Scenes All

static Expression **And** (*IConnection connection*, *Expression arg0*, *Expression arg1*)
Boolean and operator.

Parameters

Game Scenes All

static Expression **Or** (*IConnection connection*, *Expression arg0*, *Expression arg1*)
Boolean or operator.

Parameters

Game Scenes All

static Expression **ExclusiveOr** (*IConnection connection*, *Expression arg0*, *Expression arg1*)
Boolean exclusive-or operator.

Parameters

Game Scenes All

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

Parameters

Game Scenes All

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

Parameters

Game Scenes All

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

Parameters

Game Scenes All

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

Parameters

Game Scenes All

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

Parameters

Game Scenes All

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

Parameters

Returns The remainder of arg0 divided by arg1

Game Scenes All

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

Parameters

Returns arg0 raised to the power of arg1, with type of arg0

Game Scenes All

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

Parameters

Game Scenes All

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

Parameters

Game Scenes All

static Expression **Cast** (*IConnection connection*, *Expression arg*, *Type type*)
Perform a cast to the given type.

Parameters

- **type** – Type to cast the argument to.

Game Scenes All

static Expression **Parameter** (*IConnection connection*, String *name*, *Type type*)
A named parameter of type double.

Parameters

- **name** – The name of the parameter.
- **type** – The type of the parameter.

Returns A named parameter.

Game Scenes All

static Expression **Function** (*IConnection connection*, IList<*Expression*> *parameters*, *Expression body*)

A function.

Parameters

- **parameters** – The parameters of the function.
- **body** – The body of the function.

Returns A function.

Game Scenes All

static Expression **Invoke** (*IConnection connection*, *Expression function*, IDictionary<String, *Expression*> *args*)

A function call.

Parameters

- **function** – The function to call.
- **args** – The arguments to call the function with.

Returns A function call.

Game Scenes All

static Expression **CreateTuple** (*IConnection connection*, IList<*Expression*> *elements*)

Construct a tuple.

Parameters

- **elements** – The elements.

Returns The tuple.

Game Scenes All

static Expression **CreateList** (*IConnection connection*, IList<*Expression*> *values*)

Construct a list.

Parameters

- **values** – The value. Should all be of the same type.

Returns The list.

Game Scenes All

static Expression **CreateSet** (*IConnection connection*, ISet<*Expression*> *values*)

Construct a set.

Parameters

- **values** – The values. Should all be of the same type.

Returns The set.

Game Scenes All

static Expression **CreateDictionary** (*IConnection connection*, IList<*Expression*> *keys*, IList<*Expression*> *values*)

Construct a dictionary, from a list of corresponding keys and values.

Parameters

- **keys** – The keys. Should all be of the same type.
- **values** – The values. Should all be of the same type.

Returns The dictionary.

Game Scenes All

static Expression **ToList** (*IConnection connection*, *Expression arg*)

Convert a collection to a list.

Parameters

- **arg** – The collection.

Returns The collection as a list.

Game Scenes All

static Expression **ToSet** (*IConnection connection*, *Expression arg*)

Convert a collection to a set.

Parameters

- **arg** – The collection.

Returns The collection as a set.

Game Scenes All

static Expression **Get** (*IConnection connection*, *Expression arg*, *Expression index*)

Access an element in a tuple, list or dictionary.

Parameters

- **arg** – The tuple, list or dictionary.
- **index** – The index of the element to access. A zero indexed integer for a tuple or list, or a key for a dictionary.

Returns The element.

Game Scenes All

static Expression **Count** (*IConnection connection*, *Expression arg*)

Number of elements in a collection.

Parameters

- **arg** – The list, set or dictionary.

Returns The number of elements in the collection.

Game Scenes All

static Expression **Sum** (*IConnection connection*, *Expression arg*)

Sum all elements of a collection.

Parameters

- **arg** – The list or set.

Returns The sum of the elements in the collection.

Game Scenes All

static Expression **Max** (*IConnection connection*, *Expression arg*)
Maximum of all elements in a collection.

Parameters

- **arg** – The list or set.

Returns The maximum elements in the collection.

Game Scenes All

static Expression **Min** (*IConnection connection*, *Expression arg*)
Minimum of all elements in a collection.

Parameters

- **arg** – The list or set.

Returns The minimum elements in the collection.

Game Scenes All

static Expression **Average** (*IConnection connection*, *Expression arg*)
Minimum of all elements in a collection.

Parameters

- **arg** – The list or set.

Returns The minimum elements in the collection.

Game Scenes All

static Expression **Select** (*IConnection connection*, *Expression arg*, *Expression func*)
Run a function on every element in the collection.

Parameters

- **arg** – The list or set.
- **func** – The function.

Returns The modified collection.

Game Scenes All

static Expression **Where** (*IConnection connection*, *Expression arg*, *Expression func*)
Run a function on every element in the collection.

Parameters

- **arg** – The list or set.
- **func** – The function.

Returns The modified collection.

Game Scenes All

static Expression **Contains** (*IConnection connection*, *Expression arg*, *Expression value*)
Determine if a collection contains a value.

Parameters

- **arg** – The collection.
- **value** – The value to look for.

Returns Whether the collection contains a value.

Game Scenes All

static Expression **Aggregate** (*IConnection connection*, *Expression arg*, *Expression func*)

Applies an accumulator function over a sequence.

Parameters

- **arg** – The collection.
- **func** – The accumulator function.

Returns The accumulated value.

Game Scenes All

static Expression **AggregateWithSeed** (*IConnection connection*, *Expression arg*, *Expression seed*, *Expression func*)

Applies an accumulator function over a sequence, with a given seed.

Parameters

- **arg** – The collection.
- **seed** – The seed value.
- **func** – The accumulator function.

Returns The accumulated value.

Game Scenes All

static Expression **Concat** (*IConnection connection*, *Expression arg1*, *Expression arg2*)

Concatenate two sequences.

Parameters

- **arg1** – The first sequence.
- **arg2** – The second sequence.

Returns The first sequence followed by the second sequence.

Game Scenes All

static Expression **OrderBy** (*IConnection connection*, *Expression arg*, *Expression key*)

Order a collection using a key function.

Parameters

- **arg** – The collection to order.
- **key** – A function that takes a value from the collection and generates a key to sort on.

Returns The ordered collection.

Game Scenes All

static Expression **All** (*IConnection connection*, *Expression arg*, *Expression predicate*)

Determine whether all items in a collection satisfy a boolean predicate.

Parameters

- **arg** – The collection.
- **predicate** – The predicate function.

Returns Whether all items satisfy the predicate.

Game Scenes All

static Expression **Any** (*IConnection connection*, *Expression arg*, *Expression predicate*)
Determine whether any item in a collection satisfies a boolean predicate.

Parameters

- **arg** – The collection.
- **predicate** – The predicate function.

Returns Whether any item satisfies the predicate.

Game Scenes All

class Type
A server side expression.

static Type **Double** (*IConnection connection*)
Double type.

Game Scenes All

static Type **Float** (*IConnection connection*)
Float type.

Game Scenes All

static Type **Int** (*IConnection connection*)
Int type.

Game Scenes All

static Type **Bool** (*IConnection connection*)
Bool type.

Game Scenes All

static Type **String** (*IConnection connection*)
String type.

Game Scenes All

4.3 SpaceCenter API

4.3.1 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.

Single **Science** { **get;** }
The current amount of science.

Game Scenes All

Double **Funds** { **get;** }
The current amount of funds.

Game Scenes All

Single **Reputation** { **get;** }
The current amount of reputation.

Game Scenes All

Vessel **ActiveVessel** { **get; set;** }
The currently active vessel.

Game Scenes Flight

IList<*Vessel*> **Vessels** { **get;** }
A list of all the vessels in the game.

Game Scenes All

IDictionary<String, *CelestialBody*> **Bodies** { **get;** }
A dictionary of all celestial bodies (planets, moons, etc.) in the game, keyed by the name of the body.

Game Scenes All

CelestialBody **TargetBody** { **get; set;** }
The currently targeted celestial body.

Game Scenes Flight

Vessel **TargetVessel** { **get; set;** }
The currently targeted vessel.

Game Scenes Flight

DockingPort **TargetDockingPort** { **get; set;** }
The currently targeted docking port.

Game Scenes Flight

void **ClearTarget** ()
Clears the current target.

Game Scenes Flight

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"`.

Game Scenes All

void **LaunchVessel** (String *craftDirectory*, String *name*, String *launchSite*, Boolean *recover = true*)
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"`.
- **recover** – If true and there is a vessel on the launch site, recover it before launching.

Game Scenes All

Note: Throws an exception if any of the games pre-flight checks fail.

void **LaunchVesselFromVAB** (String *name*, Boolean *recover = true*)
Launch a new vessel from the VAB onto the launchpad.

Parameters

- **name** – Name of the vessel to launch.
- **recover** – If true and there is a vessel on the launch pad, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *SpaceCenter.LaunchVessel* with the craft directory set to "VAB" and the launch site set to "LaunchPad". Throws an exception if any of the games pre-flight checks fail.

void **LaunchVesselFromSPH** (String *name*, Boolean *recover = true*)
Launch a new vessel from the SPH onto the runway.

Parameters

- **name** – Name of the vessel to launch.
- **recover** – If true and there is a vessel on the runway, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *SpaceCenter.LaunchVessel* with the craft directory set to "SPH" and the launch site set to "Runway". Throws an exception if any of the games pre-flight checks fail.

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

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

Game Scenes All

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

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes All

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

Game Scenes All

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.

Game Scenes Flight

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*.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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

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.

Game Scenes Flight

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 covert the position to.

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

Game Scenes All

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 covert the direction to.

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

Game Scenes All

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 covert the rotation to.

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

Game Scenes All

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 covert the velocity to.

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

Game Scenes All

Double **`RaycastDistance`** (Tuple<Double, Double, Double> *position*, Tuple<Double, Double, Double> *direction*, *ReferenceFrame referenceFrame*)

Cast a ray from a given position in a given direction, and return the distance to the hit point. If no hit occurs, returns infinity.

Parameters

- **position** – Position, as a vector, of the origin of the ray.
- **direction** – Direction of the ray, as a unit vector.
- **referenceFrame** – The reference frame that the position and direction are in.

Returns The distance to the hit, in meters, or infinity if there was no hit.

Game Scenes All

Part **`RaycastPart`** (Tuple<Double, Double, Double> *position*, Tuple<Double, Double, Double> *direction*, *ReferenceFrame referenceFrame*)

Cast a ray from a given position in a given direction, and return the part that it hits. If no hit occurs, returns `null`.

Parameters

- **position** – Position, as a vector, of the origin of the ray.
- **direction** – Direction of the ray, as a unit vector.
- **referenceFrame** – The reference frame that the position and direction are in.

Returns The part that was hit or `null` if there was no hit.

Game Scenes Flight

Boolean **`FARAvailable`** { **`get;`** }

Whether Ferram Aerospace Research is installed.

Game Scenes All

GameMode **GameMode** { **get;** }
The current mode the game is in.

Game Scenes All

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.

Game Scenes Flight

Camera **Camera** { **get;** }
An object that can be used to control the camera.

Game Scenes Flight

WaypointManager **WaypointManager** { **get;** }
The waypoint manager.

Game Scenes Flight

ContractManager **ContractManager** { **get;** }
The contract manager.

Game Scenes All

enum GameMode
The game mode. Returned by *GameMode*

Sandbox
Sandbox mode.

Career
Career mode.

Science
Science career mode.

ScienceSandbox
Science sandbox mode.

Mission
Mission mode.

MissionBuilder
Mission builder mode.

Scenario
Scenario mode.

ScenarioNonResumable
Scenario mode that cannot be resumed.

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.

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 *SpaceCenter.ActiveVessel* or *SpaceCenter.Vessels*.

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

The name of the vessel.

Game Scenes All

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

The type of the vessel.

Game Scenes All

VesselSituation **Situation** { **get;** }

The situation the vessel is in.

Game Scenes All

Boolean **Recoverable** { **get;** }

Whether the vessel is recoverable.

Game Scenes All

void **Recover** ()

Recover the vessel.

Game Scenes All

Double **MET** { **get;** }

The mission elapsed time in seconds.

Game Scenes All

String **Biome** { **get;** }

The name of the biome the vessel is currently in.

Game Scenes All

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*).

Game Scenes 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 **Orbit** { **get;** }

The current orbit of the vessel.

Game Scenes All

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.

Game Scenes Flight

Comms **Comms** { **get;** }
Returns a `Comms` object that can be used to interact with CommNet for this vessel.

Game Scenes Flight

AutoPilot **AutoPilot** { **get;** }
An `AutoPilot` object, that can be used to perform simple auto-piloting of the vessel.

Game Scenes Flight

Int32 **CrewCapacity** { **get;** }
The number of crew that can occupy the vessel.

Game Scenes All

Int32 **CrewCount** { **get;** }
The number of crew that are occupying the vessel.

Game Scenes All

IList<*CrewMember*> **Crew** { **get;** }
The crew in the vessel.

Game Scenes All

Resources **Resources** { **get;** }
A `Resources` object, that can used to get information about resources stored in the vessel.

Game Scenes Flight

Resources **ResourcesInDecoupleStage** (Int32 *stage*, Boolean *cumulative* = *true*)
Returns a `Resources` object, that can 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.

Game Scenes Flight

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.

Game Scenes Flight

Single **Mass** { **get;** }
The total mass of the vessel, including resources, in kg.

Game Scenes Flight

Single **DryMass** { **get;** }
The total mass of the vessel, excluding resources, in kg.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Single **SpecificImpulse** { **get;** }

The combined specific impulse of all active engines, in seconds. This is computed using the formula described here.

Game Scenes Flight

Single **VacuumSpecificImpulse** { **get;** }

The combined vacuum specific impulse of all active engines, in seconds. This is computed using the formula described here.

Game Scenes Flight

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.

Game Scenes Flight

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*).

Game Scenes Flight

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.

Game Scenes All

Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> **AvailableTorque** { **get;** }

The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gimballed 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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> **AvailableEngineTorque** { **get;** }
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 (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

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.

Game Scenes Flight

Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> **AvailableOtherTorque** { **get;** }
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 (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

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.

Game Scenes Flight

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.

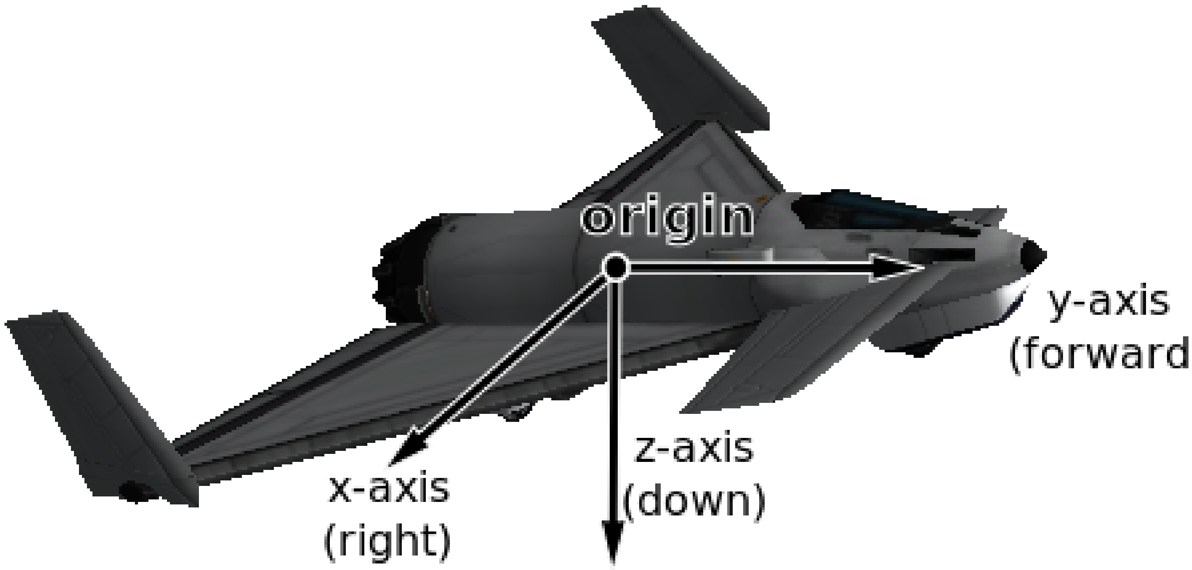


Fig. 1: Vessel reference frame origin and axes for the Aeris 3A aircraft

- The z-axis points in the orbital normal direction.

  Game Scenes Flight

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.
- 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).

  Game Scenes Flight

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.

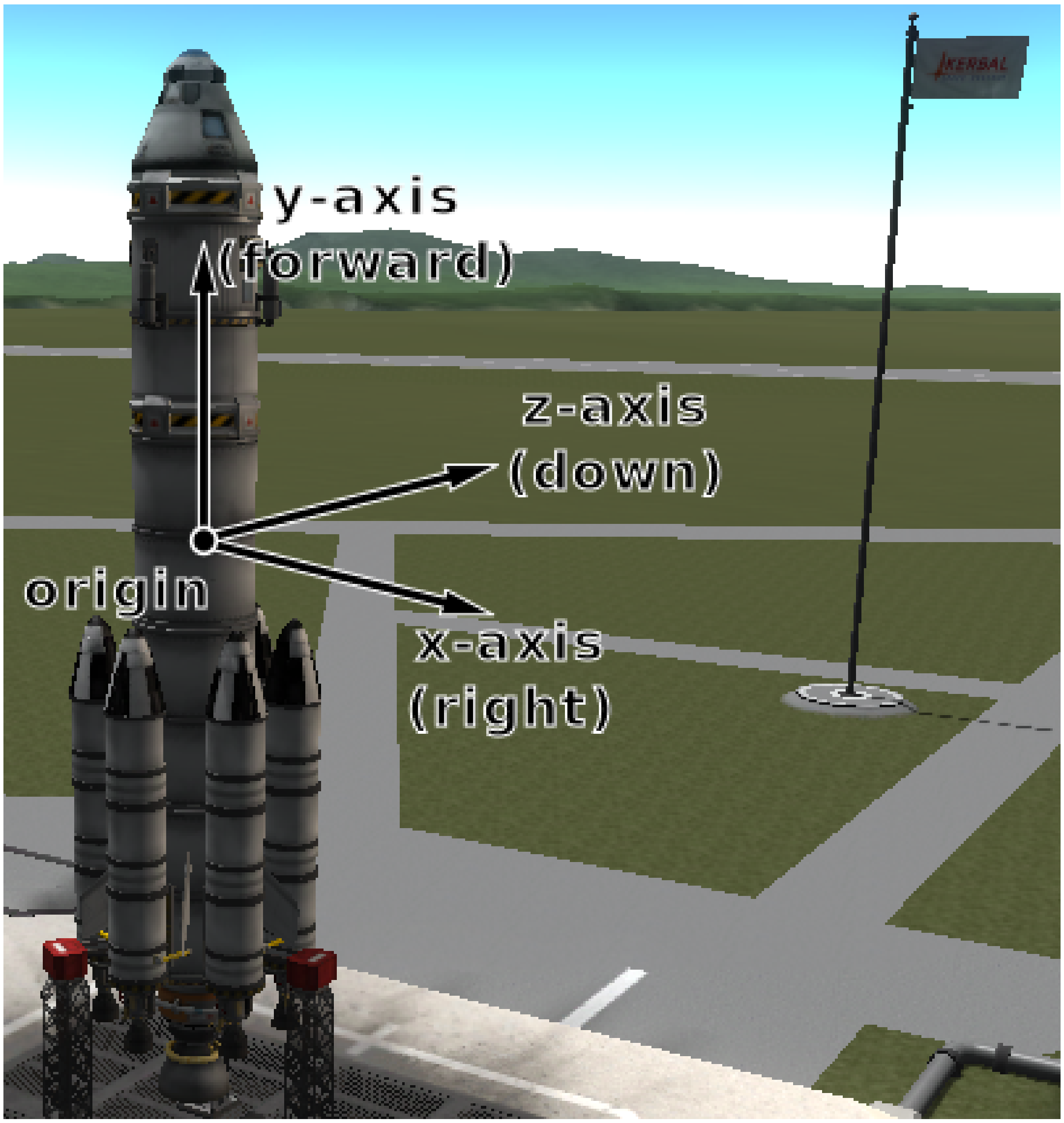


Fig. 2: Vessel reference frame origin and axes for the Kerbal-X rocket

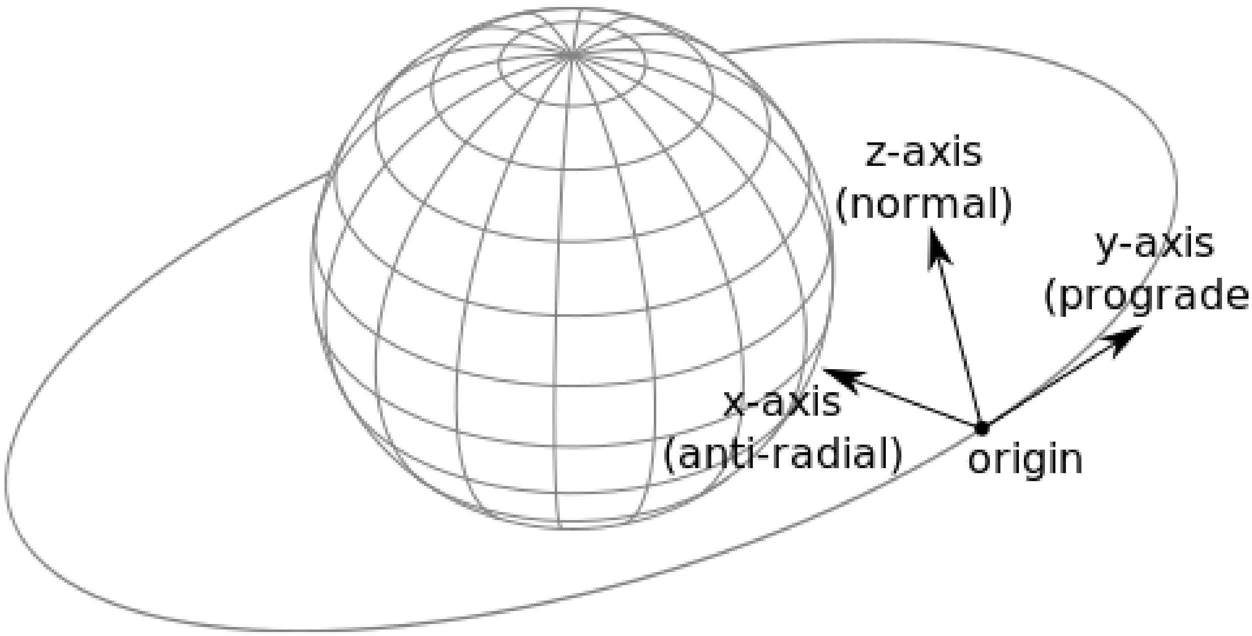


Fig. 3: Vessel orbital reference frame origin and axes

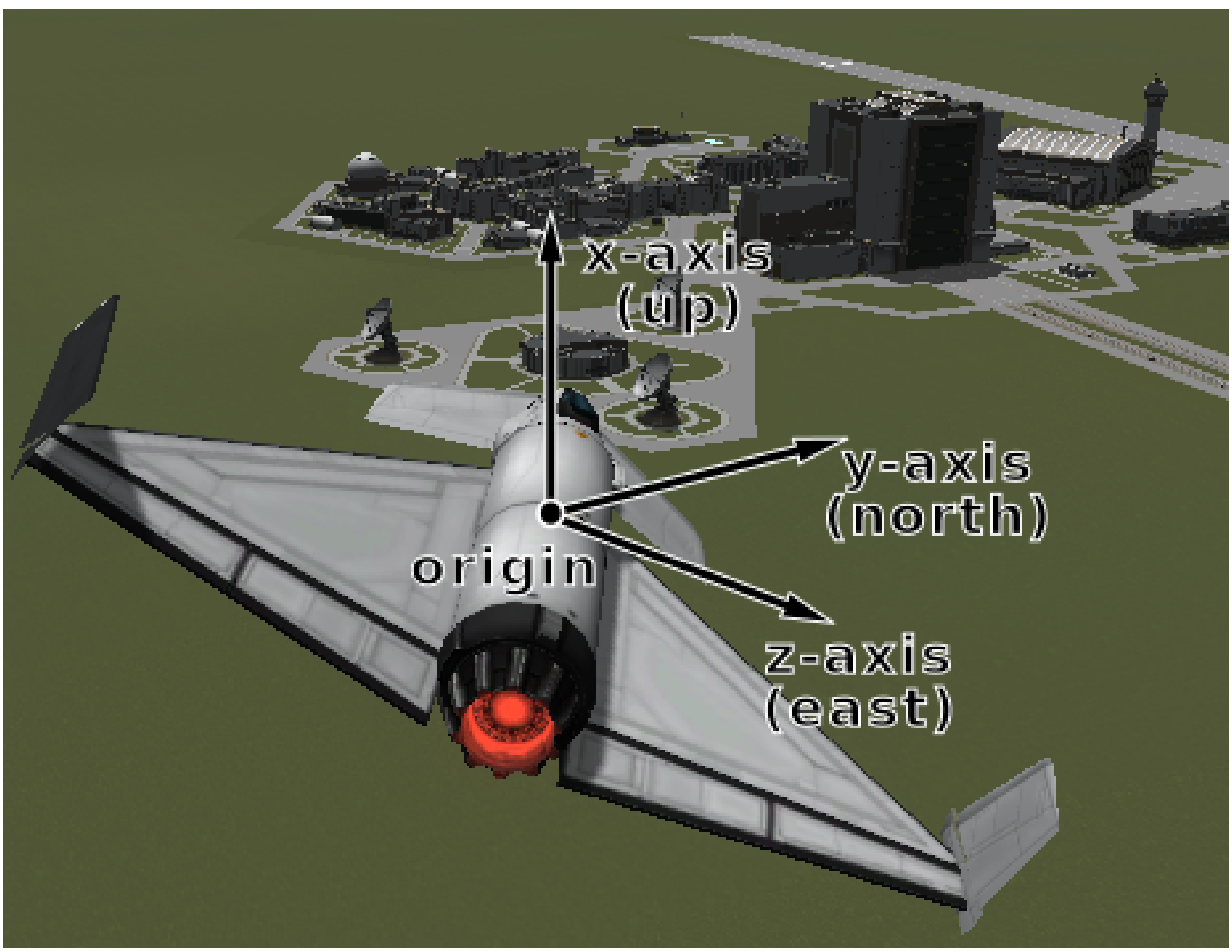


Fig. 4: Vessel surface reference frame origin and axes

- 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.

Game Scenes Flight

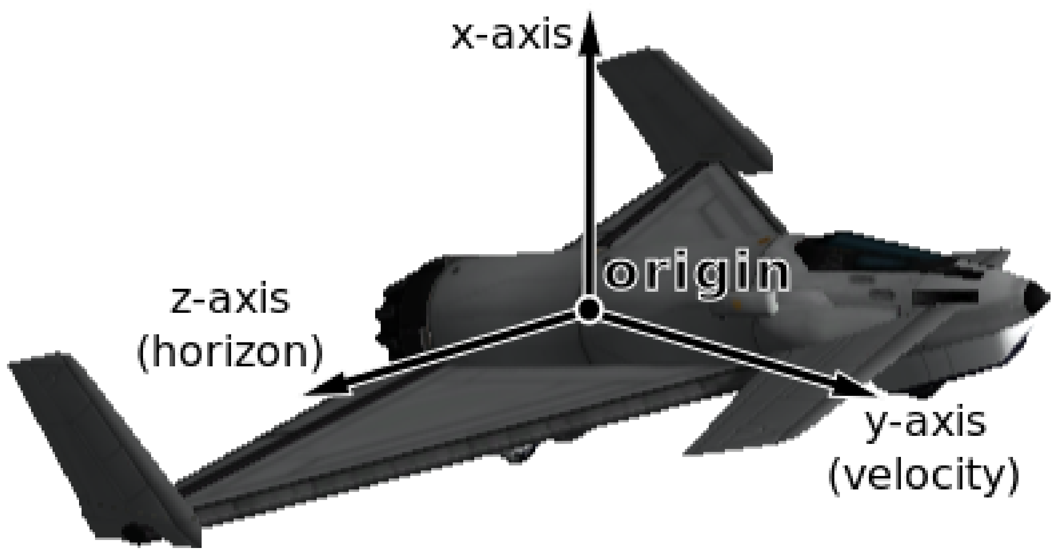


Fig. 5: Vessel surface velocity reference frame origin and axes

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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).

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

class CrewMember
Represents crew in a vessel. Can be obtained using *Vessel.Crew*.

String **Name** { **get; set;** }
The crew members name.

Game Scenes All

CrewMemberType **Type** { **get;** }
The type of crew member.

Game Scenes All

Boolean **OnMission** { **get;** }
Whether the crew member is on a mission.

Game Scenes All

Single **Courage** { **get; set;** }
The crew members courage.

Game Scenes All

Single **Stupidity** { **get; set;** }
The crew members stupidity.

Game Scenes All

Single **Experience** { **get; set;** }
The crew members experience.

Game Scenes All

Boolean **Badass** { **get; set;** }
Whether the crew member is a badass.

Game Scenes All

Boolean **Veteran** { **get; set;** }
Whether the crew member is a veteran.

Game Scenes All

enum CrewMemberType
The type of a crew member. See *CrewMember.Type*.

Applicant
An applicant for crew.

Crew
Rocket crew.

Tourist
A tourist.

Unowned
An unowned crew member.

4.3.3 CelestialBody

class CelestialBody
Represents a celestial body (such as a planet or moon). See *SpaceCenter.Bodies*.

String **Name** { **get;** }
The name of the body.

Game Scenes All

IList<*CelestialBody*> **Satellites** { **get;** }
A list of celestial bodies that are in orbit around this celestial body.

Game Scenes All

Orbit **Orbit** { **get;** }
The orbit of the body.

Game Scenes All

Single **Mass** { **get;** }
The mass of the body, in kilograms.

Game Scenes All

Single **GravitationalParameter** { **get;** }
The standard gravitational parameter of the body in m^3s^{-2}.

Game Scenes All

Single **SurfaceGravity** { **get;** }
The acceleration due to gravity at sea level (mean altitude) on the body, in m/s^2.

Game Scenes All

Single **RotationalPeriod** { **get;** }
The sidereal rotational period of the body, in seconds.

Game Scenes All

Single **RotationalSpeed** { **get;** }
The rotational speed of the body, in radians per second.

Game Scenes All

Double **RotationAngle** { **get;** }

The current rotation angle of the body, in radians. A value between 0 and 2π

Game Scenes All

Double **InitialRotation** { **get;** }

The initial rotation angle of the body (at UT 0), in radians. A value between 0 and 2π

Game Scenes All

Single **EquatorialRadius** { **get;** }

The equatorial radius of the body, in meters.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Single **SphereOfInfluence** { **get;** }

The radius of the sphere of influence of the body, in meters.

Game Scenes All

Boolean **HasAtmosphere** { **get;** }

`true` if the body has an atmosphere.

Game Scenes All

Single **AtmosphereDepth** { **get;** }

The depth of the atmosphere, in meters.

Game Scenes All

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.

Game Scenes All

Boolean **HasAtmosphericOxygen** { **get;** }

`true` if there is oxygen in the atmosphere, required for air-breathing engines.

Game Scenes All

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.

Game Scenes All

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

Game Scenes All

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

Game Scenes All

ISet<String> **Biomes** { **get;** }
The biomes present on this body.

Game Scenes All

String **BiomeAt** (Double *latitude*, Double *longitude*)
The biome at the given latitude and longitude, in degrees.

Parameters

Game Scenes All

Single **FlyingHighAltitudeThreshold** { **get;** }
The altitude, in meters, above which a vessel is considered to be flying "high" when doing science.

Game Scenes All

Single **SpaceHighAltitudeThreshold** { **get;** }
The altitude, in meters, above which a vessel is considered to be in "high" space when doing science.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

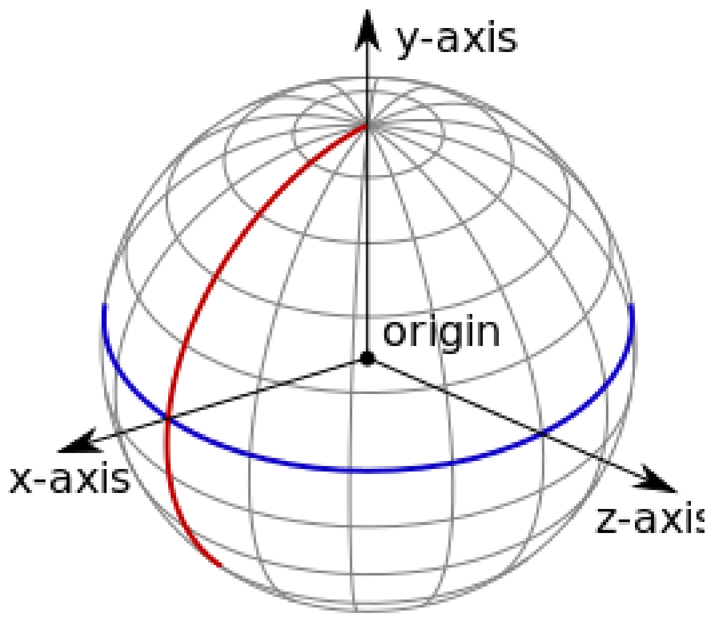


Fig. 6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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).

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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*.

Single **GForce** { **get;** }
The current G force acting on the vessel in g.

Game Scenes Flight

Double **MeanAltitude** { **get;** }
The altitude above sea level, in meters. Measured from the center of mass of the vessel.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Double **Latitude** { **get;** }

The latitude of the vessel for the body being orbited, in degrees.

Game Scenes Flight

Double **Longitude** { **get;** }

The longitude of the vessel for the body being orbited, in degrees.

Game Scenes Flight

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.

Game Scenes Flight

Double **Speed** { **get;** }

The speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Game Scenes Flight

Double **HorizontalSpeed** { **get;** }

The horizontal speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Game Scenes Flight

Double **VerticalSpeed** { **get;** }

The vertical speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Game Scenes Flight

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.

Game Scenes Flight

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).

Game Scenes Flight

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.

Game Scenes Flight

Single **Pitch** { **get;** }

The pitch of the vessel relative to the horizon, in degrees. A value between -90° and +90°.

Game Scenes Flight

Single **Heading** { **get;** }
The heading of the vessel (its angle relative to north), in degrees. A value between 0° and 360°.

Game Scenes Flight

Single **Roll** { **get;** }
The roll of the vessel relative to the horizon, in degrees. A value between -180° and +180°.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Single **AtmosphereDensity** { **get;** }
The current density of the atmosphere around the vessel, in kg/m^3.

Game Scenes Flight

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 $\frac{1}{2}$.air density.velocity^2. It is commonly denoted Q.

Game Scenes Flight

Single **StaticPressure** { **get;** }
The static atmospheric pressure acting on the vessel, in Pascals.

Game Scenes Flight

Single **StaticPressureAtMSL** { **get;** }
The static atmospheric pressure at mean sea level, in Pascals.

Game Scenes Flight

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.

Game Scenes Flight

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 where 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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Single **SpeedOfSound** { **get;** }
The speed of sound, in the atmosphere around the vessel, in m/s.

Game Scenes Flight

Single **Mach** { **get;** }
The speed of the vessel, in multiples of the speed of sound.

Game Scenes Flight

Single **ReynoldsNumber** { **get;** }
The vessels Reynolds number.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

Single **TrueAirSpeed** { **get;** }
The true air speed of the vessel, in meters per second.

Game Scenes Flight

Single **EquivalentAirSpeed** { **get;** }
The equivalent air speed of the vessel, in meters per second.

Game Scenes Flight

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.

Game Scenes Flight

Single **AngleOfAttack** { **get;** }
The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

Game Scenes Flight

Single **SideslipAngle** { **get;** }
The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

Game Scenes Flight

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.

Game Scenes Flight

Single **StaticAirTemperature** { **get;** }
The static (ambient) temperature of the atmosphere around the vessel, in Kelvin.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

Single **BallisticCoefficient** { **get;** }
The ballistic coefficient.

Game Scenes Flight

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.

Game Scenes Flight

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** { **get;** }
The celestial body (e.g. planet or moon) around which the object is orbiting.

Game Scenes All

Double **Apoapsis** { **get;** }
Gets the apoapsis of the orbit, in meters, from the center of mass of the body being orbited.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Double **SemiMinorAxis** { **get;** }
The semi-minor axis of the orbit, in meters.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Double **Speed** { **get;** }
The current orbital speed of the object in meters per second.

Game Scenes All

Note: This value will change over time if the orbit is elliptical.

Double **Period** { **get;** }
The orbital period, in seconds.

Game Scenes All

Double **TimeToApoapsis** { **get;** }
The time until the object reaches apoapsis, in seconds.

Game Scenes All

Double **TimeToPeriapsis** { **get;** }
The time until the object reaches periapsis, in seconds.

Game Scenes All

Double **Eccentricity** { **get;** }
The eccentricity of the orbit.

Game Scenes All

Double **Inclination** { **get;** }
The inclination of the orbit, in radians.

Game Scenes All

Double **LongitudeOfAscendingNode** { **get;** }
The longitude of the ascending node, in radians.

Game Scenes All

Double **ArgumentOfPeriapsis** { **get;** }
The argument of periapsis, in radians.

Game Scenes All

Double **MeanAnomalyAtEpoch** { **get;** }
The mean anomaly at epoch.

Game Scenes All

Double **Epoch** { **get;** }
The time since the epoch (the point at which the mean anomaly at epoch was measured, in seconds.

Game Scenes All

Double **MeanAnomaly** { **get;** }
The mean anomaly.

Game Scenes All

Double **MeanAnomalyAtUT** (Double *ut*)
The mean anomaly at the given time.

Parameters

- **ut** – The universal time in seconds.

Game Scenes All

Double **EccentricAnomaly** { **get;** }
The eccentric anomaly.

Game Scenes All

Double **EccentricAnomalyAtUT** (Double *ut*)
The eccentric anomaly at the given universal time.

Parameters

- **ut** – The universal time, in seconds.

Game Scenes All

Double **TrueAnomaly** { **get;** }
The true anomaly.

Game Scenes All

Double **TrueAnomalyAtUT** (Double *ut*)
The true anomaly at the given time.

Parameters

- **ut** – The universal time in seconds.

Game Scenes All

Double **TrueAnomalyAtRadius** (Double *radius*)
The true anomaly at the given orbital radius.

Parameters

- **radius** – The orbital radius in meters.

Game Scenes All

Double **UTAtTrueAnomaly** (Double *trueAnomaly*)
The universal time, in seconds, corresponding to the given true anomaly.

Parameters

- **trueAnomaly** – True anomaly.

Game Scenes All

Double **RadiusAtTrueAnomaly** (Double *trueAnomaly*)
The orbital radius at the point in the orbit given by the true anomaly.

Parameters

- **trueAnomaly** – The true anomaly.

Game Scenes All

Double **TrueAnomalyAtAN** (*Orbit target*)
The true anomaly of the ascending node with the given target orbit.

Parameters

- **target** – Target orbit.

Game Scenes All

Double **TrueAnomalyAtDN** (*Orbit target*)
The true anomaly of the descending node with the given target orbit.

Parameters

- **target** – Target orbit.

Game Scenes All

Double **OrbitalSpeed** { **get;** }
The current orbital speed in meters per second.

Game Scenes All

Double **OrbitalSpeedAt** (Double *time*)
The orbital speed at the given time, in meters per second.

Parameters

- **time** – Time from now, in seconds.

Game Scenes All

static Tuple<Double, Double, Double> **ReferencePlaneNormal** (*IConnection 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.

Game Scenes All

static Tuple<Double, Double, Double> **ReferencePlaneDirection** (*IConnection 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.

Game Scenes All

Double **RelativeInclination** (*Orbit target*)

Relative inclination of this orbit and the target orbit, in radians.

Parameters

- **target** – Target orbit.

Game Scenes All

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.

Game Scenes All

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`.

Game Scenes All

Double **TimeOfClosestApproach** (*Orbit target*)

Estimates and returns the time at closest approach to a target orbit.

Parameters

- **target** – Target orbit.

Returns The universal time at closest approach, in seconds.

Game Scenes All

Double **DistanceAtClosestApproach** (*Orbit target*)

Estimates and returns the distance at closest approach to a target orbit, in meters.

Parameters

- **target** – Target orbit.

Game Scenes All

IList<IList<Double>> **ListClosestApproaches** (*Orbit target*, Int32 *orbits*)
Returns the times at closest approach and corresponding distances, to a target orbit.

Parameters

- **target** – Target orbit.
- **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.

Game Scenes All

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** { **get;** }
The source of the vessels control, for example by a kerbal or a probe core.

Game Scenes Flight

ControlState **State** { **get;** }
The control state of the vessel.

Game Scenes Flight

Boolean **SAS** { **get; set;** }
The state of SAS.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Boolean **RCS** { **get; set;** }
The state of RCS.

Game Scenes Flight

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*.

Game Scenes Flight

Boolean **Gear** { **get; set;** }
The state of the landing gear/legs.

Game Scenes Flight

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*.

Game Scenes Flight

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*.

Game Scenes Flight

Boolean **Lights** { **get; set;** }
The state of the lights.

Game Scenes Flight

Boolean **Brakes** { **get; set;** }
The state of the wheel brakes.

Game Scenes Flight

Boolean **Antennas** { **get; set;** }
Returns whether all antennas on the vessel are deployed, and sets the deployment state of all antennas. See *Antenna.Deployed*.

Game Scenes Flight

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*.

Game Scenes Flight

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*.

Game Scenes Flight

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*.

Game Scenes Flight

Boolean **Radiators** { **get; set;** }
Returns whether all radiators on the vessel are deployed, and sets the deployment state of all radiators. See *Radiator.Deployed*.

Game Scenes Flight

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*.

Game Scenes Flight

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*.

Game Scenes Flight

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*.

Game Scenes Flight

Boolean **Abort** { **get; set;** }
The state of the abort action group.

Game Scenes Flight

Single **Throttle** { **get; set;** }
The state of the throttle. A value between 0 and 1.

Game Scenes Flight

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.

Game Scenes Flight

Single **Pitch** { **get; set;** }
The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

Game Scenes Flight

Single **Yaw** { **get; set;** }
The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

Game Scenes Flight

Single **Roll** { **get; set;** }
The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

Game Scenes Flight

Single **Forward** { **get; set;** }
The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

Game Scenes Flight

Single **Up** { **get; set;** }
The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

Game Scenes Flight

Single **Right** { **get; set;** }
The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Int32 **CurrentStage** { **get;** }
The current stage of the vessel. Corresponds to the stage number in the in-game UI.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Node **AddNode** (Double *ut*, Single *prograde = 0.0f*, Single *normal = 0.0f*, Single *radial = 0.0f*)
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.

Game Scenes Flight

IList<*Node*> **`Nodes`** { **`get;`** }

Returns a list of all existing maneuver nodes, ordered by time from first to last.

Game Scenes Flight

void **`RemoveNodes`** ()

Remove all maneuver nodes.

Game Scenes Flight

`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.

4.3.7 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.

Game Scenes Flight

Boolean **CanTransmitScience** { **get;** }
Whether the vessel can transmit science data to KSC.

Game Scenes Flight

Double **SignalStrength** { **get;** }
Signal strength to KSC.

Game Scenes Flight

Double **SignalDelay** { **get;** }
Signal delay to KSC in seconds.

Game Scenes Flight

Double **Power** { **get;** }
The combined power of all active antennae on the vessel.

Game Scenes Flight

IList<*CommLink*> **ControlPath** { **get;** }

The communication path used to control the vessel.

Game Scenes Flight

class CommLink

Represents a communication node in the network. For example, a vessel or the KSC.

CommLinkType **Type** { **get;** }

The type of link.

Game Scenes All

Double **SignalStrength** { **get;** }

Signal strength of the link.

Game Scenes All

CommNode **Start** { **get;** }

Start point of the link.

Game Scenes All

CommNode **End** { **get;** }

Start point of the link.

Game Scenes All

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.

Game Scenes All

Boolean **IsHome** { **get;** }

Whether the communication node is on Kerbin.

Game Scenes All

Boolean **IsControlPoint** { **get;** }

Whether the communication node is a control point, for example a manned vessel.

Game Scenes All

Boolean **IsVessel** { **get;** }

Whether the communication node is a vessel.

Game Scenes All

Vessel **Vessel** { **get;** }

The vessel for this communication node.

Game Scenes All

4.3.8 Parts

The following classes allow interaction with a vessels individual parts.

- *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.

Game Scenes All

Part **Root** { **get;** }

The vessels root part.

Game Scenes All

Note: See the discussion on *Trees of Parts*.

Part **Controlling** { **get; set;** }

The part from which the vessel is controlled.

Game Scenes All

IList<*Part*> **WithName** (String *name*)

A list of parts whose `Part.Name` is *name*.

Parameters

Game Scenes All

IList<*Part*> **WithTitle** (String *title*)

A list of all parts whose `Part.Title` is *title*.

Parameters

Game Scenes All

IList<*Part*> **WithTag** (String *tag*)

A list of all parts whose `Part.Tag` is *tag*.

Parameters

Game Scenes All

IList<*Part*> **WithModule** (String *moduleName*)

A list of all parts that contain a `Module` whose `Module.Name` is *moduleName*.

Parameters

Game Scenes All

IList<*Part*> **InStage** (Int32 *stage*)

A list of all parts that are activated in the given *stage*.

Parameters

Game Scenes All

Note: See the discussion on *Staging*.

IList<*Part*> **InDecoupleStage** (Int32 *stage*)
A list of all parts that are decoupled in the given *stage*.

Parameters

Game Scenes All

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

Game Scenes All

IList<*Antenna*> **Antennas** { **get;** }
A list of all antennas in the vessel.

Game Scenes All

IList<*CargoBay*> **CargoBays** { **get;** }
A list of all cargo bays in the vessel.

Game Scenes All

IList<*ControlSurface*> **ControlSurfaces** { **get;** }
A list of all control surfaces in the vessel.

Game Scenes All

IList<*Decoupler*> **Decouplers** { **get;** }
A list of all decouplers in the vessel.

Game Scenes All

IList<*DockingPort*> **DockingPorts** { **get;** }
A list of all docking ports in the vessel.

Game Scenes All

IList<*Engine*> **Engines** { **get;** }
A list of all engines in the vessel.

Game Scenes All

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.

Game Scenes All

IList<*Fairing*> **Fairings** { **get;** }
A list of all fairings in the vessel.

Game Scenes All

IList<*Intake*> **Intakes** { **get;** }
A list of all intakes in the vessel.

Game Scenes All

IList<*Leg*> **Legs** { **get;** }
A list of all landing legs attached to the vessel.

Game Scenes All

IList<*LaunchClamp*> **LaunchClamps** { **get;** }
A list of all launch clamps attached to the vessel.

Game Scenes All

IList<*Light*> **Lights** { **get;** }
A list of all lights in the vessel.

Game Scenes All

IList<*Parachute*> **Parachutes** { **get;** }
A list of all parachutes in the vessel.

Game Scenes All

IList<*Radiator*> **Radiators** { **get;** }
A list of all radiators in the vessel.

Game Scenes All

IList<*RCS*> **RCS** { **get;** }
A list of all RCS blocks/thrusters in the vessel.

Game Scenes All

IList<*ReactionWheel*> **ReactionWheels** { **get;** }
A list of all reaction wheels in the vessel.

Game Scenes All

IList<*ResourceConverter*> **ResourceConverters** { **get;** }
A list of all resource converters in the vessel.

Game Scenes All

IList<*ResourceHarvester*> **ResourceHarvesters** { **get;** }
A list of all resource harvesters in the vessel.

Game Scenes All

IList<*Sensor*> **Sensors** { **get;** }
A list of all sensors in the vessel.

Game Scenes All

IList<*SolarPanel*> **SolarPanels** { **get;** }
A list of all solar panels in the vessel.

Game Scenes All

IList<*Wheel*> **Wheels** { **get;** }
A list of all wheels in the vessel.

Game Scenes All

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".

Game Scenes All

String **Title** { **get;** }

Title of the part, as shown when the part is right clicked in-game. For example "Mk1-2 Command Pod".

Game Scenes All

String **Tag** { **get; set;** }

The name tag for the part. Can be set to a custom string using the in-game user interface.

Game Scenes All

Note: This string is shared with kOS if it is installed.

Boolean **Highlighted** { **get; set;** }

Whether the part is highlighted.

Game Scenes All

Tuple<Double, Double, Double> **HighlightColor** { **get; set;** }

The color used to highlight the part, as an RGB triple.

Game Scenes All

Double **Cost** { **get;** }

The cost of the part, in units of funds.

Game Scenes All

Vessel **Vessel** { **get;** }

The vessel that contains this part.

Game Scenes All

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.

Game Scenes All

Note: See the discussion on *Trees of Parts*.

IList<*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.

Game Scenes All

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`.

Game Scenes All

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`.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Note: See the discussion on *Staging*.

Boolean **Massless** { **get;** }
Whether the part is massless.

Game Scenes All

Double **Mass** { **get;** }
The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

Game Scenes All

Double **DryMass** { **get;** }
The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

Game Scenes All

Boolean **Shielded** { **get;** }
Whether the part is shielded from the exterior of the vessel, for example by a fairing.

Game Scenes All

Single **DynamicPressure** { **get;** }
The dynamic pressure acting on the part, in Pascals.

Game Scenes All

Double **ImpactTolerance** { **get;** }
The impact tolerance of the part, in meters per second.

Game Scenes All

Double **Temperature** { **get;** }
Temperature of the part, in Kelvin.

Game Scenes All

Double **SkinTemperature** { **get;** }
Temperature of the skin of the part, in Kelvin.

Game Scenes All

Double **MaxTemperature** { **get;** }
Maximum temperature that the part can survive, in Kelvin.

Game Scenes All

Double **MaxSkinTemperature** { **get;** }
Maximum temperature that the skin of the part can survive, in Kelvin.

Game Scenes All

Single **ThermalMass** { **get;** }
A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

Game Scenes All

Single **ThermalSkinMass** { **get;** }
A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Single **ThermalInternalFlux** { **get;** }
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.

Game Scenes All

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.

Game Scenes All

Resources **Resources** { **get;** }
A `Resources` object for the part.

Game Scenes All

Boolean **Crossfeed** { **get;** }
Whether this part is crossfeed capable.

Game Scenes All

Boolean **IsFuelLine** { **get;** }
Whether this part is a fuel line.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Note: See the discussion on *Fuel Lines*.

IList<*Module*> **Modules** { **get;** }
The modules for this part.

Game Scenes All

Antenna **Antenna** { **get;** }
A `Antenna` if the part is an antenna, otherwise `null`.

Game Scenes All

CargoBay **CargoBay** { **get;** }
A `CargoBay` if the part is a cargo bay, otherwise `null`.

Game Scenes All

ControlSurface **ControlSurface** { **get;** }
A `ControlSurface` if the part is an aerodynamic control surface, otherwise `null`.

Game Scenes All

Decoupler **Decoupler** { **get;** }
A *Decoupler* if the part is a decoupler, otherwise `null`.

Game Scenes All

DockingPort **DockingPort** { **get;** }
A *DockingPort* if the part is a docking port, otherwise `null`.

Game Scenes All

Engine **Engine** { **get;** }
An *Engine* if the part is an engine, otherwise `null`.

Game Scenes All

Experiment **Experiment** { **get;** }
An *Experiment* if the part is a science experiment, otherwise `null`.

Game Scenes All

Fairing **Fairing** { **get;** }
A *Fairing* if the part is a fairing, otherwise `null`.

Game Scenes All

Intake **Intake** { **get;** }
An *Intake* if the part is an intake, otherwise `null`.

Game Scenes All

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`.

Game Scenes All

LaunchClamp **LaunchClamp** { **get;** }
A *LaunchClamp* if the part is a launch clamp, otherwise `null`.

Game Scenes All

Light **Light** { **get;** }
A *Light* if the part is a light, otherwise `null`.

Game Scenes All

Parachute **Parachute** { **get;** }
A *Parachute* if the part is a parachute, otherwise `null`.

Game Scenes All

Radiator **Radiator** { **get;** }
A *Radiator* if the part is a radiator, otherwise `null`.

Game Scenes All

RCS **RCS** { **get;** }
A *RCS* if the part is an RCS block/thruster, otherwise `null`.

Game Scenes All

ReactionWheel **ReactionWheel** { **get;** }
A *ReactionWheel* if the part is a reaction wheel, otherwise `null`.

Game Scenes All

ResourceConverter **ResourceConverter** { **get;** }
A *ResourceConverter* if the part is a resource converter, otherwise `null`.

Game Scenes All

ResourceHarvester **ResourceHarvester** { **get;** }
A *ResourceHarvester* if the part is a resource harvester, otherwise `null`.

Game Scenes All

Sensor **Sensor** { **get;** }
A *Sensor* if the part is a sensor, otherwise `null`.

Game Scenes All

SolarPanel **SolarPanel** { **get;** }
A *SolarPanel* if the part is a solar panel, otherwise `null`.

Game Scenes All

Wheel **Wheel** { **get;** }
A *Wheel* if the part is a wheel, otherwise `null`.

Game Scenes All

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.

Game Scenes All

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* 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.

Game Scenes All

Tuple<Tuple<Double, Double, Double>, Tuple<Double, Double, Double>> **BoundingBox** (*ReferenceFrame reference-Frame*)
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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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).

Game Scenes All

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`).

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *DockingPort.ReferenceFrame*.

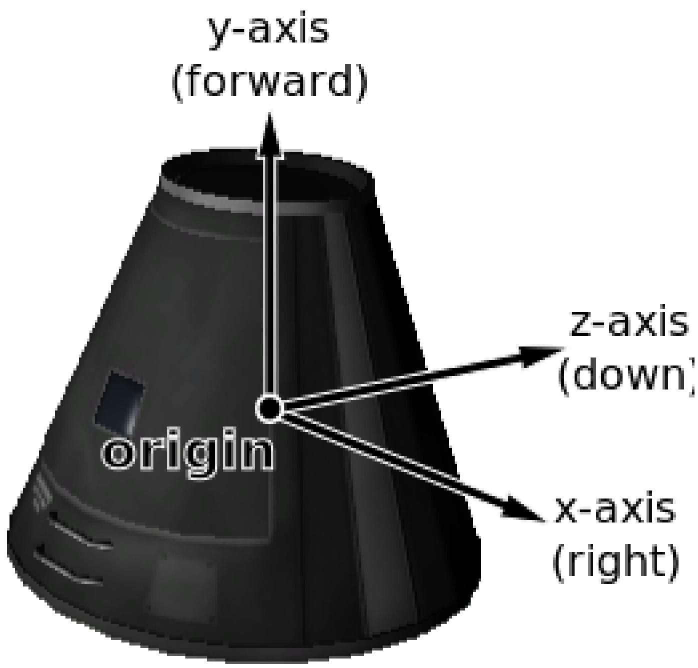


Fig. 7: Mk1 Command Pod reference frame origin and axes

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Tuple<Double, Double, Double> **Position** { **get; set;** }

The position at which the force acts, in reference frame `ReferenceFrame`.

Returns The position as a vector.

Game Scenes All

ReferenceFrame **ReferenceFrame** { **get; set;** }

The reference frame of the force vector and position.

Game Scenes All

void **Remove** ()

Remove the force.

Game Scenes All

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”.

Game Scenes All

Part **Part** { **get;** }

The part that contains this module.

Game Scenes All

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.

Game Scenes All

Boolean **HasField** (String *name*)

Returns `true` if the module has a field with the given name.

Parameters

- **name** – Name of the field.

Game Scenes All

String **GetField** (String *name*)

Returns the value of a field.

Parameters

- **name** – Name of the field.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

void **ResetField** (String *name*)
Set the value of a field to its original value.

Parameters

- **name** – Name of the field.

Game Scenes All

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.

Game Scenes All

Boolean **HasEvent** (String *name*)
`true` if the module has an event with the given name.

Parameters

Game Scenes All

void **TriggerEvent** (String *name*)
Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters

Game Scenes All

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.

Game Scenes All

Boolean **HasAction** (String *name*)
`true` if the part has an action with the given name.

Parameters

Game Scenes All

void **SetAction** (String *name*, Boolean *value* = *true*)
Set the value of an action with the given name.

Parameters

Game Scenes All

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.

Game Scenes All

AntennaState **State** { **get;** }

The current state of the antenna.

Game Scenes All

Boolean **Deployable** { **get;** }

Whether the antenna is deployable.

Game Scenes All

Boolean **Deployed** { **get; set;** }

Whether the antenna is deployed.

Game Scenes All

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.

Game Scenes All

void **Transmit** ()

Transmit data.

Game Scenes All

void **Cancel** ()

Cancel current transmission of data.

Game Scenes All

Boolean **AllowPartial** { **get; set;** }

Whether partial data transmission is permitted.

Game Scenes All

Double **Power** { **get;** }

The power of the antenna.

Game Scenes All

Boolean **Combinable** { **get;** }

Whether the antenna can be combined with other antennae on the vessel to boost the power.

Game Scenes All

Double **CombinableExponent** { **get;** }

Exponent used to calculate the combined power of multiple antennae on a vessel.

Game Scenes All

Single **PacketInterval** { **get;** }

Interval between sending packets in seconds.

Game Scenes All

Single **PacketSize** { **get;** }

Amount of data sent per packet in Mits.

Game Scenes All

Double **PacketResourceCost** { **get;** }

Units of electric charge consumed per packet sent.

Game Scenes All

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.

Game Scenes All

CargoBayState **State** { **get;** }

The state of the cargo bay.

Game Scenes All

Boolean **Open** { **get; set;** }

Whether the cargo bay is open.

Game Scenes All

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.

Game Scenes All

Boolean **PitchEnabled** { **get; set;** }

Whether the control surface has pitch control enabled.

Game Scenes All

Boolean **YawEnabled** { **get; set;** }

Whether the control surface has yaw control enabled.

Game Scenes All

Boolean **RollEnabled** { **get; set;** }

Whether the control surface has roll control enabled.

Game Scenes All

Single **AuthorityLimiter** { **get; set;** }

The authority limiter for the control surface, which controls how far the control surface will move.

Game Scenes All

Boolean **Inverted** { **get; set;** }
Whether the control surface movement is inverted.

Game Scenes All

Boolean **Deployed** { **get; set;** }
Whether the control surface has been fully deployed.

Game Scenes All

Single **SurfaceArea** { **get;** }
Surface area of the control surface in m^2.

Game Scenes All

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*.

Game Scenes All

Decoupler

class Decoupler
A decoupler. Obtained by calling *Part.Decoupler*

Part **Part** { **get;** }
The part object for this decoupler.

Game Scenes All

Vessel **Decouple** ()
Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

Game Scenes All

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.

Game Scenes All

Boolean **Staged** { **get;** }
Whether the decoupler is enabled in the staging sequence.

Game Scenes All

Single **Impulse** { **get;** }
The impulse that the decoupler imparts when it is fired, in Newton seconds.

Game Scenes All

Docking Port

class DockingPort

A docking port. Obtained by calling *Part.DockingPort*

Part **Part** { **get;** }

The part object for this docking port.

Game Scenes All

DockingPortState **State** { **get;** }

The current state of the docking port.

Game Scenes All

Part **DockedPart** { **get;** }

The part that this docking port is docked to. Returns `null` if this docking port is not docked to anything.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Boolean **HasShield** { **get;** }

Whether the docking port has a shield.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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).

Game Scenes All

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.

Game Scenes All

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by *Part.ReferenceFrame*.

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.

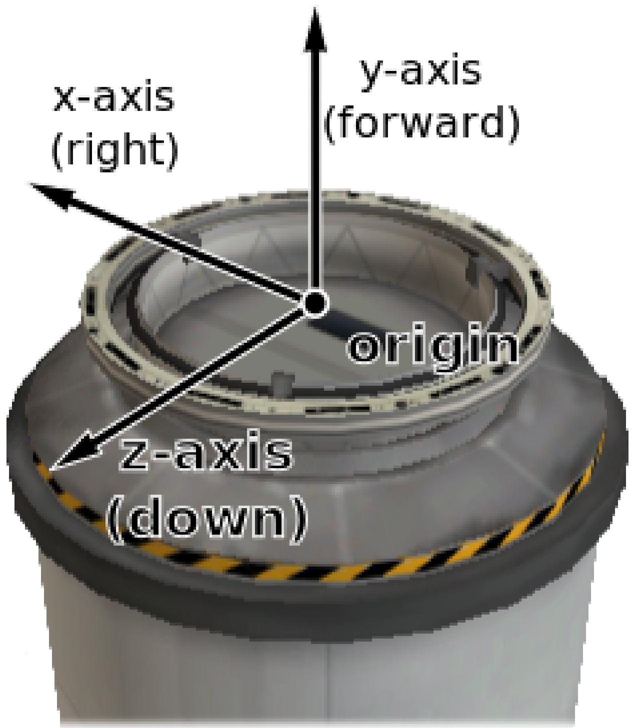


Fig. 8: Docking port reference frame origin and axes

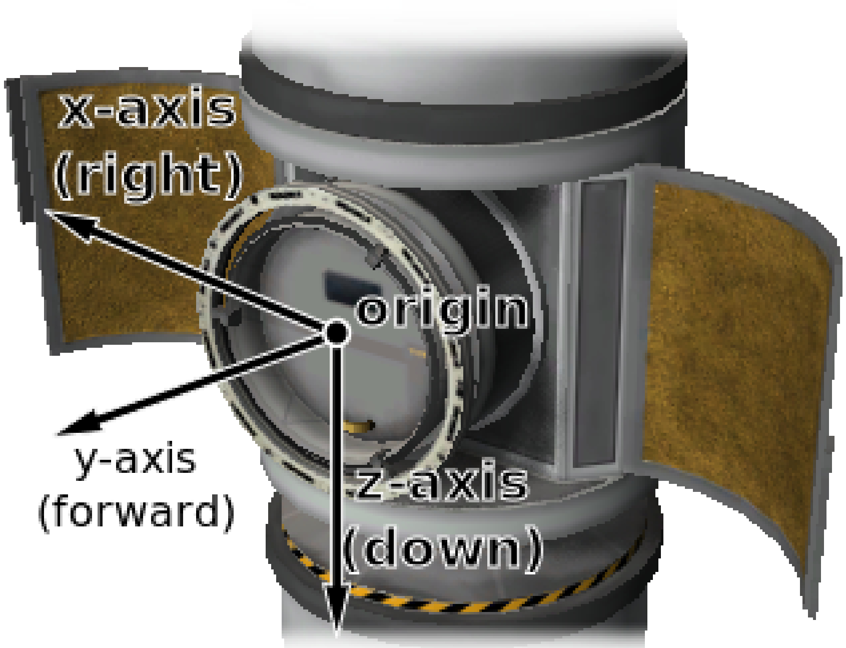


Fig. 9: Inline docking port reference frame origin and axes

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 **Part** { **get;** }

The part object for this engine.

Game Scenes All

Boolean **Active** { **get; set;** }

Whether the engine is active. Setting this attribute may have no effect, depending on *Engine.CanShutdown* and *Engine.CanRestart*.

Game Scenes All

Single **Thrust** { **get;** }

The current amount of thrust being produced by the engine, in Newtons.

Game Scenes All

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.

Game Scenes All

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%.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

IList<*Thruster*> **Thrusters** { **get;** }

The components of the engine that generate thrust.

Game Scenes All

Note: For example, this corresponds to the rocket nozzel on a solid rocket booster, or the individual nozzels 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.

Game Scenes All

Single **VacuumSpecificImpulse** { **get;** }
The vacuum specific impulse of the engine, in seconds.

Game Scenes All

Single **KerbinSeaLevelSpecificImpulse** { **get;** }
The specific impulse of the engine at sea level on Kerbin, in seconds.

Game Scenes All

IList<String> **PropellantNames** { **get;** }
The names of the propellants that the engine consumes.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Boolean **HasFuel** { **get;** }
Whether the engine has any fuel available.

Game Scenes All

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).

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Boolean **HasModes** { **get;** }
Whether the engine has multiple modes of operation.

Game Scenes All

String **Mode** { **get; set;** }
The name of the current engine mode.

Game Scenes All

IDictionary<String, *Engine*> **Modes** { **get;** }
The available modes for the engine. A dictionary mapping mode names to *Engine* objects.

Game Scenes All

void **ToggleMode** ()
Toggle the current engine mode.

Game Scenes All

Boolean **AutoModeSwitch** { **get; set;** }
Whether the engine will automatically switch modes.

Game Scenes All

Boolean **Gimballed** { **get;** }
Whether the engine is gimballed.

Game Scenes All

Single **GimbalRange** { **get;** }
The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

Game Scenes All

Boolean **GimbalLocked** { **get; set;** }
Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

Game Scenes All

Single **GimbalLimit** { **get; set;** }
The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

Game Scenes All

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 gimballed.

Game Scenes All

class Propellant
A propellant for an engine. Obtains by calling *Engine.Propellants*.

String **Name** { **get;** }
The name of the propellant.

Game Scenes All

Double **CurrentAmount** { **get;** }
The current amount of propellant.

Game Scenes All

Double **CurrentRequirement** { **get;** }
The required amount of propellant.

Game Scenes All

Double **TotalResourceAvailable** { **get;** }
The total amount of the underlying resource currently reachable given resource flow rules.

Game Scenes All

Double **TotalResourceCapacity** { **get;** }
The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

Game Scenes All

Boolean **IgnoreForIsp** { **get;** }
If this propellant should be ignored when calculating required mass flow given specific impulse.

Game Scenes All

Boolean **IgnoreForThrustCurve** { **get;** }
If this propellant should be ignored for thrust curve calculations.

Game Scenes All

Boolean **DrawStackGauge** { **get;** }
If this propellant has a stack gauge or not.

Game Scenes All

Boolean **IsDeprived** { **get;** }
If this propellant is deprived.

Game Scenes All

Single **Ratio** { **get;** }
The propellant ratio.

Game Scenes All

Experiment

class Experiment
Obtained by calling *Part.Experiment*.

Part **Part** { **get;** }
The part object for this experiment.

Game Scenes All

void **Run** ()
Run the experiment.

Game Scenes All

void **Transmit** ()
Transmit all experimental data contained by this part.

Game Scenes All

void **Dump** ()
Dump the experimental data contained by the experiment.

Game Scenes All

void **Reset** ()
Reset the experiment.

Game Scenes All

Boolean **Deployed** { **get;** }
Whether the experiment has been deployed.

Game Scenes All

Boolean **Rerunnable** { **get;** }
Whether the experiment can be re-run.

Game Scenes All

Boolean **Inoperable** { **get;** }
Whether the experiment is inoperable.

Game Scenes All

Boolean **HasData** { **get;** }
Whether the experiment contains data.

Game Scenes All

IList<*ScienceData*> **Data** { **get;** }
The data contained in this experiment.

Game Scenes All

String **Biome** { **get;** }
The name of the biome the experiment is currently in.

Game Scenes All

Boolean **Available** { **get;** }
Determines if the experiment is available given the current conditions.

Game Scenes All

ScienceSubject **ScienceSubject** { **get;** }
Containing information on the corresponding specific science result for the current conditions. Returns `null` if the experiment is unavailable.

Game Scenes All

class ScienceData
Obtained by calling *Experiment.Data*.

Single **DataAmount** { **get;** }
Data amount.

Game Scenes All

Single **ScienceValue** { **get;** }
Science value.

Game Scenes All

Single **TransmitValue** { **get;** }
Transmit value.

Game Scenes All

class ScienceSubject

Obtained by calling *Experiment.ScienceSubject*.

String **Title** { **get;** }

Title of science subject, displayed in science archives

Game Scenes All

Boolean **IsComplete** { **get;** }

Whether the experiment has been completed.

Game Scenes All

Single **Science** { **get;** }

Amount of science already earned from this subject, not updated until after transmission/recovery.

Game Scenes All

Single **ScienceCap** { **get;** }

Total science allowable for this subject.

Game Scenes All

Single **DataScale** { **get;** }

Multiply science value by this to determine data amount in mits.

Game Scenes All

Single **SubjectValue** { **get;** }

Multiplier for specific Celestial Body/Experiment Situation combination.

Game Scenes All

Single **ScientificValue** { **get;** }

Diminishing value multiplier for decreasing the science value returned from repeated experiments.

Game Scenes All

Fairing

class Fairing

A fairing. Obtained by calling *Part.Fairing*.

Part **Part** { **get;** }

The part object for this fairing.

Game Scenes All

void **Jettison** ()

Jettison the fairing. Has no effect if it has already been jettisoned.

Game Scenes All

Boolean **Jettisoned** { **get;** }

Whether the fairing has been jettisoned.

Game Scenes All

Intake

class Intake
An air intake. Obtained by calling `Part.Intake`.

Part **Part** { **get;** }
The part object for this intake.

Game Scenes All

Boolean **Open** { **get; set;** }
Whether the intake is open.

Game Scenes All

Single **Speed** { **get;** }
Speed of the flow into the intake, in m/s.

Game Scenes All

Single **Flow** { **get;** }
The rate of flow into the intake, in units of resource per second.

Game Scenes All

Single **Area** { **get;** }
The area of the intake's opening, in square meters.

Game Scenes All

Leg

class Leg
A landing leg. Obtained by calling `Part.Leg`.

Part **Part** { **get;** }
The part object for this landing leg.

Game Scenes All

LegState **State** { **get;** }
The current state of the landing leg.

Game Scenes All

Boolean **Deployable** { **get;** }
Whether the leg is deployable.

Game Scenes All

Boolean **Deployed** { **get; set;** }
Whether the landing leg is deployed.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

void **Release** ()
Releases the docking clamp. Has no effect if the clamp has already been released.

Game Scenes All

Light

class Light
A light. Obtained by calling *Part.Light*.

Part **Part** { **get;** }
The part object for this light.

Game Scenes All

Boolean **Active** { **get; set;** }
Whether the light is switched on.

Game Scenes All

Tuple<Single, Single, Single> **Color** { **get; set;** }
The color of the light, as an RGB triple.

Game Scenes All

Single **PowerUsage** { **get;** }
The current power usage, in units of charge per second.

Game Scenes All

Parachute

class Parachute
A parachute. Obtained by calling *Part.Parachute*.

Part **Part** { **get;** }
The part object for this parachute.

Game Scenes All

void **Deploy** ()
Deploys the parachute. This has no effect if the parachute has already been deployed.

Game Scenes All

Boolean **Deployed** { **get;** }
Whether the parachute has been deployed.

Game Scenes All

void **Arm** ()
Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

Game Scenes All

Boolean **Armed** { **get;** }
Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

Game Scenes All

ParachuteState **State** { **get;** }
The current state of the parachute.

Game Scenes All

Single **DeployAltitude** { **get; set;** }
The altitude at which the parachute will full deploy, in meters. Only applicable to stock parachutes.

Game Scenes All

Single **DeployMinPressure** { **get; set;** }
The minimum pressure at which the parachute will semi-deploy, in atmospheres. Only applicable to stock parachutes.

Game Scenes All

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.

Game Scenes All

Boolean **Deployable** { **get;** }
Whether the radiator is deployable.

Game Scenes All

Boolean **Deployed** { **get; set;** }
For a deployable radiator, `true` if the radiator is extended. If the radiator is not deployable, this is always `true`.

Game Scenes All

RadiatorState **State** { **get;** }
The current state of the radiator.

Game Scenes All

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.

Game Scenes All

Int32 **Count** { **get;** }

The number of converters in the part.

Game Scenes All

String **Name** (Int32 *index*)

The name of the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

Boolean **Active** (Int32 *index*)

True if the specified converter is active.

Parameters

- **index** – Index of the converter.

Game Scenes All

void **Start** (Int32 *index*)

Start the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

void **Stop** (Int32 *index*)

Stop the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

ResourceConverterState **State** (Int32 *index*)

The state of the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

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.

Game Scenes All

IList<String> **Inputs** (Int32 *index*)

List of the names of resources consumed by the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

IList<String> **Outputs** (Int32 *index*)
List of the names of resources produced by the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

Single **OptimumCoreTemperature** { **get;** }
The core temperature at which the converter will operate with peak efficiency, in Kelvin.

Game Scenes All

Single **CoreTemperature** { **get;** }
The core temperature of the converter, in Kelvin.

Game Scenes All

Single **ThermalEfficiency** { **get;** }
The thermal efficiency of the converter, as a percentage of its maximum.

Game Scenes All

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.

Game Scenes All

ResourceHarvesterState **State** { **get;** }
The state of the harvester.

Game Scenes All

Boolean **Deployed** { **get; set;** }
Whether the harvester is deployed.

Game Scenes All

Boolean **Active** { **get; set;** }
Whether the harvester is actively drilling.

Game Scenes All

Single **ExtractionRate** { **get;** }
The rate at which the drill is extracting ore, in units per second.

Game Scenes All

Single **ThermalEfficiency** { **get;** }
The thermal efficiency of the drill, as a percentage of its maximum.

Game Scenes All

Single **CoreTemperature** { **get;** }
The core temperature of the drill, in Kelvin.

Game Scenes All

Single **OptimumCoreTemperature** { **get;** }
The core temperature at which the drill will operate with peak efficiency, in Kelvin.

Game Scenes All

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.

Game Scenes All

Boolean **Active** { **get; set;** }
Whether the reaction wheel is active.

Game Scenes All

Boolean **Broken** { **get;** }
Whether the reaction wheel is broken.

Game Scenes All

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.

Game Scenes All

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*.

Game Scenes All

RCS

class RCS
An RCS block or thruster. Obtained by calling *Part.RCS*.

Part **Part** { **get;** }
The part object for this RCS.

Game Scenes All

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*).

Game Scenes All

Boolean **Enabled** { **get; set;** }
Whether the RCS thrusters are enabled.

Game Scenes All

Boolean **PitchEnabled** { **get; set;** }
Whether the RCS thruster will fire when pitch control input is given.

Game Scenes All

Boolean **YawEnabled** { **get; set;** }
Whether the RCS thruster will fire when yaw control input is given.

Game Scenes All

Boolean **RollEnabled** { **get; set;** }
Whether the RCS thruster will fire when roll control input is given.

Game Scenes All

Boolean **ForwardEnabled** { **get; set;** }
Whether the RCS thruster will fire when pitch control input is given.

Game Scenes All

Boolean **UpEnabled** { **get; set;** }
Whether the RCS thruster will fire when yaw control input is given.

Game Scenes All

Boolean **RightEnabled** { **get; set;** }
Whether the RCS thruster will fire when roll control input is given.

Game Scenes All

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.

Game Scenes All

Single **MaxThrust** { **get;** }
The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

Game Scenes All

Single **MaxVacuumThrust** { **get;** }
The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

Game Scenes All

IList<*Thruster*> **Thrusters** { **get;** }
A list of thrusters, one of each nozzel in the RCS part.

Game Scenes All

Single **SpecificImpulse** { **get;** }
The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

Game Scenes All

Single **VacuumSpecificImpulse** { **get;** }
The vacuum specific impulse of the RCS, in seconds.

Game Scenes All

Single **KerbinSeaLevelSpecificImpulse** { **get;** }
The specific impulse of the RCS at sea level on Kerbin, in seconds.

Game Scenes All

IList<String> **Propellants** { **get;** }
The names of resources that the RCS consumes.

Game Scenes All

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.

Game Scenes All

Boolean **HasFuel** { **get;** }
Whether the RCS has fuel available.

Game Scenes All

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.

Game Scenes All

Boolean **Active** { **get; set;** }
Whether the sensor is active.

Game Scenes All

String **Value** { **get;** }
The current value of the sensor.

Game Scenes All

Solar Panel

class SolarPanel
A solar panel. Obtained by calling `Part.SolarPanel`.

Part **Part** { **get;** }
The part object for this solar panel.

Game Scenes All

Boolean **Deployable** { **get;** }
Whether the solar panel is deployable.

Game Scenes All

Boolean **Deployed** { **get; set;** }
Whether the solar panel is extended.

Game Scenes All

SolarPanelState **State** { **get;** }
The current state of the solar panel.

Game Scenes All

Single **EnergyFlow** { **get;** }
The current amount of energy being generated by the solar panel, in units of charge per second.

Game Scenes All

Single **SunExposure** { **get;** }
The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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 gimballing.
- The y-axis points along the thrust direction.

- The x-axis and z-axis are perpendicular to the thrust direction.

  Game Scenes All

Boolean **Gimballed** { **get;** }
Whether the thruster is gimballed.

Game Scenes All

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.

Game Scenes All

Tuple<Double, Double, Double> **GimbalAngle** { **get;** }
The current gimbal angle in the pitch, roll and yaw axes, in degrees.

Game Scenes All

Tuple<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** – The reference frame that the returned position vector is in.

Returns The position as a vector.

Game Scenes All

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 gimballing), 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.

Game Scenes All

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.

Game Scenes All

WheelState **State** { **get;** }
The current state of the wheel.

Game Scenes All

Single **Radius** { **get;** }
Radius of the wheel, in meters.

Game Scenes All

Boolean **Grounded** { **get;** }
Whether the wheel is touching the ground.

Game Scenes All

Boolean **HasBrakes** { **get;** }
Whether the wheel has brakes.

Game Scenes All

Single **Brakes** { **get; set;** }
The braking force, as a percentage of maximum, when the brakes are applied.

Game Scenes All

Boolean **AutoFrictionControl** { **get; set;** }
Whether automatic friction control is enabled.

Game Scenes All

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.

Game Scenes All

Boolean **Deployable** { **get;** }
Whether the wheel is deployable.

Game Scenes All

Boolean **Deployed** { **get; set;** }
Whether the wheel is deployed.

Game Scenes All

Boolean **Powered** { **get;** }
Whether the wheel is powered by a motor.

Game Scenes All

Boolean **MotorEnabled** { **get; set;** }
Whether the motor is enabled.

Game Scenes All

Boolean **MotorInverted** { **get; set;** }
Whether the direction of the motor is inverted.

Game Scenes All

MotorState **MotorState** { **get;** }
Whether the direction of the motor is inverted.

Game Scenes All

Single **MotorOutput** { **get;** }
The output of the motor. This is the torque currently being generated, in Newton meters.

Game Scenes All

Boolean **TractionControlEnabled** { **get; set;** }
Whether automatic traction control is enabled. A wheel only has traction control if it is powered.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Boolean **Steerable** { **get;** }
Whether the wheel has steering.

Game Scenes All

Boolean **SteeringEnabled** { **get; set;** }
Whether the wheel steering is enabled.

Game Scenes All

Boolean **SteeringInverted** { **get; set;** }
Whether the wheel steering is inverted.

Game Scenes All

Boolean **HasSuspension** { **get;** }
Whether the wheel has suspension.

Game Scenes All

Single **SuspensionSpringStrength** { **get;** }
Suspension spring strength, as set in the editor.

Game Scenes All

Single **SuspensionDamperStrength** { **get;** }
Suspension damper strength, as set in the editor.

Game Scenes All

Boolean **Broken** { **get;** }
Whether the wheel is broken.

Game Scenes All

Boolean **Repairable** { **get;** }
Whether the wheel is repairable.

Game Scenes All

Single **Stress** { **get;** }
Current stress on the wheel.

Game Scenes All

Single **StressTolerance** { **get;** }
Stress tolerance of the wheel.

Game Scenes All

Single **StressPercentage** { **get;** }
Current stress on the wheel as a percentage of its stress tolerance.

Game Scenes All

Single **Deflection** { **get;** }
Current deflection of the wheel.

Game Scenes All

Single **Slip** { **get;** }
Current slip of the wheel.

Game Scenes All

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`.

Traversing the Tree

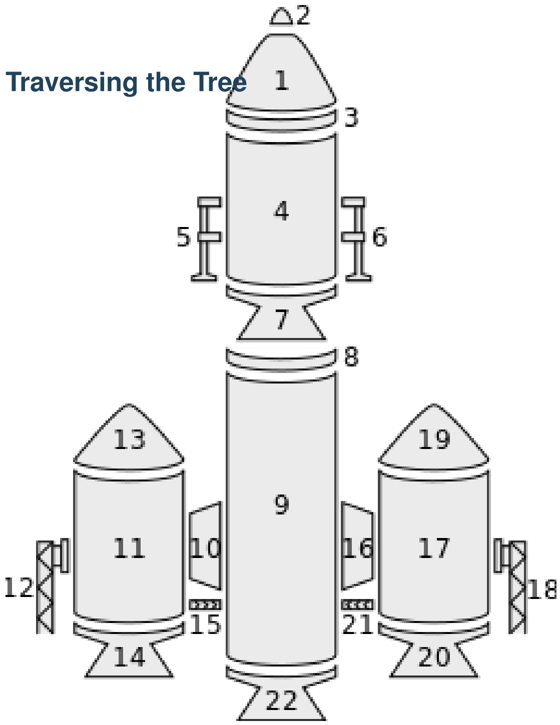


Fig. 10: **Figure 1** – Example parts making up a vessel.

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.Collect
using System.Net;
using KRPC.Client;
using KRPC.Client.Se

class AttachmentMode
{
    public static vo
    {
        
↪using (var connect
            var vess
↪= connection.Space
            var root
                
↪   var stack = new
                
↪ stack.Push (new T
            while (s
                var
                Part
                int
                Cons
↪(new String (' ',
                
↪            foreach (v
                
↪(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 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).

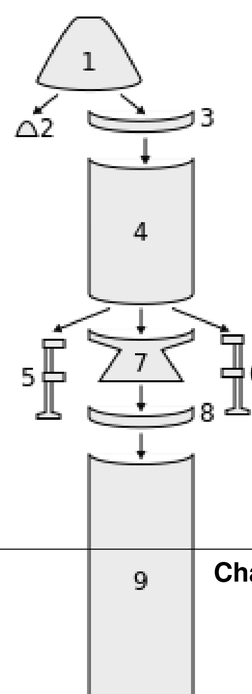
For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

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 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;
```

(continues on next page)



(continued from previous page)

```
            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));
            }
        }
    }
}
```

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

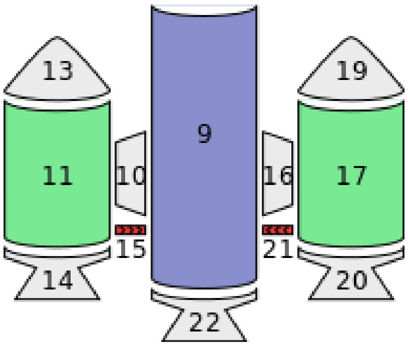


Fig. 12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

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).

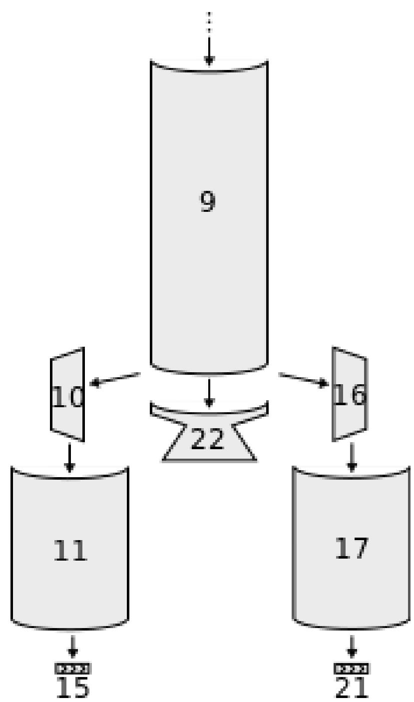


Fig. 13: **Figure 4** – A subset of the parts tree from Figure 2 above.

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.

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*.

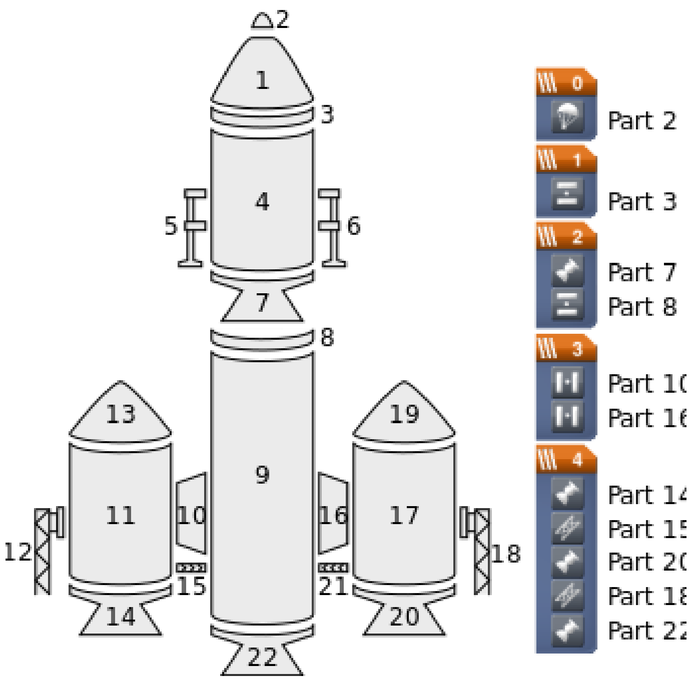


Fig. 14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

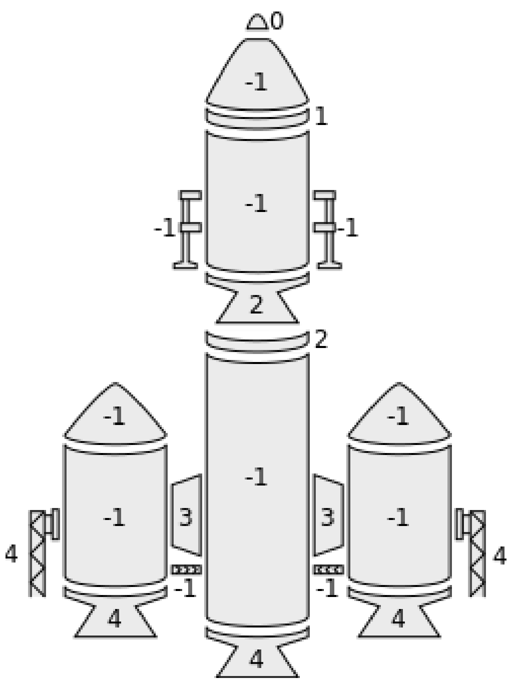


Fig. 15: **Figure 7** – The stage in which each part is *activated*.

4.3.9 Resources

class Resources

Represents the collection of resources stored in a vessel, stage or part. Created by calling `Vessel.Resources`, `Vessel.ResourcesInDecoupleStage` or `Part.Resources`.

IList<*Resource*> **All** { **get;** }
All the individual resources that can be stored.

Game Scenes Flight

IList<*Resource*> **WithResource** (String *name*)
All the individual resources with the given name that can be stored.

Parameters

Game Scenes Flight

IList<String> **Names** { **get;** }
A list of resource names that can be stored.

Game Scenes Flight

Boolean **HasResource** (String *name*)
Check whether the named resource can be stored.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

Single **Amount** (String *name*)
Returns the amount of a resource that is currently stored.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

Single **Max** (String *name*)
Returns the amount of a resource that can be stored.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

static Single **Density** (*IConnection connection*, String *name*)
Returns the density of a resource, in kg/l.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

static ResourceFlowMode **FlowMode** (*IConnection connection*, String *name*)
Returns the flow mode of a resource.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

Boolean **Enabled** { **get; set;** }
Whether use of all the resources are enabled.

Game Scenes Flight

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.

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

The name of the resource.

Game Scenes All

Part **Part** { **get;** }

The part containing the resource.

Game Scenes All

Single **Amount** { **get;** }

The amount of the resource that is currently stored in the part.

Game Scenes All

Single **Max** { **get;** }

The total amount of the resource that can be stored in the part.

Game Scenes All

Single **Density** { **get;** }

The density of the resource, in kg/l.

Game Scenes All

ResourceFlowMode **FlowMode** { **get;** }

The flow mode of the resource.

Game Scenes All

Boolean **Enabled** { **get; set;** }

Whether use of this resource is enabled.

Game Scenes All

class ResourceTransfer

Transfer resources between parts.

static ResourceTransfer **Start** (*IConnection 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.

Game Scenes All

Single **Amount** { **get;** }

The amount of the resource that has been transferred.

Game Scenes All

Boolean **Complete** { **get;** }

Whether the transfer has completed.

Game Scenes All

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.

4.3.10 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.

Game Scenes Flight

Double **Normal** { **get; set;** }

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

Game Scenes Flight

Double **Radial** { **get; set;** }

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

Game Scenes Flight

Double **DeltaV** { **get; set;** }

The delta-v of the maneuver node, in meters per second.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Double **TimeTo** { **get;** }
The time until the maneuver node will be encountered, in seconds.

Game Scenes Flight

Orbit **Orbit** { **get;** }
The orbit that results from executing the maneuver node.

Game Scenes Flight

void **Remove** ()

Removes the maneuver node.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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 **CreateRelative** (*IConnection connection*, *ReferenceFrame referenceFrame*, Tuple<Double, Double, Double> *position = { 0.0, 0.0, 0.0 }*, Tuple<Double, Double, Double, Double> *rotation = { 0.0, 0.0, 0.0, 1.0 }*, Tuple<Double, Double, Double> *velocity = { 0.0, 0.0, 0.0 }*, Tuple<Double, Double, Double> *angularVelocity = { 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

- **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 $(0, 0, 0)$.

Game Scenes All

static ReferenceFrame **CreateHybrid** (*IConnection 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.

Game Scenes All

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.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.

Game Scenes Flight

void **Disengage** ()

Disengage the auto-pilot.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

ReferenceFrame **ReferenceFrame** { **get; set;** }
The reference frame for the target direction (*AutoPilot.TargetDirection*).

Game Scenes Flight

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°.

Game Scenes Flight

Single **TargetHeading** { **get; set;** }
The target heading, in degrees, between 0° and 360°.

Game Scenes Flight

Single **TargetRoll** { **get; set;** }
The target roll, in degrees. `NaN` if no target roll is set.

Game Scenes Flight

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*.

Game Scenes Flight

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°.

Game Scenes Flight

Boolean **SAS** { **get; set;** }
The state of SAS.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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*.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Tuple<Double, Double, Double> **PitchPIDGains** { **get; set;** }
Gains for the pitch PID controller.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Note: When *AutoPilot.AutoTune* 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 *SpaceCenter.Camera*.

CameraMode **Mode** { **get; set;** }
The current mode of the camera.

Game Scenes Flight

Single **Pitch** { **get; set;** }
The pitch of the camera, in degrees. A value between *Camera.MinPitch* and *Camera.MaxPitch*

Game Scenes Flight

Single **Heading** { **get; set;** }
The heading of the camera, in degrees.

Game Scenes Flight

Single **Distance** { **get; set;** }
The distance from the camera to the subject, in meters. A value between *Camera.MinDistance* and *Camera.MaxDistance*.

Game Scenes Flight

Single **MinPitch** { **get;** }
The minimum pitch of the camera.

Game Scenes Flight

Single **MaxPitch** { **get;** }
The maximum pitch of the camera.

Game Scenes Flight

Single **MinDistance** { **get;** }
Minimum distance from the camera to the subject, in meters.

Game Scenes Flight

Single **MaxDistance** { **get;** }
Maximum distance from the camera to the subject, in meters.

Game Scenes Flight

Single **DefaultDistance** { **get;** }
Default distance from the camera to the subject, in meters.

Game Scenes Flight

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 is the camera is not in map mode.

Game Scenes Flight

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 is the camera is not in map mode.

Game Scenes Flight

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 is the camera is not in map mode.

Game Scenes Flight

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.

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 *SpaceCenter.WaypointManager*.

IList<*Waypoint*> **Waypoints** { **get;** }

A list of all existing waypoints.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

IDictionary<String, Int32> **Colors** { **get;** }

An example map of known color - seed pairs. Any other integers may be used as seed.

Game Scenes All

IList<String> **Icons** { **get;** }

Returns all available icons (from "GameData/Squad/Contracts/Icons/").

Game Scenes All

class Waypoint

Represents a waypoint. Can be created using *WaypointManager.AddWaypoint*.

CelestialBody **Body** { **get; set;** }

The celestial body the waypoint is attached to.

Game Scenes Flight

String **Name** { **get; set;** }
The name of the waypoint as it appears on the map and the contract.

Game Scenes Flight

Int32 **Color** { **get; set;** }
The seed of the icon color. See *WaypointManager.Colors* for example colors.

Game Scenes Flight

String **Icon** { **get; set;** }
The icon of the waypoint.

Game Scenes Flight

Double **Latitude** { **get; set;** }
The latitude of the waypoint.

Game Scenes Flight

Double **Longitude** { **get; set;** }
The longitude of the waypoint.

Game Scenes Flight

Double **MeanAltitude** { **get; set;** }
The altitude of the waypoint above sea level, in meters.

Game Scenes Flight

Double **SurfaceAltitude** { **get; set;** }
The altitude of the waypoint above the surface of the body or sea level, whichever is closer, in meters.

Game Scenes Flight

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.

Game Scenes Flight

Boolean **NearSurface** { **get;** }
`true` if the waypoint is near to the surface of a body.

Game Scenes Flight

Boolean **Grounded** { **get;** }
`true` if the waypoint is attached to the ground.

Game Scenes Flight

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.

Game Scenes Flight

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*.

Game Scenes Flight

Boolean **HasContract** { **get;** }
Whether the waypoint belongs to a contract.

Game Scenes Flight

Contract **Contract** { **get;** }
The associated contract.

Game Scenes Flight

void **Remove** ()
Removes the waypoint.

Game Scenes Flight

4.3.15 Contracts

class ContractManager
Contracts manager. Obtained by calling *SpaceCenter.ContractManager*.

ISet<String> **Types** { **get;** }
A list of all contract types.

Game Scenes All

IList<*Contract*> **AllContracts** { **get;** }
A list of all contracts.

Game Scenes All

IList<*Contract*> **ActiveContracts** { **get;** }
A list of all active contracts.

Game Scenes All

IList<*Contract*> **OfferedContracts** { **get;** }
A list of all offered, but unaccepted, contracts.

Game Scenes All

IList<*Contract*> **CompletedContracts** { **get;** }
A list of all completed contracts.

Game Scenes All

IList<*Contract*> **FailedContracts** { **get;** }
A list of all failed contracts.

Game Scenes All

class Contract
A contract. Can be accessed using *SpaceCenter.ContractManager*.

String **Type** { **get;** }
Type of the contract.

Game Scenes All

String **Title** { **get;** }
Title of the contract.

Game Scenes All

String **Description** { **get;** }
Description of the contract.

Game Scenes All

String **Notes** { **get;** }
Notes for the contract.

Game Scenes All

String **Synopsis** { **get;** }
Synopsis for the contract.

Game Scenes All

IList<String> **Keywords** { **get;** }
Keywords for the contract.

Game Scenes All

ContractState **State** { **get;** }
State of the contract.

Game Scenes All

Boolean **Seen** { **get;** }
Whether the contract has been seen.

Game Scenes All

Boolean **Read** { **get;** }
Whether the contract has been read.

Game Scenes All

Boolean **Active** { **get;** }
Whether the contract is active.

Game Scenes All

Boolean **Failed** { **get;** }
Whether the contract has been failed.

Game Scenes All

Boolean **CanBeCanceled** { **get;** }
Whether the contract can be canceled.

Game Scenes All

Boolean **CanBeDeclined** { **get;** }
Whether the contract can be declined.

Game Scenes All

Boolean **CanBeFailed** { **get;** }
Whether the contract can be failed.

Game Scenes All

void **Accept** ()

Accept an offered contract.

Game Scenes All

void **Cancel** ()

Cancel an active contract.

Game Scenes All

void **Decline** ()

Decline an offered contract.

Game Scenes All

Double **FundsAdvance** { **get;** }

Funds received when accepting the contract.

Game Scenes All

Double **FundsCompletion** { **get;** }

Funds received on completion of the contract.

Game Scenes All

Double **FundsFailure** { **get;** }

Funds lost if the contract is failed.

Game Scenes All

Double **ReputationCompletion** { **get;** }

Reputation gained on completion of the contract.

Game Scenes All

Double **ReputationFailure** { **get;** }

Reputation lost if the contract is failed.

Game Scenes All

Double **ScienceCompletion** { **get;** }

Science gained on completion of the contract.

Game Scenes All

IList<*ContractParameter*> **Parameters** { **get;** }

Parameters for the contract.

Game Scenes All

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.

Game Scenes All

String **Notes** { **get;** }
Notes for the parameter.

Game Scenes All

IList<*ContractParameter*> **Children** { **get;** }
Child contract parameters.

Game Scenes All

Boolean **Completed** { **get;** }
Whether the parameter has been completed.

Game Scenes All

Boolean **Failed** { **get;** }
Whether the parameter has been failed.

Game Scenes All

Boolean **Optional** { **get;** }
Whether the contract parameter is optional.

Game Scenes All

Double **FundsCompletion** { **get;** }
Funds received on completion of the contract parameter.

Game Scenes All

Double **FundsFailure** { **get;** }
Funds lost if the contract parameter is failed.

Game Scenes All

Double **ReputationCompletion** { **get;** }
Reputation gained on completion of the contract parameter.

Game Scenes All

Double **ReputationFailure** { **get;** }
Reputation lost if the contract parameter is failed.

Game Scenes All

Double **ScienceCompletion** { **get;** }
Science gained on completion of the contract parameter.

Game Scenes All

4.3.16 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;
```

(continues on next page)

(continued from previous page)

```
            Tuple<double,double,
↪double,double> q = vessel.Flight ().Rotation;
            Console.WriteLine (q.Item1 + ",
↪ " + q.Item2 + ", " + q.Item3 + ", " + q.Item4);
        }
    }
}
```

4.4 Drawing API

4.4.1 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.

Game Scenes Flight

Line **AddDirection** (Tuple<Double, Double, Double> *direction*, *SpaceCenter.ReferenceFrame referenceFrame*, Single *length = 10.0f*, 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.

Game Scenes Flight

Polygon **AddPolygon** (IList<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.

Game Scenes Flight

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.

Game Scenes Flight

void **Clear** (Boolean *clientOnly = false*)
Remove all objects being drawn.

Parameters

- **clientOnly** – If true, only remove objects created by the calling client.

Game Scenes Flight

4.4.2 Line

class Line
A line. Created using `Drawing.AddLine`.

Tuple<Double, Double, Double> **Start** { **get; set;** }
Start position of the line.

Game Scenes Flight

Tuple<Double, Double, Double> **End** { **get; set;** }
End position of the line.

Game Scenes Flight

SpaceCenter.ReferenceFrame **ReferenceFrame** { **get; set;** }
Reference frame for the positions of the object.

Game Scenes Flight

Boolean **Visible** { **get; set;** }
Whether the object is visible.

Game Scenes Flight

Tuple<Double, Double, Double> **Color** { **get; set;** }
Set the color

Game Scenes Flight

String **Material** { **get; set;** }
Material used to render the object. Creates the material from a shader with the given name.

Game Scenes Flight

Single **Thickness** { **get; set;** }
Set the thickness

Game Scenes Flight

void **Remove** ()
Remove the object.

Game Scenes Flight

4.4.3 Polygon

class Polygon
A polygon. Created using *Drawing.AddPolygon*.

IList<Tuple<Double, Double, Double>> **Vertices** { **get; set;** }
Vertices for the polygon.

Game Scenes Flight

SpaceCenter.ReferenceFrame **ReferenceFrame** { **get; set;** }
Reference frame for the positions of the object.

Game Scenes Flight

Boolean **Visible** { **get; set;** }
Whether the object is visible.

Game Scenes Flight

void **Remove** ()
Remove the object.

Game Scenes Flight

Tuple<Double, Double, Double> **Color** { **get; set;** }
Set the color

Game Scenes Flight

String **Material** { **get; set;** }
Material used to render the object. Creates the material from a shader with the given name.

Game Scenes Flight

Single **Thickness** { **get; set;** }
Set the thickness

Game Scenes Flight

4.4.4 Text

class Text
Text. Created using *Drawing.AddText*.

Tuple<Double, Double, Double> **Position** { **get; set;** }
Position of the text.

Game Scenes Flight

Tuple<Double, Double, Double, Double> **Rotation** { **get; set;** }
Rotation of the text as a quaternion.

Game Scenes Flight

SpaceCenter.ReferenceFrame **ReferenceFrame** { **get; set;** }
Reference frame for the positions of the object.

Game Scenes Flight

Boolean **Visible** { **get; set;** }
Whether the object is visible.

Game Scenes Flight

void **Remove** ()
Remove the object.

Game Scenes Flight

String **Content** { **get; set;** }
The text string

Game Scenes Flight

String **Font** { **get; set;** }
Name of the font

Game Scenes Flight

static IList<String> **AvailableFonts** (*IConnection connection*)
A list of all available fonts.

Game Scenes Flight

Int32 **Size** { **get; set;** }
Font size.

Game Scenes Flight

Single **CharacterSize** { **get; set;** }
Character size.

Game Scenes Flight

UI.FontStyle **Style** { **get; set;** }
Font style.

Game Scenes Flight

Tuple<Double, Double, Double> **Color** { **get; set;** }
Set the color

Game Scenes Flight

String **Material** { **get; set;** }
Material used to render the object. Creates the material from a shader with the given name.

Game Scenes Flight

UI.TextAlignment **Alignment** { **get; set;** }
Alignment.

Game Scenes Flight

Single **LineSpacing** { **get; set;** }
Line spacing.

Game Scenes Flight

UI.TextAnchor **Anchor** { **get; set;** }
Anchor.

Game Scenes Flight

4.5 InfernalRobotics API

Provides RPCs to interact with the InfernalRobotics mod. Both the original mod and Infernal Robotics Next are supported. Provides the following classes:

4.5.1 InfernalRobotics

class InfernalRobotics
This service provides functionality to interact with Infernal Robotics.

Boolean **Available** { **get;** }
Whether Infernal Robotics is installed.

Game Scenes Flight

Boolean **Ready** { **get;** }
Whether Infernal Robotics API is ready.

Game Scenes Flight

IList<*ServoGroup*> **ServoGroups** (*SpaceCenter.Vessel vessel*)
A list of all the servo groups in the given *vessel*.

Parameters

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

4.5.2 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.

Game Scenes Flight

String **ForwardKey** { **get; set;** }

The key assigned to be the "forward" key for the group.

Game Scenes Flight

String **ReverseKey** { **get; set;** }

The key assigned to be the "reverse" key for the group.

Game Scenes Flight

Single **Speed** { **get; set;** }

The speed multiplier for the group.

Game Scenes Flight

Boolean **Expanded** { **get; set;** }

Whether the group is expanded in the InfernalRobotics UI.

Game Scenes Flight

IList<*Servo*> **Servos** { **get;** }

The servos that are in the group.

Game Scenes Flight

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.

Game Scenes Flight

IList<*SpaceCenter.Part*> **Parts** { **get;** }

The parts containing the servos in the group.

Game Scenes Flight

void **MoveRight** ()

Moves all of the servos in the group to the right.

Game Scenes Flight

void **MoveLeft** ()

Moves all of the servos in the group to the left.

Game Scenes Flight

void **MoveCenter** ()

Moves all of the servos in the group to the center.

Game Scenes Flight

void **MoveNextPreset** ()

Moves all of the servos in the group to the next preset.

Game Scenes Flight

void **MovePrevPreset** ()

Moves all of the servos in the group to the previous preset.

Game Scenes Flight

void **Stop** ()

Stops the servos in the group.

Game Scenes Flight

4.5.3 Servo

class Servo

Represents a servo. Obtained using *ServoGroup.Servos*, *ServoGroup.ServoWithName* or *InfernalRobotics.ServoWithName*.

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

The name of the servo.

Game Scenes Flight

SpaceCenter.Part **Part** { **get;** }

The part containing the servo.

Game Scenes Flight

Boolean **Highlight** { **set;** }

Whether the servo should be highlighted in-game.

Game Scenes Flight

Single **Position** { **get;** }

The position of the servo.

Game Scenes Flight

Single **MinConfigPosition** { **get;** }

The minimum position of the servo, specified by the part configuration.

Game Scenes Flight

Single **MaxConfigPosition** { **get;** }

The maximum position of the servo, specified by the part configuration.

Game Scenes Flight

Single **MinPosition** { **get; set;** }

The minimum position of the servo, specified by the in-game tweak menu.

Game Scenes Flight

Single **MaxPosition** { **get; set;** }

The maximum position of the servo, specified by the in-game tweak menu.

Game Scenes Flight

Single **ConfigSpeed** { **get;** }

The speed multiplier of the servo, specified by the part configuration.

Game Scenes Flight

Single **Speed** { **get; set;** }

The speed multiplier of the servo, specified by the in-game tweak menu.

Game Scenes Flight

Single **CurrentSpeed** { **get; set;** }

The current speed at which the servo is moving.

Game Scenes Flight

Single **Acceleration** { **get; set;** }

The current speed multiplier set in the UI.

Game Scenes Flight

Boolean **IsMoving** { **get;** }

Whether the servo is moving.

Game Scenes Flight

Boolean **IsFreeMoving** { **get;** }

Whether the servo is freely moving.

Game Scenes Flight

Boolean **IsLocked** { **get; set;** }

Whether the servo is locked.

Game Scenes Flight

Boolean **IsAxisInverted** { **get; set;** }

Whether the servos axis is inverted.

Game Scenes Flight

void **MoveRight** ()

Moves the servo to the right.

Game Scenes Flight

void **MoveLeft** ()

Moves the servo to the left.

Game Scenes Flight

void **MoveCenter** ()

Moves the servo to the center.

Game Scenes Flight

void **MoveNextPreset** ()

Moves the servo to the next preset.

Game Scenes Flight

void **MovePrevPreset** ()

Moves the servo to the previous preset.

Game Scenes Flight

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.

Game Scenes Flight

void **Stop** ()

Stops the servo.

Game Scenes Flight

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.

```
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);
```

(continues on next page)

(continued from previous page)

```
            group.MoveRight ();
            Thread.Sleep (1000);
            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
This service provides functionality to interact with Kerbal Alarm Clock.

Boolean **Available** { **get;** }
Whether Kerbal Alarm Clock is available.

Game Scenes All

IList<*Alarm*> **Alarms** { **get;** }
A list of all the alarms.

Game Scenes All

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.

Game Scenes All

IList<*Alarm*> **AlarmsWithType** (*AlarmType type*)
Get a list of alarms of the specified *type*.

Parameters

- **type** – Type of alarm to return.

Game Scenes All

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.

Game Scenes All

4.6.2 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.

Game Scenes All

Double **Margin** { **get; set;** }

The number of seconds before the event that the alarm will fire.

Game Scenes All

Double **Time** { **get; set;** }

The time at which the alarm will fire.

Game Scenes All

AlarmType **Type** { **get;** }

The type of the alarm.

Game Scenes All

String **ID** { **get;** }

The unique identifier for the alarm.

Game Scenes All

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

The short name of the alarm.

Game Scenes All

String **Notes** { **get; set;** }

The long description of the alarm.

Game Scenes All

Double **Remaining** { **get;** }

The number of seconds until the alarm will fire.

Game Scenes All

Boolean **Repeat** { **get; set;** }

Whether the alarm will be repeated after it has fired.

Game Scenes All

Double **RepeatPeriod** { **get; set;** }

The time delay to automatically create an alarm after it has fired.

Game Scenes All

SpaceCenter.Vessel **Vessel** { **get; set;** }

The vessel that the alarm is attached to.

Game Scenes All

SpaceCenter.CelestialBody **XferOriginBody** { **get; set;** }
The celestial body the vessel is departing from.

Game Scenes All

SpaceCenter.CelestialBody **XferTargetBody** { **get; set;** }
The celestial body the vessel is arriving at.

Game Scenes All

void **Remove** ()
Removes the alarm.

Game Scenes All

4.6.3 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).

LaunchRendevous
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*.

4.6.4 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.

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.

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

(continues on next page)

(continued from previous page)

```
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;
        }
    }
}
```

4.7 RemoteTech API

Provides RPCs to interact with the RemoteTech mod. Provides the following classes:

4.7.1 RemoteTech

class RemoteTech

This service provides functionality to interact with RemoteTech.

Boolean **Available** { **get;** }

Whether RemoteTech is installed.

Game Scenes All

IList<String> **GroundStations** { **get;** }

The names of the ground stations.

Game Scenes All

Antenna **Antenna** (*SpaceCenter.Part part*)

Get the antenna object for a particular part.

Parameters

Game Scenes All

Comms **Comms** (*SpaceCenter.Vessel vessel*)

Get a communications object, representing the communication capability of a particular vessel.

Parameters

Game Scenes All

4.7.2 Comms

class Comms
Communications for a vessel.

SpaceCenter.Vessel **Vessel** { **get;** }
Get the vessel.

Game Scenes All

Boolean **HasLocalControl** { **get;** }
Whether the vessel can be controlled locally.

Game Scenes All

Boolean **HasFlightComputer** { **get;** }
Whether the vessel has a flight computer on board.

Game Scenes All

Boolean **HasConnection** { **get;** }
Whether the vessel has any connection.

Game Scenes All

Boolean **HasConnectionToGroundStation** { **get;** }
Whether the vessel has a connection to a ground station.

Game Scenes All

Double **SignalDelay** { **get;** }
The shortest signal delay to the vessel, in seconds.

Game Scenes All

Double **SignalDelayToGroundStation** { **get;** }
The signal delay between the vessel and the closest ground station, in seconds.

Game Scenes All

Double **SignalDelayToVessel** (*SpaceCenter.Vessel other*)
The signal delay between the this vessel and another vessel, in seconds.

Parameters

Game Scenes All

IList<*Antenna*> **Antennas** { **get;** }
The antennas for this vessel.

Game Scenes All

4.7.3 Antenna

class Antenna
A RemoteTech antenna. Obtained by calling `Comms.Antennas` or `RemoteTech.Antenna`.

SpaceCenter.Part **Part** { **get;** }
Get the part containing this antenna.

Game Scenes All

Boolean **HasConnection** { **get;** }
Whether the antenna has a connection.

Game Scenes All

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*.

Game Scenes All

SpaceCenter.CelestialBody **TargetBody** { **get; set;** }
The celestial body the antenna is targetting.

Game Scenes All

String **TargetGroundStation** { **get; set;** }
The ground station the antenna is targetting.

Game Scenes All

SpaceCenter.Vessel **TargetVessel** { **get; set;** }
The vessel the antenna is targetting.

Game Scenes All

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.

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.

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

class RemoteTechExample
```

(continues on next page)

(continued from previous page)

```
{
    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);
        }
    }
}
```

4.8 User Interface API

4.8.1 UI

class UI

Provides functionality for drawing and interacting with in-game user interface elements.

Canvas **StockCanvas** { **get;** }

The stock UI canvas.

Game Scenes All

Canvas **AddCanvas** ()

Add a new canvas.

Game Scenes All

Note: If you want to add UI elements to KSPs stock UI canvas, use *UI.StockCanvas*.

void **Message** (String *content*, Single *duration = 1.0f*, *MessagePosition position = 1*, Tuple<Double, Double, Double> *color = { 1.0, 0.92, 0.016 }*, Single *size = 20.0f*)

Display a message on the screen.

Parameters

- **content** – Message content.
- **duration** – Duration before the message disappears, in seconds.

- **position** – Position to display the message.
- **color** – The color of the message.
- **size** – Size of the message, differs per position.

Game Scenes All

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.

Game Scenes All

enum MessagePosition
Message position.

TopLeft
Top left.

TopCenter
Top center.

TopRight
Top right.

BottomCenter
Bottom center.

4.8.2 Canvas

class Canvas
A canvas for user interface elements. See `UI.StockCanvas` and `UI.AddCanvas`.

RectTransform **RectTransform** { **get;** }
The rect transform for the canvas.

Game Scenes All

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

Game Scenes All

Panel **AddPanel** (Boolean *visible* = *true*)
Create a new container for user interface elements.

Parameters

- **visible** – Whether the panel is visible.

Game Scenes All

Text **AddText** (String *content*, Boolean *visible = true*)
Add text to the canvas.

Parameters

- **content** – The text.
- **visible** – Whether the text is visible.

Game Scenes All

InputField **AddInputField** (Boolean *visible = true*)
Add an input field to the canvas.

Parameters

- **visible** – Whether the input field is visible.

Game Scenes All

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.

Game Scenes All

void **Remove** ()
Remove the UI object.

Game Scenes All

4.8.3 Panel

class Panel
A container for user interface elements. See `Canvas.AddPanel`.

RectTransform **RectTransform** { **get;** }
The rect transform for the panel.

Game Scenes All

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

Game Scenes All

Panel **AddPanel** (Boolean *visible = true*)
Create a panel within this panel.

Parameters

- **visible** – Whether the new panel is visible.

Game Scenes All

Text **AddText** (String *content*, Boolean *visible = true*)
Add text to the panel.

Parameters

- **content** – The text.

- **visible** – Whether the text is visible.

Game Scenes All

InputField **AddInputField** (Boolean *visible* = *true*)
Add an input field to the panel.

Parameters

- **visible** – Whether the input field is visible.

Game Scenes All

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.

Game Scenes All

void **Remove** ()
Remove the UI object.

Game Scenes All

4.8.4 Text

class Text
A text label. See *Panel.AddText*.

RectTransform **RectTransform** { **get;** }
The rect transform for the text.

Game Scenes All

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

Game Scenes All

String **Content** { **get; set;** }
The text string

Game Scenes All

String **Font** { **get; set;** }
Name of the font

Game Scenes All

IList<String> **AvailableFonts** { **get;** }
A list of all available fonts.

Game Scenes All

Int32 **Size** { **get; set;** }
Font size.

Game Scenes All

FontStyle **Style** { **get; set;** }
Font style.

Game Scenes All

Tuple<Double, Double, Double> **Color** { **get; set;** }
Set the color

Game Scenes All

TextAnchor **Alignment** { **get; set;** }
Alignment.

Game Scenes All

Single **LineSpacing** { **get; set;** }
Line spacing.

Game Scenes All

void **Remove** ()
Remove the UI object.

Game Scenes All

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.

4.8.5 Button

class Button
A text label. See *Panel.AddButton*.

RectTransform **RectTransform** { **get;** }
The rect transform for the text.

Game Scenes All

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

Game Scenes All

Text **Text** { **get;** }
The text for the button.

Game Scenes All

Boolean **Clicked** { **get; set;** }
Whether the button has been clicked.

Game Scenes All

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.

Game Scenes All

4.8.6 InputField

class InputField
An input field. See *Panel.AddInputField*.

RectTransform **RectTransform** { **get;** }
The rect transform for the input field.

Game Scenes All

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

Game Scenes All

String **Value** { **get; set;** }
The value of the input field.

Game Scenes All

Text **Text** { **get;** }
The text component of the input field.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

4.8.7 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.

Game Scenes All

Tuple<Double, Double, Double> **LocalPosition** { **get; set;** }
Position of the rectangles pivot point relative to the anchors.

Game Scenes All

Tuple<Double, Double> **Size** { **get; set;** }
Width and height of the rectangle.

Game Scenes All

Tuple<Double, Double> **UpperRight** { **get; set;** }
Position of the rectangles upper right corner relative to the anchors.

Game Scenes All

Tuple<Double, Double> **`LowerLeft`** { **`get; set;`** }
Position of the rectangles lower left corner relative to the anchors.

Game Scenes All

Tuple<Double, Double> **`Anchor`** { **`set;`** }
Set the minimum and maximum anchor points as a fraction of the size of the parent rectangle.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Tuple<Double, Double, Double, Double> **`Rotation`** { **`get; set;`** }
Rotation, as a quaternion, of the object around its pivot point.

Game Scenes All

Tuple<Double, Double, Double> **`Scale`** { **`get; set;`** }
Scale factor applied to the object in the x, y and z dimensions.

Game Scenes All

CHAPTER

FIVE

C++

5.1 C++ Client

This client provides a C++ API for interacting with a kRPC server.

5.1.1 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`.

5.1.2 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() <<␣
↪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.

5.1.3 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;
}
```

5.1.4 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;
}
```

5.1.5 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.

5.1.6 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() {
```

(continues on next page)

(continued from previous page)

```
  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.

5.1.7 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();
}
```

5.1.8 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()`.

~Client()

Destructs the client object and 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.

std::condition_variable &**get_stream_update_condition**() **const**

A condition variable that is notified whenever a stream update message finishes processing.

std::unique_lock<std::mutex> &**get_stream_update_condition_lock**() **const**

The lock for the stream update condition variable.

void **acquire_stream_update**()

Acquires a lock on the mutex for the stream update condition variable.

void **release_stream_update**()

Releases the lock on the mutex for the stream update condition variable.

void **wait_for_stream_update**(double *timeout* = -1)

This method blocks until a stream update message finishes processing or the operation times out.

The stream update condition variable must be locked (by calling `acquire_stream_update`) 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.

int **add_callback**(**const** std::function<void)

> &*callback*Adds a callback function that is invoked whenever a stream update message finishes processing. Returns a unique identifier for the callback which can be used to remove it.

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_callback**(int *tag*)

Removes a stream update callback function. The tag is the identifier returned when the callback was added.

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.

float **rate** () **const**

void **setRate** (float *rate*)

The update rate of the stream in Hertz. When set to zero, the rate is unlimited.

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).

int **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. Returns a unique identifier for the callback which can be used to remove it.

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_callback** (int *tag*)

Removes a callback from the stream. The tag is the identifier returned when the callback was added.

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.

int **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. Returns a unique identifier for the callback which can be used to remove the callback.

void **remove_callback**(int *tag*)
Removes a callback from the event. The tag is the identifier returned when the callback was added.

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.

5.2 KRPC API

5.2.1 KRPC

None 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.

Game Scenes All

std::string **get_client_name**()

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

Game Scenes All

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.

Game Scenes All

krpc::schema::Status **get_status**()

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

Game Scenes All

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.

Game Scenes All

GameScene **current_game_scene**()

Get the current game scene.

Game Scenes All

bool **paused**()

void **set_paused**(bool *value*)

Whether the game is paused.

Game Scenes All

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.

5.2.2 Expressions

class Expression
A server side expression.

static *Expression* **constant_double** (*Client &connection*, double *value*)
A constant value of double precision floating point type.

Parameters

Game Scenes All

static *Expression* **constant_float** (*Client &connection*, float *value*)
A constant value of single precision floating point type.

Parameters

Game Scenes All

static *Expression* **constant_int** (*Client &connection*, int32_t *value*)
A constant value of integer type.

Parameters

Game Scenes All

static *Expression* **constant_bool** (*Client &connection*, bool *value*)
A constant value of boolean type.

Parameters

Game Scenes All

static *Expression* **constant_string** (*Client &connection*, std::string *value*)
A constant value of string type.

Parameters

Game Scenes All

static *Expression* **call** (*Client &connection*, krpc::schema::ProcedureCall *call*)
An RPC call.

Parameters

Game Scenes All

static *Expression* **equal** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Equality comparison.

Parameters

Game Scenes All

static *Expression* **not_equal** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Inequality comparison.

Parameters

Game Scenes All

static *Expression* **greater_than** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Greater than numerical comparison.

Parameters

Game Scenes All

static *Expression* **greater_than_or_equal** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Greater than or equal numerical comparison.

Parameters

Game Scenes All

static *Expression* **less_than** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Less than numerical comparison.

Parameters

Game Scenes All

static *Expression* **less_than_or_equal** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Less than or equal numerical comparison.

Parameters

Game Scenes All

static *Expression* **and__** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Boolean and operator.

Parameters

Game Scenes All

static *Expression* **or__** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Boolean or operator.

Parameters

Game Scenes All

static *Expression* **exclusive_or** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Boolean exclusive-or operator.

Parameters

Game Scenes All

static *Expression* **not__** (*Client &connection*, *Expression arg*)
Boolean negation operator.

Parameters

Game Scenes All

`static` *Expression* **add** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Numerical addition.

Parameters

Game Scenes All

`static` *Expression* **subtract** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Numerical subtraction.

Parameters

Game Scenes All

`static` *Expression* **multiply** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Numerical multiplication.

Parameters

Game Scenes All

`static` *Expression* **divide** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Numerical division.

Parameters

Game Scenes All

`static` *Expression* **modulo** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Numerical modulo operator.

Parameters

Returns The remainder of arg0 divided by arg1

Game Scenes All

`static` *Expression* **power** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Numerical power operator.

Parameters

Returns arg0 raised to the power of arg1, with type of arg0

Game Scenes All

`static` *Expression* **left_shift** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Bitwise left shift.

Parameters

Game Scenes All

`static` *Expression* **right_shift** (*Client &connection*, *Expression arg0*, *Expression arg1*)
Bitwise right shift.

Parameters

Game Scenes All

`static` *Expression* **cast** (*Client &connection*, *Expression arg*, *Type type*)
Perform a cast to the given type.

Parameters

- **type** – Type to cast the argument to.

Game Scenes All

static *Expression* **parameter** (*Client &connection*, std::string *name*, *Type type*)
A named parameter of type double.

Parameters

- **name** – The name of the parameter.
- **type** – The type of the parameter.

Returns A named parameter.

Game Scenes All

static *Expression* **function** (*Client &connection*, std::vector<*Expression*> *parameters*, *Expression body*)
A function.

Parameters

- **parameters** – The parameters of the function.
- **body** – The body of the function.

Returns A function.

Game Scenes All

static *Expression* **invoke** (*Client &connection*, *Expression function*, std::map<std::string, *Expression*> *args*)
A function call.

Parameters

- **function** – The function to call.
- **args** – The arguments to call the function with.

Returns A function call.

Game Scenes All

static *Expression* **create_tuple** (*Client &connection*, std::vector<*Expression*> *elements*)
Construct a tuple.

Parameters

- **elements** – The elements.

Returns The tuple.

Game Scenes All

static *Expression* **create_list** (*Client &connection*, std::vector<*Expression*> *values*)
Construct a list.

Parameters

- **values** – The value. Should all be of the same type.

Returns The list.

Game Scenes All

static *Expression* **create_set** (*Client &connection*, std::set<*Expression*> *values*)
Construct a set.

Parameters

- **values** – The values. Should all be of the same type.

Returns The set.

Game Scenes All

`static` *Expression* **`create_dictionary`**(*Client &connection*, std::vector<*Expression*> *keys*, std::vector<*Expression*> *values*)
Construct a dictionary, from a list of corresponding keys and values.

Parameters

- **keys** – The keys. Should all be of the same type.
- **values** – The values. Should all be of the same type.

Returns The dictionary.

Game Scenes All

`static` *Expression* **`to_list`**(*Client &connection*, *Expression arg*)
Convert a collection to a list.

Parameters

- **arg** – The collection.

Returns The collection as a list.

Game Scenes All

`static` *Expression* **`to_set`**(*Client &connection*, *Expression arg*)
Convert a collection to a set.

Parameters

- **arg** – The collection.

Returns The collection as a set.

Game Scenes All

`static` *Expression* **`get`**(*Client &connection*, *Expression arg*, *Expression index*)
Access an element in a tuple, list or dictionary.

Parameters

- **arg** – The tuple, list or dictionary.
- **index** – The index of the element to access. A zero indexed integer for a tuple or list, or a key for a dictionary.

Returns The element.

Game Scenes All

`static` *Expression* **`count`**(*Client &connection*, *Expression arg*)
Number of elements in a collection.

Parameters

- **arg** – The list, set or dictionary.

Returns The number of elements in the collection.

Game Scenes All

`static` *Expression* **`sum`**(*Client &connection*, *Expression arg*)
Sum all elements of a collection.

Parameters

- **arg** – The list or set.

Returns The sum of the elements in the collection.

Game Scenes All

`static` *Expression* **`max`** (*Client &connection*, *Expression arg*)
Maximum of all elements in a collection.

Parameters

- **arg** – The list or set.

Returns The maximum elements in the collection.

Game Scenes All

`static` *Expression* **`min`** (*Client &connection*, *Expression arg*)
Minimum of all elements in a collection.

Parameters

- **arg** – The list or set.

Returns The minimum elements in the collection.

Game Scenes All

`static` *Expression* **`average`** (*Client &connection*, *Expression arg*)
Minimum of all elements in a collection.

Parameters

- **arg** – The list or set.

Returns The minimum elements in the collection.

Game Scenes All

`static` *Expression* **`select`** (*Client &connection*, *Expression arg*, *Expression func*)
Run a function on every element in the collection.

Parameters

- **arg** – The list or set.
- **func** – The function.

Returns The modified collection.

Game Scenes All

`static` *Expression* **`where`** (*Client &connection*, *Expression arg*, *Expression func*)
Run a function on every element in the collection.

Parameters

- **arg** – The list or set.
- **func** – The function.

Returns The modified collection.

Game Scenes All

`static` *Expression* **`contains`** (*Client &connection*, *Expression arg*, *Expression value*)
Determine if a collection contains a value.

Parameters

- **arg** – The collection.
- **value** – The value to look for.

Returns Whether the collection contains a value.

Game Scenes All

`static` *Expression* **`aggregate`** (*Client &connection*, *Expression arg*, *Expression func*)

Applies an accumulator function over a sequence.

Parameters

- **arg** – The collection.
- **func** – The accumulator function.

Returns The accumulated value.

Game Scenes All

`static` *Expression* **`aggregate_with_seed`** (*Client &connection*, *Expression arg*, *Expression seed*, *Expression func*)

Applies an accumulator function over a sequence, with a given seed.

Parameters

- **arg** – The collection.
- **seed** – The seed value.
- **func** – The accumulator function.

Returns The accumulated value.

Game Scenes All

`static` *Expression* **`concat`** (*Client &connection*, *Expression arg1*, *Expression arg2*)

Concatenate two sequences.

Parameters

- **arg1** – The first sequence.
- **arg2** – The second sequence.

Returns The first sequence followed by the second sequence.

Game Scenes All

`static` *Expression* **`order_by`** (*Client &connection*, *Expression arg*, *Expression key*)

Order a collection using a key function.

Parameters

- **arg** – The collection to order.
- **key** – A function that takes a value from the collection and generates a key to sort on.

Returns The ordered collection.

Game Scenes All

`static` *Expression* **`all`** (*Client &connection*, *Expression arg*, *Expression predicate*)

Determine whether all items in a collection satisfy a boolean predicate.

Parameters

- **arg** – The collection.
- **predicate** – The predicate function.

Returns Whether all items satisfy the predicate.

Game Scenes All

`static` *Expression* **any** (*Client &connection*, *Expression arg*, *Expression predicate*)
Determine whether any item in a collection satisfies a boolean predicate.

Parameters

- **arg** – The collection.
- **predicate** – The predicate function.

Returns Whether any item satisfies the predicate.

Game Scenes All

`class Type`
A server side expression.

`static` *Type* **double__** (*Client &connection*)
Double type.

Game Scenes All

`static` *Type* **float__** (*Client &connection*)
Float type.

Game Scenes All

`static` *Type* **int__** (*Client &connection*)
Int type.

Game Scenes All

`static` *Type* **bool__** (*Client &connection*)
Bool type.

Game Scenes All

`static` *Type* **string** (*Client &connection*)
String type.

Game Scenes All

5.3 SpaceCenter API

5.3.1 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.

float **science** ()
The current amount of science.

Game Scenes All

double **funds**()
The current amount of funds.

Game Scenes All

float **reputation**()
The current amount of reputation.

Game Scenes All

Vessel **active_vessel**()

void **set_active_vessel**(*Vessel value*)
The currently active vessel.

Game Scenes Flight

std::vector<*Vessel*> **vessels**()
A list of all the vessels in the game.

Game Scenes All

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.

Game Scenes All

CelestialBody **target_body**()

void **set_target_body**(*CelestialBody value*)
The currently targeted celestial body.

Game Scenes Flight

Vessel **target_vessel**()

void **set_target_vessel**(*Vessel value*)
The currently targeted vessel.

Game Scenes Flight

DockingPort **target_docking_port**()

void **set_target_docking_port**(*DockingPort value*)
The currently targeted docking port.

Game Scenes Flight

void **clear_target**()
Clears the current target.

Game Scenes Flight

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"`.

Game Scenes All

void **launch_vessel**(std::string *craft_directory*, std::string *name*, std::string *launch_site*, bool *recover* = true)
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"`.
- **recover** – If true and there is a vessel on the launch site, recover it before launching.

Game Scenes All

Note: Throws an exception if any of the games pre-flight checks fail.

void **launch_vessel_from_vab** (std::string *name*, bool *recover* = true)
Launch a new vessel from the VAB onto the launchpad.

Parameters

- **name** – Name of the vessel to launch.
- **recover** – If true and there is a vessel on the launch pad, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *launch_vessel()* with the craft directory set to "VAB" and the launch site set to "LaunchPad". Throws an exception if any of the games pre-flight checks fail.

void **launch_vessel_from_sph** (std::string *name*, bool *recover* = true)
Launch a new vessel from the SPH onto the runway.

Parameters

- **name** – Name of the vessel to launch.
- **recover** – If true and there is a vessel on the runway, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *launch_vessel()* with the craft directory set to "SPH" and the launch site set to "Runway". Throws an exception if any of the games pre-flight checks fail.

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

Note: This is the same as calling *save()* with the name "quicksave".

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

Game Scenes All

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.

Game Scenes Flight

bool **navball**()

void **set_navball**(bool *value*)
Whether the navball is visible.

Game Scenes Flight

double **ut**()
The current universal time in seconds.

Game Scenes All

double **g**()
The value of the gravitational constant G in $N(m/kg)^2$.

Game Scenes All

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.

Game Scenes Flight

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()*.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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 covert the position to.

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

Game Scenes All

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 covert the direction to.

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

Game Scenes All

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 covert the rotation to.

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

Game Scenes All

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 covert the velocity to.

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

Game Scenes All

double **raycast_distance** (std::tuple<double, double, double> *position*, std::tuple<double, double, double> *direction*, *ReferenceFrame reference_frame*)

Cast a ray from a given position in a given direction, and return the distance to the hit point. If no hit occurs, returns infinity.

Parameters

- **position** – Position, as a vector, of the origin of the ray.
- **direction** – Direction of the ray, as a unit vector.

- **reference_frame** – The reference frame that the position and direction are in.

Returns The distance to the hit, in meters, or infinity if there was no hit.

Game Scenes All

Part **raycast_part** (std::tuple<double, double, double> *position*, std::tuple<double, double, double> *direction*, *ReferenceFrame reference_frame*)

Cast a ray from a given position in a given direction, and return the part that it hits. If no hit occurs, returns `NULL`.

Parameters

- **position** – Position, as a vector, of the origin of the ray.
- **direction** – Direction of the ray, as a unit vector.
- **reference_frame** – The reference frame that the position and direction are in.

Returns The part that was hit or `NULL` if there was no hit.

Game Scenes Flight

bool **far_available** ()

Whether Ferram Aerospace Research is installed.

Game Scenes All

GameMode **game_mode** ()

The current mode the game is in.

Game Scenes All

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.

Game Scenes Flight

Camera **camera** ()

An object that can be used to control the camera.

Game Scenes Flight

WaypointManager **waypoint_manager** ()

The waypoint manager.

Game Scenes Flight

ContractManager **contract_manager** ()

The contract manager.

Game Scenes All

enum struct GameMode

The game mode. Returned by *GameMode*

enumerator sandbox

Sandbox mode.

enumerator career

Career mode.

enumerator science

Science career mode.

enumerator science_sandbox
Science sandbox mode.

enumerator mission
Mission mode.

enumerator mission_builder
Mission builder mode.

enumerator scenario
Scenario mode.

enumerator scenario_non_resumable
Scenario mode that cannot be resumed.

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.

5.3.2 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**()

void **set_name**(std::string *value*)
The name of the vessel.

Game Scenes All

VesselType **type**()

void **set_type**(*VesselType value*)
The type of the vessel.

Game Scenes All

VesselSituation **situation**()
The situation the vessel is in.

Game Scenes All

bool **recoverable**()
Whether the vessel is recoverable.

Game Scenes All

void **recover**()
Recover the vessel.

Game Scenes All

double **met**()
The mission elapsed time in seconds.

Game Scenes All

std::string **biome**()

The name of the biome the vessel is currently in.

Game Scenes All

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()*).

Game Scenes 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 **orbit**()

The current orbit of the vessel.

Game Scenes All

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.

Game Scenes Flight

Comms **comms**()

Returns a *Comms* object that can be used to interact with CommNet for this vessel.

Game Scenes Flight

AutoPilot **auto_pilot**()

An *AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

Game Scenes Flight

int32_t **crew_capacity**()

The number of crew that can occupy the vessel.

Game Scenes All

int32_t **crew_count**()

The number of crew that are occupying the vessel.

Game Scenes All

std::vector<*CrewMember*> **crew**()

The crew in the vessel.

Game Scenes All

Resources **resources**()

A *Resources* object, that can used to get information about resources stored in the vessel.

Game Scenes Flight

Resources **resources_in_decouple_stage** (int32_t *stage*, bool *cumulative* = true)
Returns a *Resources* object, that can 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.

Game Scenes Flight

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.

Game Scenes Flight

float **mass** ()
The total mass of the vessel, including resources, in kg.

Game Scenes Flight

float **dry_mass** ()
The total mass of the vessel, excluding resources, in kg.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

float **specific_impulse** ()
The combined specific impulse of all active engines, in seconds. This is computed using the formula described here.

Game Scenes Flight

float **vacuum_specific_impulse**()

The combined vacuum specific impulse of all active engines, in seconds. This is computed using the formula described here.

Game Scenes Flight

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.

Game Scenes Flight

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*).

Game Scenes Flight

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.

Game Scenes All

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, gimballed 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.

Game Scenes Flight

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **available_reaction_wheel_torqu**

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.

Game Scenes Flight

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.

Game Scenes Flight

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **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 (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **available_control_surface_torq**

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.

Game Scenes Flight

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **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 (`ReferenceFrame`). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

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.

Game Scenes Flight

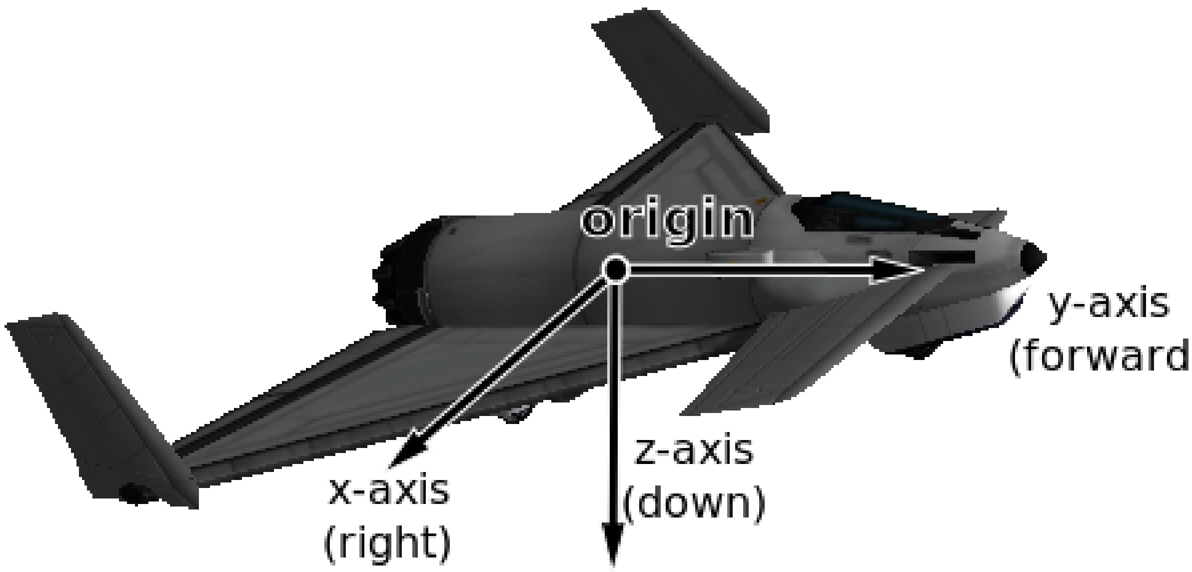


Fig. 1: Vessel reference frame origin and axes for the Aeris 3A aircraft

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.

Game Scenes Flight

Note: Be careful not to confuse this with 'orbit' mode on the navball.

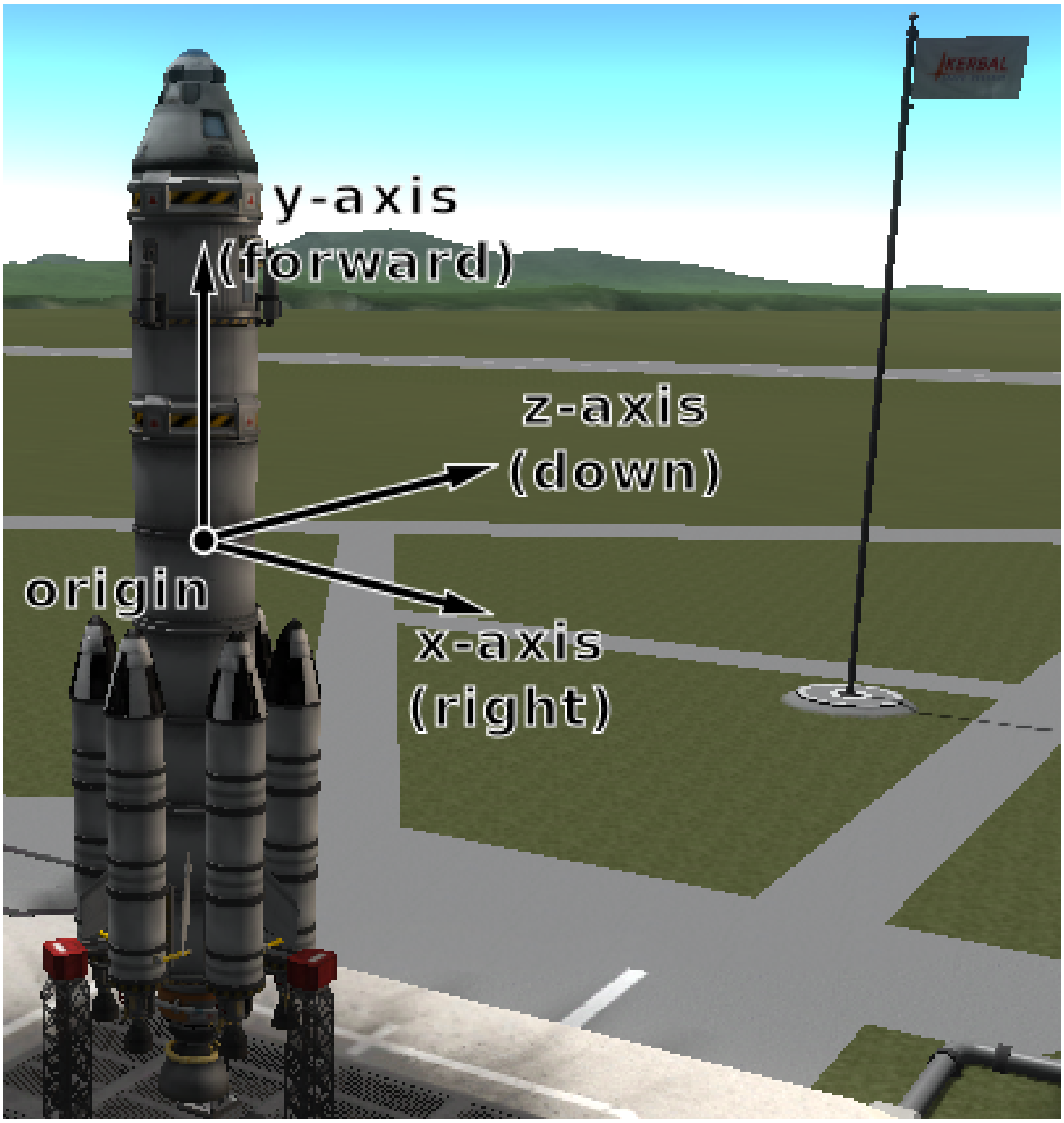


Fig. 2: Vessel reference frame origin and axes for the Kerbal-X rocket

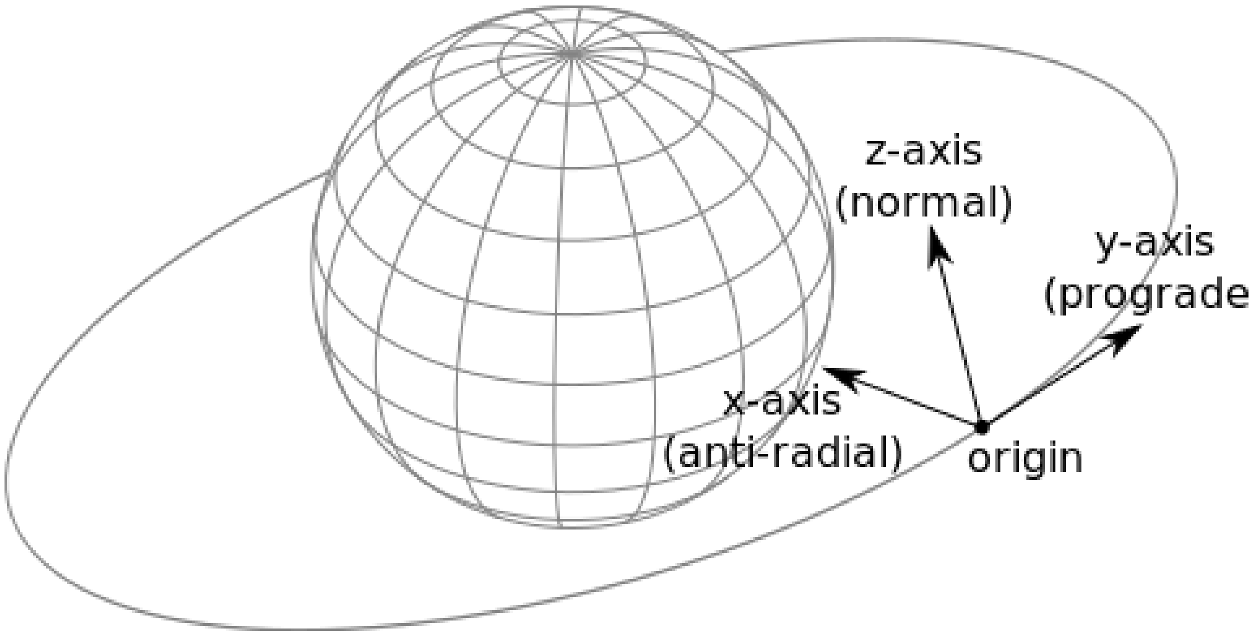


Fig. 3: Vessel orbital reference frame origin and axes

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).

Game Scenes Flight

Note: Be careful not to confuse this with 'surface' mode on the navball.

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.

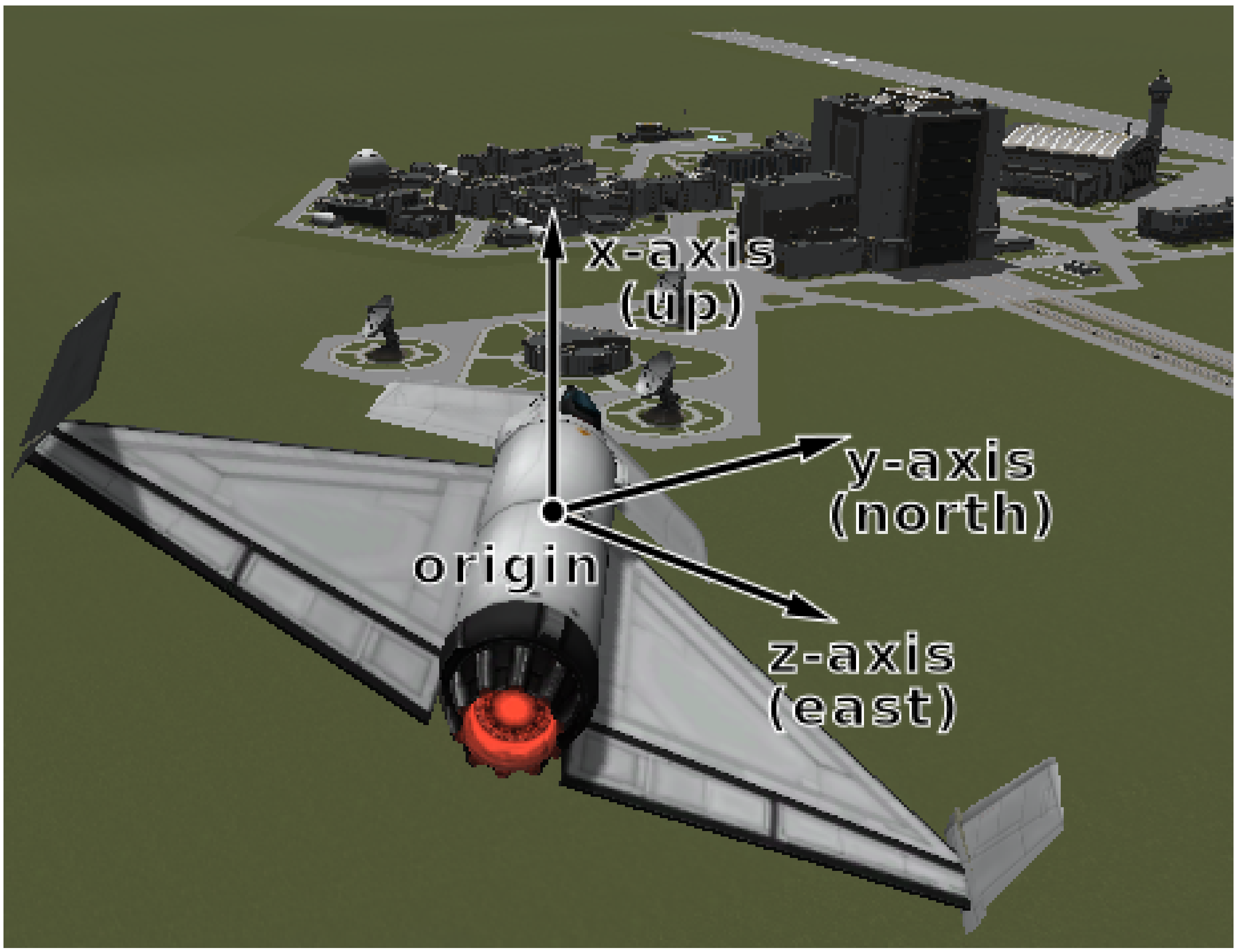


Fig. 4: Vessel surface reference frame origin and axes

- The x-axis is orthogonal to the other two axes.

Game Scenes Flight

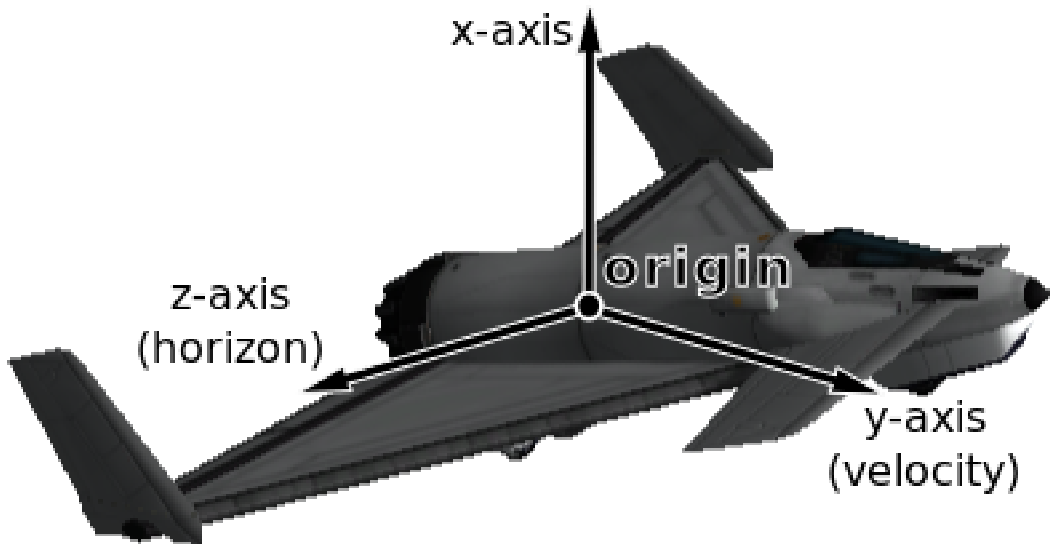


Fig. 5: Vessel surface velocity reference frame origin and axes

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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).

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

class CrewMember
Represents crew in a vessel. Can be obtained using *Vessel::crew()*.

std::string **name**()

void **set_name**(std::string *value*)
The crew members name.

Game Scenes All

CrewMemberType **type**()
The type of crew member.

Game Scenes All

bool **on_mission**()
Whether the crew member is on a mission.

Game Scenes All

float **courage**()

void **set_courage**(float *value*)
The crew members courage.

Game Scenes All

float **stupidity**()

void **set_stupidity**(float *value*)
The crew members stupidity.

Game Scenes All

float **experience**()

void **set_experience**(float *value*)
The crew members experience.

Game Scenes All

bool **badass**()

void **set_badass**(bool *value*)
Whether the crew member is a badass.

Game Scenes All

bool **veteran**()

void **set_veteran**(bool *value*)
Whether the crew member is a veteran.

Game Scenes All

enum struct CrewMemberType
The type of a crew member. See *CrewMember::type()*.

enumerator applicant
An applicant for crew.

enumerator crew
Rocket crew.

enumerator tourist
A tourist.

enumerator unowned
An unowned crew member.

5.3.3 CelestialBody

class CelestialBody
Represents a celestial body (such as a planet or moon). See *bodies()*.

std::string **name**()
The name of the body.

Game Scenes All

std::vector<*CelestialBody*> **satellites**()
A list of celestial bodies that are in orbit around this celestial body.

Game Scenes All

Orbit **orbit**()
The orbit of the body.

Game Scenes All

float **mass**()
The mass of the body, in kilograms.

Game Scenes All

float **gravitational_parameter**()
The standard gravitational parameter of the body in m^3s^{-2}.

Game Scenes All

float **surface_gravity**()
The acceleration due to gravity at sea level (mean altitude) on the body, in m/s^2.

Game Scenes All

float **rotational_period**()

The sidereal rotational period of the body, in seconds.

Game Scenes All

float **rotational_speed**()

The rotational speed of the body, in radians per second.

Game Scenes All

double **rotation_angle**()

The current rotation angle of the body, in radians. A value between 0 and 2π

Game Scenes All

double **initial_rotation**()

The initial rotation angle of the body (at UT 0), in radians. A value between 0 and 2π

Game Scenes All

float **equatorial_radius**()

The equatorial radius of the body, in meters.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

float **sphere_of_influence** ()

The radius of the sphere of influence of the body, in meters.

Game Scenes All

bool **has_atmosphere** ()

`true` if the body has an atmosphere.

Game Scenes All

float **atmosphere_depth** ()

The depth of the atmosphere, in meters.

Game Scenes All

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.

Game Scenes All

bool **has_atmospheric_oxygen** ()

`true` if there is oxygen in the atmosphere, required for air-breathing engines.

Game Scenes All

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.

Game Scenes All

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

Game Scenes All

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

Game Scenes All

std::set<std::string> **biomes** ()

The biomes present on this body.

Game Scenes All

std::string **biome_at** (double *latitude*, double *longitude*)

The biome at the given latitude and longitude, in degrees.

Parameters

Game Scenes All

float **flying_high_altitude_threshold** ()

The altitude, in meters, above which a vessel is considered to be flying "high" when doing science.

Game Scenes All

float **space_high_altitude_threshold** ()

The altitude, in meters, above which a vessel is considered to be in "high" space when doing science.

Game Scenes All

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.

Game Scenes All

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.

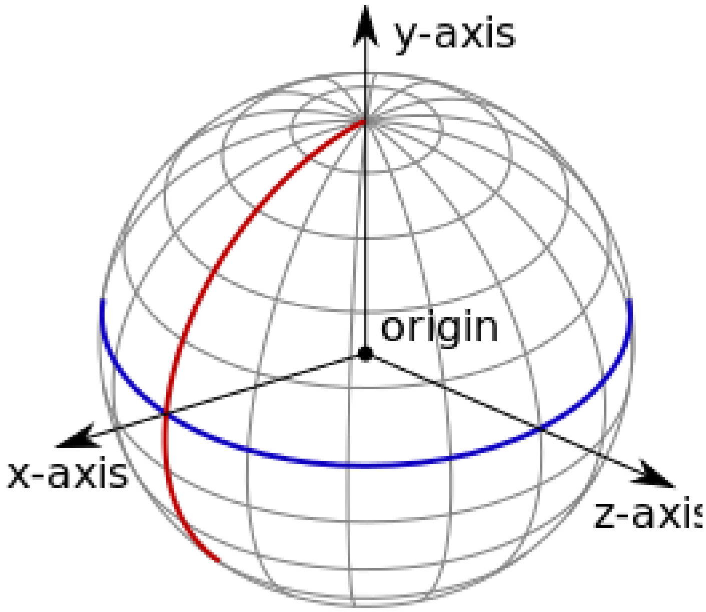


Fig. 6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

- 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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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).

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

5.3.4 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 g.

Game Scenes Flight

double **mean_altitude** ()
The altitude above sea level, in meters. Measured from the center of mass of the vessel.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

double **latitude**()
The latitude of the vessel for the body being orbited, in degrees.

Game Scenes Flight

double **longitude**()
The longitude of the vessel for the body being orbited, in degrees.

Game Scenes Flight

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.

Game Scenes Flight

double **speed**()
The speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Game Scenes Flight

double **horizontal_speed**()
The horizontal speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Game Scenes Flight

double **vertical_speed**()
The vertical speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Game Scenes Flight

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.

Game Scenes Flight

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).

Game Scenes Flight

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.

Game Scenes Flight

float **pitch**()
The pitch of the vessel relative to the horizon, in degrees. A value between -90° and +90°.

Game Scenes Flight

float **heading**()
The heading of the vessel (its angle relative to north), in degrees. A value between 0° and 360°.

Game Scenes Flight

float **roll**()
The roll of the vessel relative to the horizon, in degrees. A value between -180° and +180°.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

float **atmosphere_density**()
The current density of the atmosphere around the vessel, in kg/m^3.

Game Scenes Flight

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 $\frac{1}{2}$.air density.velocity^2. It is commonly denoted Q.

Game Scenes Flight

float **static_pressure**()

The static atmospheric pressure acting on the vessel, in Pascals.

Game Scenes Flight

float **static_pressure_at_msl**()

The static atmospheric pressure at mean sea level, in Pascals.

Game Scenes Flight

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.

Game Scenes Flight

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 where 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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

float **speed_of_sound**()

The speed of sound, in the atmosphere around the vessel, in m/s.

Game Scenes Flight

float **mach**()

The speed of the vessel, in multiples of the speed of sound.

Game Scenes Flight

float **reynolds_number**()

The vessels Reynolds number.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

float **true_air_speed**()

The true air speed of the vessel, in meters per second.

Game Scenes Flight

float **equivalent_air_speed**()

The equivalent air speed of the vessel, in meters per second.

Game Scenes Flight

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.

Game Scenes Flight

float **angle_of_attack**()

The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

Game Scenes Flight

float **sideslip_angle**()

The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

Game Scenes Flight

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.

Game Scenes Flight

float **static_air_temperature**()

The static (ambient) temperature of the atmosphere around the vessel, in Kelvin.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

float **ballistic_coefficient** ()

The ballistic coefficient.

Game Scenes Flight

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.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

5.3.5 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.

Game Scenes All

double **apoapsis** ()

Gets the apoapsis of the orbit, in meters, from the center of mass of the body being orbited.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

double **semi_minor_axis**()
The semi-minor axis of the orbit, in meters.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

double **speed**()
The current orbital speed of the object in meters per second.

Game Scenes All

Note: This value will change over time if the orbit is elliptical.

double **period** ()

The orbital period, in seconds.

Game Scenes All

double **time_to_apoapsis** ()

The time until the object reaches apoapsis, in seconds.

Game Scenes All

double **time_to_periapsis** ()

The time until the object reaches periapsis, in seconds.

Game Scenes All

double **eccentricity** ()

The eccentricity of the orbit.

Game Scenes All

double **inclination** ()

The inclination of the orbit, in radians.

Game Scenes All

double **longitude_of_ascending_node** ()

The longitude of the ascending node, in radians.

Game Scenes All

double **argument_of_periapsis** ()

The argument of periapsis, in radians.

Game Scenes All

double **mean_anomaly_at_epoch** ()

The mean anomaly at epoch.

Game Scenes All

double **epoch** ()

The time since the epoch (the point at which the mean anomaly at epoch was measured, in seconds.

Game Scenes All

double **mean_anomaly** ()

The mean anomaly.

Game Scenes All

double **mean_anomaly_at_ut** (double *ut*)

The mean anomaly at the given time.

Parameters

- **ut** – The universal time in seconds.

Game Scenes All

double **eccentric_anomaly** ()

The eccentric anomaly.

Game Scenes All

double **eccentric_anomaly_at_ut** (double *ut*)
The eccentric anomaly at the given universal time.

Parameters

- **ut** – The universal time, in seconds.

Game Scenes All

double **true_anomaly** ()
The true anomaly.

Game Scenes All

double **true_anomaly_at_ut** (double *ut*)
The true anomaly at the given time.

Parameters

- **ut** – The universal time in seconds.

Game Scenes All

double **true_anomaly_at_radius** (double *radius*)
The true anomaly at the given orbital radius.

Parameters

- **radius** – The orbital radius in meters.

Game Scenes All

double **ut_at_true_anomaly** (double *true_anomaly*)
The universal time, in seconds, corresponding to the given true anomaly.

Parameters

- **true_anomaly** – True anomaly.

Game Scenes All

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.

Game Scenes All

double **true_anomaly_at_an** (*Orbit target*)
The true anomaly of the ascending node with the given target orbit.

Parameters

- **target** – Target orbit.

Game Scenes All

double **true_anomaly_at_dn** (*Orbit target*)
The true anomaly of the descending node with the given target orbit.

Parameters

- **target** – Target orbit.

Game Scenes All

double **orbital_speed**()

The current orbital speed in meters per second.

Game Scenes All

double **orbital_speed_at**(double *time*)

The orbital speed at the given time, in meters per second.

Parameters

- **time** – Time from now, in seconds.

Game Scenes All

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.

Game Scenes All

static std::tuple<double, double, double> **reference_plane_direction**(*Client &connection*, *ReferenceFrame reference_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.

Game Scenes All

double **relative_inclination**(*Orbit target*)

Relative inclination of this orbit and the target orbit, in radians.

Parameters

- **target** – Target orbit.

Game Scenes All

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.

Game Scenes All

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`.

Game Scenes All

double **time_of_closest_approach**(*Orbit target*)

Estimates and returns the time at closest approach to a target orbit.

Parameters

- **target** – Target orbit.

Returns The universal time at closest approach, in seconds.

Game Scenes All

double **distance_at_closest_approach** (*Orbit target*)
Estimates and returns the distance at closest approach to a target orbit, in meters.

Parameters

- **target** – Target orbit.

Game Scenes All

std::vector<std::vector<double>> **list_closest_approaches** (*Orbit target*, int32_t *orbits*)
Returns the times at closest approach and corresponding distances, to a target orbit.

Parameters

- **target** – Target orbit.
- **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.

Game Scenes All

5.3.6 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.

Game Scenes Flight

ControlState **state** ()
The control state of the vessel.

Game Scenes Flight

bool **sas** ()

void **set_sas** (bool *value*)
The state of SAS.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

bool **rcs**()

void **set_rcs**(bool *value*)
The state of RCS.

Game Scenes Flight

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()*.

Game Scenes Flight

bool **gear**()

void **set_gear**(bool *value*)
The state of the landing gear/legs.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

bool **lights**()

void **set_lights**(bool *value*)
The state of the lights.

Game Scenes Flight

bool **brakes**()

void **set_brakes** (bool *value*)
The state of the wheel brakes.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

bool **abort**()

void **set_abort**(bool *value*)
The state of the abort action group.

Game Scenes Flight

float **throttle**()

void **set_throttle**(float *value*)
The state of the throttle. A value between 0 and 1.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

int32_t **current_stage**()

The current stage of the vessel. Corresponds to the stage number in the in-game UI.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

std::vector<*Node*> **nodes**()

Returns a list of all existing maneuver nodes, ordered by time from first to last.

Game Scenes Flight

void **remove_nodes**()

Remove all maneuver nodes.

Game Scenes Flight

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.

5.3.7 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.

Game Scenes Flight

bool **can_transmit_science**()
Whether the vessel can transmit science data to KSC.

Game Scenes Flight

double **signal_strength**()
Signal strength to KSC.

Game Scenes Flight

double **signal_delay**()
Signal delay to KSC in seconds.

Game Scenes Flight

double **power**()
The combined power of all active antennae on the vessel.

Game Scenes Flight

std::vector<*CommLink*> **control_path**()

The communication path used to control the vessel.

Game Scenes Flight

class CommLink

Represents a communication node in the network. For example, a vessel or the KSC.

CommLinkType **type**()

The type of link.

Game Scenes All

double **signal_strength**()

Signal strength of the link.

Game Scenes All

CommNode **start**()

Start point of the link.

Game Scenes All

CommNode **end**()

Start point of the link.

Game Scenes All

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.

Game Scenes All

bool **is_home**()

Whether the communication node is on Kerbin.

Game Scenes All

bool **is_control_point**()

Whether the communication node is a control point, for example a manned vessel.

Game Scenes All

bool **is_vessel**()

Whether the communication node is a vessel.

Game Scenes All

Vessel **vessel**()

The vessel for this communication node.

Game Scenes All

5.3.8 Parts

The following classes allow interaction with a vessels individual parts.

- *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.

Game Scenes All

Part **root**()

The vessels root part.

Game Scenes All

Note: See the discussion on *Trees of Parts*.

Part **controlling**()

void **set_controlling**(*Part value*)

The part from which the vessel is controlled.

Game Scenes All

std::vector<*Part*> **with_name**(std::string *name*)

A list of parts whose `Part::name()` is *name*.

Parameters

Game Scenes All

std::vector<*Part*> **with_title**(std::string *title*)

A list of all parts whose `Part::title()` is *title*.

Parameters

Game Scenes All

std::vector<*Part*> **with_tag**(std::string *tag*)

A list of all parts whose `Part::tag()` is *tag*.

Parameters

Game Scenes All

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

Game Scenes All

std::vector<*Part*> **in_stage**(int32_t *stage*)

A list of all parts that are activated in the given *stage*.

Parameters

Game Scenes All

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

Game Scenes All

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

Game Scenes All

std::vector<*Antenna*> **antennas** ()
A list of all antennas in the vessel.

Game Scenes All

std::vector<*CargoBay*> **cargo_bays** ()
A list of all cargo bays in the vessel.

Game Scenes All

std::vector<*ControlSurface*> **control_surfaces** ()
A list of all control surfaces in the vessel.

Game Scenes All

std::vector<*Decoupler*> **decouplers** ()
A list of all decouplers in the vessel.

Game Scenes All

std::vector<*DockingPort*> **docking_ports** ()
A list of all docking ports in the vessel.

Game Scenes All

std::vector<*Engine*> **engines** ()
A list of all engines in the vessel.

Game Scenes All

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.

Game Scenes All

std::vector<*Fairing*> **fairings** ()
A list of all fairings in the vessel.

Game Scenes All

std::vector<*Intake*> **intakes**()
A list of all intakes in the vessel.

Game Scenes All

std::vector<*Leg*> **legs**()
A list of all landing legs attached to the vessel.

Game Scenes All

std::vector<*LaunchClamp*> **launch_clamps**()
A list of all launch clamps attached to the vessel.

Game Scenes All

std::vector<*Light*> **lights**()
A list of all lights in the vessel.

Game Scenes All

std::vector<*Parachute*> **parachutes**()
A list of all parachutes in the vessel.

Game Scenes All

std::vector<*Radiator*> **radiators**()
A list of all radiators in the vessel.

Game Scenes All

std::vector<*RCS*> **rcs**()
A list of all RCS blocks/thrusters in the vessel.

Game Scenes All

std::vector<*ReactionWheel*> **reaction_wheels**()
A list of all reaction wheels in the vessel.

Game Scenes All

std::vector<*ResourceConverter*> **resource_converters**()
A list of all resource converters in the vessel.

Game Scenes All

std::vector<*ResourceHarvester*> **resource_harvesters**()
A list of all resource harvesters in the vessel.

Game Scenes All

std::vector<*Sensor*> **sensors**()
A list of all sensors in the vessel.

Game Scenes All

std::vector<*SolarPanel*> **solar_panels**()
A list of all solar panels in the vessel.

Game Scenes All

std::vector<*Wheel*> **wheels**()
A list of all wheels in the vessel.

Game Scenes All

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".

Game Scenes All

std::string **title**()

Title of the part, as shown when the part is right clicked in-game. For example "Mk1-2 Command Pod".

Game Scenes All

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.

Game Scenes All

Note: This string is shared with kOS if it is installed.

bool **highlighted**()

void **set_highlighted**(bool *value*)

Whether the part is highlighted.

Game Scenes All

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.

Game Scenes All

double **cost**()

The cost of the part, in units of funds.

Game Scenes All

Vessel **vessel**()

The vessel that contains this part.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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`.

Game Scenes All

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`.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Note: See the discussion on *Staging*.

bool **massless**()

Whether the part is massless.

Game Scenes All

double **mass**()

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

Game Scenes All

double **dry_mass**()

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

Game Scenes All

bool **shielded**()

Whether the part is shielded from the exterior of the vessel, for example by a fairing.

Game Scenes All

float **dynamic_pressure**()

The dynamic pressure acting on the part, in Pascals.

Game Scenes All

double **impact_tolerance**()

The impact tolerance of the part, in meters per second.

Game Scenes All

double **temperature**()

Temperature of the part, in Kelvin.

Game Scenes All

double **skin_temperature**()

Temperature of the skin of the part, in Kelvin.

Game Scenes All

double **max_temperature**()

Maximum temperature that the part can survive, in Kelvin.

Game Scenes All

double **max_skin_temperature**()

Maximum temperature that the skin of the part can survive, in Kelvin.

Game Scenes All

float **thermal_mass**()

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

Game Scenes All

float **thermal_skin_mass**()

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Resources **resources**()

A `Resources` object for the part.

Game Scenes All

bool **crossfeed**()

Whether this part is crossfeed capable.

Game Scenes All

bool **is_fuel_line**()

Whether this part is a fuel line.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Note: See the discussion on *Fuel Lines*.

std::vector<*Module*> **modules**()

The modules for this part.

Game Scenes All

Antenna **antenna**()

A `Antenna` if the part is an antenna, otherwise `NULL`.

Game Scenes All

CargoBay **cargo_bay**()
A *CargoBay* if the part is a cargo bay, otherwise NULL.

Game Scenes All

ControlSurface **control_surface**()
A *ControlSurface* if the part is an aerodynamic control surface, otherwise NULL.

Game Scenes All

Decoupler **decoupler**()
A *Decoupler* if the part is a decoupler, otherwise NULL.

Game Scenes All

DockingPort **docking_port**()
A *DockingPort* if the part is a docking port, otherwise NULL.

Game Scenes All

Engine **engine**()
An *Engine* if the part is an engine, otherwise NULL.

Game Scenes All

Experiment **experiment**()
An *Experiment* if the part is a science experiment, otherwise NULL.

Game Scenes All

Fairing **fairing**()
A *Fairing* if the part is a fairing, otherwise NULL.

Game Scenes All

Intake **intake**()
An *Intake* if the part is an intake, otherwise NULL.

Game Scenes All

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.

Game Scenes All

LaunchClamp **launch_clamp**()
A *LaunchClamp* if the part is a launch clamp, otherwise NULL.

Game Scenes All

Light **light**()
A *Light* if the part is a light, otherwise NULL.

Game Scenes All

Parachute **parachute**()
A *Parachute* if the part is a parachute, otherwise NULL.

Game Scenes All

Radiator **radiator**()
A *Radiator* if the part is a radiator, otherwise NULL.

Game Scenes All

RCS **rcs**()
A *RCS* if the part is an RCS block/thruster, otherwise NULL.

Game Scenes All

ReactionWheel **reaction_wheel**()
A *ReactionWheel* if the part is a reaction wheel, otherwise NULL.

Game Scenes All

ResourceConverter **resource_converter**()
A *ResourceConverter* if the part is a resource converter, otherwise NULL.

Game Scenes All

ResourceHarvester **resource_harvester**()
A *ResourceHarvester* if the part is a resource harvester, otherwise NULL.

Game Scenes All

Sensor **sensor**()
A *Sensor* if the part is a sensor, otherwise NULL.

Game Scenes All

SolarPanel **solar_panel**()
A *SolarPanel* if the part is a solar panel, otherwise NULL.

Game Scenes All

Wheel **wheel**()
A *Wheel* if the part is a wheel, otherwise NULL.

Game Scenes All

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.

Game Scenes All

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::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.

Game Scenes All

std::tuple<std::tuple<double, double, double>, std::tuple<double, double, double>> **bounding_box** (*ReferenceFrame reference_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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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).

Game Scenes All

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`).

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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()*.

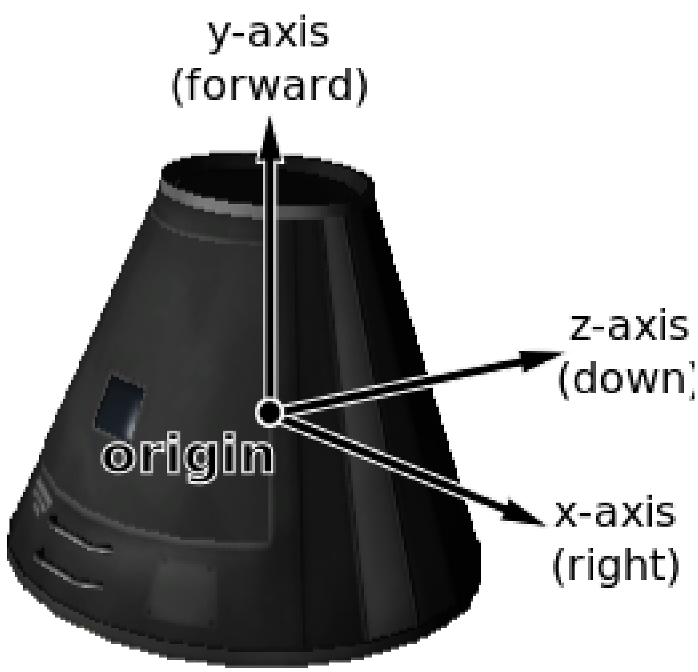


Fig. 7: Mk1 Command Pod reference frame origin and axes

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

ReferenceFrame **reference_frame** ()

void **set_reference_frame** (*ReferenceFrame value*)

The reference frame of the force vector and position.

Game Scenes All

void **remove**()
Remove the force.

Game Scenes All

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".

Game Scenes All

Part **part**()
The part that contains this module.

Game Scenes All

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.

Game Scenes All

bool **has_field**(std::string *name*)
Returns `true` if the module has a field with the given name.

Parameters

- **name** – Name of the field.

Game Scenes All

std::string **get_field**(std::string *name*)
Returns the value of a field.

Parameters

- **name** – Name of the field.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

void **reset_field** (std::string *name*)

Set the value of a field to its original value.

Parameters

- **name** – Name of the field.

Game Scenes All

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.

Game Scenes All

bool **has_event** (std::string *name*)

`true` if the module has an event with the given name.

Parameters

Game Scenes All

void **trigger_event** (std::string *name*)

Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters

Game Scenes All

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.

Game Scenes All

bool **has_action** (std::string *name*)

`true` if the part has an action with the given name.

Parameters

Game Scenes All

void **set_action** (std::string *name*, bool *value* = true)

Set the value of an action with the given name.

Parameters

Game Scenes All

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.

Game Scenes All

AntennaState **state**()

The current state of the antenna.

Game Scenes All

bool **deployable**()
Whether the antenna is deployable.

Game Scenes All

bool **deployed**()

void **set_deployed**(bool *value*)
Whether the antenna is deployed.

Game Scenes All

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.

Game Scenes All

void **transmit**()
Transmit data.

Game Scenes All

void **cancel**()
Cancel current transmission of data.

Game Scenes All

bool **allow_partial**()

void **set_allow_partial**(bool *value*)
Whether partial data transmission is permitted.

Game Scenes All

double **power**()
The power of the antenna.

Game Scenes All

bool **combinable**()
Whether the antenna can be combined with other antennae on the vessel to boost the power.

Game Scenes All

double **combinable_exponent**()
Exponent used to calculate the combined power of multiple antennae on a vessel.

Game Scenes All

float **packet_interval**()
Interval between sending packets in seconds.

Game Scenes All

float **packet_size**()
Amount of data sent per packet in Mits.

Game Scenes All

double **packet_resource_cost**()
Units of electric charge consumed per packet sent.

Game Scenes All

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.

Game Scenes All

CargoBayState **state**()
The state of the cargo bay.

Game Scenes All

bool **open**()

void **set_open**(bool *value*)
Whether the cargo bay is open.

Game Scenes All

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.

Game Scenes All

bool **pitch_enabled**()

void **set_pitch_enabled**(bool *value*)

Whether the control surface has pitch control enabled.

Game Scenes All

bool **yaw_enabled**()

void **set_yaw_enabled**(bool *value*)

Whether the control surface has yaw control enabled.

Game Scenes All

bool **roll_enabled**()

void **set_roll_enabled**(bool *value*)

Whether the control surface has roll control enabled.

Game Scenes All

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.

Game Scenes All

bool **inverted**()

void **set_inverted**(bool *value*)

Whether the control surface movement is inverted.

Game Scenes All

bool **deployed**()

void **set_deployed**(bool *value*)

Whether the control surface has been fully deployed.

Game Scenes All

float **surface_area**()

Surface area of the control surface in m^2.

Game Scenes All

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()*.

Game Scenes All

Decoupler

class Decoupler

A decoupler. Obtained by calling *Part::decoupler()*

Part **part** ()
The part object for this decoupler.

Game Scenes All

Vessel **decouple** ()
Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

Game Scenes All

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.

Game Scenes All

bool **staged** ()
Whether the decoupler is enabled in the staging sequence.

Game Scenes All

float **impulse** ()
The impulse that the decoupler imparts when it is fired, in Newton seconds.

Game Scenes All

Docking Port

class DockingPort
A docking port. Obtained by calling *Part::docking_port()*

Part **part** ()
The part object for this docking port.

Game Scenes All

DockingPortState **state** ()
The current state of the docking port.

Game Scenes All

Part **docked_part** ()
The part that this docking port is docked to. Returns `NULL` if this docking port is not docked to anything.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

bool **has_shield**()

Whether the docking port has a shield.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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).

Game Scenes All

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.

Game Scenes All

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by *Part::reference_frame()*.

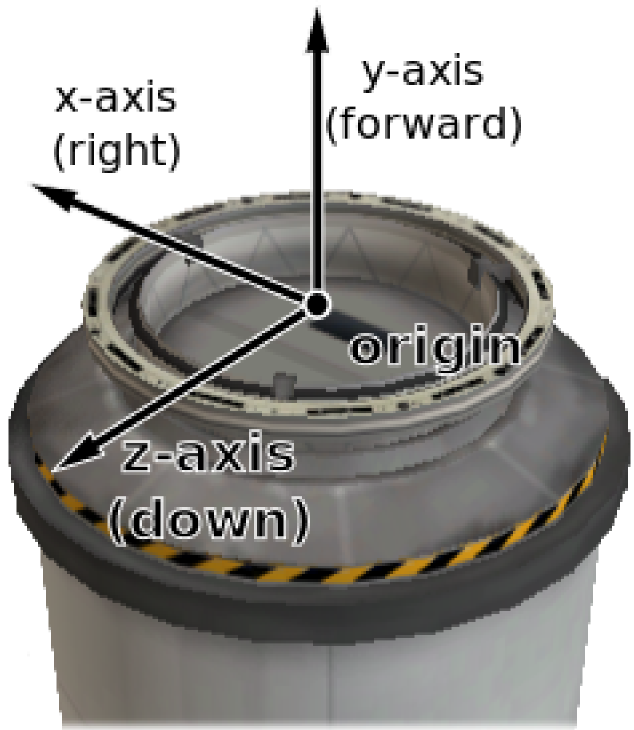


Fig. 8: Docking port reference frame origin and axes

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.

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.

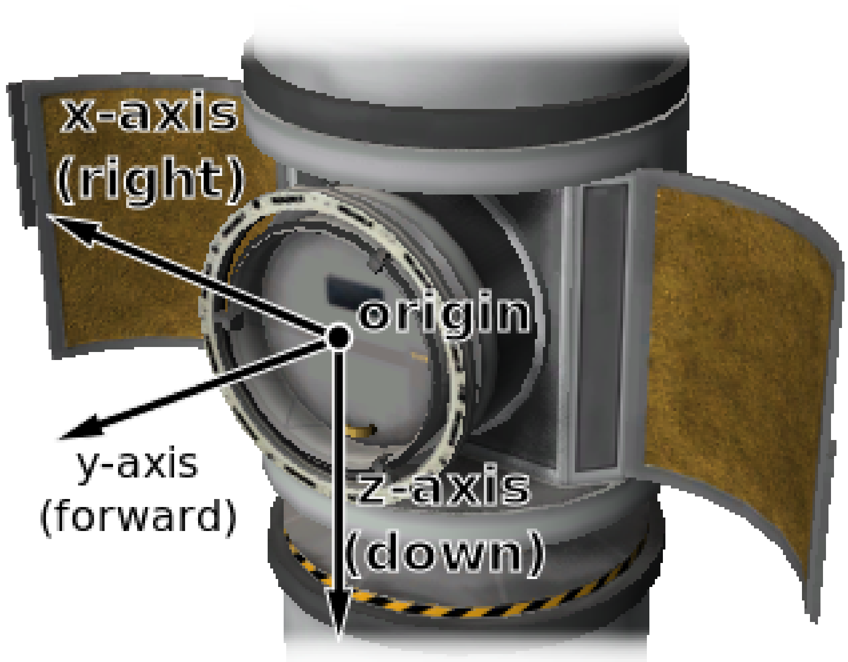


Fig. 9: Inline docking port reference frame origin and axes

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 **part**()

The part object for this engine.

Game Scenes All

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()`.

Game Scenes All

float **thrust**()

The current amount of thrust being produced by the engine, in Newtons.

Game Scenes All

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.

Game Scenes All

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%.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

std::vector<*Thruster*> **thrusters** ()

The components of the engine that generate thrust.

Game Scenes All

Note: For example, this corresponds to the rocket nozzel on a solid rocket booster, or the individual nozzels 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.

Game Scenes All

float **vacuum_specific_impulse** ()

The vacuum specific impulse of the engine, in seconds.

Game Scenes All

float **kerbin_sea_level_specific_impulse** ()

The specific impulse of the engine at sea level on Kerbin, in seconds.

Game Scenes All

std::vector<std::string> **propellant_names** ()

The names of the propellants that the engine consumes.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

bool **has_fuel**()
Whether the engine has any fuel available.

Game Scenes All

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).

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

bool **has_modes**()
Whether the engine has multiple modes of operation.

Game Scenes All

std::string **mode**()

void **set_mode**(std::string *value*)
The name of the current engine mode.

Game Scenes All

std::map<std::string, *Engine*> **modes**()
The available modes for the engine. A dictionary mapping mode names to *Engine* objects.

Game Scenes All

void **toggle_mode**()
Toggle the current engine mode.

Game Scenes All

bool **auto_mode_switch**()

void **set_auto_mode_switch**(bool *value*)
Whether the engine will automatically switch modes.

Game Scenes All

bool **gimballed**()
Whether the engine is gimballed.

Game Scenes All

float **gimbal_range**()
The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

class Propellant
A propellant for an engine. Obtains by calling *Engine::propellants()*.

std::string **name**()
The name of the propellant.

Game Scenes All

double **current_amount**()
The current amount of propellant.

Game Scenes All

double **current_requirement**()
The required amount of propellant.

Game Scenes All

double **total_resource_available**()
The total amount of the underlying resource currently reachable given resource flow rules.

Game Scenes All

double **total_resource_capacity**()
The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

Game Scenes All

bool **ignore_for_isp**()
If this propellant should be ignored when calculating required mass flow given specific impulse.

Game Scenes All

bool **ignore_for_thrust_curve**()
If this propellant should be ignored for thrust curve calculations.

Game Scenes All

bool **draw_stack_gauge**()
If this propellant has a stack gauge or not.

Game Scenes All

bool **is_deprived**()
If this propellant is deprived.

Game Scenes All

float **ratio**()
The propellant ratio.

Game Scenes All

Experiment

class Experiment
Obtained by calling *Part::experiment()*.

Part **part**()
The part object for this experiment.

Game Scenes All

void **run**()
Run the experiment.

Game Scenes All

void **transmit**()
Transmit all experimental data contained by this part.

Game Scenes All

void **dump**()
Dump the experimental data contained by the experiment.

Game Scenes All

void **reset**()
Reset the experiment.

Game Scenes All

bool **deployed**()
Whether the experiment has been deployed.

Game Scenes All

bool **rerunnable**()
Whether the experiment can be re-run.

Game Scenes All

bool **inoperable**()
Whether the experiment is inoperable.

Game Scenes All

bool **has_data**()

Whether the experiment contains data.

Game Scenes All

std::vector<*ScienceData*> **data**()

The data contained in this experiment.

Game Scenes All

std::string **biome**()

The name of the biome the experiment is currently in.

Game Scenes All

bool **available**()

Determines if the experiment is available given the current conditions.

Game Scenes All

ScienceSubject **science_subject**()

Containing information on the corresponding specific science result for the current conditions. Returns `NULL` if the experiment is unavailable.

Game Scenes All

class ScienceData

Obtained by calling *Experiment::data()*.

float **data_amount**()

Data amount.

Game Scenes All

float **science_value**()

Science value.

Game Scenes All

float **transmit_value**()

Transmit value.

Game Scenes All

class ScienceSubject

Obtained by calling *Experiment::science_subject()*.

std::string **title**()

Title of science subject, displayed in science archives

Game Scenes All

bool **is_complete**()

Whether the experiment has been completed.

Game Scenes All

float **science**()

Amount of science already earned from this subject, not updated until after transmission/recovery.

Game Scenes All

float **science_cap**()

Total science allowable for this subject.

Game Scenes All

float **data_scale**()
: Multiply science value by this to determine data amount in mits.

Game Scenes All

float **subject_value**()
: Multiplier for specific Celestial Body/Experiment Situation combination.

Game Scenes All

float **scientific_value**()
: Diminishing value multiplier for decreasing the science value returned from repeated experiments.

Game Scenes All

Fairing

class Fairing
: A fairing. Obtained by calling `Part::fairing()`.

Part **part**()
: The part object for this fairing.

Game Scenes All

void **jettison**()
: Jettison the fairing. Has no effect if it has already been jettisoned.

Game Scenes All

bool **jettisoned**()
: Whether the fairing has been jettisoned.

Game Scenes All

Intake

class Intake
: An air intake. Obtained by calling `Part::intake()`.

Part **part**()
: The part object for this intake.

Game Scenes All

bool **open**()

void **set_open**(bool *value*)
: Whether the intake is open.

Game Scenes All

float **speed**()
: Speed of the flow into the intake, in m/s.

Game Scenes All

float **flow**()
: The rate of flow into the intake, in units of resource per second.

Game Scenes All

float **area**()

The area of the intake's opening, in square meters.

Game Scenes All

Leg

class Leg

A landing leg. Obtained by calling *Part::leg()*.

Part **part**()

The part object for this landing leg.

Game Scenes All

LegState **state**()

The current state of the landing leg.

Game Scenes All

bool **deployable**()

Whether the leg is deployable.

Game Scenes All

bool **deployed**()

void **set_deployed**(bool *value*)

Whether the landing leg is deployed.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

void **release**()

Releases the docking clamp. Has no effect if the clamp has already been released.

Game Scenes All

Light

class Light

A light. Obtained by calling *Part::light()*.

Part **part**()

The part object for this light.

Game Scenes All

bool **active**()

void **set_active**(bool *value*)

Whether the light is switched on.

Game Scenes All

std::tuple<float, float, float> **color**()

void **set_color**(std::tuple<float, float, float> *value*)

The color of the light, as an RGB triple.

Game Scenes All

float **power_usage**()

The current power usage, in units of charge per second.

Game Scenes All

Parachute

class Parachute

A parachute. Obtained by calling *Part::parachute()*.

Part **part**()

The part object for this parachute.

Game Scenes All

void **deploy**()

Deploys the parachute. This has no effect if the parachute has already been deployed.

Game Scenes All

bool **deployed**()

Whether the parachute has been deployed.

Game Scenes All

void **arm**()

Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

Game Scenes All

bool **armed**()

Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

Game Scenes All

ParachuteState **state**()

The current state of the parachute.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

bool **deployable**()
Whether the radiator is deployable.

Game Scenes All

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`.

Game Scenes All

RadiatorState **state**()
The current state of the radiator.

Game Scenes All

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.

Game Scenes All

int32_t **count**()
The number of converters in the part.

Game Scenes All

std::string **name**(int32_t *index*)
The name of the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

bool **active** (int32_t *index*)
True if the specified converter is active.

Parameters

- **index** – Index of the converter.

Game Scenes All

void **start** (int32_t *index*)
Start the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

void **stop** (int32_t *index*)
Stop the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

ResourceConverterState **state** (int32_t *index*)
The state of the specified converter.

Parameters

- **index** – Index of the converter.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

float **optimum_core_temperature** ()
The core temperature at which the converter will operate with peak efficiency, in Kelvin.

Game Scenes All

float **core_temperature**()
The core temperature of the converter, in Kelvin.

Game Scenes All

float **thermal_efficiency**()
The thermal efficiency of the converter, as a percentage of its maximum.

Game Scenes All

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.

Game Scenes All

ResourceHarvesterState **state**()
The state of the harvester.

Game Scenes All

bool **deployed**()

void **set_deployed**(bool *value*)
Whether the harvester is deployed.

Game Scenes All

bool **active**()

void **set_active**(bool *value*)
Whether the harvester is actively drilling.

Game Scenes All

float **extraction_rate**()
The rate at which the drill is extracting ore, in units per second.

Game Scenes All

float **thermal_efficiency**()

The thermal efficiency of the drill, as a percentage of its maximum.

Game Scenes All

float **core_temperature**()

The core temperature of the drill, in Kelvin.

Game Scenes All

float **optimum_core_temperature**()

The core temperature at which the drill will operate with peak efficiency, in Kelvin.

Game Scenes All

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.

Game Scenes All

bool **active**()

void **set_active**(bool *value*)

Whether the reaction wheel is active.

Game Scenes All

bool **broken**()

Whether the reaction wheel is broken.

Game Scenes All

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.

Game Scenes All

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()*.

Game Scenes All

RCS

class **RCS**

An RCS block or thruster. Obtained by calling *Part::rcs()*.

Part **part**()

The part object for this RCS.

Game Scenes All

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()*).

Game Scenes All

bool **enabled**()

void **set_enabled**(bool *value*)

Whether the RCS thrusters are enabled.

Game Scenes All

bool **pitch_enabled**()

void **set_pitch_enabled**(bool *value*)

Whether the RCS thruster will fire when pitch control input is given.

Game Scenes All

bool **yaw_enabled**()

void **set_yaw_enabled**(bool *value*)

Whether the RCS thruster will fire when yaw control input is given.

Game Scenes All

bool **roll_enabled**()

void **set_roll_enabled**(bool *value*)

Whether the RCS thruster will fire when roll control input is given.

Game Scenes All

bool **forward_enabled**()

void **set_forward_enabled**(bool *value*)

Whether the RCS thruster will fire when pitch control input is given.

Game Scenes All

bool **up_enabled**()

void **set_up_enabled**(bool *value*)

Whether the RCS thruster will fire when yaw control input is given.

Game Scenes All

bool **right_enabled**()

void **set_right_enabled**(bool *value*)

Whether the RCS thruster will fire when roll control input is given.

Game Scenes All

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.

Game Scenes All

float **max_thrust**()

The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

Game Scenes All

float **max_vacuum_thrust**()

The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

Game Scenes All

std::vector<*Thruster*> **thrusters**()

A list of thrusters, one of each nozzel in the RCS part.

Game Scenes All

float **specific_impulse**()

The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

Game Scenes All

float **vacuum_specific_impulse**()

The vacuum specific impulse of the RCS, in seconds.

Game Scenes All

float **kerbin_sea_level_specific_impulse**()

The specific impulse of the RCS at sea level on Kerbin, in seconds.

Game Scenes All

std::vector<std::string> **propellants**()

The names of resources that the RCS consumes.

Game Scenes All

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.

Game Scenes All

bool **has_fuel**()

Whether the RCS has fuel available.

Game Scenes All

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.

Game Scenes All

bool **active** ()

void **set_active** (bool *value*)
Whether the sensor is active.

Game Scenes All

std::string **value** ()
The current value of the sensor.

Game Scenes All

Solar Panel

class SolarPanel
A solar panel. Obtained by calling *Part::solar_panel()*.

Part **part** ()
The part object for this solar panel.

Game Scenes All

bool **deployable** ()
Whether the solar panel is deployable.

Game Scenes All

bool **deployed** ()

void **set_deployed** (bool *value*)
Whether the solar panel is extended.

Game Scenes All

SolarPanelState **state** ()
The current state of the solar panel.

Game Scenes All

float **energy_flow** ()
The current amount of energy being generated by the solar panel, in units of charge per second.

Game Scenes All

float **sun_exposure** ()
The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

Game Scenes All

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 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**()
The *Part* that contains this thruster.

Game Scenes All

std::tuple<double, double, double> **thrust_position**(*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

- **reference_frame** – The reference frame that the returned position vector is in.

Returns The position as a vector.

Game Scenes All

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 gimballed 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.

Game Scenes All

ReferenceFrame **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 gimballing.
- The y-axis points along the thrust direction.
- The x-axis and z-axis are perpendicular to the thrust direction.

Game Scenes All

bool **gimballed**()

Whether the thruster is gimballed.

Game Scenes All

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.

Game Scenes All

std::tuple<double, double, double> **gimbal_angle**()

The current gimbal angle in the pitch, roll and yaw axes, in degrees.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

WheelState **state**()

The current state of the wheel.

Game Scenes All

float **radius**()

Radius of the wheel, in meters.

Game Scenes All

bool **grounded**()

Whether the wheel is touching the ground.

Game Scenes All

bool **has_brakes**()

Whether the wheel has brakes.

Game Scenes All

float **brakes**()

void **set_brakes**(float *value*)

The braking force, as a percentage of maximum, when the brakes are applied.

Game Scenes All

bool **auto_friction_control**()

void **set_auto_friction_control**(bool *value*)

Whether automatic friction control is enabled.

Game Scenes All

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.

Game Scenes All

bool **deployable**()

Whether the wheel is deployable.

Game Scenes All

bool **deployed**()

void **set_deployed**(bool *value*)

Whether the wheel is deployed.

Game Scenes All

bool **powered**()
Whether the wheel is powered by a motor.

Game Scenes All

bool **motor_enabled**()

void **set_motor_enabled**(bool *value*)
Whether the motor is enabled.

Game Scenes All

bool **motor_inverted**()

void **set_motor_inverted**(bool *value*)
Whether the direction of the motor is inverted.

Game Scenes All

MotorState **motor_state**()
Whether the direction of the motor is inverted.

Game Scenes All

float **motor_output**()
The output of the motor. This is the torque currently being generated, in Newton meters.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

bool **steerable**()
Whether the wheel has steering.

Game Scenes All

bool **steering_enabled**()

void **set_steering_enabled**(bool *value*)
Whether the wheel steering is enabled.

Game Scenes All

bool **steering_inverted**()

void **set_steering_inverted**(bool *value*)
Whether the wheel steering is inverted.

Game Scenes All

bool **has_suspension**()
Whether the wheel has suspension.

Game Scenes All

float **suspension_spring_strength**()
Suspension spring strength, as set in the editor.

Game Scenes All

float **suspension_damper_strength**()
Suspension damper strength, as set in the editor.

Game Scenes All

bool **broken**()
Whether the wheel is broken.

Game Scenes All

bool **repairable**()
Whether the wheel is repairable.

Game Scenes All

float **stress**()
Current stress on the wheel.

Game Scenes All

float **stress_tolerance**()
Stress tolerance of the wheel.

Game Scenes All

float **stress_percentage**()
Current stress on the wheel as a percentage of its stress tolerance.

Game Scenes All

float **deflection**()
Current deflection of the wheel.

Game Scenes All

float **slip**()
Current slip of the wheel.

Game Scenes All

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.

Traversing the Tree

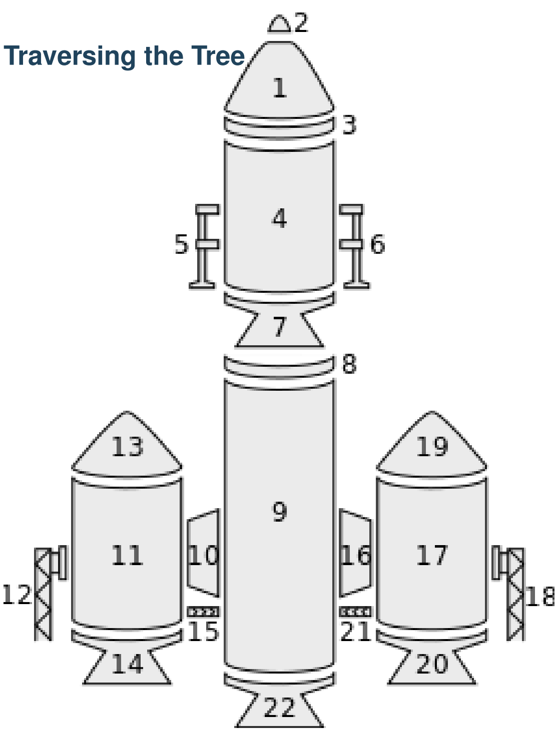


Fig. 10: **Figure 1** – Example parts making up a vessel.

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/servi

using SpaceCenter = krpc
```

(continues on next page)

(continued from previous page)

```
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.empt
    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 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 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));
    }
  }
}
```

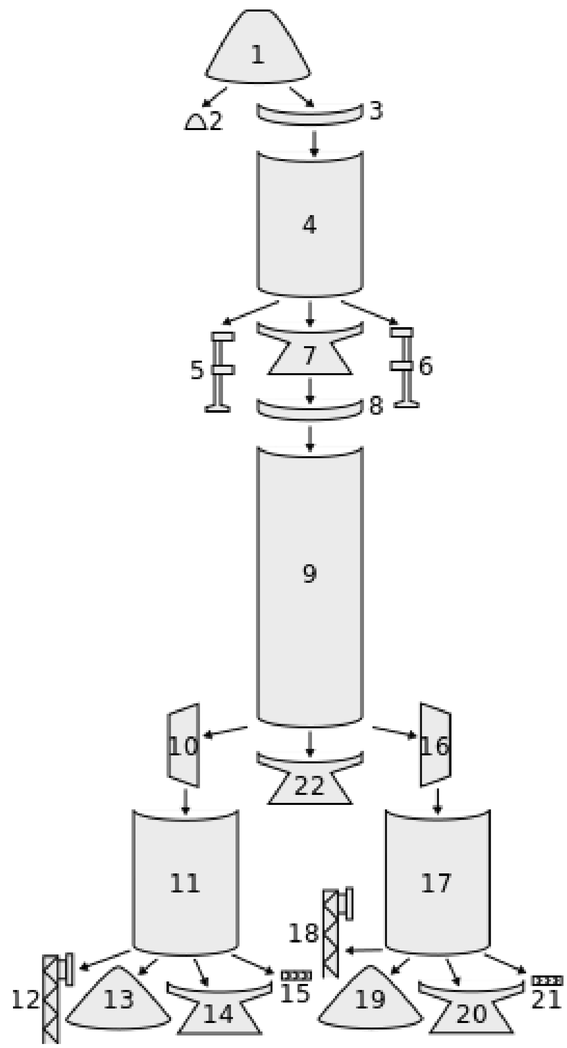


Fig. 11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the child part.

When this code is 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
```

(continues on next page)

(continued from previous page)

```
      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

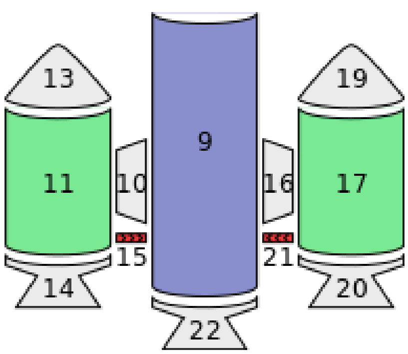
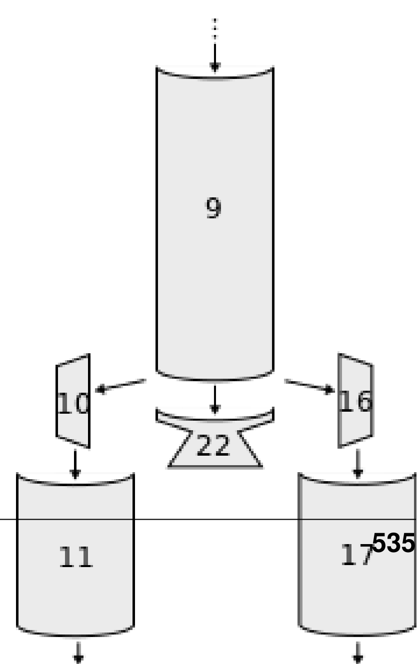


Fig. 12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

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*.

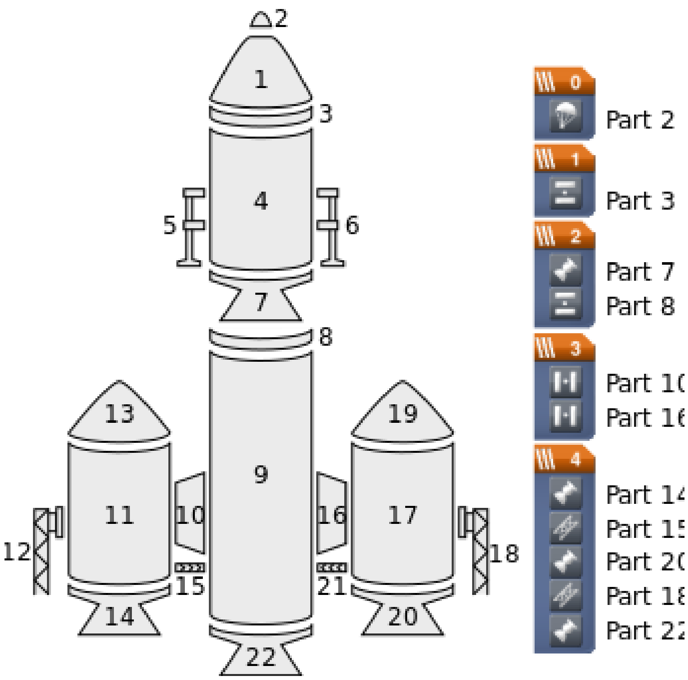


Fig. 14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

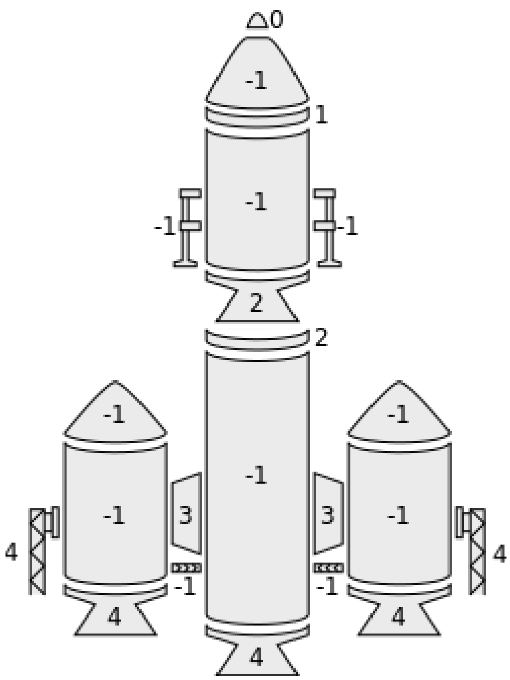


Fig. 15: **Figure 7** – The stage in which each part is *activated*.

5.3.9 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.

Game Scenes Flight

std::vector<*Resource*> **with_resource**(std::string *name*)

All the individual resources with the given name that can be stored.

Parameters

Game Scenes Flight

std::vector<std::string> **names**()

A list of resource names that can be stored.

Game Scenes Flight

bool **has_resource**(std::string *name*)

Check whether the named resource can be stored.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

float **amount** (std::string *name*)
Returns the amount of a resource that is currently stored.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

float **max** (std::string *name*)
Returns the amount of a resource that can be stored.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

static float **density** (*Client &connection*, std::string *name*)
Returns the density of a resource, in kg/l.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

static *ResourceFlowMode* **flow_mode** (*Client &connection*, std::string *name*)
Returns the flow mode of a resource.

Parameters

- **name** – The name of the resource.

Game Scenes Flight

bool **enabled** ()

void **set_enabled** (bool *value*)
Whether use of all the resources are enabled.

Game Scenes Flight

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.

Game Scenes All

Part **part** ()
The part containing the resource.

Game Scenes All

float **amount** ()
The amount of the resource that is currently stored in the part.

Game Scenes All

float **max** ()

The total amount of the resource that can be stored in the part.

Game Scenes All

float **density** ()

The density of the resource, in kg/l.

Game Scenes All

ResourceFlowMode **flow_mode** ()

The flow mode of the resource.

Game Scenes All

bool **enabled** ()

void **set_enabled** (bool *value*)

Whether use of this resource is enabled.

Game Scenes All

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.

Game Scenes All

float **amount** ()

The amount of the resource that has been transferred.

Game Scenes All

bool **complete** ()

Whether the transfer has completed.

Game Scenes All

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.

5.3.10 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.

Game Scenes Flight

double **normal**()

void **set_normal**(double *value*)
The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

Game Scenes Flight

double **radial**()

void **set_radial**(double *value*)
The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

Game Scenes Flight

double **delta_v**()

void **set_delta_v**(double *value*)
The delta-v of the maneuver node, in meters per second.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

double **time_to**()

The time until the maneuver node will be encountered, in seconds.

Game Scenes Flight

Orbit **orbit**()

The orbit that results from executing the maneuver node.

Game Scenes Flight

void **remove**()

Removes the maneuver node.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

5.3.11 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* = std::tuple<double, double, double>(0.0, 0.0, 0.0), std::tuple<double, double, double, double> *rotation* = std::tuple<double, double, double, double>(0.0, 0.0, 0.0, 1.0), std::tuple<double, double, double> *velocity* = std::tuple<double, double, double>(0.0, 0.0, 0.0), std::tuple<double, double, double> *angular_velocity* = std::tuple<double, double, double>(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 $(0, 0, 0)$.

Game Scenes All

static *ReferenceFrame* **create_hybrid** (*Client &connection*, *ReferenceFrame position*, *ReferenceFrame rotation* = *ReferenceFrame*(), *ReferenceFrame velocity* = *ReferenceFrame*(), *ReferenceFrame angular_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.

Game Scenes All

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

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.

Game Scenes Flight

void **disengage**()

Disengage the auto-pilot.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

ReferenceFrame **reference_frame**()

void **set_reference_frame**(*ReferenceFrame value*)

The reference frame for the target direction (`AutoPilot::target_direction()`).

Game Scenes Flight

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°.

Game Scenes Flight

float **target_heading**()

void **set_target_heading**(float *value*)

The target heading, in degrees, between 0° and 360°.

Game Scenes Flight

float **target_roll**()

void **set_target_roll**(float *value*)

The target roll, in degrees. `NaN` if no target roll is set.

Game Scenes Flight

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`.

Game Scenes Flight

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°.

Game Scenes Flight

bool **sas**()

void **set_sas**(bool *value*)
The state of SAS.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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()*.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Note: When *AutoPilot::auto_tune()* is true, these values are updated automatically, which will overwrite any manual changes.

5.3.13 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.

Game Scenes Flight

float **pitch** ()

void **set_pitch** (float *value*)
The pitch of the camera, in degrees. A value between *Camera::min_pitch()* and *Camera::max_pitch()*

Game Scenes Flight

float **heading** ()

void **set_heading** (float *value*)
The heading of the camera, in degrees.

Game Scenes Flight

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()*.

Game Scenes Flight

float **min_pitch** ()
The minimum pitch of the camera.

Game Scenes Flight

float **max_pitch**()

The maximum pitch of the camera.

Game Scenes Flight

float **min_distance**()

Minimum distance from the camera to the subject, in meters.

Game Scenes Flight

float **max_distance**()

Maximum distance from the camera to the subject, in meters.

Game Scenes Flight

float **default_distance**()

Default distance from the camera to the subject, in meters.

Game Scenes Flight

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 is the camera is not in map mode.

Game Scenes Flight

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 is the camera is not in map mode.

Game Scenes Flight

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 is the camera is not in map mode.

Game Scenes Flight

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.

5.3.14 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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

std::map<std::string, int32_t> **colors**()
An example map of known color - seed pairs. Any other integers may be used as seed.

Game Scenes All

std::vector<std::string> **icons**()

Returns all available icons (from "GameData/Squad/Contracts/Icons/").

Game Scenes All

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.

Game Scenes Flight

std::string **name**()

void **set_name**(std::string *value*)

The name of the waypoint as it appears on the map and the contract.

Game Scenes Flight

int32_t **color**()

void **set_color**(int32_t *value*)

The seed of the icon color. See *WaypointManager::colors()* for example colors.

Game Scenes Flight

std::string **icon**()

void **set_icon**(std::string *value*)

The icon of the waypoint.

Game Scenes Flight

double **latitude**()

void **set_latitude**(double *value*)

The latitude of the waypoint.

Game Scenes Flight

double **longitude**()

void **set_longitude**(double *value*)

The longitude of the waypoint.

Game Scenes Flight

double **mean_altitude**()

void **set_mean_altitude**(double *value*)
The altitude of the waypoint above sea level, in meters.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

bool **near_surface**()
`true` if the waypoint is near to the surface of a body.

Game Scenes Flight

bool **grounded**()
`true` if the waypoint is attached to the ground.

Game Scenes Flight

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.

Game Scenes Flight

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()*.

Game Scenes Flight

bool **has_contract**()
Whether the waypoint belongs to a contract.

Game Scenes Flight

Contract **contract**()
The associated contract.

Game Scenes Flight

void **remove**()
Removes the waypoint.

Game Scenes Flight

5.3.15 Contracts

class ContractManager
Contracts manager. Obtained by calling *contract_manager()*.

std::set<std::string> **types**()
A list of all contract types.

Game Scenes All

std::vector<*Contract*> **all_contracts**()
A list of all contracts.

Game Scenes All

std::vector<*Contract*> **active_contracts**()
A list of all active contracts.

Game Scenes All

std::vector<*Contract*> **offered_contracts**()
A list of all offered, but unaccepted, contracts.

Game Scenes All

std::vector<*Contract*> **completed_contracts**()
A list of all completed contracts.

Game Scenes All

std::vector<*Contract*> **failed_contracts**()
A list of all failed contracts.

Game Scenes All

class Contract
A contract. Can be accessed using *contract_manager()*.

std::string **type**()
Type of the contract.

Game Scenes All

std::string **title**()
Title of the contract.

Game Scenes All

std::string **description**()
Description of the contract.

Game Scenes All

std::string **notes**()
Notes for the contract.

Game Scenes All

std::string **synopsis**()
Synopsis for the contract.

Game Scenes All

std::vector<std::string> **keywords**()
Keywords for the contract.

Game Scenes All

ContractState **state**()
State of the contract.

Game Scenes All

bool **seen**()
Whether the contract has been seen.

Game Scenes All

bool **read**()
Whether the contract has been read.

Game Scenes All

bool **active**()
Whether the contract is active.

Game Scenes All

bool **failed**()
Whether the contract has been failed.

Game Scenes All

bool **can_be_canceled**()
Whether the contract can be canceled.

Game Scenes All

bool **can_be_declined**()
Whether the contract can be declined.

Game Scenes All

bool **can_be_failed**()
Whether the contract can be failed.

Game Scenes All

void **accept**()
Accept an offered contract.

Game Scenes All

void **cancel**()
Cancel an active contract.

Game Scenes All

void **decline**()
Decline an offered contract.

Game Scenes All

double **funds_advance**()
Funds received when accepting the contract.

Game Scenes All

double **funds_completion**()
Funds received on completion of the contract.

Game Scenes All

double **funds_failure**()
Funds lost if the contract is failed.

Game Scenes All

double **reputation_completion**()
Reputation gained on completion of the contract.

Game Scenes All

double **reputation_failure**()
Reputation lost if the contract is failed.

Game Scenes All

double **science_completion**()
Science gained on completion of the contract.

Game Scenes All

std::vector<*ContractParameter*> **parameters**()
Parameters for the contract.

Game Scenes All

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.

Game Scenes All

std::string **notes**()
Notes for the parameter.

Game Scenes All

std::vector<*ContractParameter*> **children**()
Child contract parameters.

Game Scenes All

bool **completed**()
Whether the parameter has been completed.

Game Scenes All

bool **failed**()
Whether the parameter has been failed.

Game Scenes All

bool **optional**()
Whether the contract parameter is optional.

Game Scenes All

double **funds_completion**()
Funds received on completion of the contract parameter.

Game Scenes All

double **funds_failure**()
Funds lost if the contract parameter is failed.

Game Scenes All

double **reputation_completion**()
Reputation gained on completion of the contract parameter.

Game Scenes All

double **reputation_failure**()
Reputation lost if the contract parameter is failed.

Game Scenes All

double **science_completion**()
Science gained on completion of the contract parameter.

Game Scenes All

5.3.16 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>
```

(continues on next page)

(continued from previous page)

```
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;
}
```

5.4 Drawing API

5.4.1 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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

void **clear** (bool *client_only* = false)
Remove all objects being drawn.

Parameters

- **client_only** – If true, only remove objects created by the calling client.

Game Scenes Flight

5.4.2 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.

Game Scenes Flight

std::tuple<double, double, double> **end**()

void **set_end**(std::tuple<double, double, double> *value*)
End position of the line.

Game Scenes Flight

SpaceCenter::ReferenceFrame **reference_frame**()

void **set_reference_frame**(*SpaceCenter::ReferenceFrame value*)
Reference frame for the positions of the object.

Game Scenes Flight

bool **visible**()

void **set_visible**(bool *value*)
Whether the object is visible.

Game Scenes Flight

std::tuple<double, double, double> **color**()

void **set_color**(std::tuple<double, double, double> *value*)
Set the color

Game Scenes Flight

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.

Game Scenes Flight

float **thickness**()

void **set_thickness**(float *value*)
Set the thickness

Game Scenes Flight

void **remove**()
Remove the object.

Game Scenes Flight

5.4.3 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.

Game Scenes Flight

SpaceCenter::ReferenceFrame **reference_frame**()

void **set_reference_frame**(*SpaceCenter::ReferenceFrame value*)
Reference frame for the positions of the object.

Game Scenes Flight

bool **visible**()

void **set_visible**(bool *value*)
Whether the object is visible.

Game Scenes Flight

void **remove**()
Remove the object.

Game Scenes Flight

std::tuple<double, double, double> **color**()

void **set_color**(std::tuple<double, double, double> *value*)
Set the color

Game Scenes Flight

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.

Game Scenes Flight

float **thickness**()

void **set_thickness**(float *value*)
Set the thickness

Game Scenes Flight

5.4.4 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.

Game Scenes Flight

std::tuple<double, double, double, double> **rotation**()

void **set_rotation**(std::tuple<double, double, double, double> *value*)
Rotation of the text as a quaternion.

Game Scenes Flight

SpaceCenter::ReferenceFrame **reference_frame**()

void **set_reference_frame**(*SpaceCenter::ReferenceFrame value*)
Reference frame for the positions of the object.

Game Scenes Flight

bool **visible**()

void **set_visible**(bool *value*)
Whether the object is visible.

Game Scenes Flight

void **remove**()
Remove the object.

Game Scenes Flight

std::string **content**()

void **set_content**(std::string *value*)
The text string

Game Scenes Flight

std::string **font**()

void **set_font**(std::string *value*)
Name of the font

Game Scenes Flight

static std::vector<std::string> **available_fonts**(*Client &connection*)
A list of all available fonts.

Game Scenes Flight

int32_t **size**()

void **set_size**(int32_t *value*)
Font size.

Game Scenes Flight

float **character_size**()

void **set_character_size**(float *value*)
Character size.

Game Scenes Flight

UI::*FontStyle* **style**()

void **set_style**(*UI*::*FontStyle value*)
Font style.

Game Scenes Flight

std::tuple<double, double, double> **color**()

void **set_color**(std::tuple<double, double, double> *value*)
Set the color

Game Scenes Flight

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.

Game Scenes Flight

UI::*TextAlignment* **alignment**()

void **set_alignment**(*UI*::*TextAlignment value*)
Alignment.

Game Scenes Flight

float **line_spacing**()

void **set_line_spacing**(float *value*)
Line spacing.

Game Scenes Flight

UI::*TextAnchor* **anchor**()

void **set_anchor**(*UI*::*TextAnchor value*)
Anchor.

Game Scenes Flight

5.5 InfernalRobotics API

Provides RPCs to interact with the InfernalRobotics mod. Both the original mod and Infernal Robotics Next are supported. Provides the following classes:

5.5.1 InfernalRobotics

class InfernalRobotics : **public** krpc::Service

This service provides functionality to interact with Infernal Robotics.

InfernalRobotics(krpc::*Client* **client*)

Construct an instance of this service.

bool **available**()

Whether Infernal Robotics is installed.

Game Scenes Flight

bool **ready**()

Whether Infernal Robotics API is ready.

Game Scenes Flight

std::vector<*ServoGroup*> **servo_groups**(*SpaceCenter*::*Vessel vessel*)

A list of all the servo groups in the given *vessel*.

Parameters

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

5.5.2 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.

Game Scenes Flight

std::string **forward_key** ()

void **set_forward_key** (std::string *value*)
The key assigned to be the "forward" key for the group.

Game Scenes Flight

std::string **reverse_key** ()

void **set_reverse_key** (std::string *value*)
The key assigned to be the "reverse" key for the group.

Game Scenes Flight

float **speed** ()

void **set_speed** (float *value*)
The speed multiplier for the group.

Game Scenes Flight

bool **expanded** ()

void **set_expanded** (bool *value*)
Whether the group is expanded in the InfernalRobotics UI.

Game Scenes Flight

std::vector<*Servo*> **servos** ()
The servos that are in the group.

Game Scenes Flight

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.

Game Scenes Flight

std::vector<*SpaceCenter*::*Part*> **parts** ()
The parts containing the servos in the group.

Game Scenes Flight

void **move_right** ()
Moves all of the servos in the group to the right.

Game Scenes Flight

void **move_left** ()
Moves all of the servos in the group to the left.

Game Scenes Flight

void **move_center**()
Moves all of the servos in the group to the center.

Game Scenes Flight

void **move_next_preset**()
Moves all of the servos in the group to the next preset.

Game Scenes Flight

void **move_prev_preset**()
Moves all of the servos in the group to the previous preset.

Game Scenes Flight

void **stop**()
Stops the servos in the group.

Game Scenes Flight

5.5.3 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.

Game Scenes Flight

SpaceCenter::Part **part**()
The part containing the servo.

Game Scenes Flight

void **set_highlight**(bool *value*)
Whether the servo should be highlighted in-game.

Game Scenes Flight

float **position**()
The position of the servo.

Game Scenes Flight

float **min_config_position**()
The minimum position of the servo, specified by the part configuration.

Game Scenes Flight

float **max_config_position**()
The maximum position of the servo, specified by the part configuration.

Game Scenes Flight

float **min_position**()

void **set_min_position**(float *value*)
The minimum position of the servo, specified by the in-game tweak menu.

Game Scenes Flight

float **max_position**()

void **set_max_position**(float *value*)
The maximum position of the servo, specified by the in-game tweak menu.

Game Scenes Flight

float **config_speed**()
The speed multiplier of the servo, specified by the part configuration.

Game Scenes Flight

float **speed**()

void **set_speed**(float *value*)
The speed multiplier of the servo, specified by the in-game tweak menu.

Game Scenes Flight

float **current_speed**()

void **set_current_speed**(float *value*)
The current speed at which the servo is moving.

Game Scenes Flight

float **acceleration**()

void **set_acceleration**(float *value*)
The current speed multiplier set in the UI.

Game Scenes Flight

bool **is_moving**()
Whether the servo is moving.

Game Scenes Flight

bool **is_free_moving**()
Whether the servo is freely moving.

Game Scenes Flight

bool **is_locked**()

void **set_is_locked**(bool *value*)
Whether the servo is locked.

Game Scenes Flight

bool **is_axis_inverted**()

void **set_is_axis_inverted** (bool *value*)
Whether the servos axis is inverted.

Game Scenes Flight

void **move_right** ()
Moves the servo to the right.

Game Scenes Flight

void **move_left** ()
Moves the servo to the left.

Game Scenes Flight

void **move_center** ()
Moves the servo to the center.

Game Scenes Flight

void **move_next_preset** ()
Moves the servo to the next preset.

Game Scenes Flight

void **move_prev_preset** ()
Moves the servo to the previous preset.

Game Scenes Flight

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.

Game Scenes Flight

void **stop** ()
Stops the servo.

Game Scenes Flight

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.

```
#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;
```

(continues on next page)

(continued from previous page)

```
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();
}
```

5.6 Kerbal Alarm Clock API

Provides RPCs to interact with the Kerbal Alarm Clock mod. Provides the following classes:

5.6.1 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.

Game Scenes All

std::vector<*Alarm*> **alarms**()
A list of all the alarms.

Game Scenes All

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.

Game Scenes All

std::vector<*Alarm*> **alarms_with_type** (*AlarmType type*)
Get a list of alarms of the specified *type*.

Parameters

- **type** – Type of alarm to return.

Game Scenes All

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.

Game Scenes All

5.6.2 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.

Game Scenes All

double **margin** ()

void **set_margin** (double *value*)
The number of seconds before the event that the alarm will fire.

Game Scenes All

double **time** ()

void **set_time** (double *value*)
The time at which the alarm will fire.

Game Scenes All

AlarmType **type** ()
The type of the alarm.

Game Scenes All

std::string **id** ()
The unique identifier for the alarm.

Game Scenes All

std::string **name**()

void **set_name**(std::string *value*)
The short name of the alarm.

Game Scenes All

std::string **notes**()

void **set_notes**(std::string *value*)
The long description of the alarm.

Game Scenes All

double **remaining**()
The number of seconds until the alarm will fire.

Game Scenes All

bool **repeat**()

void **set_repeat**(bool *value*)
Whether the alarm will be repeated after it has fired.

Game Scenes All

double **repeat_period**()

void **set_repeat_period**(double *value*)
The time delay to automatically create an alarm after it has fired.

Game Scenes All

SpaceCenter::*Vessel* **vessel**()

void **set_vessel**(*SpaceCenter*::*Vessel value*)
The vessel that the alarm is attached to.

Game Scenes All

SpaceCenter::*CelestialBody* **xfer_origin_body**()

void **set_xfer_origin_body**(*SpaceCenter*::*CelestialBody value*)
The celestial body the vessel is departing from.

Game Scenes All

SpaceCenter::*CelestialBody* **xfer_target_body**()

void **set_xfer_target_body**(*SpaceCenter*::*CelestialBody value*)
The celestial body the vessel is arriving at.

Game Scenes All

void **remove**()
Removes the alarm.

Game Scenes All

5.6.3 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*.

5.6.4 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.

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.

```cpp
#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",
```

(continues on next page)

(continued from previous page)

```
                                    sc.ut()+10);

  alarm.set_notes("10 seconds␣
↪have now passed since the alarm was created.");
  alarm.set_
↪action(KerbalAlarmClock::AlarmAction::message_
↪only);
}
```

5.7 RemoteTech API

Provides RPCs to interact with the RemoteTech mod. Provides the following classes:

5.7.1 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.

Game Scenes All

std::vector<std::string> **ground_stations**()
The names of the ground stations.

Game Scenes All

Antenna **antenna**(*SpaceCenter*::*Part part*)
Get the antenna object for a particular part.

Parameters

Game Scenes All

Comms **comms**(*SpaceCenter*::*Vessel vessel*)
Get a communications object, representing the communication capability of a particular vessel.

Parameters

Game Scenes All

5.7.2 Comms

class Comms
Communications for a vessel.

SpaceCenter::*Vessel* **vessel**()
Get the vessel.

Game Scenes All

bool **has_local_control**()
Whether the vessel can be controlled locally.

Game Scenes All

bool **has_flight_computer**()
Whether the vessel has a flight computer on board.

Game Scenes All

bool **has_connection**()
Whether the vessel has any connection.

Game Scenes All

bool **has_connection_to_ground_station**()
Whether the vessel has a connection to a ground station.

Game Scenes All

double **signal_delay**()
The shortest signal delay to the vessel, in seconds.

Game Scenes All

double **signal_delay_to_ground_station**()
The signal delay between the vessel and the closest ground station, in seconds.

Game Scenes All

double **signal_delay_to_vessel**(*SpaceCenter*::*Vessel other*)
The signal delay between the this vessel and another vessel, in seconds.

Parameters

Game Scenes All

std::vector<*Antenna*> **antennas**()
The antennas for this vessel.

Game Scenes All

5.7.3 Antenna

class Antenna
A RemoteTech antenna. Obtained by calling *Comms::antennas()* or *antenna()*.

SpaceCenter::*Part* **part**()
Get the part containing this antenna.

Game Scenes All

bool **has_connection**()
Whether the antenna has a connection.

Game Scenes All

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()`.

Game Scenes All

SpaceCenter::CelestialBody **target_body** ()

void **set_target_body** (*SpaceCenter::CelestialBody value*)
The celestial body the antenna is targetting.

Game Scenes All

std::string **target_ground_station** ()

void **set_target_ground_station** (std::string *value*)
The ground station the antenna is targetting.

Game Scenes All

SpaceCenter::Vessel **target_vessel** ()

void **set_target_vessel** (*SpaceCenter::Vessel value*)
The vessel the antenna is targetting.

Game Scenes All

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.

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.

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

(continues on next page)

(continued from previous page)

```
#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;
}
```

5.8 User Interface API

5.8.1 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.

Game Scenes All

Canvas **add_canvas**()

Add a new canvas.

Game Scenes All

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), std::tuple<double, double, double> *color* = std::tuple<double, double, double>(1.0, 0.92, 0.016), float *size* = 20.0)

Display a message on the screen.

Parameters

- **content** – Message content.

- **duration** – Duration before the message disappears, in seconds.
- **position** – Position to display the message.
- **color** – The color of the message.
- **size** – Size of the message, differs per position.

Game Scenes All

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.

Game Scenes All

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.

5.8.2 Canvas

class Canvas
A canvas for user interface elements. See *stock_canvas()* and *add_canvas()*.

RectTransform **rect_transform** ()
The rect transform for the canvas.

Game Scenes All

bool **visible** ()

void **set_visible** (bool *value*)
Whether the UI object is visible.

Game Scenes All

Panel **add_panel** (bool *visible* = true)
Create a new container for user interface elements.

Parameters

- **visible** – Whether the panel is visible.

Game Scenes All

Text **add_text** (std::string *content*, bool *visible* = true)
Add text to the canvas.

Parameters

- **content** – The text.
- **visible** – Whether the text is visible.

Game Scenes All

InputField **add_input_field** (bool *visible* = true)
Add an input field to the canvas.

Parameters

- **visible** – Whether the input field is visible.

Game Scenes All

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.

Game Scenes All

void **remove** ()
Remove the UI object.

Game Scenes All

5.8.3 Panel

class Panel
A container for user interface elements. See `Canvas::add_panel()`.

RectTransform **rect_transform** ()
The rect transform for the panel.

Game Scenes All

bool **visible** ()

void **set_visible** (bool *value*)
Whether the UI object is visible.

Game Scenes All

Panel **add_panel** (bool *visible* = true)
Create a panel within this panel.

Parameters

- **visible** – Whether the new panel is visible.

Game Scenes All

Text **add_text** (std::string *content*, bool *visible* = true)
Add text to the panel.

Parameters

- **content** – The text.
- **visible** – Whether the text is visible.

Game Scenes All

InputField **add_input_field** (bool *visible* = true)
Add an input field to the panel.

Parameters

- **visible** – Whether the input field is visible.

Game Scenes All

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.

Game Scenes All

void **remove** ()
Remove the UI object.

Game Scenes All

5.8.4 Text

class Text
A text label. See `Panel::add_text()`.

RectTransform **rect_transform** ()
The rect transform for the text.

Game Scenes All

bool **visible** ()

void **set_visible** (bool *value*)
Whether the UI object is visible.

Game Scenes All

std::string **content** ()

void **set_content** (std::string *value*)
The text string

Game Scenes All

std::string **font** ()

void **set_font** (std::string *value*)
Name of the font

Game Scenes All

std::vector<std::string> **available_fonts** ()
A list of all available fonts.

Game Scenes All

int32_t **size** ()

void **set_size** (int32_t *value*)
Font size.

Game Scenes All

FontStyle **style** ()

void **set_style** (*FontStyle value*)
Font style.

Game Scenes All

std::tuple<double, double, double> **color** ()

void **set_color** (std::tuple<double, double, double> *value*)
Set the color

Game Scenes All

TextAnchor **alignment** ()

void **set_alignment** (*TextAnchor value*)
Alignment.

Game Scenes All

float **line_spacing** ()

void **set_line_spacing** (float *value*)
Line spacing.

Game Scenes All

void **remove** ()
Remove the UI object.

Game Scenes All

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.

5.8.5 Button

class Button
A text label. See *Panel::add_button()*.

RectTransform **rect_transform**()
The rect transform for the text.

Game Scenes All

bool **visible**()

void **set_visible**(bool *value*)
Whether the UI object is visible.

Game Scenes All

Text **text**()
The text for the button.

Game Scenes All

bool **clicked**()

void **set_clicked**(bool *value*)
Whether the button has been clicked.

Game Scenes All

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.

Game Scenes All

5.8.6 InputField

class InputField
An input field. See `Panel::add_input_field()`.

RectTransform **rect_transform**()
The rect transform for the input field.

Game Scenes All

bool **visible**()

void **set_visible**(bool *value*)
Whether the UI object is visible.

Game Scenes All

std::string **value**()

void **set_value**(std::string *value*)
The value of the input field.

Game Scenes All

Text **text**()
The text component of the input field.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

5.8.7 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.

Game Scenes All

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.

Game Scenes All

std::tuple<double, double> **size** ()

void **set_size** (std::tuple<double, double> *value*)
Width and height of the rectangle.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

CHAPTER SIX

JAVA

6.1 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.

6.1.1 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 {
    try (Connection connection = Connection.newInstance()) {
      KRPC krpc = KRPC.newInstance(connection);
      System.out.println("Connected to kRPC version " + krpc.getStatus().
↪getVersion());
    }
  }
}
```

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.

6.1.2 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 {
    try (Connection connection = Connection.newInstance("Remote example", "my.domain.
↪name", 1000, 1001)) {
      System.out.println(KRPC.newInstance(connection).getStatus().getVersion());
    }
  }
}
```

6.1.3 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 {
    try (Connection connection = Connection.newInstance("Vessel Name")) {
      SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
      Vessel vessel = spaceCenter.getActiveVessel();
      System.out.println(vessel.getName());
    }
  }
}
```

6.1.4 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;
```

(continues on next page)

(continued from previous page)

```
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 {
    try (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 {
    try (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.

6.1.5 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 {
    try (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 {
    try (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.

6.1.6 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 {
    try (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");
      }
    }
  }
}
```

6.1.7 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** ()

Starts the stream. When a stream is created it does not start sending updates to the client until this method is called.

The `startAndWait` method will block until at least one update has been received from the server.

The `start` method starts the stream and returns immediately. Subsequent calls to `get()` may throw a `StreamException`.

float **getRate** ()

void **setRate** (float *rate*)
: The update rate of the stream in Hertz. When set to zero, the rate is unlimited.

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).

int **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. Returns a unique identifier for the callback which can be used to remove it.

  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 **removeCallback** (int *tag*)
: Removes a callback function from the stream. The tag is the identifier returned when the callback was added.

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.

int **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. Returns an integer tag identifying the callback which can be used to remove it later.

void **removeCallback**(int *tag*)
Removes a callback function from the event.

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.

6.2 KRPC API

6.2.1 KRPC

None 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.

Game Scenes All

String **getClientName**()
Returns the name of the current client. This is an empty string if the client has no name.

Game Scenes All

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.

Game Scenes All

krpc.schema.KRPC.Status **getStatus**()
Returns some information about the server, such as the version.

Game Scenes All

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.

Game Scenes All

GameScene **getCurrentGameScene**()
Get the current game scene.

Game Scenes All

boolean **getPaused**()

void **setPaused**(boolean *value*)
Whether the game is paused.

Game Scenes All

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.

6.2.2 Expressions

public class **Expression**
A server side expression.

static *Expression* **constantDouble**(*Connection connection*, double *value*)
A constant value of double precision floating point type.

Parameters

- **value** (*double*) –

Game Scenes All

static *Expression* **constantFloat**(*Connection connection*, float *value*)
A constant value of single precision floating point type.

Parameters

- **value** (*float*) –

Game Scenes All

static *Expression* **constantInt**(*Connection connection*, int *value*)
A constant value of integer type.

Parameters

- **value** (*int*) –

Game Scenes All

static *Expression* **constantBool** (*Connection connection*, boolean *value*)

A constant value of boolean type.

Parameters

- **value** (*boolean*) –

Game Scenes All

static *Expression* **constantString** (*Connection connection*, String *value*)

A constant value of string type.

Parameters

- **value** (*String*) –

Game Scenes All

static *Expression* **call** (*Connection connection*, krpc.schema.KRPC.ProcedureCall *call*)

An RPC call.

Parameters

- **call** (*krpc.schema.KRPC.ProcedureCall*) –

Game Scenes All

static *Expression* **equal** (*Connection connection*, *Expression arg0*, *Expression arg1*)

Equality comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **notEqual** (*Connection connection*, *Expression arg0*, *Expression arg1*)

Inequality comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **greaterThan** (*Connection connection*, *Expression arg0*, *Expression arg1*)

Greater than numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **greaterThanOrEqual** (*Connection connection*, *Expression arg0*, *Expression arg1*)

Greater than or equal numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **lessThan**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Less than numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **lessThanOrEqual**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Less than or equal numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **and**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Boolean and operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **or**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Boolean or operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **exclusiveOr**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Boolean exclusive-or operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **not**(*Connection connection*, *Expression arg*)
Boolean negation operator.

Parameters

- **arg** (Expression) –

Game Scenes All

static *Expression* **add** (*Connection connection*, *Expression arg0*, *Expression arg1*)
Numerical addition.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **subtract** (*Connection connection*, *Expression arg0*, *Expression arg1*)
Numerical subtraction.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **multiply** (*Connection connection*, *Expression arg0*, *Expression arg1*)
Numerical multiplication.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **divide** (*Connection connection*, *Expression arg0*, *Expression arg1*)
Numerical division.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

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

Game Scenes All

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, with type of arg0

Game Scenes All

static *Expression* **leftShift** (*Connection connection*, *Expression arg0*, *Expression arg1*)
Bitwise left shift.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **rightShift** (*Connection connection*, *Expression arg0*, *Expression arg1*)
Bitwise right shift.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Game Scenes All

static *Expression* **cast** (*Connection connection*, *Expression arg*, *Type type*)
Perform a cast to the given type.

Parameters

- **arg** (Expression) –
- **type** (Type) – Type to cast the argument to.

Game Scenes All

static *Expression* **parameter** (*Connection connection*, String *name*, *Type type*)
A named parameter of type double.

Parameters

- **name** (*String*) – The name of the parameter.
- **type** (Type) – The type of the parameter.

Returns A named parameter.

Game Scenes All

static *Expression* **function** (*Connection connection*, java.util.List<*Expression*> *parameters*, *Expression body*)
A function.

Parameters

- **parameters** (*java.util.List<Expression>*) – The parameters of the function.
- **body** (Expression) – The body of the function.

Returns A function.

Game Scenes All

static *Expression* **invoke** (*Connection connection*, *Expression function*, java.util.Map<String, *Expression*> *args*)
A function call.

Parameters

- **function** (Expression) – The function to call.

- **args** (*java.util.Map<String,Expression>*) – The arguments to call the function with.

Returns A function call.

Game Scenes All

static *Expression* **createTuple** (*Connection connection*, java.util.List<*Expression*> *elements*)
Construct a tuple.

Parameters

- **elements** (*java.util.List<Expression>*) – The elements.

Returns The tuple.

Game Scenes All

static *Expression* **createList** (*Connection connection*, java.util.List<*Expression*> *values*)
Construct a list.

Parameters

- **values** (*java.util.List<Expression>*) – The value. Should all be of the same type.

Returns The list.

Game Scenes All

static *Expression* **createSet** (*Connection connection*, java.util.Set<*Expression*> *values*)
Construct a set.

Parameters

- **values** (*java.util.Set<Expression>*) – The values. Should all be of the same type.

Returns The set.

Game Scenes All

static *Expression* **createDictionary** (*Connection connection*, java.util.List<*Expression*> *keys*, java.util.List<*Expression*> *values*)
Construct a dictionary, from a list of corresponding keys and values.

Parameters

- **keys** (*java.util.List<Expression>*) – The keys. Should all be of the same type.
- **values** (*java.util.List<Expression>*) – The values. Should all be of the same type.

Returns The dictionary.

Game Scenes All

static *Expression* **toList** (*Connection connection*, *Expression arg*)
Convert a collection to a list.

Parameters

- **arg** (Expression) – The collection.

Returns The collection as a list.

Game Scenes All

static *Expression* **toSet** (*Connection connection*, *Expression arg*)
Convert a collection to a set.

Parameters

- **arg** (`Expression`) – The collection.

Returns The collection as a set.

Game Scenes All

static *Expression* **get** (*Connection connection*, *Expression arg*, *Expression index*)
Access an element in a tuple, list or dictionary.

Parameters

- **arg** (`Expression`) – The tuple, list or dictionary.
- **index** (`Expression`) – The index of the element to access. A zero indexed integer for a tuple or list, or a key for a dictionary.

Returns The element.

Game Scenes All

static *Expression* **count** (*Connection connection*, *Expression arg*)
Number of elements in a collection.

Parameters

- **arg** (`Expression`) – The list, set or dictionary.

Returns The number of elements in the collection.

Game Scenes All

static *Expression* **sum** (*Connection connection*, *Expression arg*)
Sum all elements of a collection.

Parameters

- **arg** (`Expression`) – The list or set.

Returns The sum of the elements in the collection.

Game Scenes All

static *Expression* **max** (*Connection connection*, *Expression arg*)
Maximum of all elements in a collection.

Parameters

- **arg** (`Expression`) – The list or set.

Returns The maximum elements in the collection.

Game Scenes All

static *Expression* **min** (*Connection connection*, *Expression arg*)
Minimum of all elements in a collection.

Parameters

- **arg** (`Expression`) – The list or set.

Returns The minimum elements in the collection.

Game Scenes All

static *Expression* **average** (*Connection connection*, *Expression arg*)
Minimum of all elements in a collection.

Parameters

- **arg** (`Expression`) – The list or set.

Returns The minimum elements in the collection.

Game Scenes All

static *Expression* **select** (*Connection connection*, *Expression arg*, *Expression func*)
Run a function on every element in the collection.

Parameters

- **arg** (`Expression`) – The list or set.
- **func** (`Expression`) – The function.

Returns The modified collection.

Game Scenes All

static *Expression* **where** (*Connection connection*, *Expression arg*, *Expression func*)
Run a function on every element in the collection.

Parameters

- **arg** (`Expression`) – The list or set.
- **func** (`Expression`) – The function.

Returns The modified collection.

Game Scenes All

static *Expression* **contains** (*Connection connection*, *Expression arg*, *Expression value*)
Determine if a collection contains a value.

Parameters

- **arg** (`Expression`) – The collection.
- **value** (`Expression`) – The value to look for.

Returns Whether the collection contains a value.

Game Scenes All

static *Expression* **aggregate** (*Connection connection*, *Expression arg*, *Expression func*)
Applies an accumulator function over a sequence.

Parameters

- **arg** (`Expression`) – The collection.
- **func** (`Expression`) – The accumulator function.

Returns The accumulated value.

Game Scenes All

static *Expression* **aggregateWithSeed** (*Connection connection*, *Expression arg*, *Expression seed*, *Expression func*)
Applies an accumulator function over a sequence, with a given seed.

Parameters

- **arg** (`Expression`) – The collection.

- **seed** (Expression) – The seed value.
- **func** (Expression) – The accumulator function.

Returns The accumulated value.

Game Scenes All

static *Expression* **concat** (*Connection connection*, *Expression arg1*, *Expression arg2*)
Concatenate two sequences.

Parameters

- **arg1** (Expression) – The first sequence.
- **arg2** (Expression) – The second sequence.

Returns The first sequence followed by the second sequence.

Game Scenes All

static *Expression* **orderBy** (*Connection connection*, *Expression arg*, *Expression key*)
Order a collection using a key function.

Parameters

- **arg** (Expression) – The collection to order.
- **key** (Expression) – A function that takes a value from the collection and generates a key to sort on.

Returns The ordered collection.

Game Scenes All

static *Expression* **all** (*Connection connection*, *Expression arg*, *Expression predicate*)
Determine whether all items in a collection satisfy a boolean predicate.

Parameters

- **arg** (Expression) – The collection.
- **predicate** (Expression) – The predicate function.

Returns Whether all items satisfy the predicate.

Game Scenes All

static *Expression* **any** (*Connection connection*, *Expression arg*, *Expression predicate*)
Determine whether any item in a collection satisfies a boolean predicate.

Parameters

- **arg** (Expression) – The collection.
- **predicate** (Expression) – The predicate function.

Returns Whether any item satisfies the predicate.

Game Scenes All

public class **Type**
A server side expression.

static *Type* **double_** (*Connection connection*)
Double type.

Game Scenes All

static *Type* **float_**(*Connection connection*)
Float type.

Game Scenes All

static *Type* **int_**(*Connection connection*)
Int type.

Game Scenes All

static *Type* **bool**(*Connection connection*)
Bool type.

Game Scenes All

static *Type* **string**(*Connection connection*)
String type.

Game Scenes All

6.3 SpaceCenter API

6.3.1 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.

float **getScience**()
The current amount of science.

Game Scenes All

double **getFunds**()
The current amount of funds.

Game Scenes All

float **getReputation**()
The current amount of reputation.

Game Scenes All

Vessel **getActiveVessel**()

void **setActiveVessel**(*Vessel value*)
The currently active vessel.

Game Scenes Flight

java.util.List<*Vessel*> **getVessels**()
A list of all the vessels in the game.

Game Scenes All

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.

Game Scenes All

CelestialBody **getTargetBody**()

void **setTargetBody** (*CelestialBody value*)
The currently targeted celestial body.

Game Scenes Flight

Vessel **getTargetVessel** ()

void **setTargetVessel** (*Vessel value*)
The currently targeted vessel.

Game Scenes Flight

DockingPort **getTargetDockingPort** ()

void **setTargetDockingPort** (*DockingPort value*)
The currently targeted docking port.

Game Scenes Flight

void **clearTarget** ()
Clears the current target.

Game Scenes Flight

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"`.

Game Scenes All

void **launchVessel** (String *craftDirectory*, String *name*, String *launchSite*, boolean *recover*)
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"`.
- **recover** (*boolean*) – If true and there is a vessel on the launch site, recover it before launching.

Game Scenes All

Note: Throws an exception if any of the games pre-flight checks fail.

void **launchVesselFromVAB** (String *name*, boolean *recover*)
Launch a new vessel from the VAB onto the launchpad.

Parameters

- **name** (*String*) – Name of the vessel to launch.
- **recover** (*boolean*) – If true and there is a vessel on the launch pad, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *launchVessel(String, String, String, boolean)* with the craft directory set to "VAB" and the launch site set to "LaunchPad". Throws an exception if any of the games pre-flight checks fail.

void **launchVesselFromSPH**(String *name*, boolean *recover*)

Launch a new vessel from the SPH onto the runway.

Parameters

- **name** (*String*) – Name of the vessel to launch.
- **recover** (*boolean*) – If true and there is a vessel on the runway, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *launchVessel(String, String, String, boolean)* with the craft directory set to "SPH" and the launch site set to "Runway". Throws an exception if any of the games pre-flight checks fail.

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*) –

Game Scenes All

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*) –

Game Scenes All

void **quicksave**()

Save a quicksave.

Game Scenes All

Note: This is the same as calling *save(String)* with the name "quicksave".

void **quickload**()

Load a quicksave.

Game Scenes All

Note: This is the same as calling *load(String)* with the name "quicksave".

boolean **getUIVisible**()

void **setUIVisible**(boolean *value*)
Whether the UI is visible.

Game Scenes Flight

boolean **getNavball**()

void **setNavball**(boolean *value*)
Whether the navball is visible.

Game Scenes Flight

double **getUT**()
The current universal time in seconds.

Game Scenes All

double **getG**()
The value of the gravitational constant G in $N(m/kg)^2$.

Game Scenes All

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.

Game Scenes Flight

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()*.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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 covert the position to.

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

Game Scenes All

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 covert the direction to.

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

Game Scenes All

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 covert the rotation to.

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

Game Scenes All

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 covert the velocity to.

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

Game Scenes All

double **raycastDistance**(org.javatuples.Triplet<Double, Double, Double> *position*, org.javatuples.Triplet<Double, Double, Double> *direction*, *ReferenceFrame referenceFrame*)

Cast a ray from a given position in a given direction, and return the distance to the hit point. If no hit occurs, returns infinity.

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position, as a vector, of the origin of the ray.
- **direction** (*org.javatuples.Triplet<Double,Double,Double>*) – Direction of the ray, as a unit vector.

- **referenceFrame** (ReferenceFrame) – The reference frame that the position and direction are in.

Returns The distance to the hit, in meters, or infinity if there was no hit.

Game Scenes All

Part **raycastPart** (org.javatuples.Triplet<Double, Double, Double> *position*, org.javatuples.Triplet<Double, Double, Double> *direction*, *ReferenceFrame referenceFrame*)

Cast a ray from a given position in a given direction, and return the part that it hits. If no hit occurs, returns `null`.

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position, as a vector, of the origin of the ray.
- **direction** (*org.javatuples.Triplet<Double,Double,Double>*) – Direction of the ray, as a unit vector.
- **referenceFrame** (ReferenceFrame) – The reference frame that the position and direction are in.

Returns The part that was hit or `null` if there was no hit.

Game Scenes Flight

boolean **getFARAvailable** ()

Whether Ferram Aerospace Research is installed.

Game Scenes All

GameMode **getGameMode** ()

The current mode the game is in.

Game Scenes All

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.

Game Scenes Flight

Camera **getCamera** ()

An object that can be used to control the camera.

Game Scenes Flight

WaypointManager **getWaypointManager** ()

The waypoint manager.

Game Scenes Flight

ContractManager **getContractManager** ()

The contract manager.

Game Scenes All

public enum **GameMode**

The game mode. Returned by *GameMode*

public *GameMode* **SANDBOX**

Sandbox mode.

public *GameMode* **CAREER**
Career mode.

public *GameMode* **SCIENCE**
Science career mode.

public *GameMode* **SCIENCE_SANDBOX**
Science sandbox mode.

public *GameMode* **MISSION**
Mission mode.

public *GameMode* **MISSION_BUILDER**
Mission builder mode.

public *GameMode* **SCENARIO**
Scenario mode.

public *GameMode* **SCENARIO_NON_RESUMABLE**
Scenario mode that cannot be resumed.

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.

6.3.2 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.

Game Scenes All

VesselType **getType**()

void **setType**(*VesselType value*)
The type of the vessel.

Game Scenes All

VesselSituation **getSituation**()
The situation the vessel is in.

Game Scenes All

boolean **getRecoverable**()
Whether the vessel is recoverable.

Game Scenes All

void **recover**()
Recover the vessel.

Game Scenes All

double **getMET**()
The mission elapsed time in seconds.

Game Scenes All

String **getBiome**()
The name of the biome the vessel is currently in.

Game Scenes All

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()*).

Game Scenes 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 **getOrbit**()
The current orbit of the vessel.

Game Scenes All

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.

Game Scenes Flight

Comms **getComms**()
Returns a *Comms* object that can be used to interact with CommNet for this vessel.

Game Scenes Flight

AutoPilot **getAutoPilot**()
An *AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

Game Scenes Flight

int **getCrewCapacity**()
The number of crew that can occupy the vessel.

Game Scenes All

int **getCrewCount**()
The number of crew that are occupying the vessel.

Game Scenes All

java.util.List<*CrewMember*> **getCrew**()
The crew in the vessel.

Game Scenes All

Resources **getResources**()

A *Resources* object, that can used to get information about resources stored in the vessel.

Game Scenes Flight

Resources **resourcesInDecoupleStage**(int *stage*, boolean *cumulative*)

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** (*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.

Game Scenes Flight

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.

Game Scenes Flight

float **getMass**()

The total mass of the vessel, including resources, in kg.

Game Scenes Flight

float **getDryMass**()

The total mass of the vessel, excluding resources, in kg.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

float **getSpecificImpulse**()

The combined specific impulse of all active engines, in seconds. This is computed using the formula described here.

Game Scenes Flight

float **getVacuumSpecificImpulse**()

The combined vacuum specific impulse of all active engines, in seconds. This is computed using the formula described here.

Game Scenes Flight

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.

Game Scenes Flight

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*).

Game Scenes Flight

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.

Game Scenes All

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**

The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gimballed 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.

Game Scenes Flight

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**

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.

Game Scenes Flight

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**

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.

Game Scenes Flight

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**

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 (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**

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.

Game Scenes Flight

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**

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 (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

Game Scenes Flight

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.

Game Scenes Flight

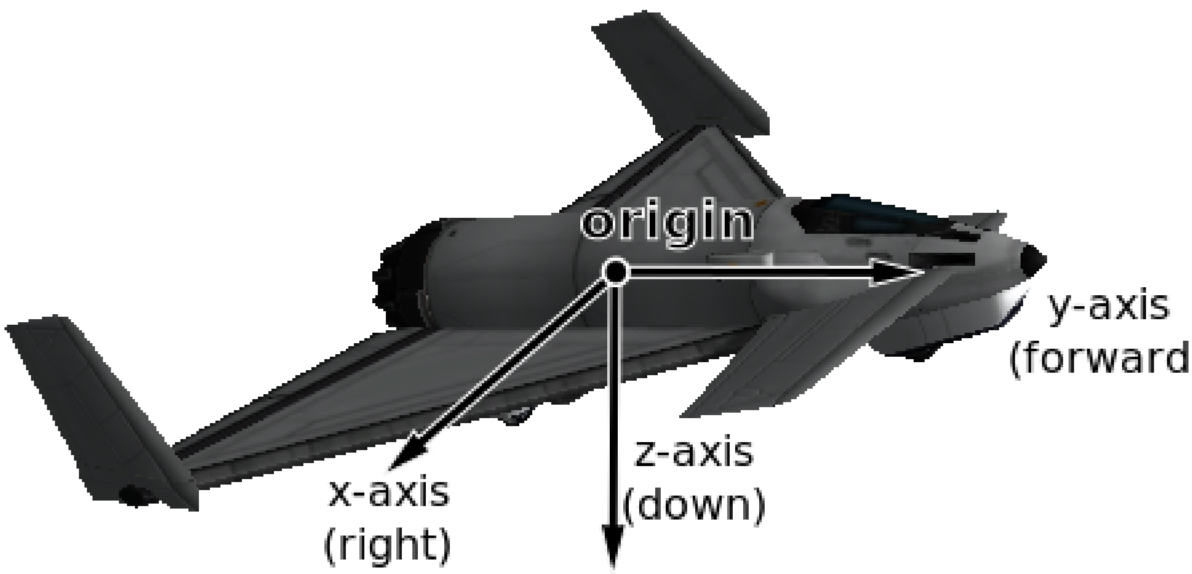


Fig. 1: Vessel reference frame origin and axes for the Aeris 3A aircraft

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.
- 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.

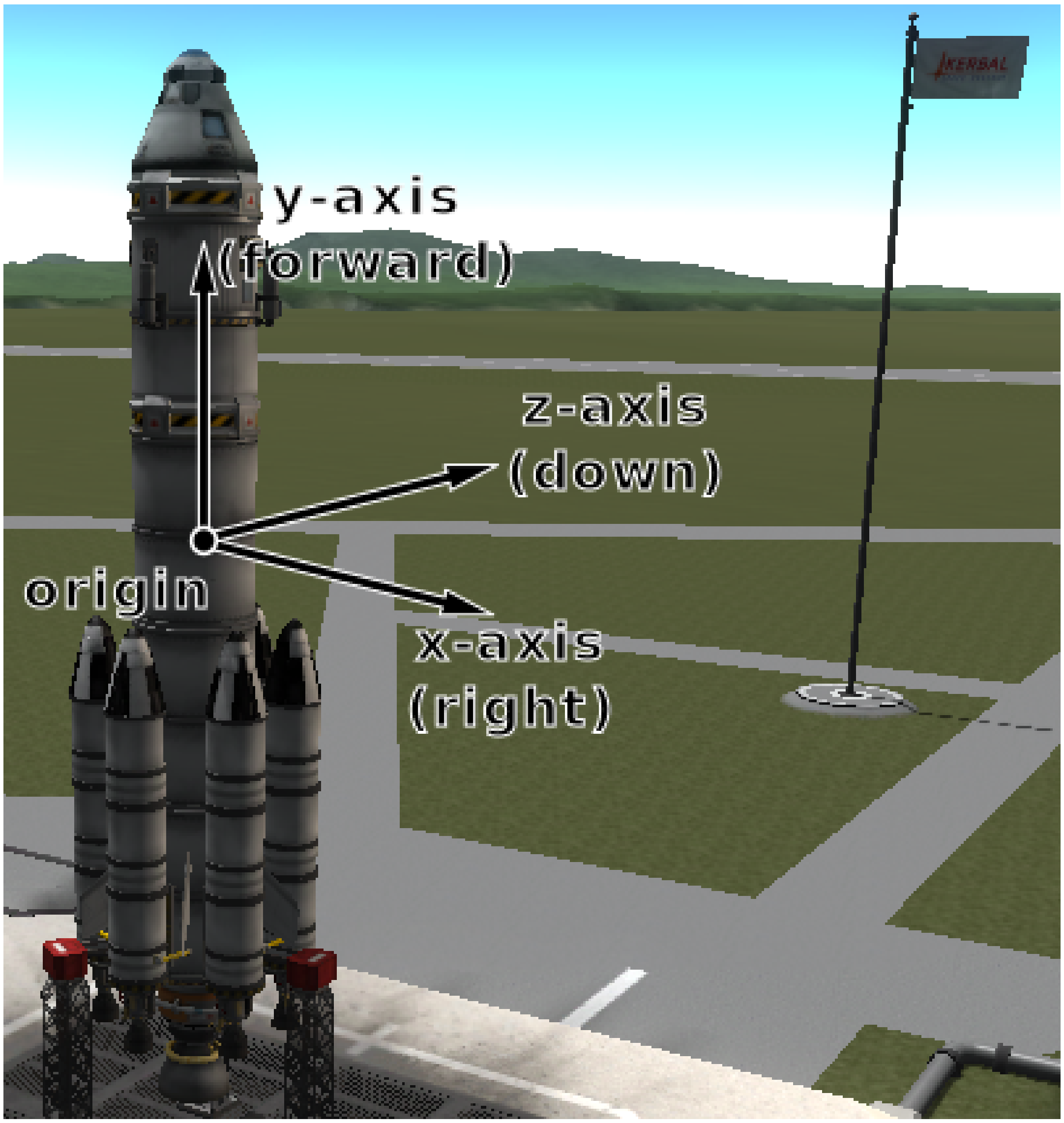


Fig. 2: Vessel reference frame origin and axes for the Kerbal-X rocket

Game Scenes Flight

Note: Be careful not to confuse this with 'orbit' mode on the navball.

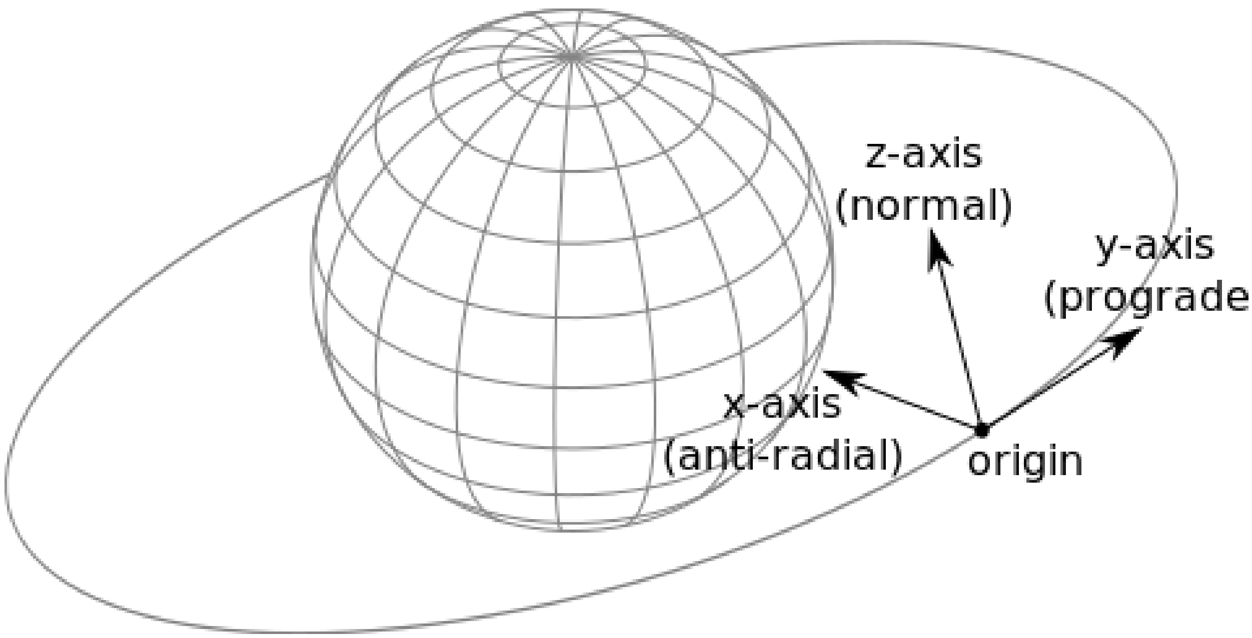


Fig. 3: Vessel orbital reference frame origin and axes

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).

Game Scenes Flight

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.

- The origin is at the center of mass of the vessel.

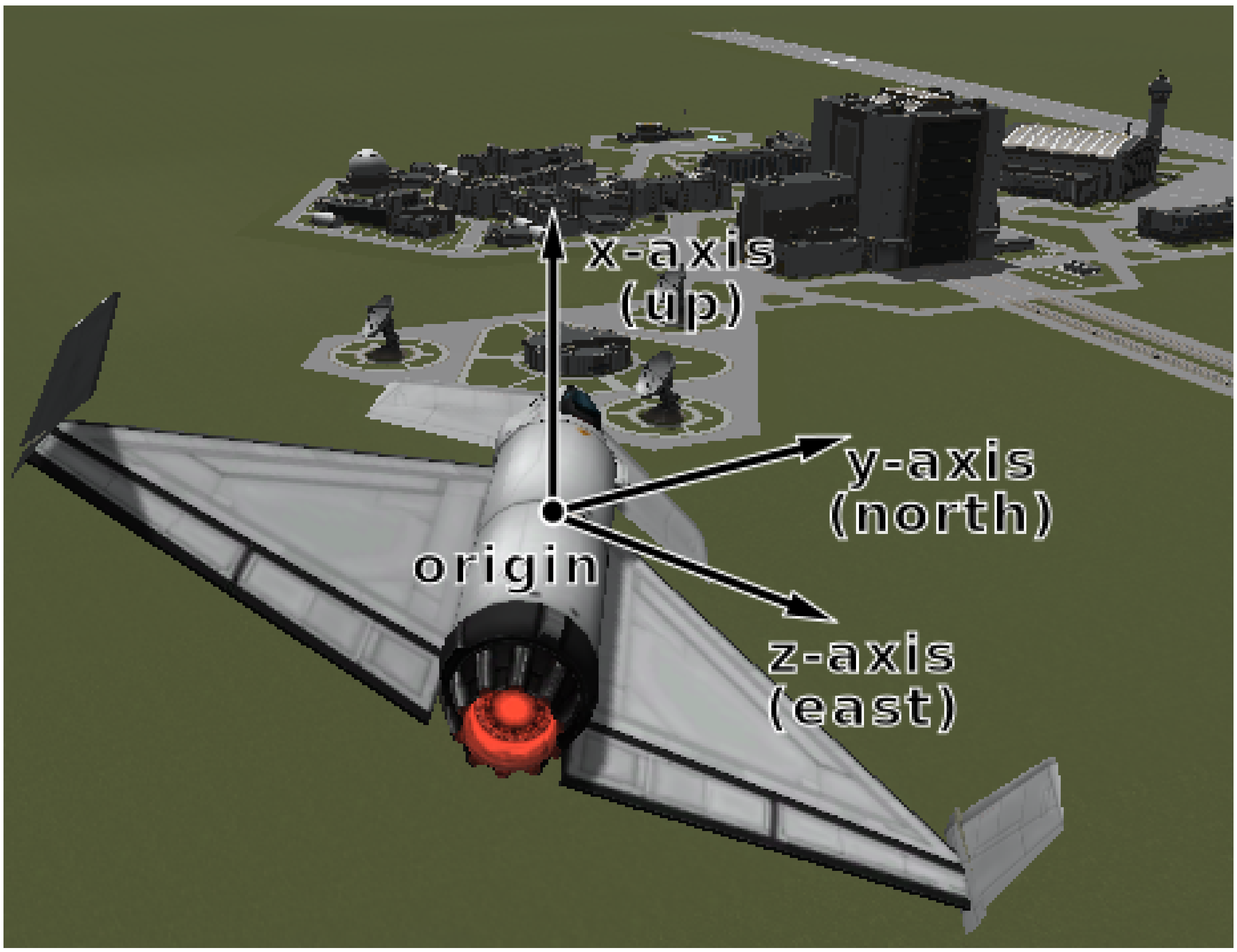


Fig. 4: Vessel surface reference frame origin and axes

- 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.

Game Scenes Flight

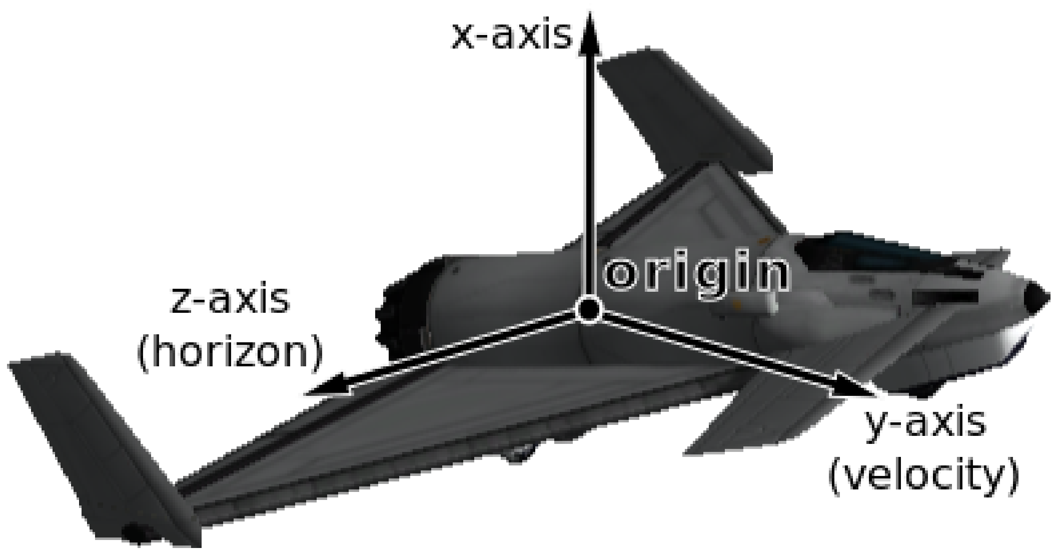


Fig. 5: Vessel surface velocity reference frame origin and axes

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.

Game Scenes Flight

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **bound**

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.

Game Scenes Flight

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.

Game Scenes Flight

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).

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

public class **CrewMember**
Represents crew in a vessel. Can be obtained using *Vessel.getCrew()*.

String **getName**()

void **setName**(String *value*)
The crew members name.

Game Scenes All

CrewMemberType **getType**()
The type of crew member.

Game Scenes All

boolean **getOnMission**()
Whether the crew member is on a mission.

Game Scenes All

float **getCourage** ()

void **setCourage** (float *value*)
The crew members courage.

Game Scenes All

float **getStupidity** ()

void **setStupidity** (float *value*)
The crew members stupidity.

Game Scenes All

float **getExperience** ()

void **setExperience** (float *value*)
The crew members experience.

Game Scenes All

boolean **getBadass** ()

void **setBadass** (boolean *value*)
Whether the crew member is a badass.

Game Scenes All

boolean **getVeteran** ()

void **setVeteran** (boolean *value*)
Whether the crew member is a veteran.

Game Scenes All

public enum **CrewMemberType**
The type of a crew member. See *CrewMember.getType()*.

public *CrewMemberType* **APPLICANT**
An applicant for crew.

public *CrewMemberType* **CREW**
Rocket crew.

public *CrewMemberType* **TOURIST**
A tourist.

public *CrewMemberType* **UNOWNED**
An unowned crew member.

6.3.3 CelestialBody

public class **CelestialBody**
Represents a celestial body (such as a planet or moon). See *getBodies()*.

String **getName** ()
The name of the body.

Game Scenes All

java.util.List<*CelestialBody*> **getSatellites** ()
A list of celestial bodies that are in orbit around this celestial body.

Game Scenes All

Orbit **getOrbit**()
The orbit of the body.

Game Scenes All

float **getMass**()
The mass of the body, in kilograms.

Game Scenes All

float **getGravitationalParameter**()
The standard gravitational parameter of the body in m^3s^{-2}.

Game Scenes All

float **getSurfaceGravity**()
The acceleration due to gravity at sea level (mean altitude) on the body, in m/s^2.

Game Scenes All

float **getRotationalPeriod**()
The sidereal rotational period of the body, in seconds.

Game Scenes All

float **getRotationalSpeed**()
The rotational speed of the body, in radians per second.

Game Scenes All

double **getRotationAngle**()
The current rotation angle of the body, in radians. A value between 0 and 2π

Game Scenes All

double **getInitialRotation**()
The initial rotation angle of the body (at UT 0), in radians. A value between 0 and 2π

Game Scenes All

float **getEquatorialRadius**()
The equatorial radius of the body, in meters.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

float **getSphereOfInfluence**()

The radius of the sphere of influence of the body, in meters.

Game Scenes All

boolean **getHasAtmosphere**()

`true` if the body has an atmosphere.

Game Scenes All

float **getAtmosphereDepth**()

The depth of the atmosphere, in meters.

Game Scenes All

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.

Game Scenes All

boolean **getHasAtmosphericOxygen** ()

`true` if there is oxygen in the atmosphere, required for air-breathing engines.

Game Scenes All

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.

Game Scenes All

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*) –

Game Scenes All

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*) –

Game Scenes All

java.util.Set<String> **getBiomes** ()

The biomes present on this body.

Game Scenes All

String **biomeAt** (double *latitude*, double *longitude*)
The biome at the given latitude and longitude, in degrees.

Parameters

- **latitude** (*double*) –
- **longitude** (*double*) –

Game Scenes All

float **getFlyingHighAltitudeThreshold** ()
The altitude, in meters, above which a vessel is considered to be flying "high" when doing science.

Game Scenes All

float **getSpaceHighAltitudeThreshold** ()
The altitude, in meters, above which a vessel is considered to be in "high" space when doing science.

Game Scenes All

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.

Game Scenes All

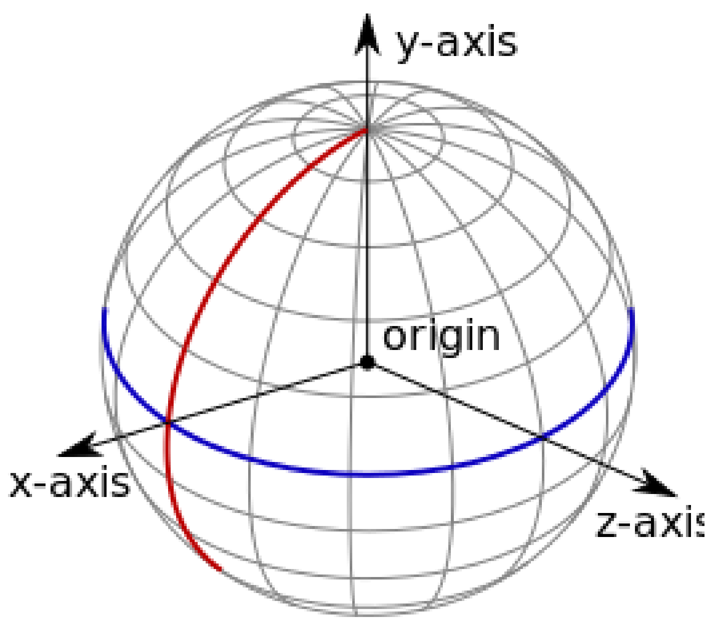


Fig. 6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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).

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

6.3.4 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 g.

Game Scenes Flight

double **getMeanAltitude** ()

The altitude above sea level, in meters. Measured from the center of mass of the vessel.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

double **getLatitude**()
The latitude of the vessel for the body being orbited, in degrees.

Game Scenes Flight

double **getLongitude**()
The longitude of the vessel for the body being orbited, in degrees.

Game Scenes Flight

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.

Game Scenes Flight

double **getSpeed**()
The speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Game Scenes Flight

double **getHorizontalSpeed**()
The horizontal speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Game Scenes Flight

double **getVerticalSpeed**()
The vertical speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Game Scenes Flight

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.

Game Scenes Flight

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).

Game Scenes Flight

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.

Game Scenes Flight

float **getPitch**()
The pitch of the vessel relative to the horizon, in degrees. A value between -90° and +90°.

Game Scenes Flight

float **getHeading**()
The heading of the vessel (its angle relative to north), in degrees. A value between 0° and 360°.

Game Scenes Flight

float **getRoll**()

The roll of the vessel relative to the horizon, in degrees. A value between -180° and +180°.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

float **getAtmosphereDensity**()

The current density of the atmosphere around the vessel, in kg/m^3.

Game Scenes Flight

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 $\frac{1}{2}$.air density.velocity^2. It is commonly denoted Q.

Game Scenes Flight

float **getStaticPressure**()

The static atmospheric pressure acting on the vessel, in Pascals.

Game Scenes Flight

float **getStaticPressureAtMSL**()

The static atmospheric pressure at mean sea level, in Pascals.

Game Scenes Flight

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.

Game Scenes Flight

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 where 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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

float **getSpeedOfSound**()

The speed of sound, in the atmosphere around the vessel, in m/s.

Game Scenes Flight

float **getMach**()

The speed of the vessel, in multiples of the speed of sound.

Game Scenes Flight

float **getReynoldsNumber**()

The vessels Reynolds number.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

float **getTrueAirSpeed**()

The true air speed of the vessel, in meters per second.

Game Scenes Flight

float **getEquivalentAirSpeed**()

The equivalent air speed of the vessel, in meters per second.

Game Scenes Flight

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.

Game Scenes Flight

float **getAngleOfAttack**()

The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

Game Scenes Flight

float **getSideslipAngle**()

The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

Game Scenes Flight

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.

Game Scenes Flight

float **getStaticAirTemperature**()

The static (ambient) temperature of the atmosphere around the vessel, in Kelvin.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

float **getBallisticCoefficient**()

The ballistic coefficient.

Game Scenes Flight

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.

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

6.3.5 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.

Game Scenes All

double **getApoapsis**()

Gets the apoapsis of the orbit, in meters, from the center of mass of the body being orbited.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

double **getSemiMinorAxis**()
The semi-minor axis of the orbit, in meters.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

double **getSpeed**()
The current orbital speed of the object in meters per second.

Game Scenes All

Note: This value will change over time if the orbit is elliptical.

double **getPeriod**()
The orbital period, in seconds.

Game Scenes All

double **getTimeToApoapsis**()
The time until the object reaches apoapsis, in seconds.

Game Scenes All

double **getTimeToPeriapsis**()
The time until the object reaches periapsis, in seconds.

Game Scenes All

double **getEccentricity**()
The eccentricity of the orbit.

Game Scenes All

double **getInclination**()
The inclination of the orbit, in radians.

Game Scenes All

double **getLongitudeOfAscendingNode**()
The longitude of the ascending node, in radians.

Game Scenes All

double **getArgumentOfPeriapsis**()
The argument of periapsis, in radians.

Game Scenes All

double **getMeanAnomalyAtEpoch**()
The mean anomaly at epoch.

Game Scenes All

double **getEpoch**()
The time since the epoch (the point at which the mean anomaly at epoch was measured, in seconds.

Game Scenes All

double **getMeanAnomaly**()
The mean anomaly.

Game Scenes All

double **meanAnomalyAtUT**(double *ut*)
The mean anomaly at the given time.

Parameters

- **ut** (*double*) – The universal time in seconds.

Game Scenes All

double **getEccentricAnomaly**()
The eccentric anomaly.

Game Scenes All

double **`eccentricAnomalyAtUT`** (double *ut*)
The eccentric anomaly at the given universal time.

Parameters

- **ut** (*double*) – The universal time, in seconds.

Game Scenes All

double **`getTrueAnomaly`** ()
The true anomaly.

Game Scenes All

double **`trueAnomalyAtUT`** (double *ut*)
The true anomaly at the given time.

Parameters

- **ut** (*double*) – The universal time in seconds.

Game Scenes All

double **`trueAnomalyAtRadius`** (double *radius*)
The true anomaly at the given orbital radius.

Parameters

- **radius** (*double*) – The orbital radius in meters.

Game Scenes All

double **`uTAtTrueAnomaly`** (double *trueAnomaly*)
The universal time, in seconds, corresponding to the given true anomaly.

Parameters

- **trueAnomaly** (*double*) – True anomaly.

Game Scenes All

double **`radiusAtTrueAnomaly`** (double *trueAnomaly*)
The orbital radius at the point in the orbit given by the true anomaly.

Parameters

- **trueAnomaly** (*double*) – The true anomaly.

Game Scenes All

double **`trueAnomalyAtAN`** (*Orbit target*)
The true anomaly of the ascending node with the given target orbit.

Parameters

- **target** (Orbit) – Target orbit.

Game Scenes All

double **`trueAnomalyAtDN`** (*Orbit target*)
The true anomaly of the descending node with the given target orbit.

Parameters

- **target** (Orbit) – Target orbit.

Game Scenes All

double **getOrbitalSpeed**()

The current orbital speed in meters per second.

Game Scenes All

double **orbitalSpeedAt**(double *time*)

The orbital speed at the given time, in meters per second.

Parameters

- **time** (*double*) – Time from now, in seconds.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

double **relativeInclination**(*Orbit target*)

Relative inclination of this orbit and the target orbit, in radians.

Parameters

- **target** (`Orbit`) – Target orbit.

Game Scenes All

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.

Game Scenes All

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`.

Game Scenes All

double **timeOfClosestApproach** (*Orbit target*)
Estimates and returns the time at closest approach to a target orbit.

Parameters

- **target** (Orbit) – Target orbit.

Returns The universal time at closest approach, in seconds.

Game Scenes All

double **distanceAtClosestApproach** (*Orbit target*)
Estimates and returns the distance at closest approach to a target orbit, in meters.

Parameters

- **target** (Orbit) – Target orbit.

Game Scenes All

java.util.List<java.util.List<Double>> **listClosestApproaches** (*Orbit target*, int *orbits*)
Returns the times at closest approach and corresponding distances, to a target orbit.

Parameters

- **target** (Orbit) – Target orbit.
- **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.

Game Scenes All

6.3.6 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.

Game Scenes Flight

ControlState **getState** ()
The control state of the vessel.

Game Scenes Flight

boolean **getSAS** ()

void **setSAS** (boolean *value*)
The state of SAS.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

boolean **getRCS**()

void **setRCS**(boolean *value*)
The state of RCS.

Game Scenes Flight

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()*.

Game Scenes Flight

boolean **getGear**()

void **setGear**(boolean *value*)
The state of the landing gear/legs.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

boolean **getLights**()

void **setLights**(boolean *value*)
The state of the lights.

Game Scenes Flight

boolean **getBrakes**()

void **setBrakes**(boolean *value*)
The state of the wheel brakes.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

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()*.

Game Scenes Flight

boolean **getAbort**()

void **setAbort**(boolean *value*)
The state of the abort action group.

Game Scenes Flight

float **getThrottle**()

void **setThrottle**(float *value*)
The state of the throttle. A value between 0 and 1.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

int **getCurrentStage** ()
The current stage of the vessel. Corresponds to the stage number in the in-game UI.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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*) –

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

java.util.List<*Node*> **getNodes** ()

Returns a list of all existing maneuver nodes, ordered by time from first to last.

Game Scenes Flight

void **removeNodes** ()

Remove all maneuver nodes.

Game Scenes Flight

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.

6.3.7 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.

Game Scenes Flight

boolean **getCanTransmitScience**()
Whether the vessel can transmit science data to KSC.

Game Scenes Flight

double **getSignalStrength**()
Signal strength to KSC.

Game Scenes Flight

double **getSignalDelay**()
Signal delay to KSC in seconds.

Game Scenes Flight

double **getPower**()
The combined power of all active antennae on the vessel.

Game Scenes Flight

java.util.List<*CommLink*> **getControlPath**()
The communication path used to control the vessel.

Game Scenes Flight

public class **CommLink**
Represents a communication node in the network. For example, a vessel or the KSC.

CommLinkType **getType**()
The type of link.

Game Scenes All

double **getSignalStrength**()
Signal strength of the link.

Game Scenes All

CommNode **getStart**()
Start point of the link.

Game Scenes All

CommNode **getEnd**()
Start point of the link.

Game Scenes All

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.

Game Scenes All

boolean **getIsHome**()
Whether the communication node is on Kerbin.

Game Scenes All

boolean **getIsControlPoint**()

Whether the communication node is a control point, for example a manned vessel.

Game Scenes All

boolean **getIsVessel**()

Whether the communication node is a vessel.

Game Scenes All

Vessel **getVessel**()

The vessel for this communication node.

Game Scenes All

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*

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.

Game Scenes All

Part **getRoot**()

The vessels root part.

Game Scenes All

Note: See the discussion on *Trees of Parts*.

Part **getControlling**()

void **setControlling**(*Part value*)

The part from which the vessel is controlled.

Game Scenes All

java.util.List<*Part*> **withName**(String *name*)

A list of parts whose `Part.getName()` is *name*.

Parameters

- **name** (`String`) –

Game Scenes All

java.util.List<*Part*> **withTitle**(String *title*)

A list of all parts whose `Part.getTitle()` is *title*.

Parameters

- **title** (`String`) –

Game Scenes All

java.util.List<*Part*> **withTag**(String *tag*)

A list of all parts whose `Part.getTag()` is *tag*.

Parameters

- **tag** (*String*) –

Game Scenes All

java.util.List<*Part*> **withModule** (String *moduleName*)

A list of all parts that contain a *Module* whose *Module.getName()* is *moduleName*.

Parameters

- **moduleName** (*String*) –

Game Scenes All

java.util.List<*Part*> **inStage** (int *stage*)

A list of all parts that are activated in the given *stage*.

Parameters

- **stage** (*int*) –

Game Scenes All

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*) –

Game Scenes All

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*) –

Game Scenes All

java.util.List<*Antenna*> **getAntennas** ()

A list of all antennas in the vessel.

Game Scenes All

java.util.List<*CargoBay*> **getCargoBays** ()

A list of all cargo bays in the vessel.

Game Scenes All

java.util.List<*ControlSurface*> **getControlSurfaces** ()

A list of all control surfaces in the vessel.

Game Scenes All

java.util.List<*Decoupler*> **getDecouplers** ()

A list of all decouplers in the vessel.

Game Scenes All

java.util.List<*DockingPort*> **getDockingPorts**()
A list of all docking ports in the vessel.

Game Scenes All

java.util.List<*Engine*> **getEngines**()
A list of all engines in the vessel.

Game Scenes All

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.

Game Scenes All

java.util.List<*Fairing*> **getFairings**()
A list of all fairings in the vessel.

Game Scenes All

java.util.List<*Intake*> **getIntakes**()
A list of all intakes in the vessel.

Game Scenes All

java.util.List<*Leg*> **getLegs**()
A list of all landing legs attached to the vessel.

Game Scenes All

java.util.List<*LaunchClamp*> **getLaunchClamps**()
A list of all launch clamps attached to the vessel.

Game Scenes All

java.util.List<*Light*> **getLights**()
A list of all lights in the vessel.

Game Scenes All

java.util.List<*Parachute*> **getParachutes**()
A list of all parachutes in the vessel.

Game Scenes All

java.util.List<*Radiator*> **getRadiators**()
A list of all radiators in the vessel.

Game Scenes All

java.util.List<*RCS*> **getRCS**()
A list of all RCS blocks/thrusters in the vessel.

Game Scenes All

java.util.List<*ReactionWheel*> **getReactionWheels**()
A list of all reaction wheels in the vessel.

Game Scenes All

java.util.List<*ResourceConverter*> **getResourceConverters**()
A list of all resource converters in the vessel.

Game Scenes All

java.util.List<*ResourceHarvester*> **getResourceHarvesters**()
A list of all resource harvesters in the vessel.

Game Scenes All

java.util.List<*Sensor*> **getSensors**()
A list of all sensors in the vessel.

Game Scenes All

java.util.List<*SolarPanel*> **getSolarPanels**()
A list of all solar panels in the vessel.

Game Scenes All

java.util.List<*Wheel*> **getWheels**()
A list of all wheels in the vessel.

Game Scenes All

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".

Game Scenes All

String **getTitle**()
Title of the part, as shown when the part is right clicked in-game. For example "Mk1-2 Command Pod".

Game Scenes All

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.

Game Scenes All

Note: This string is shared with kOS if it is installed.

boolean **getHighlighted**()

void **setHighlighted**(boolean *value*)
Whether the part is highlighted.

Game Scenes All

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.

Game Scenes All

double **getCost** ()

The cost of the part, in units of funds.

Game Scenes All

Vessel **getVessel** ()

The vessel that contains this part.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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`.

Game Scenes All

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`.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Note: See the discussion on *Staging*.

boolean **getMassless**()
Whether the part is massless.

Game Scenes All

double **getMass**()
The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

Game Scenes All

double **getDryMass**()
The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

Game Scenes All

boolean **getShielded**()
Whether the part is shielded from the exterior of the vessel, for example by a fairing.

Game Scenes All

float **getDynamicPressure**()
The dynamic pressure acting on the part, in Pascals.

Game Scenes All

double **getImpactTolerance**()
The impact tolerance of the part, in meters per second.

Game Scenes All

double **getTemperature**()
Temperature of the part, in Kelvin.

Game Scenes All

double **getSkinTemperature**()
Temperature of the skin of the part, in Kelvin.

Game Scenes All

double **getMaxTemperature**()
Maximum temperature that the part can survive, in Kelvin.

Game Scenes All

double **getMaxSkinTemperature**()
Maximum temperature that the skin of the part can survive, in Kelvin.

Game Scenes All

float **getThermalMass**()
A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

Game Scenes All

float **getThermalSkinMass**()
A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Resources **getResources**()

A `Resources` object for the part.

Game Scenes All

boolean **getCrossfeed**()

Whether this part is crossfeed capable.

Game Scenes All

boolean **getIsFuelLine**()

Whether this part is a fuel line.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Note: See the discussion on *Fuel Lines*.

java.util.List<*Module*> **getModules**()
The modules for this part.

Game Scenes All

Antenna **getAntenna**()
A `Antenna` if the part is an antenna, otherwise `null`.

Game Scenes All

CargoBay **getCargoBay**()
A `CargoBay` if the part is a cargo bay, otherwise `null`.

Game Scenes All

ControlSurface **getControlSurface**()
A `ControlSurface` if the part is an aerodynamic control surface, otherwise `null`.

Game Scenes All

Decoupler **getDecoupler**()
A `Decoupler` if the part is a decoupler, otherwise `null`.

Game Scenes All

DockingPort **getDockingPort**()
A `DockingPort` if the part is a docking port, otherwise `null`.

Game Scenes All

Engine **getEngine**()
An `Engine` if the part is an engine, otherwise `null`.

Game Scenes All

Experiment **getExperiment**()
An `Experiment` if the part is a science experiment, otherwise `null`.

Game Scenes All

Fairing **getFairing**()
A `Fairing` if the part is a fairing, otherwise `null`.

Game Scenes All

Intake **getIntake**()
An `Intake` if the part is an intake, otherwise `null`.

Game Scenes All

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`.

Game Scenes All

LaunchClamp **getLaunchClamp**()

A *LaunchClamp* if the part is a launch clamp, otherwise `null`.

Game Scenes All

Light **getLight**()

A *Light* if the part is a light, otherwise `null`.

Game Scenes All

Parachute **getParachute**()

A *Parachute* if the part is a parachute, otherwise `null`.

Game Scenes All

Radiator **getRadiator**()

A *Radiator* if the part is a radiator, otherwise `null`.

Game Scenes All

RCS **getRCS**()

A *RCS* if the part is an RCS block/thruster, otherwise `null`.

Game Scenes All

ReactionWheel **getReactionWheel**()

A *ReactionWheel* if the part is a reaction wheel, otherwise `null`.

Game Scenes All

ResourceConverter **getResourceConverter**()

A *ResourceConverter* if the part is a resource converter, otherwise `null`.

Game Scenes All

ResourceHarvester **getResourceHarvester**()

A *ResourceHarvester* if the part is a resource harvester, otherwise `null`.

Game Scenes All

Sensor **getSensor**()

A *Sensor* if the part is a sensor, otherwise `null`.

Game Scenes All

SolarPanel **getSolarPanel**()

A *SolarPanel* if the part is a solar panel, otherwise `null`.

Game Scenes All

Wheel **getWheel**()

A *Wheel* if the part is a wheel, otherwise `null`.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **bound:**

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

org.javatuples.Quartet<Double, Double, Double, Double> **rotation** (*ReferenceFrame referenceFrame*)

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).

Game Scenes All

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`).

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by `DockingPort.getReferenceFrame()`.

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.

Game Scenes All

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.

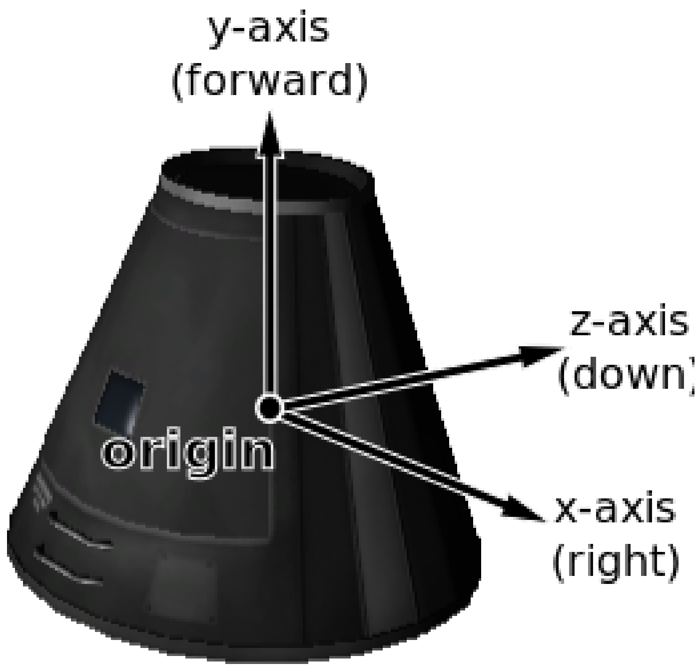


Fig. 7: Mk1 Command Pod reference frame origin and axes

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

ReferenceFrame **getReferenceFrame** ()

void **setReferenceFrame** (*ReferenceFrame value*)

The reference frame of the force vector and position.

Game Scenes All

void **remove** ()

Remove the force.

Game Scenes All

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".

Game Scenes All

Part **getPart** ()

The part that contains this module.

Game Scenes All

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.

Game Scenes All

boolean **hasField**(String *name*)

Returns `true` if the module has a field with the given name.

Parameters

- **name** (*String*) – Name of the field.

Game Scenes All

String **getField**(String *name*)

Returns the value of a field.

Parameters

- **name** (*String*) – Name of the field.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

void **resetField**(String *name*)

Set the value of a field to its original value.

Parameters

- **name** (*String*) – Name of the field.

Game Scenes All

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.

Game Scenes All

boolean **hasEvent**(String *name*)
`true` if the module has an event with the given name.

Parameters

- **name** (*String*) –

Game Scenes All

void **triggerEvent**(String *name*)
Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters

- **name** (*String*) –

Game Scenes All

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.

Game Scenes All

boolean **hasAction**(String *name*)
`true` if the part has an action with the given name.

Parameters

- **name** (*String*) –

Game Scenes All

void **setAction**(String *name*, boolean *value*)
Set the value of an action with the given name.

Parameters

- **name** (*String*) –
- **value** (*boolean*) –

Game Scenes All

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.

Game Scenes All

AntennaState **getState**()

The current state of the antenna.

Game Scenes All

boolean **getDeployable**()

Whether the antenna is deployable.

Game Scenes All

boolean **getDeployed**()

void **setDeployed**(boolean *value*)

Whether the antenna is deployed.

Game Scenes All

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.

Game Scenes All

void **transmit**()
Transmit data.

Game Scenes All

void **cancel**()
Cancel current transmission of data.

Game Scenes All

boolean **getAllowPartial**()

void **setAllowPartial**(boolean *value*)
Whether partial data transmission is permitted.

Game Scenes All

double **getPower**()
The power of the antenna.

Game Scenes All

boolean **getCombinable**()
Whether the antenna can be combined with other antennae on the vessel to boost the power.

Game Scenes All

double **getCombinableExponent**()
Exponent used to calculate the combined power of multiple antennae on a vessel.

Game Scenes All

float **getPacketInterval**()
Interval between sending packets in seconds.

Game Scenes All

float **getPacketSize**()
Amount of data sent per packet in Mits.

Game Scenes All

double **getPacketResourceCost**()
Units of electric charge consumed per packet sent.

Game Scenes All

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.

Game Scenes All

CargoBayState **getState**()
The state of the cargo bay.

Game Scenes All

boolean **getOpen**()

void **setOpen**(boolean *value*)
Whether the cargo bay is open.

Game Scenes All

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.

Game Scenes All

boolean **getPitchEnabled**()

void **setPitchEnabled**(boolean *value*)
Whether the control surface has pitch control enabled.

Game Scenes All

boolean **getYawEnabled**()

void **setYawEnabled**(boolean *value*)
Whether the control surface has yaw control enabled.

Game Scenes All

boolean **getRollEnabled**()

void **setRollEnabled**(boolean *value*)

Whether the control surface has roll control enabled.

Game Scenes All

float **getAuthorityLimiter**()

void **setAuthorityLimiter**(float *value*)

The authority limiter for the control surface, which controls how far the control surface will move.

Game Scenes All

boolean **getInverted**()

void **setInverted**(boolean *value*)

Whether the control surface movement is inverted.

Game Scenes All

boolean **getDeployed**()

void **setDeployed**(boolean *value*)

Whether the control surface has been fully deployed.

Game Scenes All

float **getSurfaceArea**()

Surface area of the control surface in m^2.

Game Scenes All

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**

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()`.

Game Scenes All

Decoupler

public class **Decoupler**

A decoupler. Obtained by calling `Part.getDecoupler()`

Part **getPart**()

The part object for this decoupler.

Game Scenes All

Vessel **decouple**()

Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

Game Scenes All

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.

Game Scenes All

boolean **getStaged**()
Whether the decoupler is enabled in the staging sequence.

Game Scenes All

float **getImpulse**()
The impulse that the decoupler imparts when it is fired, in Newton seconds.

Game Scenes All

Docking Port

public class **DockingPort**
A docking port. Obtained by calling *Part.getDockingPort()*

Part **getPart**()
The part object for this docking port.

Game Scenes All

DockingPortState **getState**()
The current state of the docking port.

Game Scenes All

Part **getDockedPart**()
The part that this docking port is docked to. Returns `null` if this docking port is not docked to anything.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

boolean **getHasShield**()
Whether the docking port has a shield.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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).

Game Scenes All

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.

Game Scenes All

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by *`Part.getReferenceFrame()`*.

---

public enum **`DockingPortState`**

The state of a docking port. See *`DockingPort.getState()`*.

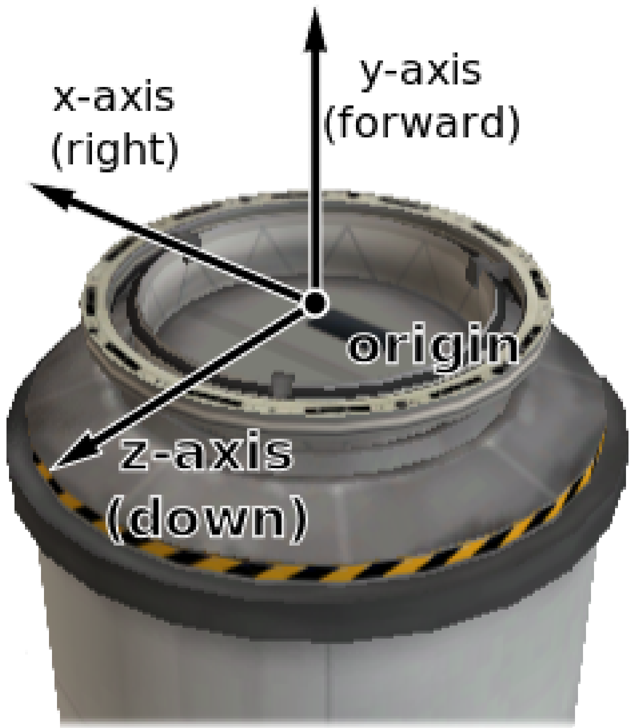


Fig. 8: Docking port reference frame origin and axes

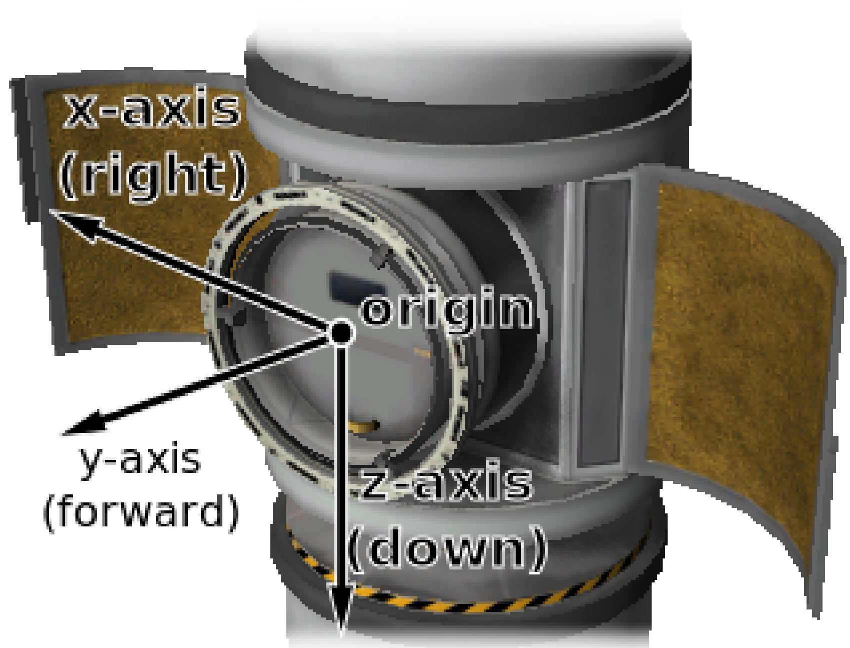


Fig. 9: Inline docking port reference frame origin and axes

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.

Game Scenes All

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()*.

Game Scenes All

float **getThrust**()
The current amount of thrust being produced by the engine, in Newtons.

Game Scenes All

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.

Game Scenes All

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%.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

java.util.List<*Thruster*> **getThrusters**()
The components of the engine that generate thrust.

Game Scenes All

Note: For example, this corresponds to the rocket nozzel on a solid rocket booster, or the individual nozzels 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.

Game Scenes All

float **getVacuumSpecificImpulse**()
The vacuum specific impulse of the engine, in seconds.

Game Scenes All

float **getKerbinSeaLevelSpecificImpulse**()
The specific impulse of the engine at sea level on Kerbin, in seconds.

Game Scenes All

java.util.List<String> **getPropellantNames**()
The names of the propellants that the engine consumes.

Game Scenes All

java.util.Map<String, Float> **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.

Game Scenes All

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.

Game Scenes All

boolean **getHasFuel**()

Whether the engine has any fuel available.

Game Scenes All

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).

Game Scenes All

boolean **getThrottleLocked**()

Whether the *Control.getThrottle()* affects the engine. For example, this is `true` for liquid fueled rockets, and `false` for solid rocket boosters.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

boolean **getHasModes**()

Whether the engine has multiple modes of operation.

Game Scenes All

String **getMode**()

void **setMode**(String *value*)

The name of the current engine mode.

Game Scenes All

java.util.Map<String, *Engine*> **getModes**()

The available modes for the engine. A dictionary mapping mode names to *Engine* objects.

Game Scenes All

void **toggleMode**()

Toggle the current engine mode.

Game Scenes All

boolean **getAutoModeSwitch**()

void **setAutoModeSwitch**(boolean *value*)

Whether the engine will automatically switch modes.

Game Scenes All

boolean **getGimballed**()

Whether the engine is gimballed.

Game Scenes All

float **getGimbalRange**()

The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**

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.

Game Scenes All

public class **Propellant**

A propellant for an engine. Obtains by calling *Engine.getPropellants()*.

String **getName**()

The name of the propellant.

Game Scenes All

double **getCurrentAmount**()

The current amount of propellant.

Game Scenes All

double **getCurrentRequirement**()

The required amount of propellant.

Game Scenes All

double **getTotalResourceAvailable**()

The total amount of the underlying resource currently reachable given resource flow rules.

Game Scenes All

double **getTotalResourceCapacity**()

The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

Game Scenes All

boolean **getIgnoreForIsp**()

If this propellant should be ignored when calculating required mass flow given specific impulse.

Game Scenes All

boolean **getIgnoreForThrustCurve**()

If this propellant should be ignored for thrust curve calculations.

Game Scenes All

boolean **getDrawStackGauge**()
If this propellant has a stack gauge or not.

Game Scenes All

boolean **getIsDeprived**()
If this propellant is deprived.

Game Scenes All

float **getRatio**()
The propellant ratio.

Game Scenes All

Experiment

public class **Experiment**
Obtained by calling *Part.getExperiment()*.

Part **getPart**()
The part object for this experiment.

Game Scenes All

void **run**()
Run the experiment.

Game Scenes All

void **transmit**()
Transmit all experimental data contained by this part.

Game Scenes All

void **dump**()
Dump the experimental data contained by the experiment.

Game Scenes All

void **reset**()
Reset the experiment.

Game Scenes All

boolean **getDeployed**()
Whether the experiment has been deployed.

Game Scenes All

boolean **getRerunnable**()
Whether the experiment can be re-run.

Game Scenes All

boolean **getInoperable**()
Whether the experiment is inoperable.

Game Scenes All

boolean **getHasData**()
Whether the experiment contains data.

Game Scenes All

java.util.List<*ScienceData*> **getData**()
The data contained in this experiment.

Game Scenes All

String **getBiome**()
The name of the biome the experiment is currently in.

Game Scenes All

boolean **getAvailable**()
Determines if the experiment is available given the current conditions.

Game Scenes All

ScienceSubject **getScienceSubject**()
Containing information on the corresponding specific science result for the current conditions. Returns `null` if the experiment is unavailable.

Game Scenes All

public class **ScienceData**
Obtained by calling *Experiment.getData()*.

float **getDataAmount**()
Data amount.

Game Scenes All

float **getScienceValue**()
Science value.

Game Scenes All

float **getTransmitValue**()
Transmit value.

Game Scenes All

public class **ScienceSubject**
Obtained by calling *Experiment.getScienceSubject()*.

String **getTitle**()
Title of science subject, displayed in science archives

Game Scenes All

boolean **getIsComplete**()
Whether the experiment has been completed.

Game Scenes All

float **getScience**()
Amount of science already earned from this subject, not updated until after transmission/recovery.

Game Scenes All

float **getScienceCap**()
Total science allowable for this subject.

Game Scenes All

float **getDataScale**()
Multiply science value by this to determine data amount in mits.

Game Scenes All

float **getSubjectValue**()
Multiplier for specific Celestial Body/Experiment Situation combination.

Game Scenes All

float **getScientificValue**()
Diminishing value multiplier for decreasing the science value returned from repeated experiments.

Game Scenes All

Fairing

public class **Fairing**
A fairing. Obtained by calling *Part.getFairing()*.

Part **getPart**()
The part object for this fairing.

Game Scenes All

void **jettison**()
Jettison the fairing. Has no effect if it has already been jettisoned.

Game Scenes All

boolean **getJettisoned**()
Whether the fairing has been jettisoned.

Game Scenes All

Intake

public class **Intake**
An air intake. Obtained by calling *Part.getIntake()*.

Part **getPart**()
The part object for this intake.

Game Scenes All

boolean **getOpen**()

void **setOpen**(boolean *value*)
Whether the intake is open.

Game Scenes All

float **getSpeed**()
Speed of the flow into the intake, in m/s.

Game Scenes All

float **getFlow**()
The rate of flow into the intake, in units of resource per second.

Game Scenes All

float **getArea**()
The area of the intake's opening, in square meters.

Game Scenes All

Leg

public class **Leg**

A landing leg. Obtained by calling `Part.getLeg()`.

Part **getPart**()

The part object for this landing leg.

Game Scenes All

LegState **getState**()

The current state of the landing leg.

Game Scenes All

boolean **getDeployable**()

Whether the leg is deployable.

Game Scenes All

boolean **getDeployed**()

void **setDeployed**(boolean *value*)

Whether the landing leg is deployed.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

void **release**()

Releases the docking clamp. Has no effect if the clamp has already been released.

Game Scenes All

Light

public class **Light**

A light. Obtained by calling *Part.getLight()*.

Part **getPart**()

The part object for this light.

Game Scenes All

boolean **getActive**()

void **setActive**(boolean *value*)

Whether the light is switched on.

Game Scenes All

org.javatuples.Triplet<Float, Float, Float> **getColor**()

void **setColor**(org.javatuples.Triplet<Float, Float, Float> *value*)

The color of the light, as an RGB triple.

Game Scenes All

float **getPowerUsage**()

The current power usage, in units of charge per second.

Game Scenes All

Parachute

public class **Parachute**

A parachute. Obtained by calling *Part.getParachute()*.

Part **getPart**()

The part object for this parachute.

Game Scenes All

void **deploy**()

Deploys the parachute. This has no effect if the parachute has already been deployed.

Game Scenes All

boolean **getDeployed**()

Whether the parachute has been deployed.

Game Scenes All

void **arm**()

Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

Game Scenes All

boolean **getArmed**()

Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

Game Scenes All

ParachuteState **getState**()
The current state of the parachute.

Game Scenes All

float **getDeployAltitude**()

void **setDeployAltitude**(float *value*)
The altitude at which the parachute will full deploy, in meters. Only applicable to stock parachutes.

Game Scenes All

float **getDeployMinPressure**()

void **setDeployMinPressure**(float *value*)
The minimum pressure at which the parachute will semi-deploy, in atmospheres. Only applicable to stock parachutes.

Game Scenes All

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.

Game Scenes All

boolean **getDeployable**()
Whether the radiator is deployable.

Game Scenes All

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`.

Game Scenes All

RadiatorState **getState** ()
The current state of the radiator.

Game Scenes All

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.

Game Scenes All

int **getCount** ()
The number of converters in the part.

Game Scenes All

String **name** (int *index*)
The name of the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

Game Scenes All

boolean **active** (int *index*)
True if the specified converter is active.

Parameters

- **index** (*int*) – Index of the converter.

Game Scenes All

void **start** (int *index*)
Start the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

Game Scenes All

void **stop** (int *index*)
Stop the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

Game Scenes All

ResourceConverterState **state** (int *index*)
The state of the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

float **getOptimumCoreTemperature** ()
The core temperature at which the converter will operate with peak efficiency, in Kelvin.

Game Scenes All

float **getCoreTemperature** ()
The core temperature of the converter, in Kelvin.

Game Scenes All

float **getThermalEfficiency** ()
The thermal efficiency of the converter, as a percentage of its maximum.

Game Scenes All

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.

Game Scenes All

ResourceHarvesterState **getState**()
The state of the harvester.

Game Scenes All

boolean **getDeployed**()

void **setDeployed**(boolean *value*)
Whether the harvester is deployed.

Game Scenes All

boolean **getActive**()

void **setActive**(boolean *value*)
Whether the harvester is actively drilling.

Game Scenes All

float **getExtractionRate**()
The rate at which the drill is extracting ore, in units per second.

Game Scenes All

float **getThermalEfficiency**()
The thermal efficiency of the drill, as a percentage of its maximum.

Game Scenes All

float **getCoreTemperature**()
The core temperature of the drill, in Kelvin.

Game Scenes All

float **getOptimumCoreTemperature**()
The core temperature at which the drill will operate with peak efficiency, in Kelvin.

Game Scenes All

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.

Game Scenes All

boolean **getActive**()

void **setActive**(boolean *value*)
Whether the reaction wheel is active.

Game Scenes All

boolean **getBroken**()
Whether the reaction wheel is broken.

Game Scenes All

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
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.

Game Scenes All

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getMa**
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()*.

Game Scenes All

RCS

public class **RCS**

An RCS block or thruster. Obtained by calling `Part.getRCS()`.

Part **getPart**()

The part object for this RCS.

Game Scenes All

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()`).

Game Scenes All

boolean **getEnabled**()

void **setEnabled**(boolean *value*)

Whether the RCS thrusters are enabled.

Game Scenes All

boolean **getPitchEnabled**()

void **setPitchEnabled**(boolean *value*)

Whether the RCS thruster will fire when pitch control input is given.

Game Scenes All

boolean **getYawEnabled**()

void **setYawEnabled**(boolean *value*)

Whether the RCS thruster will fire when yaw control input is given.

Game Scenes All

boolean **getRollEnabled**()

void **setRollEnabled**(boolean *value*)

Whether the RCS thruster will fire when roll control input is given.

Game Scenes All

boolean **getForwardEnabled**()

void **setForwardEnabled**(boolean *value*)

Whether the RCS thruster will fire when pitch control input is given.

Game Scenes All

boolean **getUpEnabled**()

void **setUpEnabled**(boolean *value*)

Whether the RCS thruster will fire when yaw control input is given.

Game Scenes All

boolean **getRightEnabled**()

void **setRightEnabled**(boolean *value*)

Whether the RCS thruster will fire when roll control input is given.

Game Scenes All

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
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.

Game Scenes All

float **getMaxThrust**()
The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

Game Scenes All

float **getMaxVacuumThrust**()
The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

Game Scenes All

java.util.List<*Thruster*> **getThrusters**()
A list of thrusters, one of each nozzel in the RCS part.

Game Scenes All

float **getSpecificImpulse**()
The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

Game Scenes All

float **getVacuumSpecificImpulse**()
The vacuum specific impulse of the RCS, in seconds.

Game Scenes All

float **getKerbinSeaLevelSpecificImpulse**()
The specific impulse of the RCS at sea level on Kerbin, in seconds.

Game Scenes All

java.util.List<String> **getPropellants**()
The names of resources that the RCS consumes.

Game Scenes All

java.util.Map<String, Float> **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.

Game Scenes All

boolean **getHasFuel**()
Whether the RCS has fuel available.

Game Scenes All

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.

Game Scenes All

boolean **getActive**()

void **setActive**(boolean *value*)

Whether the sensor is active.

Game Scenes All

String **getValue**()

The current value of the sensor.

Game Scenes All

Solar Panel

public class **SolarPanel**

A solar panel. Obtained by calling *Part.getSolarPanel()*.

Part **getPart**()

The part object for this solar panel.

Game Scenes All

boolean **getDeployable**()

Whether the solar panel is deployable.

Game Scenes All

boolean **getDeployed**()

void **setDeployed**(boolean *value*)

Whether the solar panel is extended.

Game Scenes All

SolarPanelState **getState**()

The current state of the solar panel.

Game Scenes All

float **getEnergyFlow**()

The current amount of energy being generated by the solar panel, in units of charge per second.

Game Scenes All

float **getSunExposure**()

The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

Game Scenes All

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.

Game Scenes All

org.javatuples.Triplet<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** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

Game Scenes All

org.javatuples.Triplet<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** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

Game Scenes All

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 gimballing.
- The y-axis points along the thrust direction.
- The x-axis and z-axis are perpendicular to the thrust direction.

Game Scenes All

boolean **`getGimballed`**()

Whether the thruster is gimballed.

Game Scenes All

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.

Game Scenes All

org.javatuples.Triplet<Double, Double, Double> **`getGimbalAngle`**()

The current gimbal angle in the pitch, roll and yaw axes, in degrees.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

WheelState **getState**()

The current state of the wheel.

Game Scenes All

float **getRadius**()

Radius of the wheel, in meters.

Game Scenes All

boolean **getGrounded**()

Whether the wheel is touching the ground.

Game Scenes All

boolean **getHasBrakes**()

Whether the wheel has brakes.

Game Scenes All

float **getBrakes**()

void **setBrakes**(float *value*)

The braking force, as a percentage of maximum, when the brakes are applied.

Game Scenes All

boolean **getAutoFrictionControl**()

void **setAutoFrictionControl**(boolean *value*)

Whether automatic friction control is enabled.

Game Scenes All

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.

Game Scenes All

boolean **getDeployable**()

Whether the wheel is deployable.

Game Scenes All

boolean **getDeployed**()

void **setDeployed**(boolean *value*)

Whether the wheel is deployed.

Game Scenes All

boolean **getPowered**()
Whether the wheel is powered by a motor.

Game Scenes All

boolean **getMotorEnabled**()

void **setMotorEnabled**(boolean *value*)
Whether the motor is enabled.

Game Scenes All

boolean **getMotorInverted**()

void **setMotorInverted**(boolean *value*)
Whether the direction of the motor is inverted.

Game Scenes All

MotorState **getMotorState**()
Whether the direction of the motor is inverted.

Game Scenes All

float **getMotorOutput**()
The output of the motor. This is the torque currently being generated, in Newton meters.

Game Scenes All

boolean **getTractionControlEnabled**()

void **setTractionControlEnabled**(boolean *value*)
Whether automatic traction control is enabled. A wheel only has traction control if it is powered.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

boolean **getSteerable**()
Whether the wheel has steering.

Game Scenes All

boolean **getSteeringEnabled**()

void **setSteeringEnabled**(boolean *value*)
Whether the wheel steering is enabled.

Game Scenes All

boolean **getSteeringInverted**()

void **setSteeringInverted**(boolean *value*)
Whether the wheel steering is inverted.

Game Scenes All

boolean **getHasSuspension**()
Whether the wheel has suspension.

Game Scenes All

float **getSuspensionSpringStrength**()
Suspension spring strength, as set in the editor.

Game Scenes All

float **getSuspensionDamperStrength**()
Suspension damper strength, as set in the editor.

Game Scenes All

boolean **getBroken**()
Whether the wheel is broken.

Game Scenes All

boolean **getRepairable**()
Whether the wheel is repairable.

Game Scenes All

float **getStress**()
Current stress on the wheel.

Game Scenes All

float **getStressTolerance**()
Stress tolerance of the wheel.

Game Scenes All

float **getStressPercentage**()
Current stress on the wheel as a percentage of its stress tolerance.

Game Scenes All

float **getDeflection**()
Current deflection of the wheel.

Game Scenes All

float **getSlip**()
Current slip of the wheel.

Game Scenes All

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.

Traversing the Tree

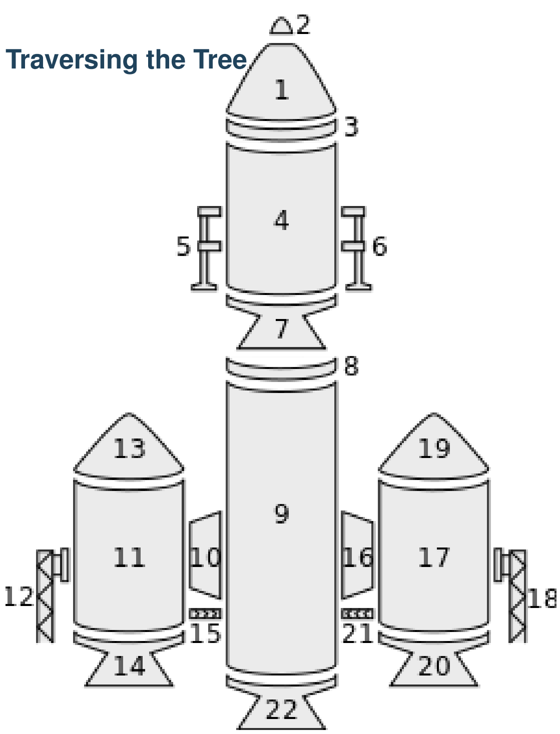


Fig. 10: **Figure 1** – Example parts making up a vessel.

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
```

(continues on next page)

(continued from previous page)

```
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(connec
  ␣
↪    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
```

(continues on next page)

(continued from previous page)

```
FL-T400 Fuel
 TT18-A Launc
 FTX-2 Extern
 LV-909 Liqui
 Aerodynamic
LT-1 Landing Stru
LT-1 Landing Stru
Mk16 Parachute
```

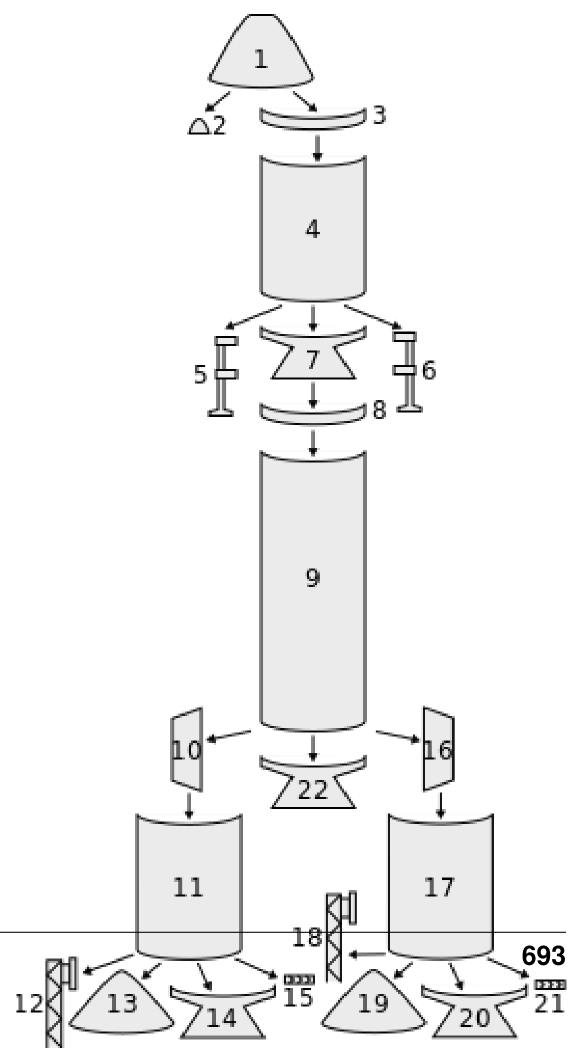
Attachment Modes

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) {
```

(continues on next page)

(continued from previous page)

```java
            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);
            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

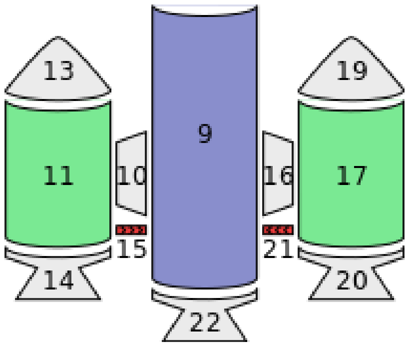


Fig. 12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

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).

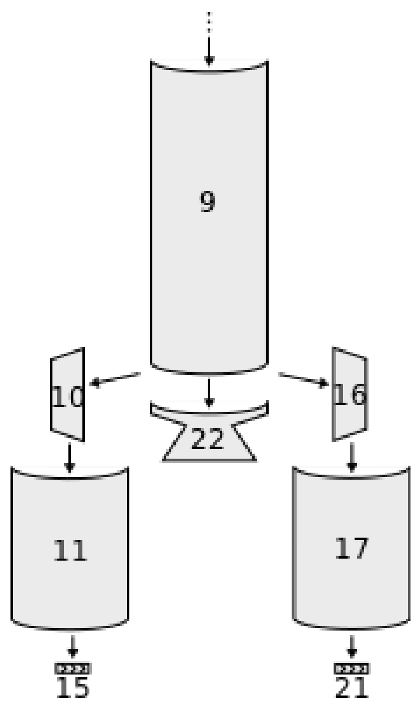


Fig. 13: **Figure 4** – A subset of the parts tree from Figure 2 above.

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*.

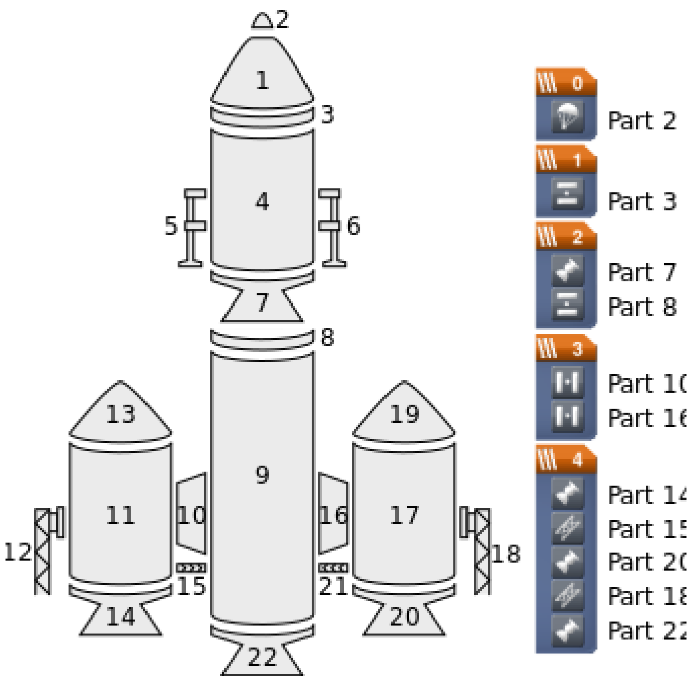


Fig. 14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

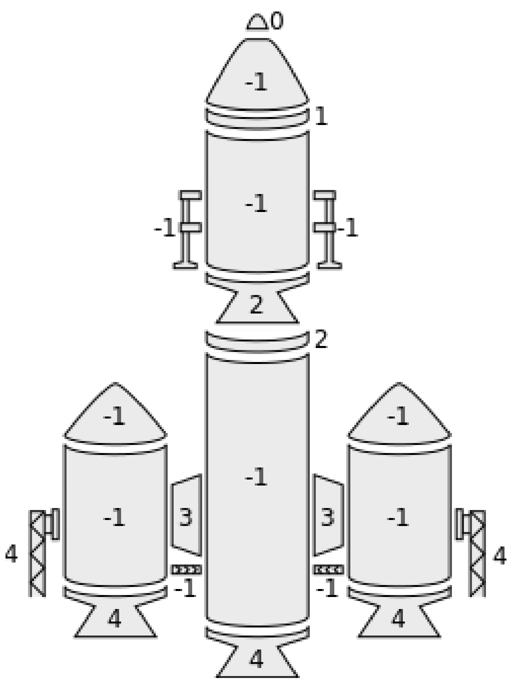


Fig. 15: **Figure 7** – The stage in which each part is *activated*.

6.3.9 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.

Game Scenes Flight

java.util.List<*Resource*> **withResource**(String *name*)
All the individual resources with the given name that can be stored.

Parameters

- **name** (`String`) –

Game Scenes Flight

java.util.List<String> **getNames**()
A list of resource names that can be stored.

Game Scenes Flight

boolean **hasResource**(String *name*)
Check whether the named resource can be stored.

Parameters

- **name** (`String`) – The name of the resource.

Game Scenes Flight

float **amount**(String *name*)
Returns the amount of a resource that is currently stored.

Parameters

- **name** (`String`) – The name of the resource.

Game Scenes Flight

float **max**(String *name*)
Returns the amount of a resource that can be stored.

Parameters

- **name** (`String`) – The name of the resource.

Game Scenes Flight

static float **density**(*Connection connection*, String *name*)
Returns the density of a resource, in kg/l.

Parameters

- **name** (`String`) – The name of the resource.

Game Scenes Flight

static *ResourceFlowMode* **flowMode**(*Connection connection*, String *name*)
Returns the flow mode of a resource.

Parameters

- **name** (`String`) – The name of the resource.

Game Scenes Flight

boolean **getEnabled**()

void **setEnabled**(boolean *value*)
Whether use of all the resources are enabled.

Game Scenes Flight

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.

Game Scenes All

Part **getPart**()
The part containing the resource.

Game Scenes All

float **getAmount**()
The amount of the resource that is currently stored in the part.

Game Scenes All

float **getMax**()
The total amount of the resource that can be stored in the part.

Game Scenes All

float **getDensity**()
The density of the resource, in kg/l.

Game Scenes All

ResourceFlowMode **getFlowMode**()
The flow mode of the resource.

Game Scenes All

boolean **getEnabled**()

void **setEnabled**(boolean *value*)
Whether use of this resource is enabled.

Game Scenes All

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.

Game Scenes All

float **getAmount**()

The amount of the resource that has been transferred.

Game Scenes All

boolean **getComplete**()

Whether the transfer has completed.

Game Scenes All

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.

6.3.10 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.

Game Scenes Flight

double **getNormal**()

void **setNormal** (double *value*)
The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

Game Scenes Flight

double **getRadial** ()

void **setRadial** (double *value*)
The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

Game Scenes Flight

double **getDeltaV** ()

void **setDeltaV** (double *value*)
The delta-v of the maneuver node, in meters per second.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

double **getTimeTo**()
The time until the maneuver node will be encountered, in seconds.

Game Scenes Flight

Orbit **getOrbit**()
The orbit that results from executing the maneuver node.

Game Scenes Flight

void **remove**()
Removes the maneuver node.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

6.3.11 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 $(0, 0, 0)$.

Game Scenes All

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.

Game Scenes All

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

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.

Game Scenes Flight

void **disengage**()
Disengage the auto-pilot.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

float **getPitchError**()
The error, in degrees, between the vessels current and target pitch. Throws an exception if the auto-pilot has not been engaged.

Game Scenes Flight

float **getHeadingError**()
The error, in degrees, between the vessels current and target heading. Throws an exception if the auto-pilot has not been engaged.

Game Scenes Flight

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.

Game Scenes Flight

ReferenceFrame **getReferenceFrame**()

void **setReferenceFrame**(*ReferenceFrame value*)
The reference frame for the target direction (`AutoPilot.getTargetDirection()`).

Game Scenes Flight

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°.

Game Scenes Flight

float **getTargetHeading** ()

void **setTargetHeading** (float *value*)
The target heading, in degrees, between 0° and 360°.

Game Scenes Flight

float **getTargetRoll** ()

void **setTargetRoll** (float *value*)
The target roll, in degrees. `NaN` if no target roll is set.

Game Scenes Flight

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*.

Game Scenes Flight

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°.

Game Scenes Flight

boolean **getSAS** ()

void **setSAS** (boolean *value*)
The state of SAS.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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()*.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

org.javatuples.Triplet<Double, Double, Double> **getPitchPIDGains** ()

void **setPitchPIDGains** (org.javatuples.Triplet<Double, Double, Double> *value*)
Gains for the pitch PID controller.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

Note: When *AutoPilot.getAutoTune()* is true, these values are updated automatically, which will overwrite any manual changes.

6.3.13 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.

Game Scenes Flight

float **getPitch** ()

void **setPitch** (float *value*)
The pitch of the camera, in degrees. A value between *Camera.getMinPitch()* and *Camera.getMaxPitch()*

Game Scenes Flight

float **getHeading** ()

void **setHeading** (float *value*)
The heading of the camera, in degrees.

Game Scenes Flight

float **getDistance** ()

void **setDistance** (float *value*)
The distance from the camera to the subject, in meters. A value between *Camera.getMinDistance()* and *Camera.getMaxDistance()*.

Game Scenes Flight

float **getMinPitch** ()
The minimum pitch of the camera.

Game Scenes Flight

float **getMaxPitch** ()
The maximum pitch of the camera.

Game Scenes Flight

float **getMinDistance** ()
Minimum distance from the camera to the subject, in meters.

Game Scenes Flight

float **getMaxDistance** ()
Maximum distance from the camera to the subject, in meters.

Game Scenes Flight

float **getDefaultDistance** ()
Default distance from the camera to the subject, in meters.

Game Scenes Flight

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 is the camera is not in map mode.

Game Scenes Flight

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 is the camera is not in map mode.

Game Scenes Flight

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 is the camera is not in map mode.

Game Scenes Flight

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.

6.3.14 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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

java.util.Map<String, Integer> **getColors**()

An example map of known color - seed pairs. Any other integers may be used as seed.

Game Scenes All

java.util.List<String> **getIcons**()

Returns all available icons (from "GameData/Squad/Contracts/Icons/").

Game Scenes All

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.

Game Scenes Flight

String **getName**()

void **setName**(String *value*)

The name of the waypoint as it appears on the map and the contract.

Game Scenes Flight

int **getColor**()

void **setColor**(int *value*)
The seed of the icon color. See `WaypointManager.getColors()` for example colors.

Game Scenes Flight

String **getIcon**()

void **setIcon**(String *value*)
The icon of the waypoint.

Game Scenes Flight

double **getLatitude**()

void **setLatitude**(double *value*)
The latitude of the waypoint.

Game Scenes Flight

double **getLongitude**()

void **setLongitude**(double *value*)
The longitude of the waypoint.

Game Scenes Flight

double **getMeanAltitude**()

void **setMeanAltitude**(double *value*)
The altitude of the waypoint above sea level, in meters.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

boolean **getNearSurface**()
`true` if the waypoint is near to the surface of a body.

Game Scenes Flight

boolean **getGrounded**()
`true` if the waypoint is attached to the ground.

Game Scenes Flight

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.

Game Scenes Flight

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()*.

Game Scenes Flight

boolean **getHasContract**()
Whether the waypoint belongs to a contract.

Game Scenes Flight

Contract **getContract**()
The associated contract.

Game Scenes Flight

void **remove**()
Removes the waypoint.

Game Scenes Flight

6.3.15 Contracts

public class **ContractManager**
Contracts manager. Obtained by calling *getContractManager()*.

java.util.Set<String> **getTypes**()
A list of all contract types.

Game Scenes All

java.util.List<*Contract*> **getAllContracts**()
A list of all contracts.

Game Scenes All

java.util.List<*Contract*> **getActiveContracts**()
A list of all active contracts.

Game Scenes All

java.util.List<*Contract*> **getOfferedContracts**()
A list of all offered, but unaccepted, contracts.

Game Scenes All

java.util.List<*Contract*> **getCompletedContracts**()
A list of all completed contracts.

Game Scenes All

java.util.List<*Contract*> **getFailedContracts**()
A list of all failed contracts.

Game Scenes All

public class **Contract**
A contract. Can be accessed using *getContractManager()*.

String **getType**()
Type of the contract.

Game Scenes All

String **getTitle**()
Title of the contract.

Game Scenes All

String **getDescription**()
Description of the contract.

Game Scenes All

String **getNotes**()
Notes for the contract.

Game Scenes All

String **getSynopsis**()
Synopsis for the contract.

Game Scenes All

java.util.List<String> **getKeywords**()
Keywords for the contract.

Game Scenes All

ContractState **getState**()
State of the contract.

Game Scenes All

boolean **getSeen**()
Whether the contract has been seen.

Game Scenes All

boolean **getRead**()
Whether the contract has been read.

Game Scenes All

boolean **getActive**()
Whether the contract is active.

Game Scenes All

boolean **getFailed**()
Whether the contract has been failed.

Game Scenes All

boolean **getCanBeCanceled**()
Whether the contract can be canceled.

Game Scenes All

boolean **getCanBeDeclined**()
Whether the contract can be declined.

Game Scenes All

boolean **getCanBeFailed**()
Whether the contract can be failed.

Game Scenes All

void **accept**()
Accept an offered contract.

Game Scenes All

void **cancel**()
Cancel an active contract.

Game Scenes All

void **decline**()
Decline an offered contract.

Game Scenes All

double **getFundsAdvance**()
Funds received when accepting the contract.

Game Scenes All

double **getFundsCompletion**()
Funds received on completion of the contract.

Game Scenes All

double **getFundsFailure**()
Funds lost if the contract is failed.

Game Scenes All

double **getReputationCompletion**()
Reputation gained on completion of the contract.

Game Scenes All

double **getReputationFailure**()
Reputation lost if the contract is failed.

Game Scenes All

double **getScienceCompletion**()
Science gained on completion of the contract.

Game Scenes All

java.util.List<*ContractParameter*> **getParameters**()
Parameters for the contract.

Game Scenes All

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.

Game Scenes All

String **getNotes**()
Notes for the parameter.

Game Scenes All

java.util.List<*ContractParameter*> **getChildren**()
Child contract parameters.

Game Scenes All

boolean **getCompleted**()
Whether the parameter has been completed.

Game Scenes All

boolean **getFailed**()
Whether the parameter has been failed.

Game Scenes All

boolean **getOptional**()
Whether the contract parameter is optional.

Game Scenes All

double **getFundsCompletion**()
Funds received on completion of the contract parameter.

Game Scenes All

double **getFundsFailure**()

Funds lost if the contract parameter is failed.

Game Scenes All

double **getReputationCompletion**()

Reputation gained on completion of the contract parameter.

Game Scenes All

double **getReputationFailure**()

Reputation lost if the contract parameter is failed.

Game Scenes All

double **getScienceCompletion**()

Science gained on completion of the contract parameter.

Game Scenes All

6.3.16 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();
    }
}
```

6.4 Drawing API

6.4.1 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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

void **clear** (boolean *clientOnly*)
Remove all objects being drawn.

Parameters

- **clientOnly** (*boolean*) – If true, only remove objects created by the calling client.

Game Scenes Flight

6.4.2 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.

Game Scenes Flight

org.javatuples.Triplet<Double, Double, Double> **getEnd**()

void **setEnd**(org.javatuples.Triplet<Double, Double, Double> *value*)

End position of the line.

Game Scenes Flight

SpaceCenter.ReferenceFrame **getReferenceFrame**()

void **setReferenceFrame**(*SpaceCenter.ReferenceFrame value*)

Reference frame for the positions of the object.

Game Scenes Flight

boolean **getVisible**()

void **setVisible**(boolean *value*)

Whether the object is visible.

Game Scenes Flight

org.javatuples.Triplet<Double, Double, Double> **getColor**()

void **setColor**(org.javatuples.Triplet<Double, Double, Double> *value*)

Set the color

Game Scenes Flight

String **getMaterial**()

void **setMaterial**(String *value*)

Material used to render the object. Creates the material from a shader with the given name.

Game Scenes Flight

float **getThickness**()

void **setThickness**(float *value*)

Set the thickness

Game Scenes Flight

void **remove**()
Remove the object.

Game Scenes Flight

6.4.3 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.

Game Scenes Flight

SpaceCenter.ReferenceFrame **getReferenceFrame**()

void **setReferenceFrame**(*SpaceCenter.ReferenceFrame value*)
Reference frame for the positions of the object.

Game Scenes Flight

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the object is visible.

Game Scenes Flight

void **remove**()
Remove the object.

Game Scenes Flight

org.javatuples.Triplet<Double, Double, Double> **getColor**()

void **setColor**(org.javatuples.Triplet<Double, Double, Double> *value*)
Set the color

Game Scenes Flight

String **getMaterial**()

void **setMaterial**(String *value*)
Material used to render the object. Creates the material from a shader with the given name.

Game Scenes Flight

float **getThickness**()

void **setThickness** (float *value*)
Set the thickness

Game Scenes Flight

6.4.4 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.

Game Scenes Flight

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.

Game Scenes Flight

SpaceCenter.ReferenceFrame **getReferenceFrame** ()

void **setReferenceFrame** (*SpaceCenter.ReferenceFrame value*)
Reference frame for the positions of the object.

Game Scenes Flight

boolean **getVisible** ()

void **setVisible** (boolean *value*)
Whether the object is visible.

Game Scenes Flight

void **remove** ()
Remove the object.

Game Scenes Flight

String **getContent** ()

void **setContent** (String *value*)
The text string

Game Scenes Flight

String **getFont** ()

void **setFont** (String *value*)
Name of the font

Game Scenes Flight

static java.util.List<String> **availableFonts** (*Connection connection*)
A list of all available fonts.

Game Scenes Flight

int **getSize** ()

void **setSize** (int *value*)
Font size.

Game Scenes Flight

float **getCharacterSize** ()

void **setCharacterSize** (float *value*)
Character size.

Game Scenes Flight

UI.FontStyle **getStyle** ()

void **setStyle** (*UI.FontStyle value*)
Font style.

Game Scenes Flight

org.javatuples.Triplet<Double, Double, Double> **getColor** ()

void **setColor** (org.javatuples.Triplet<Double, Double, Double> *value*)
Set the color

Game Scenes Flight

String **getMaterial** ()

void **setMaterial** (String *value*)
Material used to render the object. Creates the material from a shader with the given name.

Game Scenes Flight

UI.TextAlignment **getAlignment** ()

void **setAlignment** (*UI.TextAlignment value*)
Alignment.

Game Scenes Flight

float **getLineSpacing** ()

void **setLineSpacing** (float *value*)
Line spacing.

Game Scenes Flight

UI.TextAnchor **getAnchor**()

void **setAnchor**(*UI.TextAnchor value*)
Anchor.

Game Scenes Flight

6.5 InfernalRobotics API

Provides RPCs to interact with the InfernalRobotics mod. Both the original mod and Infernal Robotics Next are supported. Provides the following classes:

6.5.1 InfernalRobotics

public class **InfernalRobotics**
This service provides functionality to interact with Infernal Robotics.

boolean **getAvailable**()
Whether Infernal Robotics is installed.

Game Scenes Flight

boolean **getReady**()
Whether Infernal Robotics API is ready.

Game Scenes Flight

java.util.List<*ServoGroup*> **servoGroups**(*SpaceCenter.Vessel vessel*)
A list of all the servo groups in the given *vessel*.

Parameters

- **vessel** (`SpaceCenter.Vessel`) –

Game Scenes Flight

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.

Game Scenes Flight

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.

Game Scenes Flight

6.5.2 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.

Game Scenes Flight

String **getForwardKey**()

void **setForwardKey**(String *value*)

The key assigned to be the "forward" key for the group.

Game Scenes Flight

String **getReverseKey**()

void **setReverseKey**(String *value*)

The key assigned to be the "reverse" key for the group.

Game Scenes Flight

float **getSpeed**()

void **setSpeed**(float *value*)

The speed multiplier for the group.

Game Scenes Flight

boolean **getExpanded**()

void **setExpanded**(boolean *value*)

Whether the group is expanded in the InfernalRobotics UI.

Game Scenes Flight

java.util.List<*Servo*> **getServos**()

The servos that are in the group.

Game Scenes Flight

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.

Game Scenes Flight

java.util.List<*SpaceCenter.Part*> **getParts**()
The parts containing the servos in the group.

Game Scenes Flight

void **moveRight**()
Moves all of the servos in the group to the right.

Game Scenes Flight

void **moveLeft**()
Moves all of the servos in the group to the left.

Game Scenes Flight

void **moveCenter**()
Moves all of the servos in the group to the center.

Game Scenes Flight

void **moveNextPreset**()
Moves all of the servos in the group to the next preset.

Game Scenes Flight

void **movePrevPreset**()
Moves all of the servos in the group to the previous preset.

Game Scenes Flight

void **stop**()
Stops the servos in the group.

Game Scenes Flight

6.5.3 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.

Game Scenes Flight

SpaceCenter.Part **getPart**()
The part containing the servo.

Game Scenes Flight

void **setHighlight**(boolean *value*)
Whether the servo should be highlighted in-game.

Game Scenes Flight

float **getPosition**()
The position of the servo.

Game Scenes Flight

float **getMinConfigPosition**()
The minimum position of the servo, specified by the part configuration.

Game Scenes Flight

float **getMaxConfigPosition**()
The maximum position of the servo, specified by the part configuration.

Game Scenes Flight

float **getMinPosition**()

void **setMinPosition**(float *value*)
The minimum position of the servo, specified by the in-game tweak menu.

Game Scenes Flight

float **getMaxPosition**()

void **setMaxPosition**(float *value*)
The maximum position of the servo, specified by the in-game tweak menu.

Game Scenes Flight

float **getConfigSpeed**()
The speed multiplier of the servo, specified by the part configuration.

Game Scenes Flight

float **getSpeed**()

void **setSpeed**(float *value*)
The speed multiplier of the servo, specified by the in-game tweak menu.

Game Scenes Flight

float **getCurrentSpeed**()

void **setCurrentSpeed**(float *value*)
The current speed at which the servo is moving.

Game Scenes Flight

float **getAcceleration**()

void **setAcceleration**(float *value*)
The current speed multiplier set in the UI.

Game Scenes Flight

boolean **getIsMoving**()
Whether the servo is moving.

Game Scenes Flight

boolean **getIsFreeMoving**()
Whether the servo is freely moving.

Game Scenes Flight

boolean **getIsLocked**()

void **setIsLocked**(boolean *value*)
Whether the servo is locked.

Game Scenes Flight

boolean **getIsAxisInverted**()

void **setIsAxisInverted**(boolean *value*)
Whether the servos axis is inverted.

Game Scenes Flight

void **moveRight**()
Moves the servo to the right.

Game Scenes Flight

void **moveLeft**()
Moves the servo to the left.

Game Scenes Flight

void **moveCenter**()
Moves the servo to the center.

Game Scenes Flight

void **moveNextPreset**()
Moves the servo to the next preset.

Game Scenes Flight

void **movePrevPreset**()
Moves the servo to the previous preset.

Game Scenes Flight

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.

Game Scenes Flight

void **stop**()
Stops the servo.

Game Scenes Flight

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.

```java
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();
    }
}
```

6.6 Kerbal Alarm Clock API

Provides RPCs to interact with the Kerbal Alarm Clock mod. Provides the following classes:

6.6.1 KerbalAlarmClock

public class **KerbalAlarmClock**

This service provides functionality to interact with Kerbal Alarm Clock.

boolean **getAvailable**()

Whether Kerbal Alarm Clock is available.

Game Scenes All

java.util.List<*Alarm*> **getAlarms**()

A list of all the alarms.

Game Scenes All

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.

Game Scenes All

java.util.List<*Alarm*> **alarmsWithType**(*AlarmType type*)

Get a list of alarms of the specified *type*.

Parameters

- **type** (AlarmType) – Type of alarm to return.

Game Scenes All

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.

Game Scenes All

6.6.2 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.

Game Scenes All

double **getMargin**()

void **setMargin**(double *value*)
The number of seconds before the event that the alarm will fire.

Game Scenes All

double **getTime**()

void **setTime**(double *value*)
The time at which the alarm will fire.

Game Scenes All

AlarmType **getType**()
The type of the alarm.

Game Scenes All

String **getID**()
The unique identifier for the alarm.

Game Scenes All

String **getName**()

void **setName**(String *value*)
The short name of the alarm.

Game Scenes All

String **getNotes**()

void **setNotes**(String *value*)
The long description of the alarm.

Game Scenes All

double **getRemaining**()
The number of seconds until the alarm will fire.

Game Scenes All

boolean **getRepeat**()

void **setRepeat**(boolean *value*)
Whether the alarm will be repeated after it has fired.

Game Scenes All

double **getRepeatPeriod**()

void **setRepeatPeriod**(double *value*)
The time delay to automatically create an alarm after it has fired.

Game Scenes All

SpaceCenter.Vessel **getVessel**()

void **setVessel** (*SpaceCenter.Vessel value*)
The vessel that the alarm is attached to.

Game Scenes All

SpaceCenter.CelestialBody **getXferOriginBody** ()

void **setXferOriginBody** (*SpaceCenter.CelestialBody value*)
The celestial body the vessel is departing from.

Game Scenes All

SpaceCenter.CelestialBody **getXferTargetBody** ()

void **setXferTargetBody** (*SpaceCenter.CelestialBody value*)
The celestial body the vessel is arriving at.

Game Scenes All

void **remove** ()
Removes the alarm.

Game Scenes All

6.6.3 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`.

6.6.4 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.

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.

```
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();
    }
}
```

6.7 RemoteTech API

Provides RPCs to interact with the RemoteTech mod. Provides the following classes:

6.7.1 RemoteTech

public class **RemoteTech**

This service provides functionality to interact with RemoteTech.

boolean **getAvailable**()

Whether RemoteTech is installed.

Game Scenes All

java.util.List<String> **getGroundStations**()

The names of the ground stations.

Game Scenes All

Antenna **antenna** (*SpaceCenter.Part part*)
Get the antenna object for a particular part.

Parameters

- **part** (SpaceCenter.Part) –

Game Scenes All

Comms **comms** (*SpaceCenter.Vessel vessel*)
Get a communications object, representing the communication capability of a particular vessel.

Parameters

- **vessel** (SpaceCenter.Vessel) –

Game Scenes All

6.7.2 Comms

public class **Comms**
Communications for a vessel.

SpaceCenter.Vessel **getVessel** ()
Get the vessel.

Game Scenes All

boolean **getHasLocalControl** ()
Whether the vessel can be controlled locally.

Game Scenes All

boolean **getHasFlightComputer** ()
Whether the vessel has a flight computer on board.

Game Scenes All

boolean **getHasConnection** ()
Whether the vessel has any connection.

Game Scenes All

boolean **getHasConnectionToGroundStation** ()
Whether the vessel has a connection to a ground station.

Game Scenes All

double **getSignalDelay** ()
The shortest signal delay to the vessel, in seconds.

Game Scenes All

double **getSignalDelayToGroundStation** ()
The signal delay between the vessel and the closest ground station, in seconds.

Game Scenes All

double **signalDelayToVessel** (*SpaceCenter.Vessel other*)
The signal delay between the this vessel and another vessel, in seconds.

Parameters

- **other** (SpaceCenter.Vessel) –

Game Scenes All

java.util.List<*Antenna*> **getAntennas** ()
The antennas for this vessel.

Game Scenes All

6.7.3 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.

Game Scenes All

boolean **getHasConnection** ()
Whether the antenna has a connection.

Game Scenes All

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()*.

Game Scenes All

SpaceCenter.CelestialBody **getTargetBody** ()

void **setTargetBody** (*SpaceCenter.CelestialBody value*)
The celestial body the antenna is targetting.

Game Scenes All

String **getTargetGroundStation** ()

void **setTargetGroundStation** (String *value*)
The ground station the antenna is targetting.

Game Scenes All

SpaceCenter.Vessel **getTargetVessel** ()

void **setTargetVessel** (*SpaceCenter.Vessel value*)
The vessel the antenna is targetting.

Game Scenes All

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.

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.

```java
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);
        antenna.setTargetBody(sc.getBodies().get("Jool"));

        // Get info about the vessels communications
        Comms comms = rt.comms(vessel);
        System.out.printf("Signal delay = %.1f seconds\n", comms.getSignalDelay());
```

(continues on next page)

(continued from previous page)

```
        connection.close();
    }
}
```

6.8 User Interface API

6.8.1 UI

public class **UI**

Provides functionality for drawing and interacting with in-game user interface elements.

Canvas **getStockCanvas**()

The stock UI canvas.

Game Scenes All

Canvas **addCanvas**()

Add a new canvas.

Game Scenes All

Note: If you want to add UI elements to KSPs stock UI canvas, use *getStockCanvas()*.

void **message**(String *content*, float *duration*, *MessagePosition position*, org.javatuples.Triplet<Double, Double, Double> *color*, float *size*)

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.
- **color** (*org.javatuples.Triplet<Double,Double, Double>*) – The color of the message.
- **size** (*float*) – Size of the message, differs per position.

Game Scenes All

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.

Game Scenes All

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.

6.8.2 Canvas

public class **Canvas**
A canvas for user interface elements. See *getStockCanvas()* and *addCanvas()*.

RectTransform **getRectTransform**()
The rect transform for the canvas.

Game Scenes All

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the UI object is visible.

Game Scenes All

Panel **addPanel**(boolean *visible*)
Create a new container for user interface elements.

Parameters

- **visible** (*boolean*) – Whether the panel is visible.

Game Scenes All

Text **addText**(String *content*, boolean *visible*)
Add text to the canvas.

Parameters

- **content** (*String*) – The text.
- **visible** (*boolean*) – Whether the text is visible.

Game Scenes All

InputField **addInputField**(boolean *visible*)
Add an input field to the canvas.

Parameters

- **visible** (*boolean*) – Whether the input field is visible.

Game Scenes All

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.

Game Scenes All

void **remove**()
Remove the UI object.

Game Scenes All

6.8.3 Panel

public class **Panel**
A container for user interface elements. See *Canvas.addPanel(boolean)*.

RectTransform **getRectTransform**()
The rect transform for the panel.

Game Scenes All

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the UI object is visible.

Game Scenes All

Panel **addPanel**(boolean *visible*)
Create a panel within this panel.

Parameters

- **visible** (*boolean*) – Whether the new panel is visible.

Game Scenes All

Text **addText**(String *content*, boolean *visible*)
Add text to the panel.

Parameters

- **content** (*String*) – The text.
- **visible** (*boolean*) – Whether the text is visible.

Game Scenes All

InputField **addInputField**(boolean *visible*)
Add an input field to the panel.

Parameters

- **visible** (*boolean*) – Whether the input field is visible.

Game Scenes All

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.

Game Scenes All

void **remove**()
Remove the UI object.

Game Scenes All

6.8.4 Text

public class **Text**
A text label. See `Panel.addText(String, boolean)`.

RectTransform **getRectTransform**()
The rect transform for the text.

Game Scenes All

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the UI object is visible.

Game Scenes All

String **getContent**()

void **setContent**(String *value*)
The text string

Game Scenes All

String **getFont**()

void **setFont**(String *value*)
Name of the font

Game Scenes All

java.util.List<String> **getAvailableFonts**()
A list of all available fonts.

Game Scenes All

int **getSize**()

void **setSize**(int *value*)
Font size.

Game Scenes All

FontStyle **getStyle**()

void **setStyle** (*FontStyle value*)
Font style.

Game Scenes All

org.javatuples.Triplet<Double, Double, Double> **getColor** ()

void **setColor** (org.javatuples.Triplet<Double, Double, Double> *value*)
Set the color

Game Scenes All

TextAnchor **getAlignment** ()

void **setAlignment** (*TextAnchor value*)
Alignment.

Game Scenes All

float **getLineSpacing** ()

void **setLineSpacing** (float *value*)
Line spacing.

Game Scenes All

void **remove** ()
Remove the UI object.

Game Scenes All

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.

6.8.5 Button

public class **Button**
A text label. See *Panel.addButton(String, boolean)*.

RectTransform **getRectTransform**()
The rect transform for the text.

Game Scenes All

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the UI object is visible.

Game Scenes All

Text **getText**()
The text for the button.

Game Scenes All

boolean **getClicked**()

void **setClicked**(boolean *value*)
Whether the button has been clicked.

Game Scenes All

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.

Game Scenes All

6.8.6 InputField

public class **InputField**
An input field. See *Panel.addInputField(boolean)*.

RectTransform **getRectTransform**()
The rect transform for the input field.

Game Scenes All

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the UI object is visible.

Game Scenes All

String **getValue**()

void **setValue**(String *value*)
The value of the input field.

Game Scenes All

Text **getText**()
The text component of the input field.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

6.8.7 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.

Game Scenes All

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.

Game Scenes All

org.javatuples.Pair<Double, Double> **getSize**()

void **setSize**(org.javatuples.Pair<Double, Double> *value*)

Width and height of the rectangle.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

CHAPTER

SEVEN

LUA

7.1 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.

7.1.1 Installing the Library

The Lua client and all of its dependencies can be installed using luarocks with a single command:

```
luarocks install krpc
```

7.1.2 Using the Library

Once it's installed, simply `require 'krpc'` and you are good to go!

7.1.3 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:

```lua
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:

```lua
local krpc = require 'krpc'
local conn = krpc.connect('Remote example', 'my.domain.name', 1000, 1001)
print(conn.krpc:get_status().version)
```

7.1.4 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.

7.1.5 Streaming Data from the Server

Streams are not yet supported by the Lua client.

7.1.6 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:

```lua
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))
```

7.2 KRPC API

7.2.1 KRPC

None 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.

7.2.2 Expressions

class Expression
A server side expression.

static constant_double(*value*)
A constant value of double precision floating point type.

Parameters **value** (*number*) –

Return type *KRPC.Expression*

static constant_float(*value*)
A constant value of single precision floating point type.

Parameters **value** (*number*) –

Return type *KRPC.Expression*

static constant_int(*value*)
A constant value of integer type.

Parameters **value** (*number*) –

Return type *KRPC.Expression*

static constant_bool(*value*)
A constant value of boolean type.

Parameters **value** (*boolean*) –

Return type *KRPC.Expression*

static constant_string(*value*)
A constant value of string type.

Parameters **value** (*string*) –

Return type *KRPC.Expression*

static call (*call*)
An RPC call.

Parameters **call** (*krpc.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, with type of arg0

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 cast (*arg*, *type*)
Perform a cast to the given type.

Parameters

- **arg** (`KRPC.Expression`) –
- **type** (`KRPC.Type`) – Type to cast the argument to.

Return type *KRPC.Expression*

static parameter (*name*, *type*)
A named parameter of type double.

Parameters

- **name** (*string*) – The name of the parameter.
- **type** (`KRPC.Type`) – The type of the parameter.

Returns A named parameter.

Return type *KRPC.Expression*

static function (*parameters*, *body*)
A function.

Parameters

- **parameters** (*List*) – The parameters of the function.
- **body** (`KRPC.Expression`) – The body of the function.

Returns A function.

Return type *KRPC.Expression*

static invoke (*function*, *args*)
A function call.

Parameters

- **function** (`KRPC.Expression`) – The function to call.
- **args** (*Map*) – The arguments to call the function with.

Returns A function call.

Return type *KRPC.Expression*

static create_tuple (*elements*)
Construct a tuple.

Parameters **elements** (*List*) – The elements.

Returns The tuple.

Return type *KRPC.Expression*

static create_list (*values*)
Construct a list.

Parameters **values** (*List*) – The value. Should all be of the same type.

Returns The list.

Return type *KRPC.Expression*

static create_set (*values*)

Construct a set.

Parameters **values** (*Set*) – The values. Should all be of the same type.

Returns The set.

Return type *KRPC.Expression*

static create_dictionary (*keys*, *values*)

Construct a dictionary, from a list of corresponding keys and values.

Parameters

- **keys** (*List*) – The keys. Should all be of the same type.
- **values** (*List*) – The values. Should all be of the same type.

Returns The dictionary.

Return type *KRPC.Expression*

static to_list (*arg*)

Convert a collection to a list.

Parameters **arg** (KRPC.Expression) – The collection.

Returns The collection as a list.

Return type *KRPC.Expression*

static to_set (*arg*)

Convert a collection to a set.

Parameters **arg** (KRPC.Expression) – The collection.

Returns The collection as a set.

Return type *KRPC.Expression*

static get (*arg*, *index*)

Access an element in a tuple, list or dictionary.

Parameters

- **arg** (KRPC.Expression) – The tuple, list or dictionary.
- **index** (KRPC.Expression) – The index of the element to access. A zero indexed integer for a tuple or list, or a key for a dictionary.

Returns The element.

Return type *KRPC.Expression*

static count (*arg*)

Number of elements in a collection.

Parameters **arg** (KRPC.Expression) – The list, set or dictionary.

Returns The number of elements in the collection.

Return type *KRPC.Expression*

static sum (*arg*)

Sum all elements of a collection.

Parameters **arg** (KRPC.Expression) – The list or set.

Returns The sum of the elements in the collection.

Return type *KRPC.Expression*

static max(*arg*)
Maximum of all elements in a collection.

Parameters **arg** (KRPC.Expression) – The list or set.

Returns The maximum elements in the collection.

Return type *KRPC.Expression*

static min(*arg*)
Minimum of all elements in a collection.

Parameters **arg** (KRPC.Expression) – The list or set.

Returns The minimum elements in the collection.

Return type *KRPC.Expression*

static average(*arg*)
Minimum of all elements in a collection.

Parameters **arg** (KRPC.Expression) – The list or set.

Returns The minimum elements in the collection.

Return type *KRPC.Expression*

static select(*arg*, *func*)
Run a function on every element in the collection.

Parameters

- **arg** (KRPC.Expression) – The list or set.
- **func** (KRPC.Expression) – The function.

Returns The modified collection.

Return type *KRPC.Expression*

static where(*arg*, *func*)
Run a function on every element in the collection.

Parameters

- **arg** (KRPC.Expression) – The list or set.
- **func** (KRPC.Expression) – The function.

Returns The modified collection.

Return type *KRPC.Expression*

static contains(*arg*, *value*)
Determine if a collection contains a value.

Parameters

- **arg** (KRPC.Expression) – The collection.
- **value** (KRPC.Expression) – The value to look for.

Returns Whether the collection contains a value.

Return type *KRPC.Expression*

static aggregate(*arg*, *func*)
Applies an accumulator function over a sequence.

Parameters

- **arg** (KRPC.Expression) – The collection.
- **func** (KRPC.Expression) – The accumulator function.

Returns The accumulated value.

Return type *KRPC.Expression*

static aggregate_with_seed(*arg*, *seed*, *func*)
Applies an accumulator function over a sequence, with a given seed.

Parameters

- **arg** (KRPC.Expression) – The collection.
- **seed** (KRPC.Expression) – The seed value.
- **func** (KRPC.Expression) – The accumulator function.

Returns The accumulated value.

Return type *KRPC.Expression*

static concat(*arg1*, *arg2*)
Concatenate two sequences.

Parameters

- **arg1** (KRPC.Expression) – The first sequence.
- **arg2** (KRPC.Expression) – The second sequence.

Returns The first sequence followed by the second sequence.

Return type *KRPC.Expression*

static order_by(*arg*, *key*)
Order a collection using a key function.

Parameters

- **arg** (KRPC.Expression) – The collection to order.
- **key** (KRPC.Expression) – A function that takes a value from the collection and generates a key to sort on.

Returns The ordered collection.

Return type *KRPC.Expression*

static all(*arg*, *predicate*)
Determine whether all items in a collection satisfy a boolean predicate.

Parameters

- **arg** (KRPC.Expression) – The collection.
- **predicate** (KRPC.Expression) – The predicate function.

Returns Whether all items satisfy the predicate.

Return type *KRPC.Expression*

static any (*arg*, *predicate*)
Determine whether any item in a collection satisfies a boolean predicate.

Parameters

- **arg** (KRPC.Expression) – The collection.
- **predicate** (KRPC.Expression) – The predicate function.

Returns Whether any item satisfies the predicate.

Return type *KRPC.Expression*

class Type
A server side expression.

static double ()
Double type.

Return type *KRPC.Type*

static float ()
Float type.

Return type *KRPC.Type*

static int ()
Int type.

Return type *KRPC.Type*

static bool ()
Bool type.

Return type *KRPC.Type*

static string ()
String type.

Return type *KRPC.Type*

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.

science
The current amount of science.

Attribute Read-only, cannot be set

Return type number

funds
The current amount of funds.

Attribute Read-only, cannot be set

Return type number

reputation
The current amount of reputation.

Attribute Read-only, cannot be set

Return type number

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*[, *recover = True*])
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".
- **recover** (*boolean*) – If true and there is a vessel on the launch site, recover it before launching.

Note: Throws an exception if any of the games pre-flight checks fail.

static launch_vessel_from_vab(*name*[, *recover = True*])

Launch a new vessel from the VAB onto the launchpad.

Parameters

- **name** (*string*) – Name of the vessel to launch.
- **recover** (*boolean*) – If true and there is a vessel on the launch pad, recover it before launching.

Note: This is equivalent to calling *SpaceCenter.launch_vessel()* with the craft directory set to "VAB" and the launch site set to "LaunchPad". Throws an exception if any of the games pre-flight checks fail.

static launch_vessel_from_sph(*name*[, *recover = True*])

Launch a new vessel from the SPH onto the runway.

Parameters

- **name** (*string*) – Name of the vessel to launch.
- **recover** (*boolean*) – If true and there is a vessel on the runway, recover it before launching.

Note: This is equivalent to calling *SpaceCenter.launch_vessel()* with the craft directory set to "SPH" and the launch site set to "Runway". Throws an exception if any of the games pre-flight checks fail.

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 covert 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 covert 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 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 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 covert the velocity to.

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

Return type Tuple of (number, number, number)

static raycast_distance (*position*, *direction*, *reference_frame*)

Cast a ray from a given position in a given direction, and return the distance to the hit point. If no hit occurs, returns infinity.

Parameters

- **position** (*Tuple*) – Position, as a vector, of the origin of the ray.
- **direction** (*Tuple*) – Direction of the ray, as a unit vector.
- **reference_frame** (SpaceCenter.ReferenceFrame) – The reference frame that the position and direction are in.

Returns The distance to the hit, in meters, or infinity if there was no hit.

Return type number

static raycast_part (*position*, *direction*, *reference_frame*)

Cast a ray from a given position in a given direction, and return the part that it hits. If no hit occurs, returns `nil`.

Parameters

- **position** (*Tuple*) – Position, as a vector, of the origin of the ray.
- **direction** (*Tuple*) – Direction of the ray, as a unit vector.
- **reference_frame** (SpaceCenter.ReferenceFrame) – The reference frame that the position and direction are in.

Returns The part that was hit or `nil` if there was no hit.

Return type *SpaceCenter.Part*

far_available

Whether Ferram Aerospace Research is installed.

Attribute Read-only, cannot be set

Return type boolean

game_mode

The current mode the game is in.

Attribute Read-only, cannot be set

Return type *SpaceCenter.GameMode*

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 GameMode

The game mode. Returned by *SpaceCenter.GameMode*

sandbox

Sandbox mode.

career

Career mode.

science

Science career mode.

science_sandbox

Science sandbox mode.

mission

Mission mode.

mission_builder

Mission builder mode.

scenario
Scenario mode.

scenario_non_resumable
Scenario mode that cannot be resumed.

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.

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 *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*

crew_capacity
The number of crew that can occupy the vessel.

Attribute Read-only, cannot be set

Return type number

crew_count
The number of crew that are occupying the vessel.

Attribute Read-only, cannot be set

Return type number

crew
The crew in the vessel.

Attribute Read-only, cannot be set

Return type List of *SpaceCenter.CrewMember*

resources
A *SpaceCenter.Resources* object, that can 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 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 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, gimballed 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*

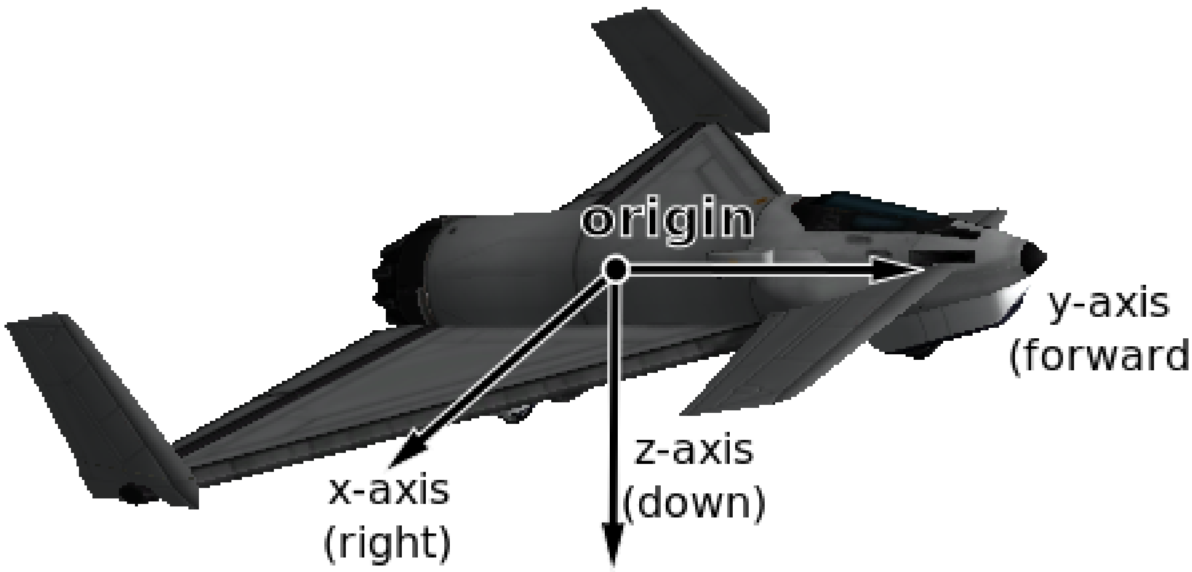


Fig. 1: Vessel reference frame origin and axes for the Aeris 3A aircraft

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 *SpaceCenter.ReferenceFrame*

Note: Be careful not to confuse this with 'orbit' mode on the navball.

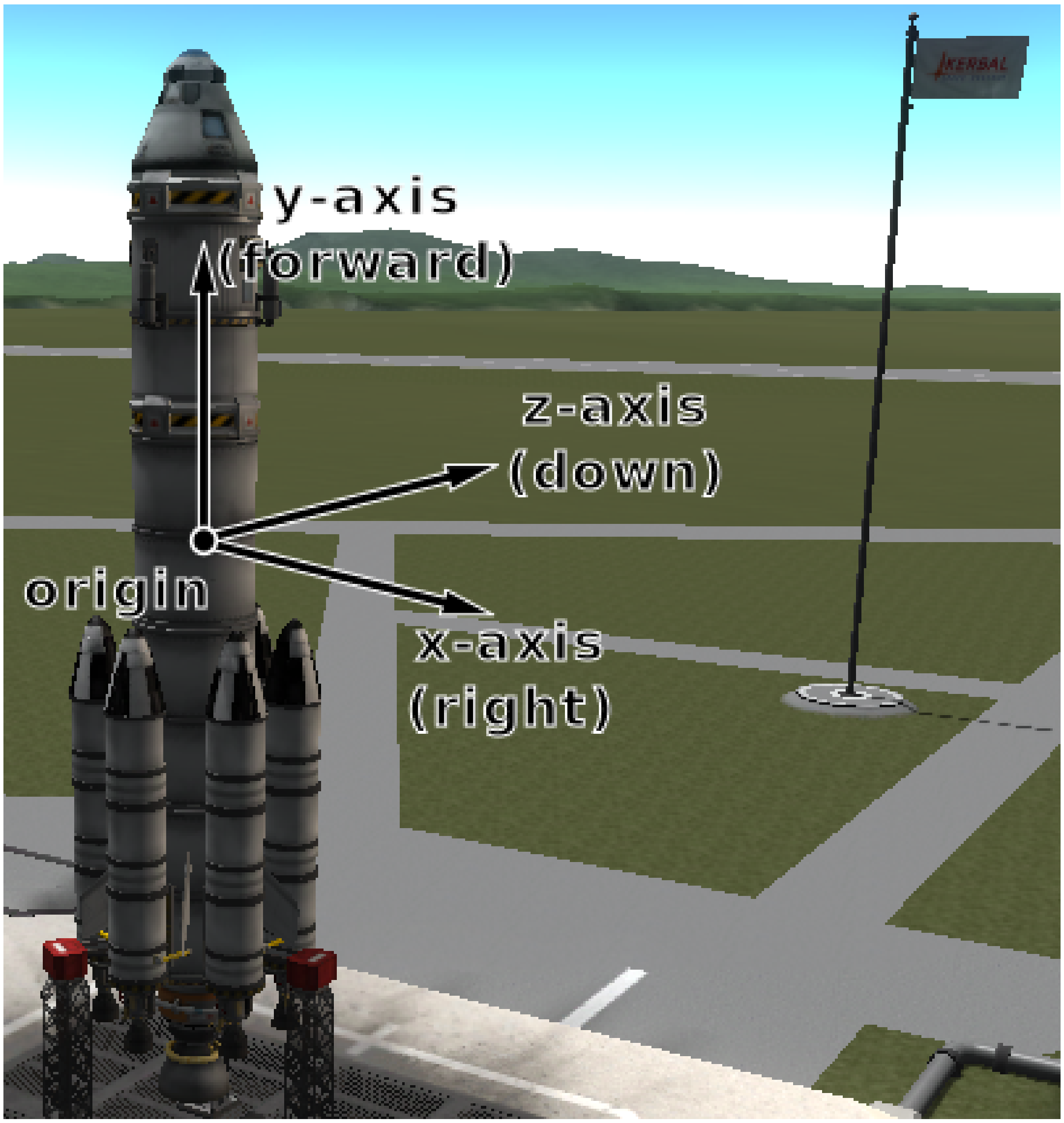


Fig. 2: Vessel reference frame origin and axes for the Kerbal-X rocket

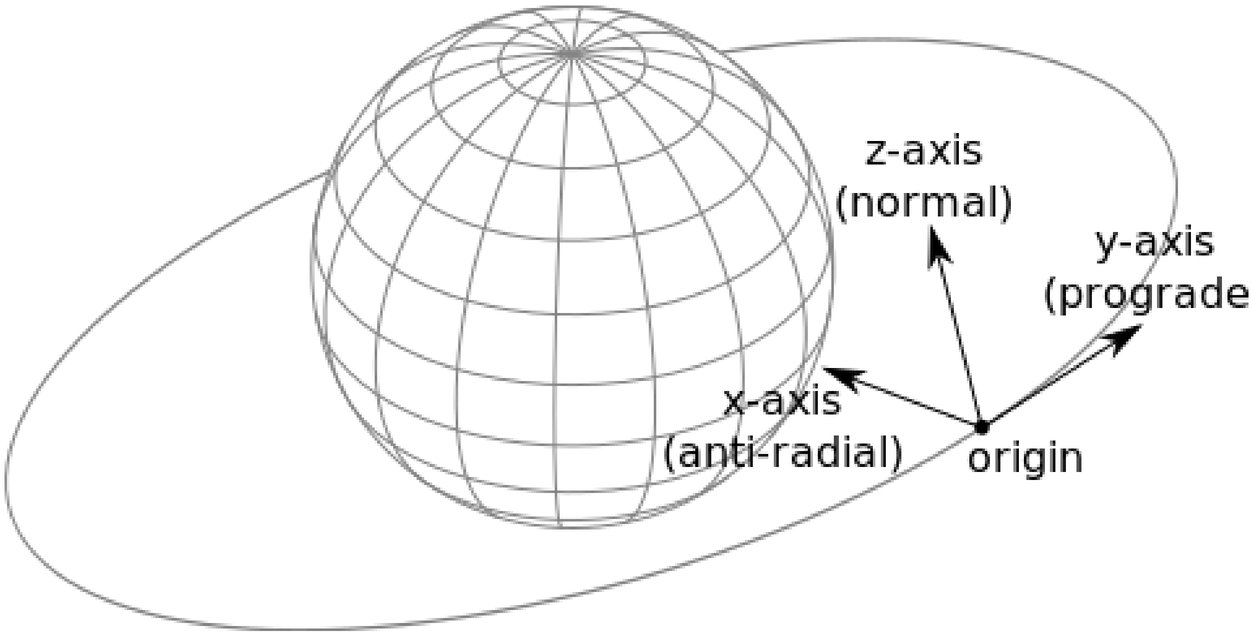


Fig. 3: Vessel orbital reference frame origin and axes

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*

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.

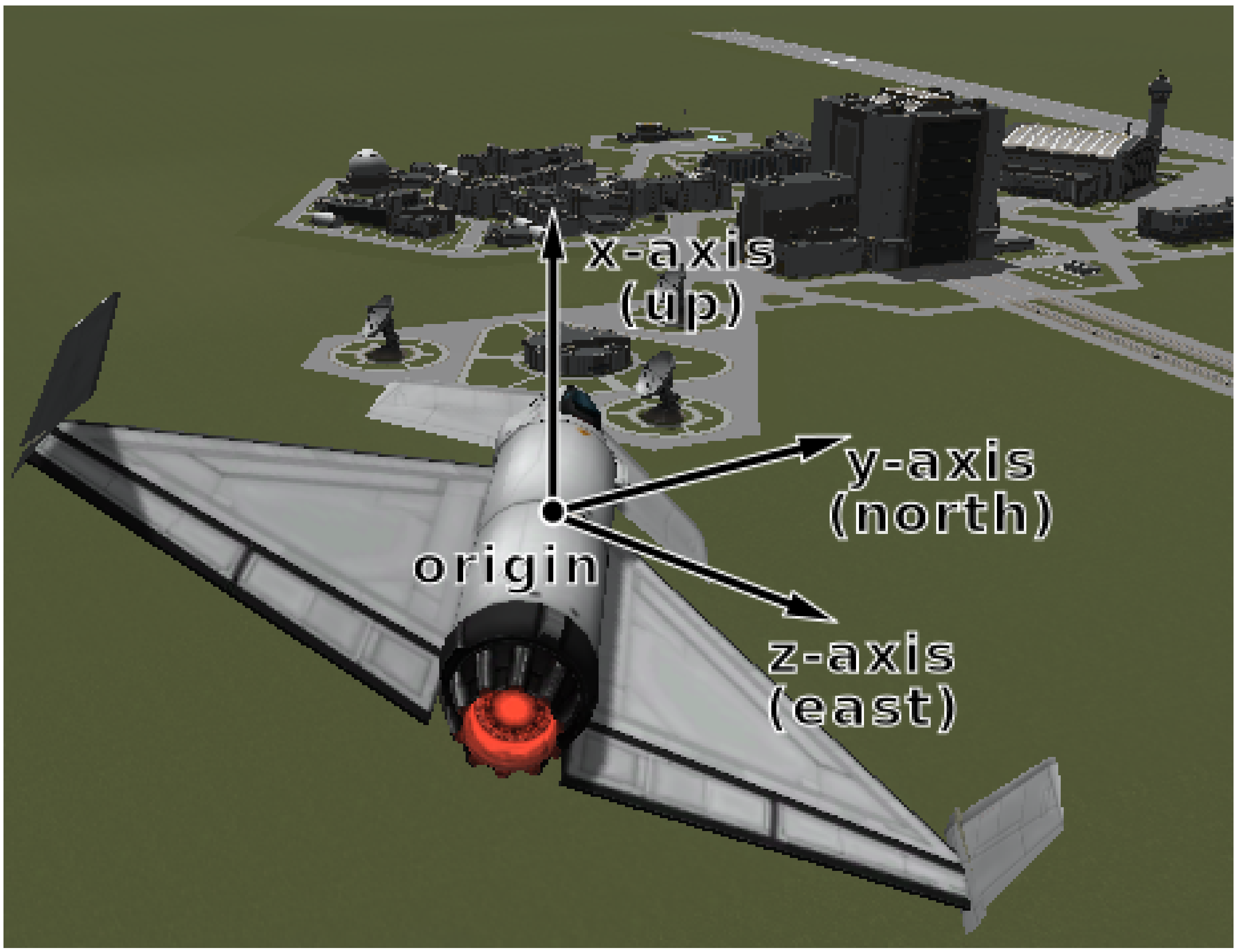


Fig. 4: Vessel surface reference frame origin and axes

- 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*

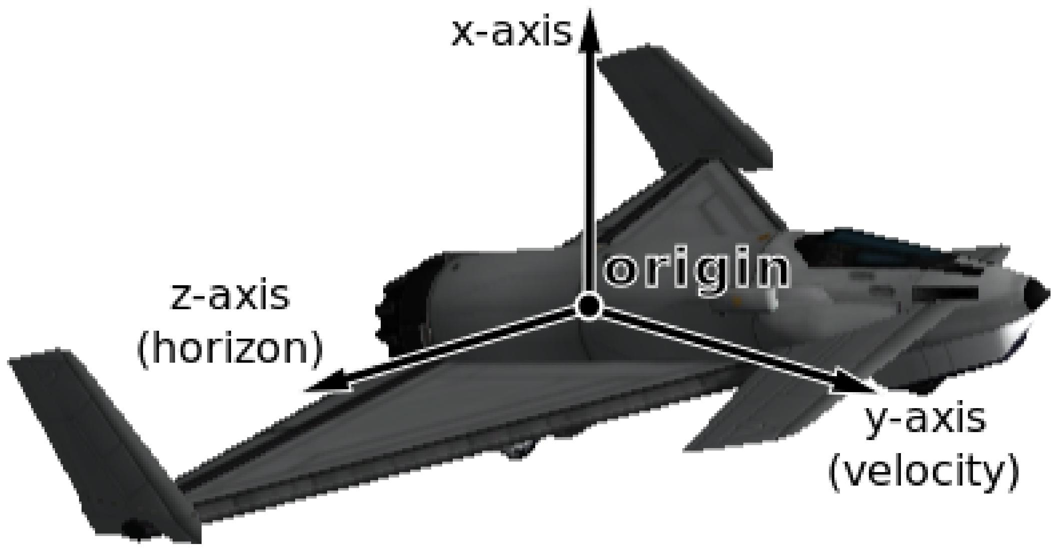


Fig. 5: Vessel surface velocity reference frame origin and axes

position(*reference_frame*)
The position of the center of mass of the vessel, in the given reference frame.

Parameters **reference_frame** (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.

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.

class CrewMember
Represents crew in a vessel. Can be obtained using *SpaceCenter.Vessel.crew*.

name
The crew members name.

Attribute Can be read or written

Return type string

type
The type of crew member.

Attribute Read-only, cannot be set

Return type *SpaceCenter.CrewMemberType*

on_mission
Whether the crew member is on a mission.

Attribute Read-only, cannot be set

Return type boolean

courage
The crew members courage.

Attribute Can be read or written

Return type number

stupidity
The crew members stupidity.

Attribute Can be read or written

Return type number

experience
The crew members experience.

Attribute Can be read or written

Return type number

badass
Whether the crew member is a badass.

Attribute Can be read or written

Return type boolean

veteran
Whether the crew member is a veteran.

Attribute Can be read or written

Return type boolean

class CrewMemberType
The type of a crew member. See *SpaceCenter.CrewMember.type*.

applicant
An applicant for crew.

crew
Rocket crew.

tourist
A tourist.

unowned
An unowned crew member.

7.3.3 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 2π

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 2π

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*

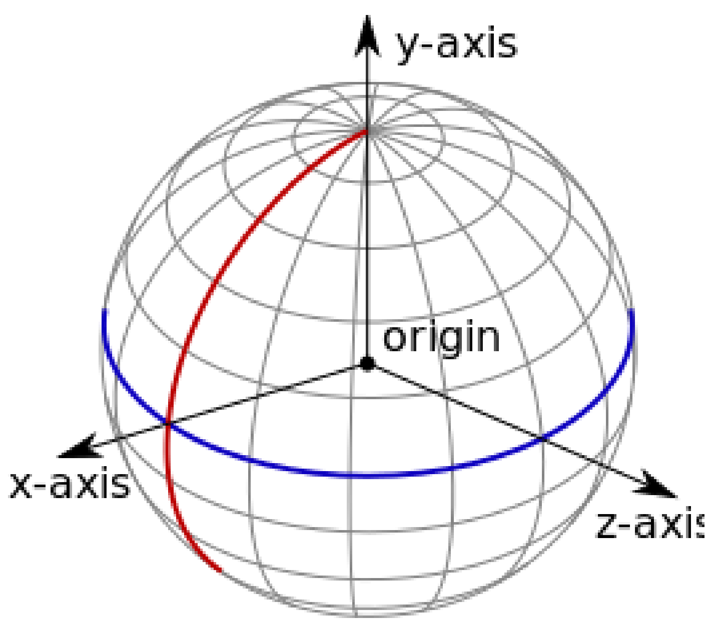


Fig. 6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

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.

- 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)

7.3.4 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 g.

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 $\frac{1}{2}$.air density.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 where 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.

7.3.5 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 orbit.

Parameters **target** (`SpaceCenter.Orbit`) – Target orbit.

Return type number

true_anomaly_at_dn (*target*)
The true anomaly of the descending node with the given target orbit.

Parameters **target** (`SpaceCenter.Orbit`) – Target orbit.

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 target orbit, in radians.

Parameters **target** (`SpaceCenter.Orbit`) – Target orbit.

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 orbit.

Parameters **target** (SpaceCenter.Orbit) – Target orbit.

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 orbit, in meters.

Parameters **target** (SpaceCenter.Orbit) – Target orbit.

Return type number

list_closest_approaches (*target*, *orbits*)

Returns the times at closest approach and corresponding distances, to a target orbit.

Parameters

- **target** (SpaceCenter.Orbit) – Target orbit.
- **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

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 *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.

7.3.7 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*

7.3.8 Parts

The following classes allow interaction with a vessels individual parts.

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 string is shared with kOS if it is 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 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 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 s 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*.

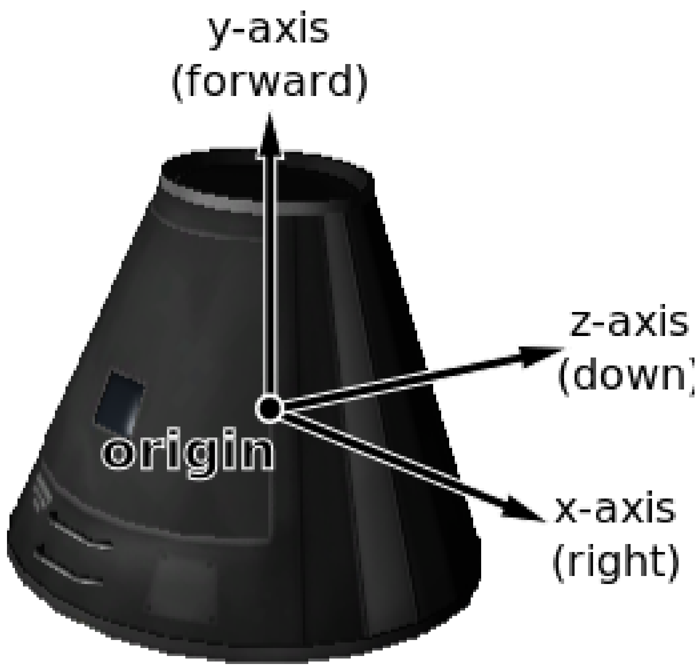


Fig. 7: Mk1 Command Pod reference frame origin and axes

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.

- *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.

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.

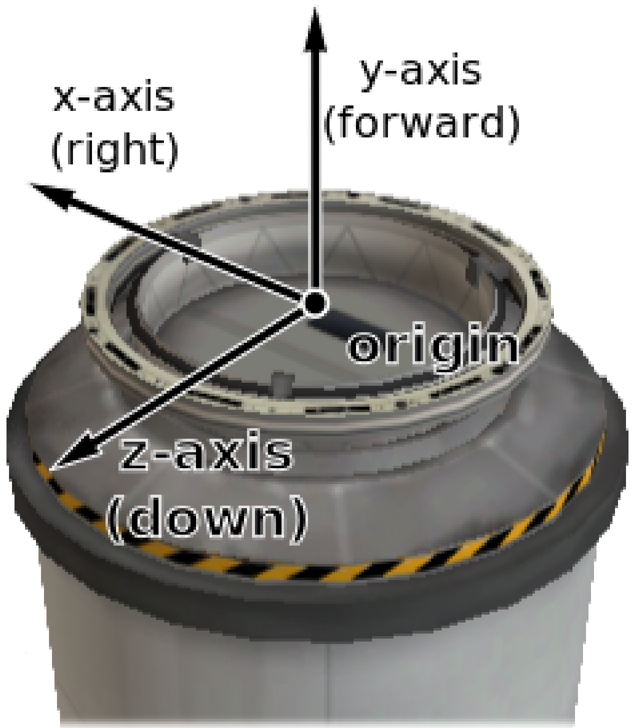


Fig. 8: Docking port reference frame origin and axes

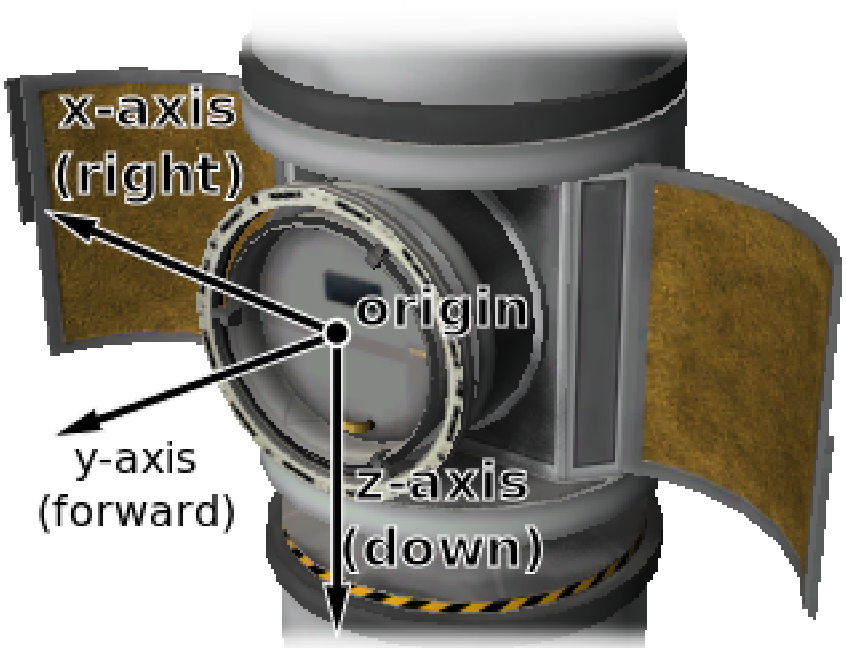


Fig. 9: Inline docking port reference frame origin and axes

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 nozzel on a solid rocket booster, or the individual nozzels 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

optimum_core_temperature
The core temperature at which the converter will operate with peak efficiency, in Kelvin.

Attribute Read-only, cannot be set

Return type number

core_temperature
The core temperature of the converter, in Kelvin.

Attribute Read-only, cannot be set

Return type number

thermal_efficiency
The thermal efficiency of the converter, as a percentage of its maximum.

Attribute Read-only, cannot be set

Return type number

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 disable.

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 nozzel 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 gimballing.
- 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 gimballing), 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 gimballing), 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`.

Traversing the Tree

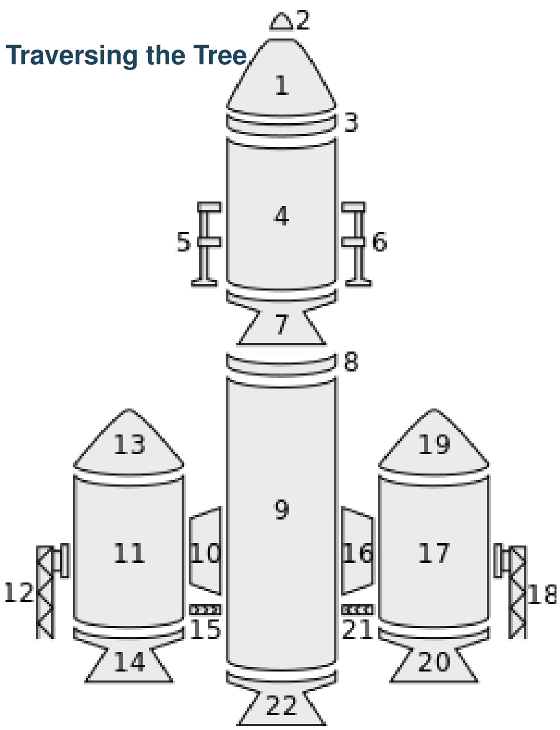


Fig. 10: **Figure 1** – Example parts making up a vessel.

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 vessels *root part* (part number 1 in the example above) and can be obtained by calling *`SpaceCenter.Parts.root`*. A parts 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 parts 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
local conn = krpc.co
local vessel = conn.

local root = vessel.
local stack = {{root
while #stack > 0 do
  local
↪part, depth = unpac
  print(string.rep('
  for _,child in ipa
    table.insert(sta
  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
 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).

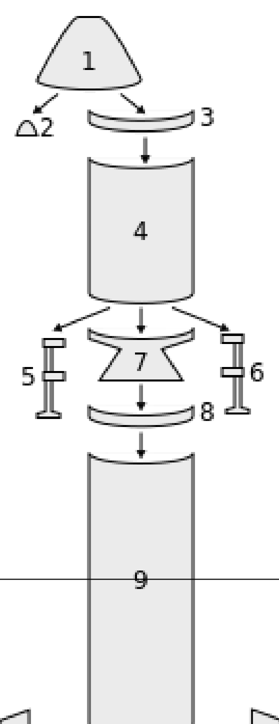
For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

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 Lua example does a depth-first traversal as before, but also prints out the attachment mode used by the part:

```lua
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
```

(continues on next page)



(continued from previous page)

```
  print(string.rep(' ', depth) .. part.title .. ' - ' .. attach_mode)
  for _,child in ipairs(part.children) do
    table.insert(stack, {child, depth+1})
  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
       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

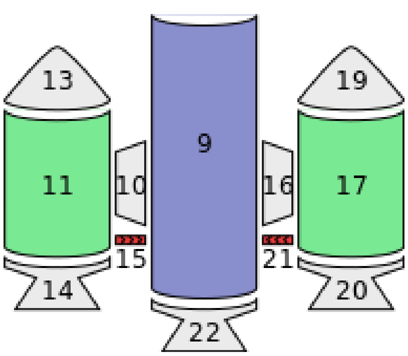


Fig. 12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

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).

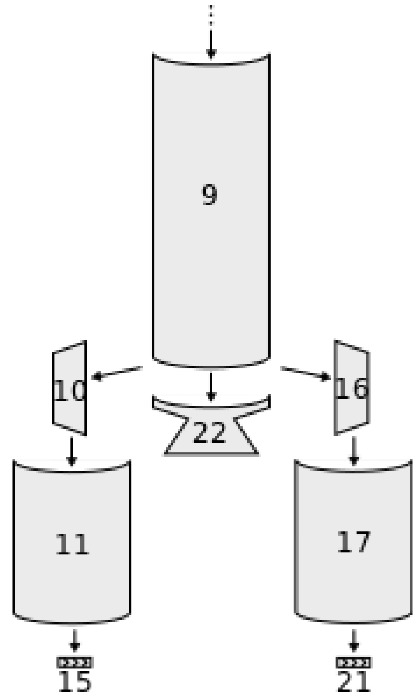


Fig. 13: **Figure 4** – A subset of the parts tree from Figure 2 above.

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*.

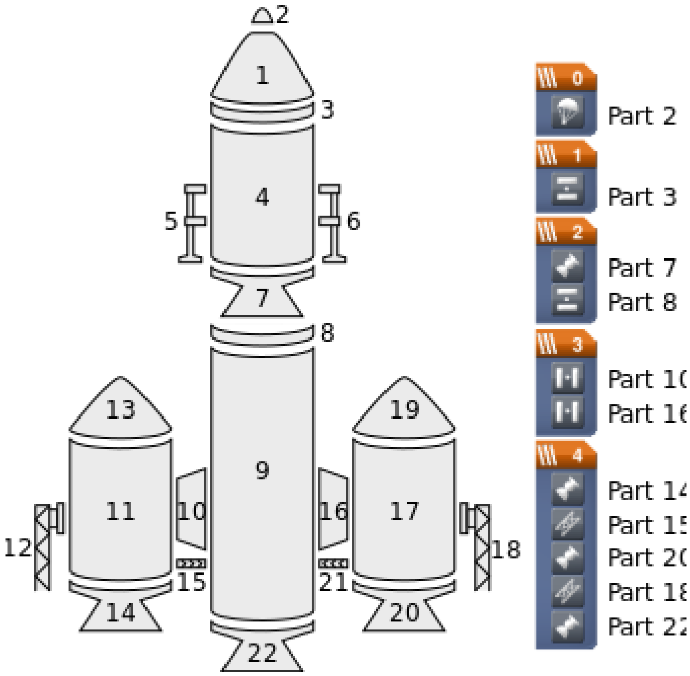


Fig. 14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

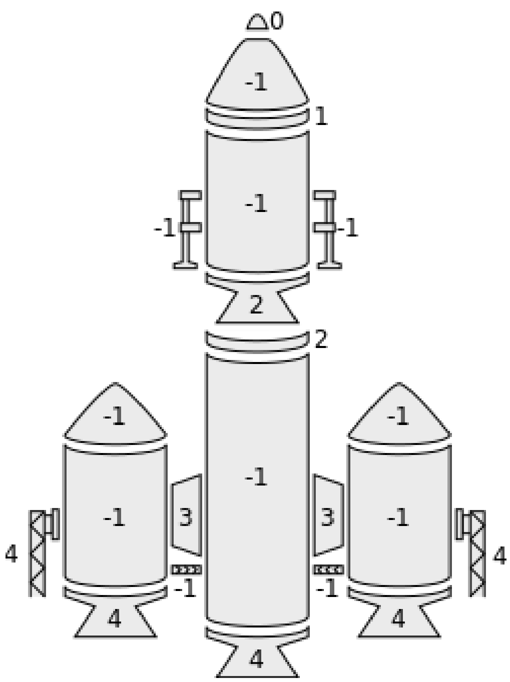


Fig. 15: **Figure 7** – The stage in which each part is *activated*.

7.3.9 Resources

class Resources

Represents the collection of resources stored in a vessel, stage or part. Created by calling *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.

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.

7.3.10 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)

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** (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.

7.3.12 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.

7.3.13 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 is 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 is 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 is 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.

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 *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 "GameData/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.

7.3.15 Contracts

class ContractManager
Contracts manager. Obtained by calling *SpaceCenter.contract_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

7.3.16 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])
```

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** (*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.

7.4.2 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.

7.4.3 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

7.4.4 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*

7.5 InfernalRobotics API

Provides RPCs to interact with the InfernalRobotics mod. Both the original mod and Infernal Robotics Next are supported. Provides the following classes:

7.5.1 InfernalRobotics

This service provides functionality to interact with Infernal Robotics.

available
Whether Infernal Robotics is installed.

Attribute Read-only, cannot be set

Return type boolean

ready
Whether Infernal Robotics API is ready.

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*

7.5.2 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.

7.5.3 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.

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.

```
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)
```

(continues on next page)

(continued from previous page)

```
end

for _,servo in ipairs(group.servos) do
  print(servo.name, servo.position)
end

group:move_right()
krpc.platform.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 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*

7.6.2 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.

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 *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 *KerbalAlarmClock.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 *KerbalAlarmClock.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.

```
local krpc = require 'krpc'

local conn␣
↪= krpc.connect('Kerbal Alarm Clock Example')

local␣
↪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 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*

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 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*

7.7.3 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.

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.

```
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)
```

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 *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*][, *color = (1.0, 0.92, 0.016)*][, *size = 20.0*])
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.
- **color** (*Tuple*) – The color of the message.
- **size** (*number*) – Size of the message, differs per position.

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.

7.8.2 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.

7.8.3 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.

7.8.4 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.

7.8.5 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.

7.8.6 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.

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 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)

CHAPTER

EIGHT

PYTHON

8.1 Python Client

This client provides a Python API for interacting with a kRPC server. It supports Python 2.7+ and 3.x

8.1.1 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
```

8.1.2 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)
```

8.1.3 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 provides by the `SpaceCenter` service. This works similarly for class types, for example: `help(conn.space_center.Vessel)`.

8.1.4 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.

8.1.5 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.conrol.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.

8.1.6 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'
```

8.1.7 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*.

stream_update_condition
A condition variable (of type `threading.Condition`) that is notified whenever a stream update finishes processing.

wait_for_stream_update(*timeout=None*)
This method blocks until a stream update finishes processing or the operation times out.

The stream update 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.

add_stream_update_callback(*callback*)
Adds a callback function that is invoked whenever a stream update finishes processing.

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_callback(*callback*)
Removes a stream update callback function.

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.

rate
The update rate of the stream in Hertz. When set to zero, the rate is unlimited.

__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_callback(*callback*)
Removes a callback function from the stream.

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_callback(*callback*)
Removes a callback function from the event.

remove()
Removes the event from the server.

stream
Returns the underlying stream for the event.

8.2 KRPC API

8.2.1 KRPC

None 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 bytes

Game Scenes All

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

Game Scenes All

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(bytes, str, str))

Game Scenes All

static get_status()
Returns some information about the server, such as the version.

Return type `krpc.schema.KRPC.Status`

Game Scenes All

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`

Game Scenes All

current_game_scene
Get the current game scene.

Attribute Read-only, cannot be set

Return type *GameScene*

Game Scenes All

paused
Whether the game is paused.

Attribute Can be read or written

Return type bool

Game Scenes All

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.

8.2.2 Expressions

class Expression
A server side expression.

static constant_double(*value*)
A constant value of double precision floating point type.

Parameters **value** (*double*) –

Return type *Expression*

Game Scenes All

static constant_float (*value*)
A constant value of single precision floating point type.

Parameters **value** (*float*) –

Return type *Expression*

Game Scenes All

static constant_int (*value*)
A constant value of integer type.

Parameters **value** (*int*) –

Return type *Expression*

Game Scenes All

static constant_bool (*value*)
A constant value of boolean type.

Parameters **value** (*bool*) –

Return type *Expression*

Game Scenes All

static constant_string (*value*)
A constant value of string type.

Parameters **value** (*str*) –

Return type *Expression*

Game Scenes All

static call (*call*)
An RPC call.

Parameters **call** (*krpc.schema.KRPC.ProcedureCall*) –

Return type *Expression*

Game Scenes All

static equal (*arg0*, *arg1*)
Equality comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static not_equal (*arg0*, *arg1*)
Inequality comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static greater_than (*arg0*, *arg1*)
Greater than numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static greater_than_or_equal (*arg0*, *arg1*)
Greater than or equal numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static less_than (*arg0*, *arg1*)
Less than numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static less_than_or_equal (*arg0*, *arg1*)
Less than or equal numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static and_ (*arg0*, *arg1*)
Boolean and operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static or_ (*arg0*, *arg1*)
Boolean or operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static exclusive_or(*arg0*, *arg1*)
Boolean exclusive-or operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static not_(*arg*)
Boolean negation operator.

Parameters **arg** (Expression) –

Return type *Expression*

Game Scenes All

static add(*arg0*, *arg1*)
Numerical addition.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static subtract(*arg0*, *arg1*)
Numerical subtraction.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static multiply(*arg0*, *arg1*)
Numerical multiplication.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static divide(*arg0*, *arg1*)
Numerical division.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static modulo(*arg0*, *arg1*)
Numerical modulo operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Returns The remainder of arg0 divided by arg1

Return type *Expression*

Game Scenes All

static power(*arg0*, *arg1*)
Numerical power operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Returns arg0 raised to the power of arg1, with type of arg0

Return type *Expression*

Game Scenes All

static left_shift(*arg0*, *arg1*)
Bitwise left shift.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static right_shift(*arg0*, *arg1*)
Bitwise right shift.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Return type *Expression*

Game Scenes All

static cast (*arg*, *type*)
Perform a cast to the given type.

Parameters

- **arg** (`Expression`) –
- **type** (`Type`) – Type to cast the argument to.

Return type *Expression*

Game Scenes All

static parameter (*name*, *type*)
A named parameter of type double.

Parameters

- **name** (*str*) – The name of the parameter.
- **type** (`Type`) – The type of the parameter.

Returns A named parameter.

Return type *Expression*

Game Scenes All

static function (*parameters*, *body*)
A function.

Parameters

- **parameters** (*list*) – The parameters of the function.
- **body** (`Expression`) – The body of the function.

Returns A function.

Return type *Expression*

Game Scenes All

static invoke (*function*, *args*)
A function call.

Parameters

- **function** (`Expression`) – The function to call.
- **args** (*dict*) – The arguments to call the function with.

Returns A function call.

Return type *Expression*

Game Scenes All

static create_tuple (*elements*)
Construct a tuple.

Parameters **elements** (*list*) – The elements.

Returns The tuple.

Return type *Expression*

Game Scenes All

static create_list (*values*)
Construct a list.

Parameters **values** (*list*) – The value. Should all be of the same type.

Returns The list.

Return type *Expression*

Game Scenes All

static create_set (*values*)
Construct a set.

Parameters **values** (*set*) – The values. Should all be of the same type.

Returns The set.

Return type *Expression*

Game Scenes All

static create_dictionary (*keys*, *values*)
Construct a dictionary, from a list of corresponding keys and values.

Parameters

- **keys** (*list*) – The keys. Should all be of the same type.
- **values** (*list*) – The values. Should all be of the same type.

Returns The dictionary.

Return type *Expression*

Game Scenes All

static to_list (*arg*)
Convert a collection to a list.

Parameters **arg** (Expression) – The collection.

Returns The collection as a list.

Return type *Expression*

Game Scenes All

static to_set (*arg*)
Convert a collection to a set.

Parameters **arg** (Expression) – The collection.

Returns The collection as a set.

Return type *Expression*

Game Scenes All

static get (*arg*, *index*)
Access an element in a tuple, list or dictionary.

Parameters

- **arg** (Expression) – The tuple, list or dictionary.
- **index** (Expression) – The index of the element to access. A zero indexed integer for a tuple or list, or a key for a dictionary.

Returns The element.

Return type *Expression*

Game Scenes All

static count(*arg*)
Number of elements in a collection.

Parameters **arg** (Expression) – The list, set or dictionary.

Returns The number of elements in the collection.

Return type *Expression*

Game Scenes All

static sum(*arg*)
Sum all elements of a collection.

Parameters **arg** (Expression) – The list or set.

Returns The sum of the elements in the collection.

Return type *Expression*

Game Scenes All

static max(*arg*)
Maximum of all elements in a collection.

Parameters **arg** (Expression) – The list or set.

Returns The maximum elements in the collection.

Return type *Expression*

Game Scenes All

static min(*arg*)
Minimum of all elements in a collection.

Parameters **arg** (Expression) – The list or set.

Returns The minimum elements in the collection.

Return type *Expression*

Game Scenes All

static average(*arg*)
Minimum of all elements in a collection.

Parameters **arg** (Expression) – The list or set.

Returns The minimum elements in the collection.

Return type *Expression*

Game Scenes All

static select(*arg*, *func*)
Run a function on every element in the collection.

Parameters

- **arg** (Expression) – The list or set.
- **func** (Expression) – The function.

Returns The modified collection.

Return type *Expression*

Game Scenes All

static where (*arg*, *func*)

Run a function on every element in the collection.

Parameters

- **arg** (Expression) – The list or set.
- **func** (Expression) – The function.

Returns The modified collection.

Return type *Expression*

Game Scenes All

static contains (*arg*, *value*)

Determine if a collection contains a value.

Parameters

- **arg** (Expression) – The collection.
- **value** (Expression) – The value to look for.

Returns Whether the collection contains a value.

Return type *Expression*

Game Scenes All

static aggregate (*arg*, *func*)

Applies an accumulator function over a sequence.

Parameters

- **arg** (Expression) – The collection.
- **func** (Expression) – The accumulator function.

Returns The accumulated value.

Return type *Expression*

Game Scenes All

static aggregate_with_seed (*arg*, *seed*, *func*)

Applies an accumulator function over a sequence, with a given seed.

Parameters

- **arg** (Expression) – The collection.
- **seed** (Expression) – The seed value.
- **func** (Expression) – The accumulator function.

Returns The accumulated value.

Return type *Expression*

Game Scenes All

static concat (*arg1*, *arg2*)

Concatenate two sequences.

Parameters

- **arg1** (Expression) – The first sequence.
- **arg2** (Expression) – The second sequence.

Returns The first sequence followed by the second sequence.

Return type *Expression*

Game Scenes All

static order_by(*arg*, *key*)

Order a collection using a key function.

Parameters

- **arg** (Expression) – The collection to order.
- **key** (Expression) – A function that takes a value from the collection and generates a key to sort on.

Returns The ordered collection.

Return type *Expression*

Game Scenes All

static all(*arg*, *predicate*)

Determine whether all items in a collection satisfy a boolean predicate.

Parameters

- **arg** (Expression) – The collection.
- **predicate** (Expression) – The predicate function.

Returns Whether all items satisfy the predicate.

Return type *Expression*

Game Scenes All

static any(*arg*, *predicate*)

Determine whether any item in a collection satisfies a boolean predicate.

Parameters

- **arg** (Expression) – The collection.
- **predicate** (Expression) – The predicate function.

Returns Whether any item satisfies the predicate.

Return type *Expression*

Game Scenes All

class Type

A server side expression.

static double()

Double type.

Return type *Type*

Game Scenes All

static float()
Float type.

Return type *Type*

Game Scenes All

static int()
Int type.

Return type *Type*

Game Scenes All

static bool()
Bool type.

Return type *Type*

Game Scenes All

static string()
String type.

Return type *Type*

Game Scenes All

8.3 SpaceCenter API

8.3.1 SpaceCenter

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

science
The current amount of science.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

funds
The current amount of funds.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

reputation
The current amount of reputation.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

active_vessel
The currently active vessel.

Attribute Can be read or written

Return type *Vessel*

Game Scenes Flight

vessels

A list of all the vessels in the game.

Attribute Read-only, cannot be set

Return type list(*Vessel*)

Game Scenes All

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*)

Game Scenes All

target_body

The currently targeted celestial body.

Attribute Can be read or written

Return type *CelestialBody*

Game Scenes Flight

target_vessel

The currently targeted vessel.

Attribute Can be read or written

Return type *Vessel*

Game Scenes Flight

target_docking_port

The currently targeted docking port.

Attribute Can be read or written

Return type *DockingPort*

Game Scenes Flight

static clear_target()

Clears the current target.

Game Scenes Flight

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)

Game Scenes All

static launch_vessel(*craft_directory*, *name*, *launch_site*[, *recover = True*])

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".
- **recover** (*bool*) – If true and there is a vessel on the launch site, recover it before launching.

Game Scenes All

Note: Throws an exception if any of the games pre-flight checks fail.

static launch_vessel_from_vab(*name*[, *recover = True*])

Launch a new vessel from the VAB onto the launchpad.

Parameters

- **name** (*str*) – Name of the vessel to launch.
- **recover** (*bool*) – If true and there is a vessel on the launch pad, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *launch_vessel()* with the craft directory set to "VAB" and the launch site set to "LaunchPad". Throws an exception if any of the games pre-flight checks fail.

static launch_vessel_from_sph(*name*[, *recover = True*])

Launch a new vessel from the SPH onto the runway.

Parameters

- **name** (*str*) – Name of the vessel to launch.
- **recover** (*bool*) – If true and there is a vessel on the runway, recover it before launching.

Game Scenes All

Note: This is equivalent to calling *launch_vessel()* with the craft directory set to "SPH" and the launch site set to "Runway". Throws an exception if any of the games pre-flight checks fail.

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*) –

Game Scenes All

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*) –

Game Scenes All

static quicksave()

Save a quicksave.

Game Scenes All

Note: This is the same as calling *save()* with the name "quicksave".

static quickload()

Load a quicksave.

Game Scenes All

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

Game Scenes Flight

navball

Whether the navball is visible.

Attribute Can be read or written

Return type bool

Game Scenes Flight

ut

The current universal time in seconds.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

g

The value of the gravitational constant G in $N(m/kg)^2$.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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** (*double*) – 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.

Game Scenes Flight

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(double, double, double)

Game Scenes All

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(double, double, double)

Game Scenes All

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(double, double, double, double)

Game Scenes All

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(double, double, double)

Game Scenes All

static raycast_distance (*position*, *direction*, *reference_frame*)

Cast a ray from a given position in a given direction, and return the distance to the hit point. If no hit occurs, returns infinity.

Parameters

- **position** (*tuple*) – Position, as a vector, of the origin of the ray.
- **direction** (*tuple*) – Direction of the ray, as a unit vector.
- **reference_frame** (ReferenceFrame) – The reference frame that the position and direction are in.

Returns The distance to the hit, in meters, or infinity if there was no hit.

Return type double

Game Scenes All

static raycast_part (*position*, *direction*, *reference_frame*)

Cast a ray from a given position in a given direction, and return the part that it hits. If no hit occurs, returns `None`.

Parameters

- **position** (*tuple*) – Position, as a vector, of the origin of the ray.
- **direction** (*tuple*) – Direction of the ray, as a unit vector.
- **reference_frame** (ReferenceFrame) – The reference frame that the position and direction are in.

Returns The part that was hit or `None` if there was no hit.

Return type *Part*

Game Scenes Flight

far_available

Whether Ferram Aerospace Research is installed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

game_mode

The current mode the game is in.

Attribute Read-only, cannot be set

Return type *GameMode*

Game Scenes All

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*

Game Scenes Flight

camera

An object that can be used to control the camera.

Attribute Read-only, cannot be set

Return type *Camera*

Game Scenes Flight

waypoint_manager

The waypoint manager.

Attribute Read-only, cannot be set

Return type *WaypointManager*

Game Scenes Flight

contract_manager

The contract manager.

Attribute Read-only, cannot be set

Return type *ContractManager*

Game Scenes All

class GameMode

The game mode. Returned by *GameMode*

sandbox

Sandbox mode.

career

Career mode.

science

Science career mode.

science_sandbox

Science sandbox mode.

mission

Mission mode.

mission_builder

Mission builder mode.

scenario

Scenario mode.

scenario_non_resumable

Scenario mode that cannot be resumed.

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.

8.3.2 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

Game Scenes All

type
The type of the vessel.

Attribute Can be read or written

Return type *VesselType*

Game Scenes All

situation
The situation the vessel is in.

Attribute Read-only, cannot be set

Return type *VesselSituation*

Game Scenes All

recoverable
Whether the vessel is recoverable.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

recover()
Recover the vessel.

Game Scenes All

met
The mission elapsed time in seconds.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

biome

The name of the biome the vessel is currently in.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

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*

Game Scenes 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*

Game Scenes All

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*

Game Scenes Flight

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*

Game Scenes Flight

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*

Game Scenes Flight

crew_capacity

The number of crew that can occupy the vessel.

Attribute Read-only, cannot be set

Return type int

Game Scenes All

crew_count

The number of crew that are occupying the vessel.

Attribute Read-only, cannot be set

Return type int

Game Scenes All

crew

The crew in the vessel.

Attribute Read-only, cannot be set

Return type list(*CrewMember*)

Game Scenes All

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*

Game Scenes Flight

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*

Game Scenes Flight

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*

Game Scenes Flight

mass

The total mass of the vessel, including resources, in kg.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

dry_mass
The total mass of the vessel, excluding resources, in kg.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double)

Game Scenes All

available_torque

The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gimballed 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(double, double, double), tuple(double, double, double))

Game Scenes Flight

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(double, double, double), tuple(double, double, double))

Game Scenes Flight

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(double, double, double), tuple(double, double, double))

Game Scenes Flight

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 (*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(double, double, double), tuple(double, double, double))

Game Scenes Flight

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(double, double, double), tuple(double, double, double))

Game Scenes Flight

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 (*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(double, double, double), tuple(double, double, double))

Game Scenes Flight

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 *ReferenceFrame*

Game Scenes Flight

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.

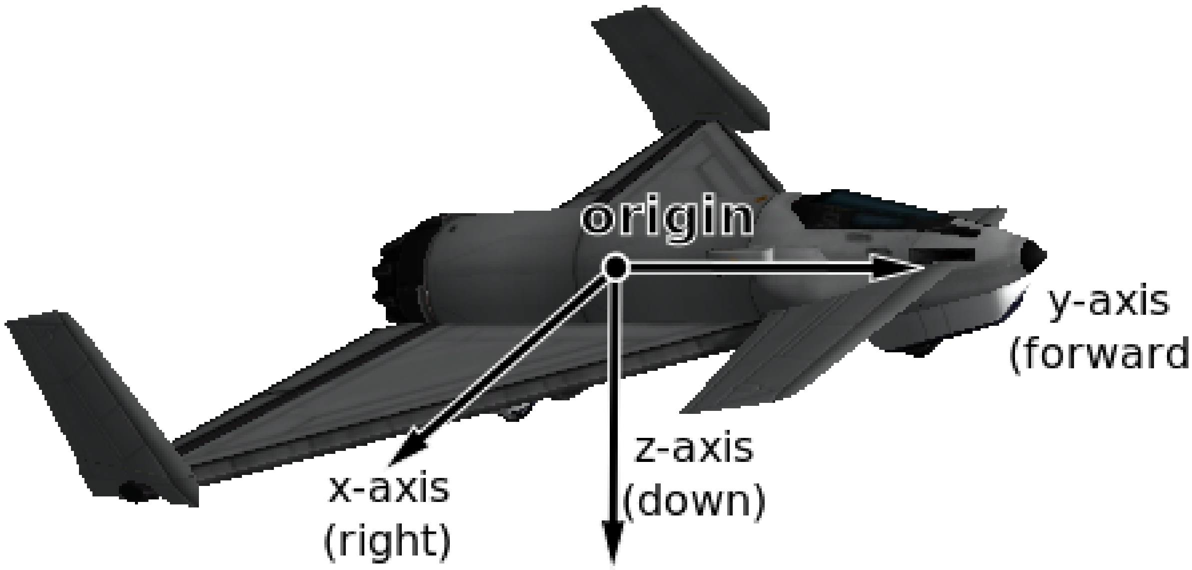


Fig. 1: Vessel reference frame origin and axes for the Aeris 3A aircraft

Attribute Read-only, cannot be set

Return type *ReferenceFrame*

Game Scenes Flight

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 *ReferenceFrame*

Game Scenes Flight

Note: Be careful not to confuse this with 'surface' mode on the navball.

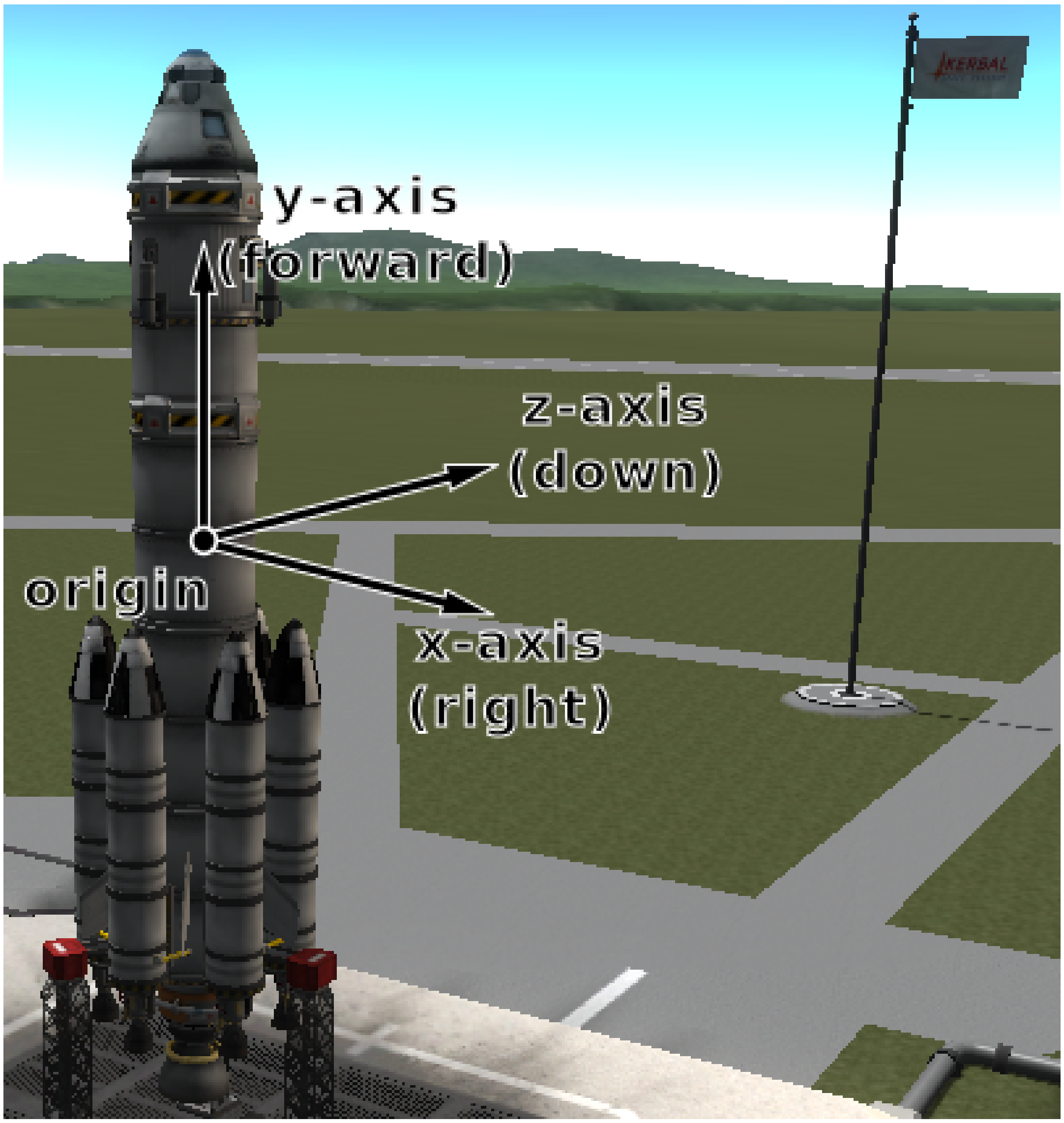


Fig. 2: Vessel reference frame origin and axes for the Kerbal-X rocket

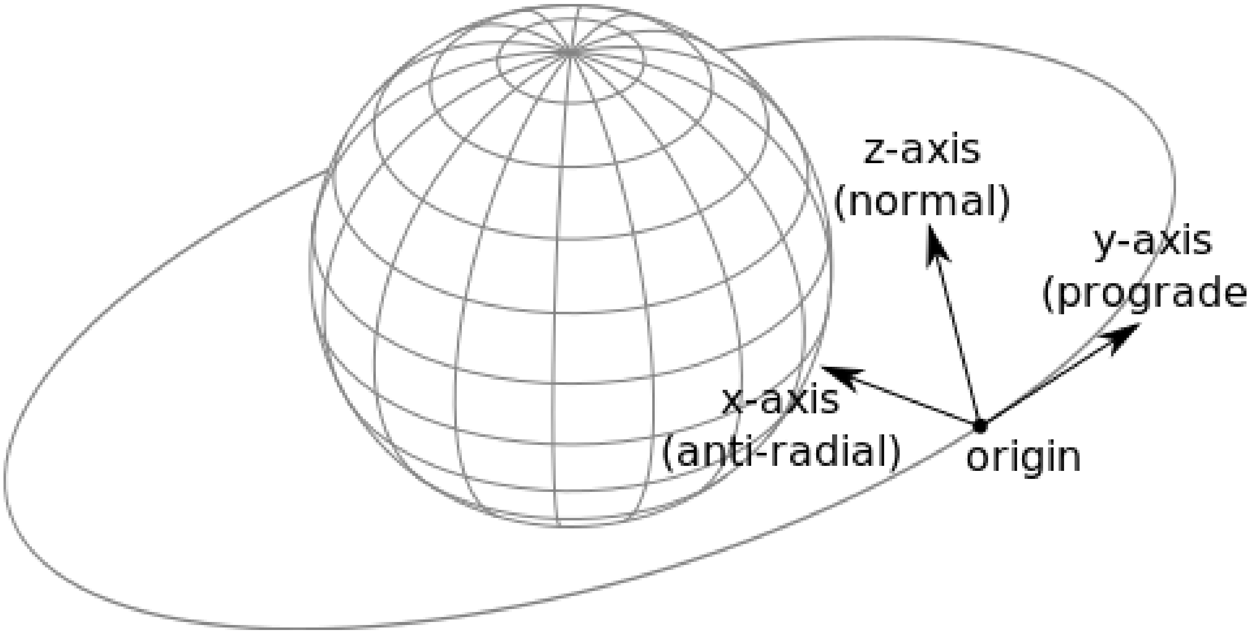


Fig. 3: Vessel orbital reference frame origin and axes

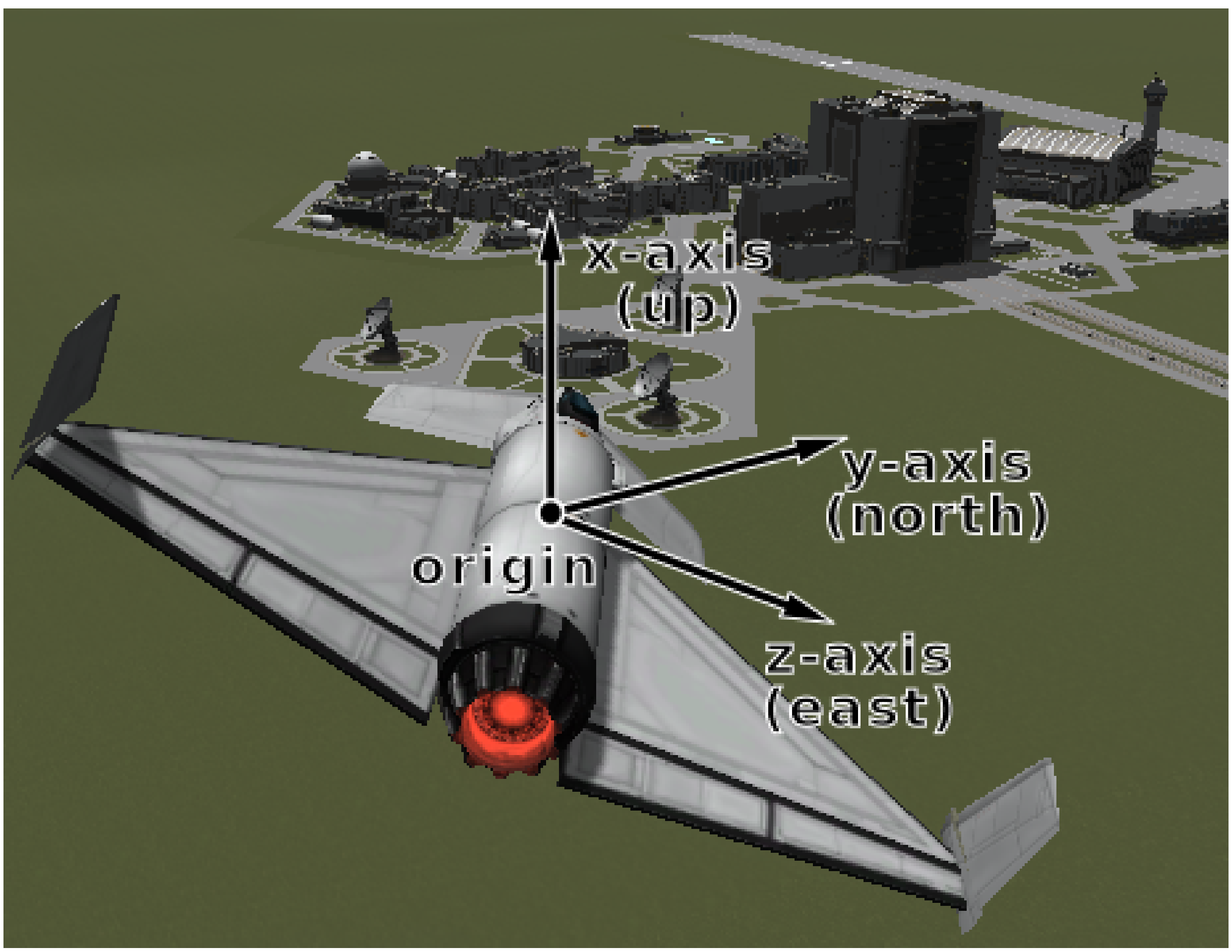


Fig. 4: Vessel surface reference frame origin and axes

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*

Game Scenes Flight

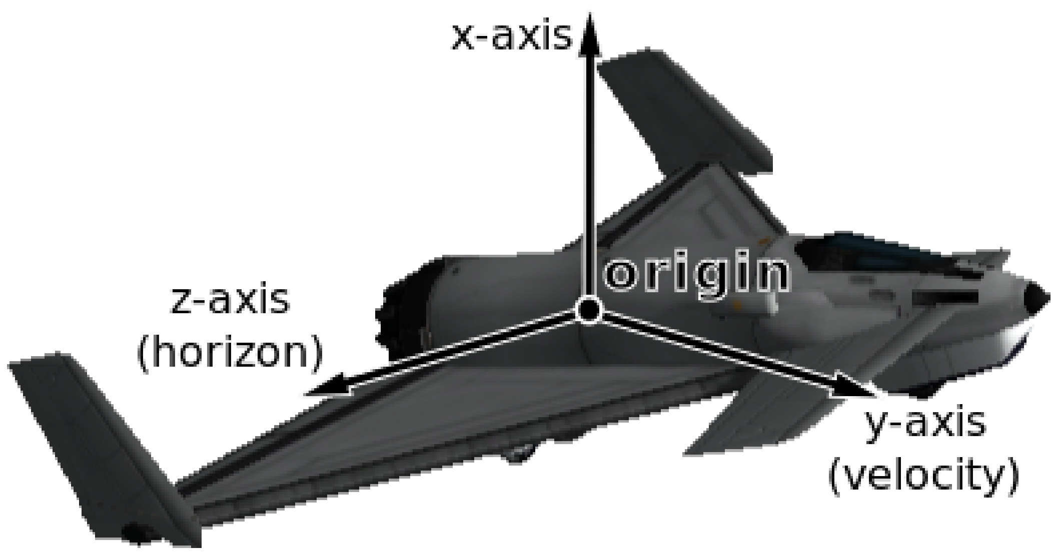


Fig. 5: Vessel surface velocity reference frame origin and axes

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(double, double, double)

Game Scenes Flight

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(double, double, double), tuple(double, double, double))

Game Scenes Flight

`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(double, double, double)

Game Scenes Flight

`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(double, double, double, double)

Game Scenes Flight

`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(double, double, double)

Game Scenes Flight

`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(double, double, double)

Game Scenes Flight

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.

class CrewMember
Represents crew in a vessel. Can be obtained using *Vessel.crew*.

name
The crew members name.

Attribute Can be read or written

Return type str

Game Scenes All

type
The type of crew member.

Attribute Read-only, cannot be set

Return type *CrewMemberType*

Game Scenes All

on_mission
Whether the crew member is on a mission.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

courage
The crew members courage.

Attribute Can be read or written

Return type float

Game Scenes All

stupidity
The crew members stupidity.

Attribute Can be read or written

Return type float

Game Scenes All

experience
The crew members experience.

Attribute Can be read or written

Return type float

Game Scenes All

badass
Whether the crew member is a badass.

Attribute Can be read or written

Return type bool

Game Scenes All

veteran
Whether the crew member is a veteran.

Attribute Can be read or written

Return type bool

Game Scenes All

class CrewMemberType
The type of a crew member. See *CrewMember.type*.

applicant
An applicant for crew.

crew
Rocket crew.

tourist
A tourist.

unowned
An unowned crew member.

8.3.3 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

Game Scenes All

satellites

A list of celestial bodies that are in orbit around this celestial body.

Attribute Read-only, cannot be set

Return type list(*CelestialBody*)

Game Scenes All

orbit

The orbit of the body.

Attribute Read-only, cannot be set

Return type *Orbit*

Game Scenes All

mass

The mass of the body, in kilograms.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

gravitational_parameter

The standard gravitational parameter of the body in $m^3 s^{-2}$.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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

Game Scenes All

rotational_period

The sidereal rotational period of the body, in seconds.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

rotational_speed
The rotational speed of the body, in radians per second.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

rotation_angle
The current rotation angle of the body, in radians. A value between 0 and 2π

Attribute Read-only, cannot be set

Return type double

Game Scenes All

initial_rotation
The initial rotation angle of the body (at UT 0), in radians. A value between 0 and 2π

Attribute Read-only, cannot be set

Return type double

Game Scenes All

equatorial_radius
The equatorial radius of the body, in meters.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.

Return type double

Game Scenes All

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** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.

Return type double

Game Scenes All

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** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **reference_frame** (ReferenceFrame) – Reference frame for the returned position vector.

Returns Position as a vector.

Return type tuple(double, double, double)

Game Scenes All

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** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **reference_frame** (ReferenceFrame) – Reference frame for the returned position vector.

Returns Position as a vector.

Return type tuple(double, double, double)

Game Scenes All

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** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **reference_frame** (ReferenceFrame) – Reference frame for the returned position vector.

Returns Position as a vector.

Return type tuple(double, double, double)

Game Scenes All

position_at_altitude (*latitude*, *longitude*, *altitude*, *reference_frame*)

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.
- **reference_frame** (ReferenceFrame) – Reference frame for the returned position vector.

Returns Position as a vector.

Return type tuple(double, double, double)

Game Scenes All

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 double

Game Scenes All

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 double

Game Scenes All

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 double

Game Scenes All

sphere_of_influence
The radius of the sphere of influence of the body, in meters.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

has_atmosphere
`True` if the body has an atmosphere.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

atmosphere_depth
The depth of the atmosphere, in meters.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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 double

Game Scenes All

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

Game Scenes All

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 double

Game Scenes All

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** (*double*) –

Return type double

Game Scenes All

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** (*double*) –

Return type double

Game Scenes All

biomes

The biomes present on this body.

Attribute Read-only, cannot be set

Return type set(str)

Game Scenes All

biome_at (*latitude*, *longitude*)

The biome at the given latitude and longitude, in degrees.

Parameters

- **latitude** (*double*) –
- **longitude** (*double*) –

Return type str

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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*

Game Scenes All

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.

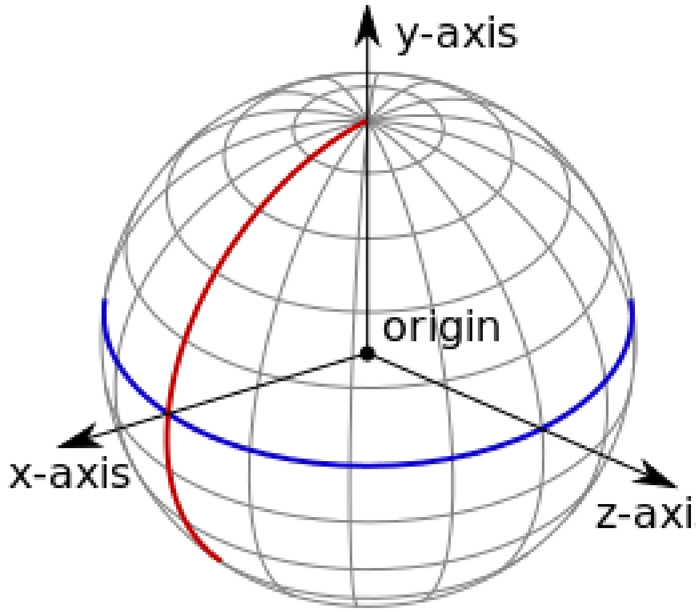


Fig. 6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

- 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 *ReferenceFrame*

Game Scenes All

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*

Game Scenes All

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(double, double, double)

Game Scenes All

`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(double, double, double)

Game Scenes All

`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(double, double, double, double)

Game Scenes All

`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(double, double, double)

Game Scenes All

`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(double, double, double)

Game Scenes All

8.3.4 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 g.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`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 double

Game Scenes Flight

`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 double

Game Scenes Flight

`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 double

Game Scenes Flight

`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 double

Game Scenes Flight

`latitude`
The latitude of the vessel for the body being orbited, in degrees.

Attribute Read-only, cannot be set

Return type double

Game Scenes Flight

`longitude`
The longitude of the vessel for the body being orbited, in degrees.

Attribute Read-only, cannot be set

Return type double

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

speed

The speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

Attribute Read-only, cannot be set

Return type double

Game Scenes Flight

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 double

Game Scenes Flight

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 double

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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 float

Game Scenes Flight

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 float

Game Scenes Flight

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 float

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

atmosphere_density

The current density of the atmosphere around the vessel, in kg/m^3.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

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 $\frac{1}{2}$.air density.velocity^2. It is commonly denoted Q.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

static_pressure

The static atmospheric pressure acting on the vessel, in Pascals.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

static_pressure_at_msl

The static atmospheric pressure at mean sea level, in Pascals.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

simulate_aerodynamic_force_at (*body*, *position*, *velocity*)

Simulate and return the total aerodynamic forces acting on the vessel, if it where 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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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

Game Scenes Flight

mach

The speed of the vessel, in multiples of the speed of sound.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

reynolds_number

The vessels Reynolds number.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

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

Game Scenes Flight

equivalent_air_speed

The equivalent air speed of the vessel, in meters per second.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

`static_air_temperature`

The static (ambient) temperature of the atmosphere around the vessel, in Kelvin.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

`ballistic_coefficient`

The ballistic coefficient.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

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

Game Scenes Flight

Note: Requires Ferram Aerospace Research.

8.3.5 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*

Game Scenes All

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 double

Game Scenes All

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 double

Game Scenes All

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 double

Game Scenes All

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 double

Game Scenes All

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 double

Game Scenes All

semi_minor_axis
The semi-minor axis of the orbit, in meters.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

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 double

Game Scenes All

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** (*double*) – The universal time to measure the radius at.

Return type double

Game Scenes All

position_at (*ut*, *reference_frame*)
The position at a given time, in the specified reference frame.

Parameters

- **ut** (*double*) – 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(double, double, double)

Game Scenes All

speed

The current orbital speed of the object in meters per second.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

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 double

Game Scenes All

time_to_apoapsis

The time until the object reaches apoapsis, in seconds.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

time_to_periapsis

The time until the object reaches periapsis, in seconds.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

eccentricity

The eccentricity of the orbit.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

inclination

The inclination of the orbit, in radians.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

longitude_of_ascending_node
The longitude of the ascending node, in radians.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

argument_of_periapsis
The argument of periapsis, in radians.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

mean_anomaly_at_epoch
The mean anomaly at epoch.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

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 double

Game Scenes All

mean_anomaly
The mean anomaly.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

mean_anomaly_at_ut (*ut*)
The mean anomaly at the given time.

Parameters **ut** (*double*) – The universal time in seconds.

Return type double

Game Scenes All

eccentric_anomaly
The eccentric anomaly.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

eccentric_anomaly_at_ut (*ut*)
The eccentric anomaly at the given universal time.

Parameters **ut** (*double*) – The universal time, in seconds.

Return type double

Game Scenes All

true_anomaly
The true anomaly.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

true_anomaly_at_ut (*ut*)
The true anomaly at the given time.

Parameters **ut** (*double*) – The universal time in seconds.

Return type double

Game Scenes All

true_anomaly_at_radius (*radius*)
The true anomaly at the given orbital radius.

Parameters **radius** (*double*) – The orbital radius in meters.

Return type double

Game Scenes All

ut_at_true_anomaly (*true_anomaly*)
The universal time, in seconds, corresponding to the given true anomaly.

Parameters **true_anomaly** (*double*) – True anomaly.

Return type double

Game Scenes All

radius_at_true_anomaly (*true_anomaly*)
The orbital radius at the point in the orbit given by the true anomaly.

Parameters **true_anomaly** (*double*) – The true anomaly.

Return type double

Game Scenes All

true_anomaly_at_an (*target*)
The true anomaly of the ascending node with the given target orbit.

Parameters **target** (Orbit) – Target orbit.

Return type double

Game Scenes All

true_anomaly_at_dn (*target*)
The true anomaly of the descending node with the given target orbit.

Parameters **target** (Orbit) – Target orbit.

Return type double

Game Scenes All

orbital_speed
The current orbital speed in meters per second.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

orbital_speed_at(*time*)

The orbital speed at the given time, in meters per second.

Parameters **time** (*double*) – Time from now, in seconds.

Return type double

Game Scenes All

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(double, double, double)

Game Scenes All

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(double, double, double)

Game Scenes All

relative_inclination(*target*)

Relative inclination of this orbit and the target orbit, in radians.

Parameters **target** (Orbit) – Target orbit.

Return type double

Game Scenes All

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 double

Game Scenes All

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*

Game Scenes All

time_of_closest_approach (*target*)
Estimates and returns the time at closest approach to a target orbit.

Parameters **target** (`Orbit`) – Target orbit.

Returns The universal time at closest approach, in seconds.

Return type double

Game Scenes All

distance_at_closest_approach (*target*)
Estimates and returns the distance at closest approach to a target orbit, in meters.

Parameters **target** (`Orbit`) – Target orbit.

Return type double

Game Scenes All

list_closest_approaches (*target*, *orbits*)
Returns the times at closest approach and corresponding distances, to a target orbit.

Parameters

- **target** (`Orbit`) – Target orbit.
- **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(double))

Game Scenes All

8.3.6 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`*

Game Scenes Flight

state
The control state of the vessel.

Attribute Read-only, cannot be set

Return type *`ControlState`*

Game Scenes Flight

sas

The state of SAS.

Attribute Can be read or written

Return type bool

Game Scenes Flight

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*

Game Scenes Flight

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*

Game Scenes Flight

rcs

The state of RCS.

Attribute Can be read or written

Return type bool

Game Scenes Flight

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

Game Scenes Flight

gear

The state of the landing gear/legs.

Attribute Can be read or written

Return type bool

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

lights

The state of the lights.

Attribute Can be read or written

Return type bool

Game Scenes Flight

brakes

The state of the wheel brakes.

Attribute Can be read or written

Return type bool

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

abort
The state of the abort action group.

Attribute Can be read or written

Return type bool

Game Scenes Flight

throttle
The state of the throttle. A value between 0 and 1.

Attribute Can be read or written

Return type float

Game Scenes Flight

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*

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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*)

Game Scenes Flight

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

Game Scenes Flight

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*) –

Game Scenes Flight

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.

Game Scenes Flight

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** (*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.

Return type *Node*

Game Scenes Flight

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*)

Game Scenes Flight

remove_nodes ()

Remove all maneuver nodes.

Game Scenes Flight

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.

8.3.7 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

Game Scenes Flight

can_transmit_science
Whether the vessel can transmit science data to KSC.

Attribute Read-only, cannot be set

Return type bool

Game Scenes Flight

signal_strength
Signal strength to KSC.

Attribute Read-only, cannot be set

Return type double

Game Scenes Flight

signal_delay
Signal delay to KSC in seconds.

Attribute Read-only, cannot be set

Return type double

Game Scenes Flight

power
The combined power of all active antennae on the vessel.

Attribute Read-only, cannot be set

Return type double

Game Scenes Flight

control_path
The communication path used to control the vessel.

Attribute Read-only, cannot be set

Return type list(*CommLink*)

Game Scenes Flight

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*

Game Scenes All

signal_strength
Signal strength of the link.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

start
Start point of the link.

Attribute Read-only, cannot be set

Return type *CommNode*

Game Scenes All

end
Start point of the link.

Attribute Read-only, cannot be set

Return type *CommNode*

Game Scenes All

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

Game Scenes All

is_home
Whether the communication node is on Kerbin.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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

Game Scenes All

is_vessel
Whether the communication node is a vessel.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

vessel

The vessel for this communication node.

Attribute Read-only, cannot be set

Return type *Vessel*

Game Scenes All

8.3.8 Parts

The following classes allow interaction with a vessels individual parts.

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`)

Game Scenes All

root

The vessels root part.

Attribute Read-only, cannot be set

Return type `Part`

Game Scenes All

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`

Game Scenes All

with_name (*name*)

A list of parts whose `Part.name` is *name*.

Parameters **name** (`str`) –

Return type list(`Part`)

Game Scenes All

with_title (*title*)

A list of all parts whose `Part.title` is *title*.

Parameters **title** (`str`) –

Return type list(`Part`)

Game Scenes All

with_tag (*tag*)

A list of all parts whose `Part.tag` is *tag*.

Parameters **tag** (*str*) –

Return type list(*Part*)

Game Scenes All

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*)

Game Scenes All

in_stage (*stage*)

A list of all parts that are activated in the given *stage*.

Parameters **stage** (*int*) –

Return type list(*Part*)

Game Scenes All

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*)

Game Scenes All

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*)

Game Scenes All

antennas

A list of all antennas in the vessel.

Attribute Read-only, cannot be set

Return type list(*Antenna*)

Game Scenes All

cargo_bays

A list of all cargo bays in the vessel.

Attribute Read-only, cannot be set

Return type list(*CargoBay*)

Game Scenes All

control_surfaces
A list of all control surfaces in the vessel.

Attribute Read-only, cannot be set

Return type list(*ControlSurface*)

Game Scenes All

decouplers
A list of all decouplers in the vessel.

Attribute Read-only, cannot be set

Return type list(*Decoupler*)

Game Scenes All

docking_ports
A list of all docking ports in the vessel.

Attribute Read-only, cannot be set

Return type list(*DockingPort*)

Game Scenes All

engines
A list of all engines in the vessel.

Attribute Read-only, cannot be set

Return type list(*Engine*)

Game Scenes All

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*)

Game Scenes All

fairings
A list of all fairings in the vessel.

Attribute Read-only, cannot be set

Return type list(*Fairing*)

Game Scenes All

intakes
A list of all intakes in the vessel.

Attribute Read-only, cannot be set

Return type list(*Intake*)

Game Scenes All

legs
A list of all landing legs attached to the vessel.

Attribute Read-only, cannot be set

Return type list(*Leg*)

Game Scenes All

launch_clamps
A list of all launch clamps attached to the vessel.

Attribute Read-only, cannot be set

Return type list(*LaunchClamp*)

Game Scenes All

lights
A list of all lights in the vessel.

Attribute Read-only, cannot be set

Return type list(*Light*)

Game Scenes All

parachutes
A list of all parachutes in the vessel.

Attribute Read-only, cannot be set

Return type list(*Parachute*)

Game Scenes All

radiators
A list of all radiators in the vessel.

Attribute Read-only, cannot be set

Return type list(*Radiator*)

Game Scenes All

rcs
A list of all RCS blocks/thrusters in the vessel.

Attribute Read-only, cannot be set

Return type list(*RCS*)

Game Scenes All

reaction_wheels
A list of all reaction wheels in the vessel.

Attribute Read-only, cannot be set

Return type list(*ReactionWheel*)

Game Scenes All

resource_converters
A list of all resource converters in the vessel.

Attribute Read-only, cannot be set

Return type list(*ResourceConverter*)

Game Scenes All

resource_harvesters
A list of all resource harvesters in the vessel.

Attribute Read-only, cannot be set

Return type list(*ResourceHarvester*)

Game Scenes All

sensors
A list of all sensors in the vessel.

Attribute Read-only, cannot be set

Return type list(*Sensor*)

Game Scenes All

solar_panels
A list of all solar panels in the vessel.

Attribute Read-only, cannot be set

Return type list(*SolarPanel*)

Game Scenes All

wheels
A list of all wheels in the vessel.

Attribute Read-only, cannot be set

Return type list(*Wheel*)

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

Note: This string is shared with kOS if it is installed.

`highlighted`

Whether the part is highlighted.

Attribute Can be read or written

Return type bool

Game Scenes All

`highlight_color`

The color used to highlight the part, as an RGB triple.

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes All

`cost`

The cost of the part, in units of funds.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

`vessel`

The vessel that contains this part.

Attribute Read-only, cannot be set

Return type *Vessel*

Game Scenes All

`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*

Game Scenes All

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*)

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

Note: See the discussion on *Staging*.

massless

Whether the part is massless.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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 double

Game Scenes All

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 double

Game Scenes All

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

Game Scenes All

dynamic_pressure

The dynamic pressure acting on the part, in Pascals.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

impact_tolerance

The impact tolerance of the part, in meters per second.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

temperature

Temperature of the part, in Kelvin.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

skin_temperature

Temperature of the skin of the part, in Kelvin.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

max_temperature

Maximum temperature that the part can survive, in Kelvin.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

`max_skin_temperature`
Maximum temperature that the skin of the part can survive, in Kelvin.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

`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

Game Scenes All

`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

Game Scenes All

`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

Game Scenes All

`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

Game Scenes All

`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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

resources

A *Resources* object for the part.

Attribute Read-only, cannot be set

Return type *Resources*

Game Scenes All

crossfeed

Whether this part is crossfeed capable.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

is_fuel_line

Whether this part is a fuel line.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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*)

Game Scenes All

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*)

Game Scenes All

Note: See the discussion on *Fuel Lines*.

modules

The modules for this part.

Attribute Read-only, cannot be set

Return type list(*Module*)

Game Scenes All

antenna

A *Antenna* if the part is an antenna, otherwise `None`.

Attribute Read-only, cannot be set

Return type *Antenna*

Game Scenes All

cargo_bay

A *CargoBay* if the part is a cargo bay, otherwise `None`.

Attribute Read-only, cannot be set

Return type *CargoBay*

Game Scenes All

control_surface

A *ControlSurface* if the part is an aerodynamic control surface, otherwise `None`.

Attribute Read-only, cannot be set

Return type *ControlSurface*

Game Scenes All

decoupler

A *Decoupler* if the part is a decoupler, otherwise `None`.

Attribute Read-only, cannot be set

Return type *Decoupler*

Game Scenes All

docking_port

A *DockingPort* if the part is a docking port, otherwise `None`.

Attribute Read-only, cannot be set

Return type *DockingPort*

Game Scenes All

engine

An *Engine* if the part is an engine, otherwise None.

Attribute Read-only, cannot be set

Return type *Engine*

Game Scenes All

experiment

An *Experiment* if the part is a science experiment, otherwise None.

Attribute Read-only, cannot be set

Return type *Experiment*

Game Scenes All

fairing

A *Fairing* if the part is a fairing, otherwise None.

Attribute Read-only, cannot be set

Return type *Fairing*

Game Scenes All

intake

An *Intake* if the part is an intake, otherwise None.

Attribute Read-only, cannot be set

Return type *Intake*

Game Scenes All

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*

Game Scenes All

launch_clamp

A *LaunchClamp* if the part is a launch clamp, otherwise None.

Attribute Read-only, cannot be set

Return type *LaunchClamp*

Game Scenes All

light

A *Light* if the part is a light, otherwise None.

Attribute Read-only, cannot be set

Return type *Light*

Game Scenes All

parachute
A *Parachute* if the part is a parachute, otherwise `None`.

Attribute Read-only, cannot be set

Return type *Parachute*

Game Scenes All

radiator
A *Radiator* if the part is a radiator, otherwise `None`.

Attribute Read-only, cannot be set

Return type *Radiator*

Game Scenes All

rcs
A *RCS* if the part is an RCS block/thruster, otherwise `None`.

Attribute Read-only, cannot be set

Return type *RCS*

Game Scenes All

reaction_wheel
A *ReactionWheel* if the part is a reaction wheel, otherwise `None`.

Attribute Read-only, cannot be set

Return type *ReactionWheel*

Game Scenes All

resource_converter
A *ResourceConverter* if the part is a resource converter, otherwise `None`.

Attribute Read-only, cannot be set

Return type *ResourceConverter*

Game Scenes All

resource_harvester
A *ResourceHarvester* if the part is a resource harvester, otherwise `None`.

Attribute Read-only, cannot be set

Return type *ResourceHarvester*

Game Scenes All

sensor
A *Sensor* if the part is a sensor, otherwise `None`.

Attribute Read-only, cannot be set

Return type *Sensor*

Game Scenes All

solar_panel
A *SolarPanel* if the part is a solar panel, otherwise `None`.

Attribute Read-only, cannot be set

Return type *SolarPanel*

Game Scenes All

wheel

A *Wheel* if the part is a wheel, otherwise None.

Attribute Read-only, cannot be set

Return type *Wheel*

Game Scenes All

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(double, double, double)

Game Scenes All

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.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(double, double, double)

Game Scenes All

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(double, double, double), tuple(double, double, double))

Game Scenes All

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(double, double, double)

Game Scenes All

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(double, double, double)

Game Scenes All

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(double, double, double, double)

Game Scenes All

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(double, double, double)

Game Scenes All

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(double)

Game Scenes All

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*

Game Scenes All

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*.

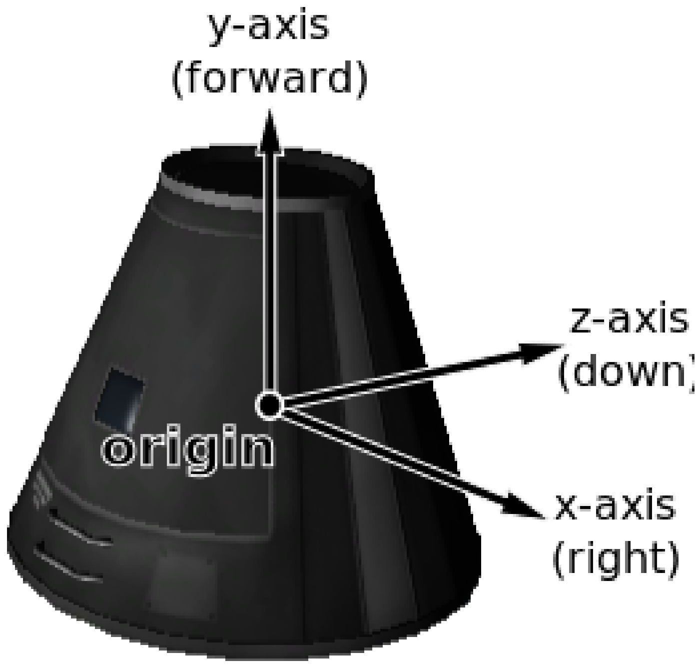


Fig. 7: Mk1 Command Pod reference frame origin and axes

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*

Game Scenes All

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*

Game Scenes All

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.

Game Scenes All

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*

Game Scenes All

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(double, double, double)

Game Scenes All

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(double, double, double)

Game Scenes All

reference_frame
The reference frame of the force vector and position.

Attribute Can be read or written

Return type *ReferenceFrame*

Game Scenes All

remove()

Remove the force.

Game Scenes All

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

Game Scenes All

part

The part that contains this module.

Attribute Read-only, cannot be set

Return type *Part*

Game Scenes All

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)

Game Scenes All

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

Game Scenes All

get_field(*name*)

Returns the value of a field.

Parameters **name** (*str*) – Name of the field.

Return type str

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

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.

Game Scenes All

reset_field (*name*)
Set the value of a field to its original value.

Parameters **name** (*str*) – Name of the field.

Game Scenes All

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)

Game Scenes All

has_event (*name*)
`True` if the module has an event with the given name.

Parameters **name** (*str*) –

Return type bool

Game Scenes All

trigger_event (*name*)
Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters **name** (*str*) –

Game Scenes All

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)

Game Scenes All

has_action (*name*)

`True` if the part has an action with the given name.

Parameters **name** (*str*) –

Return type bool

Game Scenes All

set_action (*name*[, *value = True*])

Set the value of an action with the given name.

Parameters

- **name** (*str*) –
- **value** (*bool*) –

Game Scenes All

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*

Game Scenes All

state

The current state of the antenna.

Attribute Read-only, cannot be set

Return type *AntennaState*

Game Scenes All

deployable

Whether the antenna is deployable.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

deployed

Whether the antenna is deployed.

Attribute Can be read or written

Return type bool

Game Scenes All

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

Game Scenes All

transmit()
Transmit data.

Game Scenes All

cancel()
Cancel current transmission of data.

Game Scenes All

allow_partial
Whether partial data transmission is permitted.

Attribute Can be read or written

Return type bool

Game Scenes All

power
The power of the antenna.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

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

Game Scenes All

combinable_exponent
Exponent used to calculate the combined power of multiple antennae on a vessel.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

packet_interval
Interval between sending packets in seconds.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

packet_size
Amount of data sent per packet in Mits.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

packet_resource_cost
Units of electric charge consumed per packet sent.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

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*

Game Scenes All

state

The state of the cargo bay.

Attribute Read-only, cannot be set

Return type *CargoBayState*

Game Scenes All

open

Whether the cargo bay is open.

Attribute Can be read or written

Return type bool

Game Scenes All

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*

Game Scenes All

pitch_enabled
Whether the control surface has pitch control enabled.

Attribute Can be read or written

Return type bool

Game Scenes All

yaw_enabled
Whether the control surface has yaw control enabled.

Attribute Can be read or written

Return type bool

Game Scenes All

roll_enabled
Whether the control surface has roll control enabled.

Attribute Can be read or written

Return type bool

Game Scenes All

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

Game Scenes All

inverted
Whether the control surface movement is inverted.

Attribute Can be read or written

Return type bool

Game Scenes All

deployed

Whether the control surface has been fully deployed.

Attribute Can be read or written

Return type bool

Game Scenes All

surface_area

Surface area of the control surface in m^2.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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(double, double, double), tuple(double, double, double))

Game Scenes All

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*

Game Scenes All

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*

Game Scenes All

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

Game Scenes All

staged
Whether the decoupler is enabled in the staging sequence.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

impulse
The impulse that the decoupler imparts when it is fired, in Newton seconds.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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*

Game Scenes All

state
The current state of the docking port.

Attribute Read-only, cannot be set

Return type *DockingPortState*

Game Scenes All

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*

Game Scenes All

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*

Game Scenes All

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

Game Scenes All

has_shield

Whether the docking port has a shield.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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

Game Scenes All

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(double, double, double)

Game Scenes All

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(double, double, double)

Game Scenes All

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(double, double, double, double)

Game Scenes All

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*

Game Scenes All

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by *Part.reference_frame*.

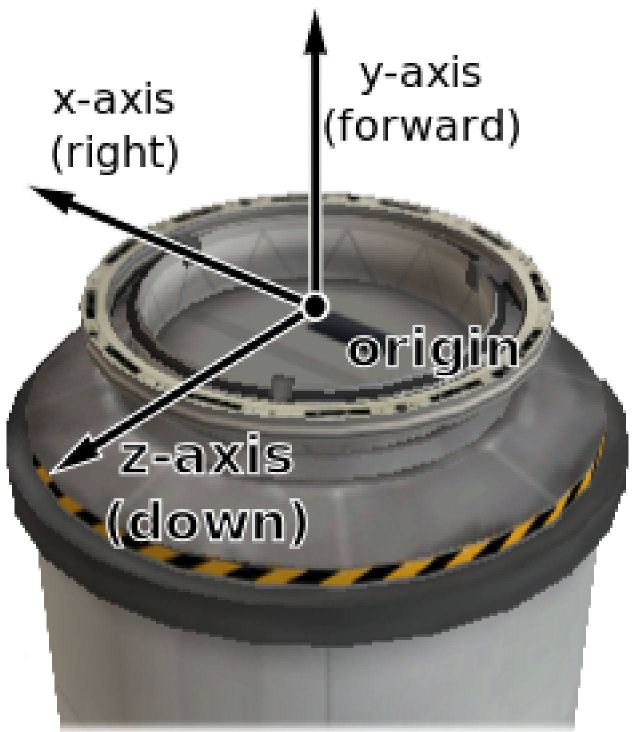


Fig. 8: Docking port reference frame origin and axes

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).

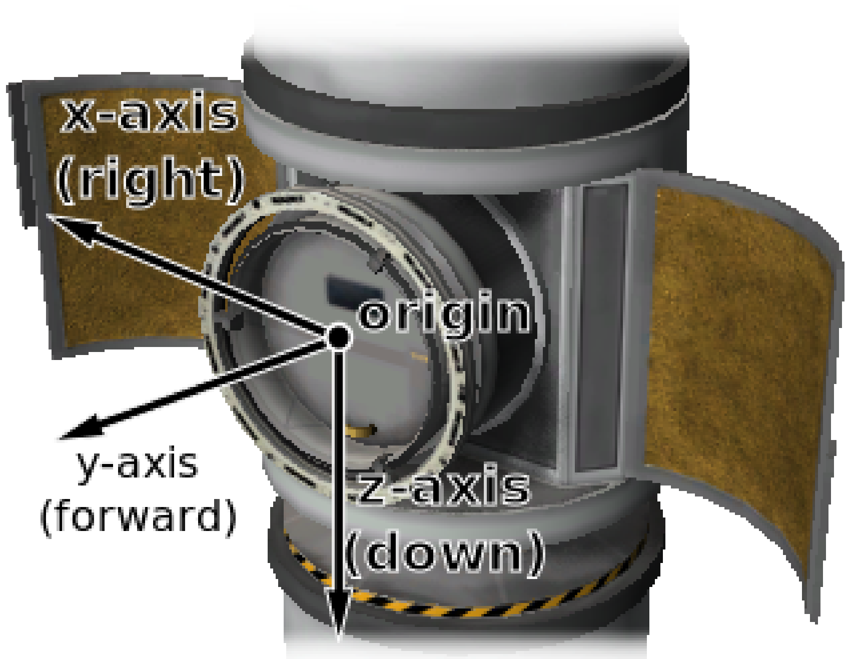


Fig. 9: Inline docking port reference frame origin and axes

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.

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*

Game Scenes All

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

Game Scenes All

thrust
The current amount of thrust being produced by the engine, in Newtons.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

thrusters
The components of the engine that generate thrust.

Attribute Read-only, cannot be set

Return type list(*Thruster*)

Game Scenes All

Note: For example, this corresponds to the rocket nozzel on a solid rocket booster, or the individual nozzels 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

Game Scenes All

vacuum_specific_impulse
The vacuum specific impulse of the engine, in seconds.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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

Game Scenes All

propellant_names
The names of the propellants that the engine consumes.

Attribute Read-only, cannot be set

Return type list(str)

Game Scenes All

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)

Game Scenes All

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*)

Game Scenes All

has_fuel

Whether the engine has any fuel available.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

has_modes

Whether the engine has multiple modes of operation.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

mode

The name of the current engine mode.

Attribute Can be read or written

Return type str

Game Scenes All

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*)

Game Scenes All

toggle_mode()

Toggle the current engine mode.

Game Scenes All

auto_mode_switch

Whether the engine will automatically switch modes.

Attribute Can be read or written

Return type bool

Game Scenes All

gimballed

Whether the engine is gimballed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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(double, double, double), tuple(double, double, double))

Game Scenes All

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

Game Scenes All

current_amount

The current amount of propellant.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

current_requirement

The required amount of propellant.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

total_resource_available

The total amount of the underlying resource currently reachable given resource flow rules.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

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 double

Game Scenes All

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

Game Scenes All

ignore_for_thrust_curve
If this propellant should be ignored for thrust curve calculations.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

draw_stack_gauge
If this propellant has a stack gauge or not.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

is_deprived
If this propellant is deprived.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

ratio
The propellant ratio.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

Experiment

class Experiment
Obtained by calling *Part.experiment*.

part
The part object for this experiment.

Attribute Read-only, cannot be set

Return type *Part*

Game Scenes All

run()
Run the experiment.

Game Scenes All

transmit()
Transmit all experimental data contained by this part.

Game Scenes All

dump()
Dump the experimental data contained by the experiment.

Game Scenes All

reset()
Reset the experiment.

Game Scenes All

deployed
Whether the experiment has been deployed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

rerunnable
Whether the experiment can be re-run.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

inoperable
Whether the experiment is inoperable.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

has_data
Whether the experiment contains data.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

data
The data contained in this experiment.

Attribute Read-only, cannot be set

Return type list(*ScienceData*)

Game Scenes All

biome
The name of the biome the experiment is currently in.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

available
Determines if the experiment is available given the current conditions.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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*

Game Scenes All

class ScienceData
Obtained by calling *Experiment.data*.

data_amount
Data amount.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

science_value
Science value.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

transmit_value
Transmit value.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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

Game Scenes All

is_complete
Whether the experiment has been completed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

science
Amount of science already earned from this subject, not updated until after transmission/recovery.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

science_cap
Total science allowable for this subject.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

data_scale
Multiply science value by this to determine data amount in mits.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

subject_value
Multiplier for specific Celestial Body/Experiment Situation combination.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

scientific_value
Diminishing value multiplier for decreasing the science value returned from repeated experiments.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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*

Game Scenes All

jettison()
Jettison the fairing. Has no effect if it has already been jettisoned.

Game Scenes All

jettisoned
Whether the fairing has been jettisoned.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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*

Game Scenes All

open

Whether the intake is open.

Attribute Can be read or written

Return type bool

Game Scenes All

speed

Speed of the flow into the intake, in m/s.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

flow

The rate of flow into the intake, in units of resource per second.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

area

The area of the intake's opening, in square meters.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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*

Game Scenes All

state

The current state of the landing leg.

Attribute Read-only, cannot be set

Return type *LegState*

Game Scenes All

deployable

Whether the leg is deployable.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

deployed

Whether the landing leg is deployed.

Attribute Can be read or written

Return type bool

Game Scenes All

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

Game Scenes All

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*

Game Scenes All

release()
Releases the docking clamp. Has no effect if the clamp has already been released.

Game Scenes All

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*

Game Scenes All

active
Whether the light is switched on.

Attribute Can be read or written

Return type bool

Game Scenes All

color
The color of the light, as an RGB triple.

Attribute Can be read or written

Return type tuple(float, float, float)

Game Scenes All

power_usage
The current power usage, in units of charge per second.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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*

Game Scenes All

deploy()

Deploys the parachute. This has no effect if the parachute has already been deployed.

Game Scenes All

deployed

Whether the parachute has been deployed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

arm()

Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

Game Scenes All

armed

Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

state

The current state of the parachute.

Attribute Read-only, cannot be set

Return type *ParachuteState*

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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*

Game Scenes All

deployable
Whether the radiator is deployable.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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

Game Scenes All

state
The current state of the radiator.

Attribute Read-only, cannot be set

Return type *RadiatorState*

Game Scenes All

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`

Game Scenes All

count
The number of converters in the part.

Attribute Read-only, cannot be set

Return type int

Game Scenes All

name (*index*)
The name of the specified converter.

Parameters **index** (`int`) – Index of the converter.

Return type str

Game Scenes All

active (*index*)
True if the specified converter is active.

Parameters **index** (`int`) – Index of the converter.

Return type bool

Game Scenes All

start (*index*)
Start the specified converter.

Parameters **index** (`int`) – Index of the converter.

Game Scenes All

stop (*index*)
Stop the specified converter.

Parameters **index** (*int*) – Index of the converter.

Game Scenes All

state (*index*)

The state of the specified converter.

Parameters **index** (*int*) – Index of the converter.

Return type *ResourceConverterState*

Game Scenes All

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

Game Scenes All

inputs (*index*)

List of the names of resources consumed by the specified converter.

Parameters **index** (*int*) – Index of the converter.

Return type list(str)

Game Scenes All

outputs (*index*)

List of the names of resources produced by the specified converter.

Parameters **index** (*int*) – Index of the converter.

Return type list(str)

Game Scenes All

optimum_core_temperature

The core temperature at which the converter will operate with peak efficiency, in Kelvin.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

core_temperature

The core temperature of the converter, in Kelvin.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

thermal_efficiency

The thermal efficiency of the converter, as a percentage of its maximum.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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*

Game Scenes All

state
The state of the harvester.

Attribute Read-only, cannot be set

Return type *ResourceHarvesterState*

Game Scenes All

deployed
Whether the harvester is deployed.

Attribute Can be read or written

Return type bool

Game Scenes All

active
Whether the harvester is actively drilling.

Attribute Can be read or written

Return type bool

Game Scenes All

extraction_rate
The rate at which the drill is extracting ore, in units per second.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

thermal_efficiency
The thermal efficiency of the drill, as a percentage of its maximum.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

core_temperature
The core temperature of the drill, in Kelvin.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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

Game Scenes All

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*

Game Scenes All

active
Whether the reaction wheel is active.

Attribute Can be read or written

Return type bool

Game Scenes All

broken

Whether the reaction wheel is broken.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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(double, double, double), tuple(double, double, double))

Game Scenes All

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(double, double, double), tuple(double, double, double))

Game Scenes All

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*

Game Scenes All

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

Game Scenes All

enabled

Whether the RCS thrusters are enabled.

Attribute Can be read or written

Return type bool

Game Scenes All

pitch_enabled
Whether the RCS thruster will fire when pitch control input is given.

Attribute Can be read or written

Return type bool

Game Scenes All

yaw_enabled
Whether the RCS thruster will fire when yaw control input is given.

Attribute Can be read or written

Return type bool

Game Scenes All

roll_enabled
Whether the RCS thruster will fire when roll control input is given.

Attribute Can be read or written

Return type bool

Game Scenes All

forward_enabled
Whether the RCS thruster will fire when pitch control input is given.

Attribute Can be read or written

Return type bool

Game Scenes All

up_enabled
Whether the RCS thruster will fire when yaw control input is given.

Attribute Can be read or written

Return type bool

Game Scenes All

right_enabled
Whether the RCS thruster will fire when roll control input is given.

Attribute Can be read or written

Return type bool

Game Scenes All

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(double, double, double), tuple(double, double, double))

Game Scenes All

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

Game Scenes All

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

Game Scenes All

thrusters

A list of thrusters, one of each nozzel in the RCS part.

Attribute Read-only, cannot be set

Return type list(*Thruster*)

Game Scenes All

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

Game Scenes All

vacuum_specific_impulse

The vacuum specific impulse of the RCS, in seconds.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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

Game Scenes All

propellants

The names of resources that the RCS consumes.

Attribute Read-only, cannot be set

Return type list(str)

Game Scenes All

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)

Game Scenes All

has_fuel

Whether the RCS has fuel available.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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*

Game Scenes All

active

Whether the sensor is active.

Attribute Can be read or written

Return type bool

Game Scenes All

value

The current value of the sensor.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

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*

Game Scenes All

deployable

Whether the solar panel is deployable.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

deployed

Whether the solar panel is extended.

Attribute Can be read or written

Return type bool

Game Scenes All

state

The current state of the solar panel.

Attribute Read-only, cannot be set

Return type *SolarPanelState*

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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*

Game Scenes All

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(double, double, double)

Game Scenes All

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(double, double, double)

Game Scenes All

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 gimballing.
- 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*

Game Scenes All

gimballed

Whether the thruster is gimballed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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(double, double, double)

Game Scenes All

gimbal_angle

The current gimbal angle in the pitch, roll and yaw axes, in degrees.

Attribute Read-only, cannot be set

Return type tuple(double, double, double)

Game Scenes All

initial_thrust_position(*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** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

Return type tuple(double, double, double)

Game Scenes All

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 gimballing), 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(double, double, double)

Game Scenes All

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*

Game Scenes All

state

The current state of the wheel.

Attribute Read-only, cannot be set

Return type *WheelState*

Game Scenes All

radius

Radius of the wheel, in meters.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

grounded

Whether the wheel is touching the ground.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

has_brakes

Whether the wheel has brakes.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

brakes

The braking force, as a percentage of maximum, when the brakes are applied.

Attribute Can be read or written

Return type float

Game Scenes All

auto_friction_control

Whether automatic friction control is enabled.

Attribute Can be read or written

Return type bool

Game Scenes All

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

Game Scenes All

deployable
Whether the wheel is deployable.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

deployed
Whether the wheel is deployed.

Attribute Can be read or written

Return type bool

Game Scenes All

powered
Whether the wheel is powered by a motor.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

motor_enabled
Whether the motor is enabled.

Attribute Can be read or written

Return type bool

Game Scenes All

motor_inverted
Whether the direction of the motor is inverted.

Attribute Can be read or written

Return type bool

Game Scenes All

motor_state
Whether the direction of the motor is inverted.

Attribute Read-only, cannot be set

Return type *MotorState*

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

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

Game Scenes All

steerable

Whether the wheel has steering.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

steering_enabled

Whether the wheel steering is enabled.

Attribute Can be read or written

Return type bool

Game Scenes All

steering_inverted

Whether the wheel steering is inverted.

Attribute Can be read or written

Return type bool

Game Scenes All

has_suspension

Whether the wheel has suspension.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

`suspension_spring_strength`
Suspension spring strength, as set in the editor.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

`suspension_damper_strength`
Suspension damper strength, as set in the editor.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

`broken`
Whether the wheel is broken.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

`repairable`
Whether the wheel is repairable.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

`stress`
Current stress on the wheel.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

`stress_tolerance`
Stress tolerance of the wheel.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

`stress_percentage`
Current stress on the wheel as a percentage of its stress tolerance.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

deflection
Current deflection of the wheel.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

slip
Current slip of the wheel.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

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.

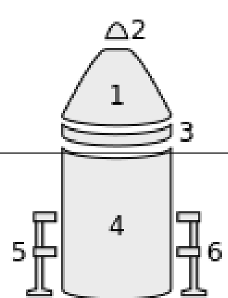
disabled
The motor is disabled.

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.
stack = [(root, 0)]
while stack:
    part, depth = st
    print(' '*depth,
    for child in par
        stack.append
```

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 Python example does a depth-first traversal as before, but also prints out the attachment mode used by the part:

```python
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
```

(continues on next page)

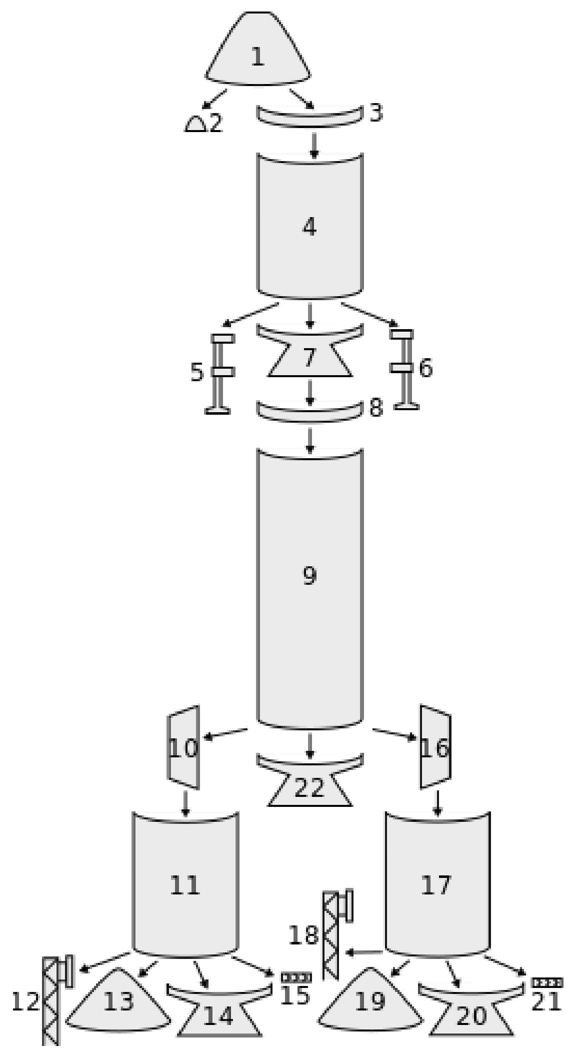


Fig. 11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the child part.

(continued from previous page)

```
        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

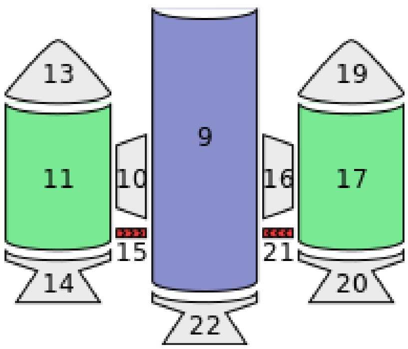


Fig. 12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

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).

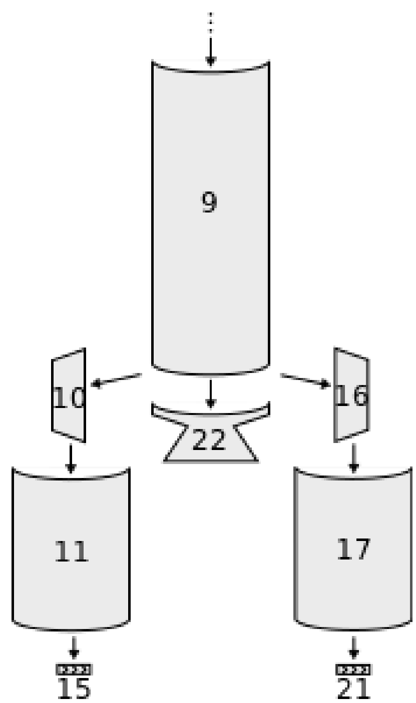


Fig. 13: **Figure 4** – A subset of the parts tree from Figure 2 above.

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*.

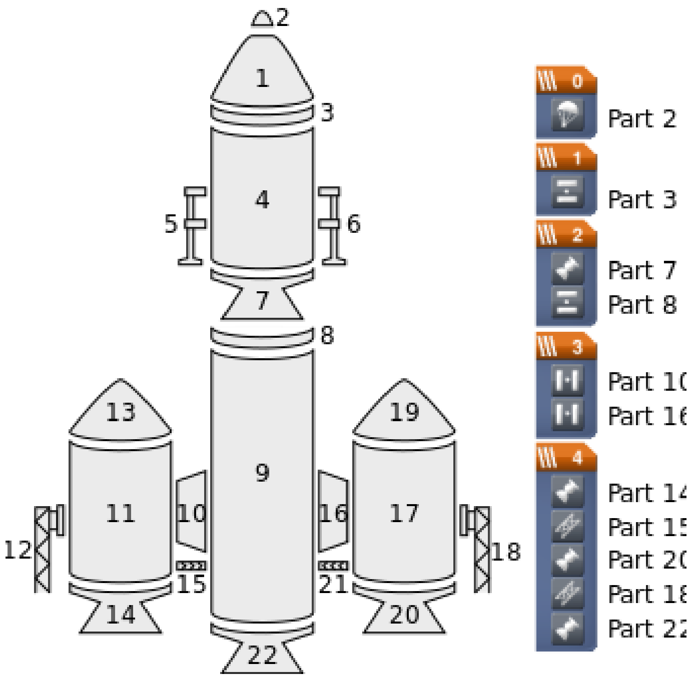


Fig. 14: **Figure 6 –** Example vessel from Figure 1 with a staging sequence.

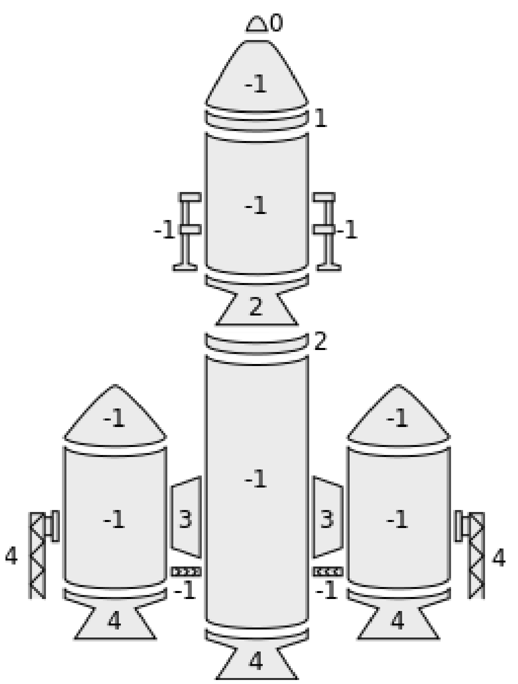


Fig. 15: **Figure 7** – The stage in which each part is *activated*.

8.3.9 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*)

Game Scenes Flight

with_resource(*name*)

All the individual resources with the given name that can be stored.

Parameters **name** (*str*) –

Return type list(*Resource*)

Game Scenes Flight

names

A list of resource names that can be stored.

Attribute Read-only, cannot be set

Return type list(str)

Game Scenes Flight

has_resource(*name*)

Check whether the named resource can be stored.

Parameters **name** (*str*) – The name of the resource.

Return type bool

Game Scenes Flight

amount(*name*)

Returns the amount of a resource that is currently stored.

Parameters **name** (*str*) – The name of the resource.

Return type float

Game Scenes Flight

max(*name*)

Returns the amount of a resource that can be stored.

Parameters **name** (*str*) – The name of the resource.

Return type float

Game Scenes Flight

static density(*name*)

Returns the density of a resource, in kg/l.

Parameters **name** (*str*) – The name of the resource.

Return type float

Game Scenes Flight

static flow_mode(*name*)
Returns the flow mode of a resource.

Parameters **name** (*str*) – The name of the resource.

Return type *ResourceFlowMode*

Game Scenes Flight

enabled
Whether use of all the resources are enabled.

Attribute Can be read or written

Return type bool

Game Scenes Flight

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

Game Scenes All

part
The part containing the resource.

Attribute Read-only, cannot be set

Return type *Part*

Game Scenes All

amount
The amount of the resource that is currently stored in the part.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

max
The total amount of the resource that can be stored in the part.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

density

The density of the resource, in kg/l.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

flow_mode

The flow mode of the resource.

Attribute Read-only, cannot be set

Return type *ResourceFlowMode*

Game Scenes All

enabled

Whether use of this resource is enabled.

Attribute Can be read or written

Return type bool

Game Scenes All

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*

Game Scenes All

amount

The amount of the resource that has been transferred.

Attribute Read-only, cannot be set

Return type float

Game Scenes All

complete

Whether the transfer has completed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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.

8.3.10 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 double

Game Scenes Flight

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 double

Game Scenes Flight

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 double

Game Scenes Flight

delta_v
The delta-v of the maneuver node, in meters per second.

Attribute Can be read or written

Return type double

Game Scenes Flight

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 double

Game Scenes Flight

`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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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 double

Game Scenes Flight

time_to
The time until the maneuver node will be encountered, in seconds.

Attribute Read-only, cannot be set

Return type double

Game Scenes Flight

orbit
The orbit that results from executing the maneuver node.

Attribute Read-only, cannot be set

Return type *Orbit*

Game Scenes Flight

remove()
Removes the maneuver node.

Game Scenes Flight

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*

Game Scenes Flight

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*

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

8.3.11 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*

Game Scenes All

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*

Game Scenes All

Note: The *position* reference frame is required but all other reference frames are optional. If omitted, they are set to the *position* reference frame.

8.3.12 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.

Game Scenes Flight

disengage()

Disengage the auto-pilot.

Game Scenes Flight

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.

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

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

Game Scenes Flight

reference_frame
The reference frame for the target direction (*AutoPilot.target_direction*).

Attribute Can be read or written

Return type *ReferenceFrame*

Game Scenes Flight

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

Game Scenes Flight

target_heading
The target heading, in degrees, between 0° and 360°.

Attribute Can be read or written

Return type float

Game Scenes Flight

target_roll
The target roll, in degrees. `NaN` if no target roll is set.

Attribute Can be read or written

Return type float

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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°.

Game Scenes Flight

sas
The state of SAS.

Attribute Can be read or written

Return type bool

Game Scenes Flight

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*

Game Scenes Flight

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 double

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

pitch_pid_gains

Gains for the pitch PID controller.

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

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(double, double, double)

Game Scenes Flight

Note: When *AutoPilot.auto_tune* is true, these values are updated automatically, which will overwrite any manual changes.

8.3.13 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*

Game Scenes Flight

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

Game Scenes Flight

heading

The heading of the camera, in degrees.

Attribute Can be read or written

Return type float

Game Scenes Flight

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

Game Scenes Flight

min_pitch

The minimum pitch of the camera.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`max_pitch`
The maximum pitch of the camera.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`min_distance`
Minimum distance from the camera to the subject, in meters.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`max_distance`
Maximum distance from the camera to the subject, in meters.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`default_distance`
Default distance from the camera to the subject, in meters.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`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 is the camera is not in map mode.

Attribute Can be read or written

Return type *CelestialBody*

Game Scenes Flight

`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 is the camera is not in map mode.

Attribute Can be read or written

Return type *Vessel*

Game Scenes Flight

`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 is the camera is not in map mode.

Attribute Can be read or written

Return type *Node*

Game Scenes Flight

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.

8.3.14 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*)

Game Scenes All

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** (*double*) – Latitude of the waypoint.
- **longitude** (*double*) – Longitude of the waypoint.
- **body** (CelestialBody) – Celestial body the waypoint is attached to.
- **name** (*str*) – Name of the waypoint.

Return type *Waypoint*

Game Scenes All

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** (`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** (`str`) – Name of the waypoint.

Return type `Waypoint`

Game Scenes All

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)

Game Scenes All

icons
Returns all available icons (from "GameData/Squad/Contracts/Icons/").

Attribute Read-only, cannot be set

Return type list(str)

Game Scenes All

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`

Game Scenes Flight

name
The name of the waypoint as it appears on the map and the contract.

Attribute Can be read or written

Return type str

Game Scenes Flight

color
The seed of the icon color. See `WaypointManager.colors` for example colors.

Attribute Can be read or written

Return type int

Game Scenes Flight

icon
The icon of the waypoint.

Attribute Can be read or written

Return type str

Game Scenes Flight

latitude
The latitude of the waypoint.

Attribute Can be read or written

Return type double

Game Scenes Flight

longitude
The longitude of the waypoint.

Attribute Can be read or written

Return type double

Game Scenes Flight

mean_altitude
The altitude of the waypoint above sea level, in meters.

Attribute Can be read or written

Return type double

Game Scenes Flight

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 double

Game Scenes Flight

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 double

Game Scenes Flight

near_surface
`True` if the waypoint is near to the surface of a body.

Attribute Read-only, cannot be set

Return type bool

Game Scenes Flight

grounded
`True` if the waypoint is attached to the ground.

Attribute Read-only, cannot be set

Return type bool

Game Scenes Flight

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.

Attribute Read-only, cannot be set

Return type int

Game Scenes Flight

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*.

Attribute Read-only, cannot be set

Return type bool

Game Scenes Flight

has_contract
Whether the waypoint belongs to a contract.

Attribute Read-only, cannot be set

Return type bool

Game Scenes Flight

contract
The associated contract.

Attribute Read-only, cannot be set

Return type *Contract*

Game Scenes Flight

remove()
Removes the waypoint.

Game Scenes Flight

8.3.15 Contracts

class ContractManager
Contracts manager. Obtained by calling *contract_manager*.

types
A list of all contract types.

Attribute Read-only, cannot be set

Return type set(str)

Game Scenes All

all_contracts
A list of all contracts.

Attribute Read-only, cannot be set

Return type list(*Contract*)

Game Scenes All

active_contracts
A list of all active contracts.

Attribute Read-only, cannot be set

Return type list(*Contract*)

Game Scenes All

offered_contracts
A list of all offered, but unaccepted, contracts.

Attribute Read-only, cannot be set

Return type list(*Contract*)

Game Scenes All

completed_contracts
A list of all completed contracts.

Attribute Read-only, cannot be set

Return type list(*Contract*)

Game Scenes All

failed_contracts
A list of all failed contracts.

Attribute Read-only, cannot be set

Return type list(*Contract*)

Game Scenes All

class Contract
A contract. Can be accessed using *contract_manager*.

type
Type of the contract.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

title
Title of the contract.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

description

Description of the contract.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

notes

Notes for the contract.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

synopsis

Synopsis for the contract.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

keywords

Keywords for the contract.

Attribute Read-only, cannot be set

Return type list(str)

Game Scenes All

state

State of the contract.

Attribute Read-only, cannot be set

Return type *ContractState*

Game Scenes All

seen

Whether the contract has been seen.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

read

Whether the contract has been read.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

active
Whether the contract is active.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

failed
Whether the contract has been failed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

can_be_canceled
Whether the contract can be canceled.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

can_be_declined
Whether the contract can be declined.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

can_be_failed
Whether the contract can be failed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

accept()
Accept an offered contract.

Game Scenes All

cancel()
Cancel an active contract.

Game Scenes All

decline()
Decline an offered contract.

Game Scenes All

funds_advance
Funds received when accepting the contract.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

funds_completion
Funds received on completion of the contract.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

funds_failure
Funds lost if the contract is failed.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

reputation_completion
Reputation gained on completion of the contract.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

reputation_failure
Reputation lost if the contract is failed.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

science_completion
Science gained on completion of the contract.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

parameters
Parameters for the contract.

Attribute Read-only, cannot be set

Return type list(*ContractParameter*)

Game Scenes All

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

Game Scenes All

notes
Notes for the parameter.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

children
Child contract parameters.

Attribute Read-only, cannot be set

Return type list(*ContractParameter*)

Game Scenes All

completed
Whether the parameter has been completed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

failed
Whether the parameter has been failed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

optional
Whether the contract parameter is optional.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

funds_completion
Funds received on completion of the contract parameter.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

funds_failure
Funds lost if the contract parameter is failed.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

reputation_completion
Reputation gained on completion of the contract parameter.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

reputation_failure
Reputation lost if the contract parameter is failed.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

science_completion
Science gained on completion of the contract parameter.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

8.3.16 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])
```

8.4 Drawing API

8.4.1 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*

Game Scenes Flight

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*

Game Scenes Flight

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*

Game Scenes Flight

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*

Game Scenes Flight

static clear ([*client_only = False*])

Remove all objects being drawn.

Parameters **client_only** (*bool*) – If true, only remove objects created by the calling client.

Game Scenes Flight

8.4.2 Line

class Line

A line. Created using *add_line()*.

start

Start position of the line.

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes Flight

end

End position of the line.

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes Flight

reference_frame

Reference frame for the positions of the object.

Attribute Can be read or written

Return type *SpaceCenter.ReferenceFrame*

Game Scenes Flight

visible
Whether the object is visible.

Attribute Can be read or written

Return type bool

Game Scenes Flight

color
Set the color

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes Flight

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

Game Scenes Flight

thickness
Set the thickness

Attribute Can be read or written

Return type float

Game Scenes Flight

remove()
Remove the object.

Game Scenes Flight

8.4.3 Polygon

class Polygon
A polygon. Created using *add_polygon()*.

vertices
Vertices for the polygon.

Attribute Can be read or written

Return type list(tuple(double, double, double))

Game Scenes Flight

reference_frame
Reference frame for the positions of the object.

Attribute Can be read or written

Return type *SpaceCenter.ReferenceFrame*

Game Scenes Flight

visible
Whether the object is visible.

Attribute Can be read or written

Return type bool

Game Scenes Flight

remove()
Remove the object.

Game Scenes Flight

color
Set the color

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes Flight

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

Game Scenes Flight

thickness
Set the thickness

Attribute Can be read or written

Return type float

Game Scenes Flight

8.4.4 Text

class Text
Text. Created using `add_text()`.

position
Position of the text.

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes Flight

rotation
Rotation of the text as a quaternion.

Attribute Can be read or written

Return type tuple(double, double, double, double)

Game Scenes Flight

reference_frame
Reference frame for the positions of the object.

Attribute Can be read or written

Return type *SpaceCenter.ReferenceFrame*

Game Scenes Flight

visible
Whether the object is visible.

Attribute Can be read or written

Return type bool

Game Scenes Flight

remove()
Remove the object.

Game Scenes Flight

content
The text string

Attribute Can be read or written

Return type str

Game Scenes Flight

font
Name of the font

Attribute Can be read or written

Return type str

Game Scenes Flight

static available_fonts()
A list of all available fonts.

Return type list(str)

Game Scenes Flight

size
Font size.

Attribute Can be read or written

Return type int

Game Scenes Flight

character_size
Character size.

Attribute Can be read or written

Return type float

Game Scenes Flight

style
Font style.

Attribute Can be read or written

Return type *UI.FontStyle*

Game Scenes Flight

color

Set the color

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes Flight

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

Game Scenes Flight

alignment

Alignment.

Attribute Can be read or written

Return type *UI.TextAlignment*

Game Scenes Flight

line_spacing

Line spacing.

Attribute Can be read or written

Return type float

Game Scenes Flight

anchor

Anchor.

Attribute Can be read or written

Return type *UI.TextAnchor*

Game Scenes Flight

8.5 InfernalRobotics API

Provides RPCs to interact with the InfernalRobotics mod. Both the original mod and Infernal Robotics Next are supported. Provides the following classes:

8.5.1 InfernalRobotics

This service provides functionality to interact with Infernal Robotics.

available

Whether Infernal Robotics is installed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes Flight

ready

Whether Infernal Robotics API is ready.

Attribute Read-only, cannot be set

Return type bool

Game Scenes Flight

static servo_groups (*vessel*)

A list of all the servo groups in the given *vessel*.

Parameters **vessel** (`SpaceCenter.Vessel`) –

Return type list(*ServoGroup*)

Game Scenes Flight

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*

Game Scenes Flight

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*

Game Scenes Flight

8.5.2 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

Game Scenes Flight

forward_key
The key assigned to be the "forward" key for the group.

Attribute Can be read or written

Return type str

Game Scenes Flight

reverse_key
The key assigned to be the "reverse" key for the group.

Attribute Can be read or written

Return type str

Game Scenes Flight

speed
The speed multiplier for the group.

Attribute Can be read or written

Return type float

Game Scenes Flight

expanded
Whether the group is expanded in the InfernalRobotics UI.

Attribute Can be read or written

Return type bool

Game Scenes Flight

servos
The servos that are in the group.

Attribute Read-only, cannot be set

Return type list(*Servo*)

Game Scenes Flight

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*

Game Scenes Flight

parts
The parts containing the servos in the group.

Attribute Read-only, cannot be set

Return type list(*SpaceCenter.Part*)

Game Scenes Flight

move_right()

Moves all of the servos in the group to the right.

Game Scenes Flight

move_left()

Moves all of the servos in the group to the left.

Game Scenes Flight

move_center()

Moves all of the servos in the group to the center.

Game Scenes Flight

move_next_preset()

Moves all of the servos in the group to the next preset.

Game Scenes Flight

move_prev_preset()

Moves all of the servos in the group to the previous preset.

Game Scenes Flight

stop()

Stops the servos in the group.

Game Scenes Flight

8.5.3 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

Game Scenes Flight

part

The part containing the servo.

Attribute Read-only, cannot be set

Return type *SpaceCenter.Part*

Game Scenes Flight

highlight

Whether the servo should be highlighted in-game.

Attribute Write-only, cannot be read

Return type bool

Game Scenes Flight

`position`
The position of the servo.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`min_config_position`
The minimum position of the servo, specified by the part configuration.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`max_config_position`
The maximum position of the servo, specified by the part configuration.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`min_position`
The minimum position of the servo, specified by the in-game tweak menu.

Attribute Can be read or written

Return type float

Game Scenes Flight

`max_position`
The maximum position of the servo, specified by the in-game tweak menu.

Attribute Can be read or written

Return type float

Game Scenes Flight

`config_speed`
The speed multiplier of the servo, specified by the part configuration.

Attribute Read-only, cannot be set

Return type float

Game Scenes Flight

`speed`
The speed multiplier of the servo, specified by the in-game tweak menu.

Attribute Can be read or written

Return type float

Game Scenes Flight

current_speed
The current speed at which the servo is moving.

Attribute Can be read or written

Return type float

Game Scenes Flight

acceleration
The current speed multiplier set in the UI.

Attribute Can be read or written

Return type float

Game Scenes Flight

is_moving
Whether the servo is moving.

Attribute Read-only, cannot be set

Return type bool

Game Scenes Flight

is_free_moving
Whether the servo is freely moving.

Attribute Read-only, cannot be set

Return type bool

Game Scenes Flight

is_locked
Whether the servo is locked.

Attribute Can be read or written

Return type bool

Game Scenes Flight

is_axis_inverted
Whether the servos axis is inverted.

Attribute Can be read or written

Return type bool

Game Scenes Flight

move_right()
Moves the servo to the right.

Game Scenes Flight

move_left()
Moves the servo to the left.

Game Scenes Flight

move_center()
Moves the servo to the center.

Game Scenes Flight

move_next_preset()
Moves the servo to the next preset.

Game Scenes Flight

move_prev_preset()
Moves the servo to the previous preset.

Game Scenes Flight

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.

Game Scenes Flight

stop()
Stops the servo.

Game Scenes Flight

8.5.4 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()
```

8.6 Kerbal Alarm Clock API

Provides RPCs to interact with the Kerbal Alarm Clock mod. Provides the following classes:

8.6.1 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

Game Scenes All

alarms
A list of all the alarms.

Attribute Read-only, cannot be set

Return type list(*Alarm*)

Game Scenes All

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*

Game Scenes All

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*)

Game Scenes All

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** (*double*) – Time at which the new alarm should trigger.

Return type *Alarm*

Game Scenes All

8.6.2 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*

Game Scenes All

margin
The number of seconds before the event that the alarm will fire.

Attribute Can be read or written

Return type double

Game Scenes All

time
The time at which the alarm will fire.

Attribute Can be read or written

Return type double

Game Scenes All

type
The type of the alarm.

Attribute Read-only, cannot be set

Return type *AlarmType*

Game Scenes All

id
The unique identifier for the alarm.

Attribute Read-only, cannot be set

Return type str

Game Scenes All

name
The short name of the alarm.

Attribute Can be read or written

Return type str

Game Scenes All

notes
The long description of the alarm.

Attribute Can be read or written

Return type str

Game Scenes All

remaining
The number of seconds until the alarm will fire.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

repeat
Whether the alarm will be repeated after it has fired.

Attribute Can be read or written

Return type bool

Game Scenes All

repeat_period
The time delay to automatically create an alarm after it has fired.

Attribute Can be read or written

Return type double

Game Scenes All

vessel
The vessel that the alarm is attached to.

Attribute Can be read or written

Return type *SpaceCenter.Vessel*

Game Scenes All

xfer_origin_body
The celestial body the vessel is departing from.

Attribute Can be read or written

Return type *SpaceCenter.CelestialBody*

Game Scenes All

xfer_target_body
The celestial body the vessel is arriving at.

Attribute Can be read or written

Return type *SpaceCenter.CelestialBody*

Game Scenes All

remove()
Removes the alarm.

Game Scenes All

8.6.3 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*.

8.6.4 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.

8.6.5 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.

```python
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
```

8.7 RemoteTech API

Provides RPCs to interact with the RemoteTech mod. Provides the following classes:

8.7.1 RemoteTech

This service provides functionality to interact with RemoteTech.

available
Whether RemoteTech is installed.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

ground_stations
The names of the ground stations.

Attribute Read-only, cannot be set

Return type list(str)

Game Scenes All

static antenna (*part*)
Get the antenna object for a particular part.

Parameters **part** (SpaceCenter.Part) –

Return type *Antenna*

Game Scenes All

static comms (*vessel*)
Get a communications object, representing the communication capability of a particular vessel.

Parameters **vessel** (SpaceCenter.Vessel) –

Return type *Comms*

Game Scenes All

8.7.2 Comms

class Comms
Communications for a vessel.

vessel
Get the vessel.

Attribute Read-only, cannot be set

Return type *SpaceCenter.Vessel*

Game Scenes All

has_local_control
Whether the vessel can be controlled locally.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

has_flight_computer
Whether the vessel has a flight computer on board.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

has_connection
Whether the vessel has any connection.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

has_connection_to_ground_station
Whether the vessel has a connection to a ground station.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

signal_delay
The shortest signal delay to the vessel, in seconds.

Attribute Read-only, cannot be set

Return type double

Game Scenes All

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 double

Game Scenes All

signal_delay_to_vessel (*other*)
The signal delay between the this vessel and another vessel, in seconds.

Parameters **other** (`SpaceCenter.Vessel`) –

Return type double

Game Scenes All

antennas
The antennas for this vessel.

Attribute Read-only, cannot be set

Return type list(*Antenna*)

Game Scenes All

8.7.3 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*

Game Scenes All

has_connection

Whether the antenna has a connection.

Attribute Read-only, cannot be set

Return type bool

Game Scenes All

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*

Game Scenes All

target_body

The celestial body the antenna is targetting.

Attribute Can be read or written

Return type *SpaceCenter.CelestialBody*

Game Scenes All

target_ground_station

The ground station the antenna is targetting.

Attribute Can be read or written

Return type str

Game Scenes All

target_vessel

The vessel the antenna is targetting.

Attribute Can be read or written

Return type *SpaceCenter.Vessel*

Game Scenes All

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.

8.7.4 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)
```

8.8 User Interface API

8.8.1 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*

Game Scenes All

static add_canvas()
Add a new canvas.

Return type *Canvas*

Game Scenes All

Note: If you want to add UI elements to KSPs stock UI canvas, use *stock_canvas*.

static message(*content*[, *duration = 1.0*][, *position = 1*][, *color = (1.0, 0.92, 0.016)*][, *size = 20.0*])
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.
- **color** (*tuple*) – The color of the message.
- **size** (*float*) – Size of the message, differs per position.

Game Scenes All

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.

Game Scenes All

class MessagePosition

Message position.

top_left

Top left.

top_center

Top center.

top_right

Top right.

bottom_center

Bottom center.

8.8.2 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*

Game Scenes All

visible

Whether the UI object is visible.

Attribute Can be read or written

Return type bool

Game Scenes All

`add_panel` ([*visible = True*])
Create a new container for user interface elements.

Parameters **`visible`** (*`bool`*) – Whether the panel is visible.

Return type *`Panel`*

Game Scenes All

`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`*

Game Scenes All

`add_input_field` ([*visible = True*])
Add an input field to the canvas.

Parameters **`visible`** (*`bool`*) – Whether the input field is visible.

Return type *`InputField`*

Game Scenes All

`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`*

Game Scenes All

`remove` ()
Remove the UI object.

Game Scenes All

8.8.3 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`*

Game Scenes All

visible
Whether the UI object is visible.

Attribute Can be read or written

Return type bool

Game Scenes All

add_panel ([*visible = True*])
Create a panel within this panel.

Parameters **visible** (*bool*) – Whether the new panel is visible.

Return type *Panel*

Game Scenes All

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*

Game Scenes All

add_input_field ([*visible = True*])
Add an input field to the panel.

Parameters **visible** (*bool*) – Whether the input field is visible.

Return type *InputField*

Game Scenes All

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*

Game Scenes All

remove ()
Remove the UI object.

Game Scenes All

8.8.4 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*

Game Scenes All

visible
Whether the UI object is visible.

Attribute Can be read or written

Return type bool

Game Scenes All

content
The text string

Attribute Can be read or written

Return type str

Game Scenes All

font
Name of the font

Attribute Can be read or written

Return type str

Game Scenes All

available_fonts
A list of all available fonts.

Attribute Read-only, cannot be set

Return type list(str)

Game Scenes All

size
Font size.

Attribute Can be read or written

Return type int

Game Scenes All

style
Font style.

Attribute Can be read or written

Return type *FontStyle*

Game Scenes All

color
Set the color

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes All

alignment
Alignment.

Attribute Can be read or written

Return type *TextAnchor*

Game Scenes All

line_spacing
Line spacing.

Attribute Can be read or written

Return type float

Game Scenes All

remove()
Remove the UI object.

Game Scenes All

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.

8.8.5 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*

Game Scenes All

visible
Whether the UI object is visible.

Attribute Can be read or written

Return type bool

Game Scenes All

text
The text for the button.

Attribute Read-only, cannot be set

Return type *Text*

Game Scenes All

clicked
Whether the button has been clicked.

Attribute Can be read or written

Return type bool

Game Scenes All

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.

Game Scenes All

8.8.6 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*

Game Scenes All

visible

Whether the UI object is visible.

Attribute Can be read or written

Return type bool

Game Scenes All

value

The value of the input field.

Attribute Can be read or written

Return type str

Game Scenes All

text

The text component of the input field.

Attribute Read-only, cannot be set

Return type *Text*

Game Scenes All

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

Game Scenes All

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.

Game Scenes All

8.8.7 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(double, double)

Game Scenes All

local_position
Position of the rectangles pivot point relative to the anchors.

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes All

size
Width and height of the rectangle.

Attribute Can be read or written

Return type tuple(double, double)

Game Scenes All

upper_right
Position of the rectangles upper right corner relative to the anchors.

Attribute Can be read or written

Return type tuple(double, double)

Game Scenes All

lower_left
Position of the rectangles lower left corner relative to the anchors.

Attribute Can be read or written

Return type tuple(double, double)

Game Scenes All

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(double, double)

Game Scenes All

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(double, double)

Game Scenes All

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(double, double)

Game Scenes All

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(double, double)

Game Scenes All

rotation
Rotation, as a quaternion, of the object around its pivot point.

Attribute Can be read or written

Return type tuple(double, double, double, double)

Game Scenes All

scale
Scale factor applied to the object in the x, y and z dimensions.

Attribute Can be read or written

Return type tuple(double, double, double)

Game Scenes All

CHAPTER

NINE

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.

9.1 Clients

- Ruby client
- Haskell client
- Node.js client (requires the v0.4.0 pre-release version of kRPC)
- Using the plugin in F#

9.2 Services

- kRPC.MechJeb – remote procedures to interact with MechJeb (an up-to-date service based on krpcmj)

9.3 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.

CHAPTER

TEN

COMPILING KRPC

10.1 Getting the source code

First you need to download a copy of the source code, which is available from GitHub or using the following on the command line:

```
git clone https://github.com/krpc/krpc
```

10.2 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
```

10.3 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.

10.4 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`.

10.5 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.

10.5.1 Running the Tests

kRPC contains a suite of tests for the server plugin, services, client libraries and others.

To run the tests, the following dependencies should be installed. Without them, some of the tests will fail.

- Gendarme
- CppCheck
- socat

To install these dependencies via apt on Ubuntu run the following command:

```
sudo apt-get install gendarme cppcheck socat
```

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 krpctest package (built by target `//tools/krpctest`) 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 `krpctest`, launch a vessel and run various tests on it.

CHAPTER

ELEVEN

EXTENDING KRPC

11.1 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`.

11.2 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 {
```

(continues on next page)

(continued from previous page)

```
            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:

```python
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.

11.2.1 Attributes

The following C# attributes can be used to add functionality to the kRPC server.

KRPCService (String *Name*, *KRPC.Service.GameScene GameScene* = *KRPC.Service.GameScene.All*)

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. Defaults to all game scenes. If this is set to `KRPC.Service.GameScene.Inherit` the service will be available in all game scenes.

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 (Boolean *Nullable* = *false*, *KRPC.Service.GameScene GameScene* = *KRPC.Service.GameScene.Inherit*)

Parameters

- **Nullable** – Whether the return value of the procedure can be null. Defaults to false.
- **GameScene** – The game scenes in which the procedure is available. Defaults to inherit this setting from the service the procedure is defined in.

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 = "")
    {
```

(continues on next page)

(continued from previous page)

```
        ...
    }
}
```

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*, *KRPC.Service.GameScene GameScene = KRPC.Service.GameScene.Inherit*)

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.
- **GameScene** – The game scenes in which the class' methods and properties are available. Defaults to inherit this setting from the service the class is defined in.

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.

A class' methods and properties can be configured to be available in specific *game scenes* via the `GameScene` parameter on the class. If the `GameScene` parameter is not specified, the class' methods and properties are available in the same game scenes as the service the class is defined in. Individual class methods and properties can override this setting.

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 {
```

(continues on next page)

(continued from previous page)

```
    ...
}
```

KRPCMethod (Boolean *Nullable = false*, *KRPC.Service.GameScene GameScene = KRPC.Service.GameScene.Inherit*)

Parameters

- **Nullable** – Whether the return value of the procedure can be null. Defaults to false.
- **GameScene** – The game scenes in which the method is available. Defaults to inherit this setting from the class the method is defined in.

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 (Boolean *Nullable = false*, *KRPC.Service.GameScene GameScene = KRPC.Service.GameScene.Inherit*)

Parameters

- **Nullable** – Whether the return value of the procedure can be null. Defaults to false.
- **GameScene** – The game scenes in which the property is available. Defaults to inherit this setting from the class the property is defined in.

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 it's `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 it's `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")
{
    ...
}
```

11.2.2 Identifiers

An identifier can only contain letters and numbers, and must start with an upper case letter. They should follow CamelCase capitalization conventions.

11.2.3 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`

11.2.4 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;
}
```

11.2.5 Game Scenes

Anything that can be called is configured to be available from a particular game scene, or scenes. This includes service procedures and properties, and class methods and properties. This is controlled by the following enumeration:

enum KRPC.Service.GameScene

Inherit
Inherit the game scene from the containing service or class. This is the default.

All
All game scenes.

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.

MissionBuilder
The mission builder.

The special enumeration value *KRPC.Service.GameScene.Inherit* can be used to inherit the game scene setting from the containing service or class. Game scene inheritance works as follows:

- A service set to *KRPC.Service.GameScene.Inherit* will be available in all game scenes. This is equivalent to setting it to *KRPC.Service.GameScene.All*.
- A service procedure or property set to *KRPC.Service.GameScene.Inherit* will be available in the same game scenes as the containing service.
- A class set to *KRPC.Service.GameScene.Inherit* will be available in the same game scenes as the containing service.
- A class method or property set to *KRPC.Service.GameScene.Inherit* will be available in the same game scenes as the containing class. If the containing class is set to *KRPC.Service.GameScene.Inherit* the class method or property will be available in the same game scenes as the containing service.

Examples

- Declare a service whose procedures are available in the *KRPC.Service.GameScene.Flight* game scene:

```
[KRPCService (GameScene = GameScene.Flight)]
public static class MyService {
    ...
}
```

- Declare a service whose procedures are available in the *KRPC.Service.GameScene.Flight* and *KRPC.Service.GameScene.Editor* game scenes:

```
[KRPCService (GameScene = (GameScene.Flight | GameScene.Editor))]
public static class MyService {
    ...
}
```

- Declare a service whose procedures are available in the *KRPC.Service.GameScene.Flight* game scene by default, and declare a procedure which overrides this and is available in all game scenes:

```
[KRPCService (GameScene = GameScene.Flight)]
public static class MyService {
   ...

   [KRPCProcedure (GameScene = GameScene.All)]
   public static string MyProcedure() {
      ...
   }

   ...
}
```

11.3 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.

11.4 Further Examples

See the SpaceCenter service implementation for more extensive examples.

11.5 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)
```

(continues on next page)

(continued from previous page)

```
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 generting 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.

11.5.1 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.

CHAPTER

TWELVE

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*.

12.1 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.

12.1.1 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.

12.1.2 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).

12.1.3 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.

12.1.4 Invoking Remote Procedures

See *Messaging Protocol*.

12.1.5 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_pb2 as KRPC


def encode_varint(value):
    """ Encode an int as a protobuf varint """
    data = []
    _VarintEncoder()(data.append, value, False)
    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
```

(continues on next page)

(continued from previous page)

```
        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.results[0].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:

```python
import binascii
import socket
from google.protobuf.internal.encoder import _VarintEncoder
from google.protobuf.internal.decoder import _DecodeVarint
from krpc.schema import KRPC_pb2 as KRPC


def encode_varint(value):
    """ Encode an int as a protobuf varint """
    data = []
    _VarintEncoder()(data.append, value, False)
    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)
```

(continues on next page)

(continued from previous page)

```
# 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 idenfitier = %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.results[0].error))
```

(continues on next page)

(continued from previous page)

```
# 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)
```

12.2 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.

12.2.1 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`

12.2.2 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=Eeh8Vbj2DkWTcJZTkYlEhQ==`

Connecting to the stream server is optional. If the client doesn't require stream functionality, there is no need to connect.

12.2.3 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.

12.2.4 Invoking Remote Procedures

See *Messaging Protocol*.

12.2.5 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');
```

(continues on next page)

(continued from previous page)

```
  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);
  };
};
```

12.3 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.

12.3.1 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 it's 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;
  uint32 serviceId = 4;
  uint32 procedureId = 5;
  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.
- `service_id` - The integer identifier of the service in which the remote procedure is defined.
- `procedure_id` - The integer identifier 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 service containing the procedure to call is specified by setting either `service` or `service_id`. The procedure to call is specified by setting either `procedure` or `procedure_id`. Use of `service` and `procedure` (i.e. descriptive strings) should be preferred as these will not change between server versions. For clients where code size or communication overhead must be kept to an absolute minimum, `service_id` and `procedure_id` can be used. However, note that the identifiers may change between server versions.

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
```

(continues on next page)

(continued from previous page)

```
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.results[0].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

12.3.2 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. A client can also control the rate of the stream, by specifying a target number of Hertz. The server computes a time delay from the target rate, and only updates the stream if at least that time has passed since the last time the stream was updated.

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.

12.3.3 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;
}
```

12.3.4 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;
```

(continues on next page)

(continued from previous page)

```
  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.

12.3.5 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;
```

(continues on next page)

(continued from previous page)

```
  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;
  repeated GameScene game_scenes = 5;
  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.
- game_scenes - The *game scenes* that the procedure is available in. If this repeated field is empty, the procedure is available in all game scenes.
- 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.

12.3.6 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.

12.3.7 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;
```

(continues on next page)

(continued from previous page)

```
    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.

12.3.8 GameScene

The GetServices procedure returns information about which game scenes a procedure can be called from. These are returned as a repeated field containing GameScene enumeration values. If the procedure is available in all game scenes, this field is empty. The enumeration is defined as follows:

```
enum GameScene {
  SPACE_CENTER = 1;
  FLIGHT = 2;
  TRACKING_STATION = 3;
  EDITOR_VAB = 4;
  EDITOR_SPH = 5;
  MISSION_BUILDER = 6;
};
```

12.3.9 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.

CHAPTER

THIRTEEN

INTERNALS OF KRPC

13.1 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 games 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-

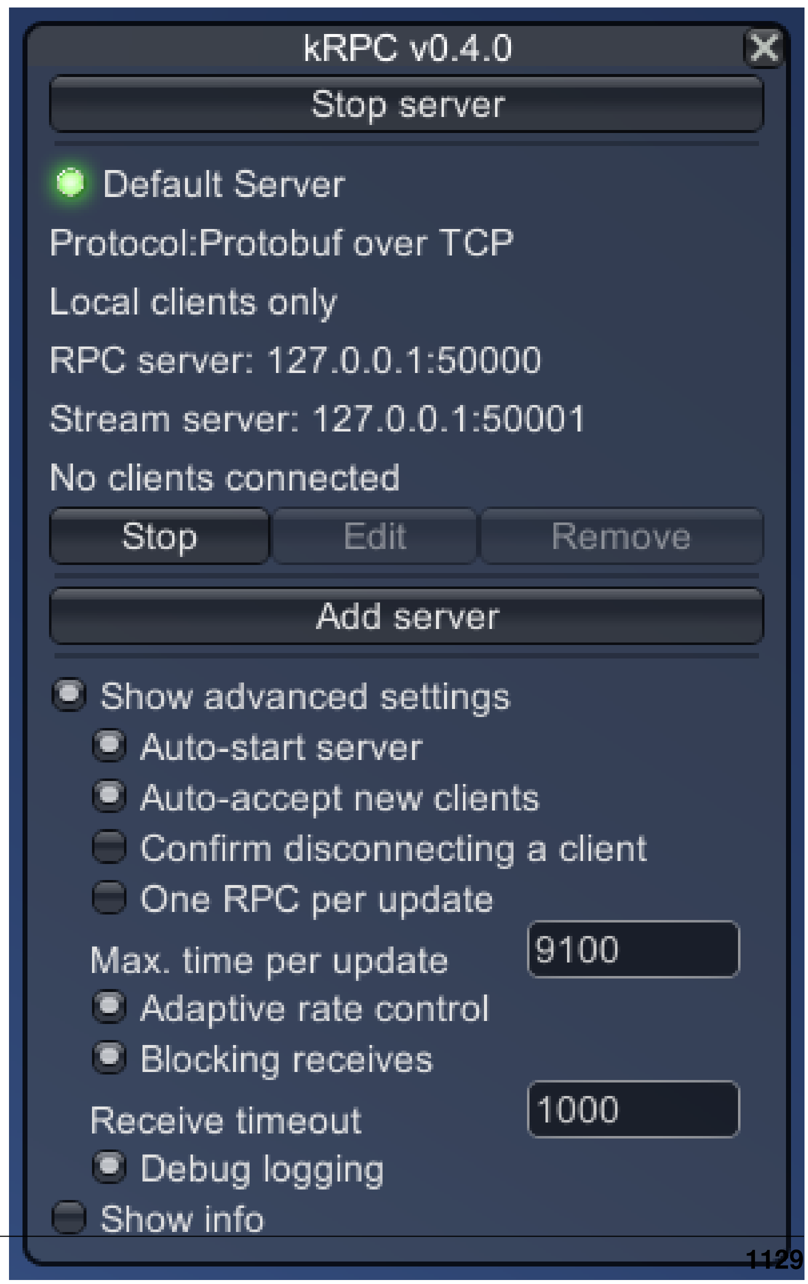


Fig. 1: Server window showing the advanced settings.

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

LUA MODULE INDEX