
kRPC

Release 0.2.3

April 06, 2016

CONTENTS

1	Getting Started	3
1.1	The Server Plugin	3
1.2	The Python Client	4
1.3	'Hello World' Script	5
1.4	Going further...	6
2	Tutorials and Examples	7
2.1	Sub-Orbital Flight	7
2.2	Reference Frames	10
2.3	Launch into Orbit	18
2.4	Pitch, Heading and Roll	21
2.5	Interacting with Parts	22
2.6	Docking Guidance	23
3	C#	27
3.1	C# Client	27
3.2	KRPC API	29
3.3	SpaceCenter API	30
3.4	InfernalRobotics API	83
3.5	Kerbal Alarm Clock API	87
4	C++	91
4.1	C++ Client	91
4.2	KRPC API	95
4.3	SpaceCenter API	96
4.4	InfernalRobotics API	151
4.5	Kerbal Alarm Clock API	156
5	Java	161
5.1	Java Client	161
5.2	KRPC API	164
5.3	SpaceCenter API	165
5.4	InfernalRobotics API	221
5.5	Kerbal Alarm Clock API	225
6	Lua	231
6.1	Lua Client	231
6.2	KRPC API	233
6.3	SpaceCenter API	234
6.4	InfernalRobotics API	305
6.5	Kerbal Alarm Clock API	310

7	Python	315
7.1	Python Client	315
7.2	KRPC API	318
7.3	SpaceCenter API	319
7.4	InfernalRobotics API	390
7.5	Kerbal Alarm Clock API	395
8	Other Clients, Services and Scripts	401
8.1	Clients	401
8.2	Services	401
8.3	Scripts/Tools/Libraries etc.	401
9	Compiling kRPC	403
9.1	Install Dependencies	403
9.2	Setup your Environment	403
9.3	Building using Bazel	403
9.4	Building the C# projects using an IDE	404
10	Extending kRPC	407
10.1	The kRPC Architecture	407
10.2	Service API	407
10.3	Documentation	413
10.4	Further Examples	413
10.5	Generating Service Code for Static Clients	414
11	Communication Protocol	417
11.1	Establishing a Connection	417
11.2	Remote Procedures	418
11.3	Protocol Buffer Encoding	421
11.4	Streams	421
11.5	KRPC Service	422
11.6	Service Description Message	423
12	Internals of kRPC	427
12.1	Server Performance Settings	427
	Python Module Index	429
	Lua Module Index	431
	Index	433

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](#).

- [Getting Started Guide](#)
- [Tutorials and Examples](#)
- [Clients, services and tools made by others](#)

The mod exposes most of KSP's API and includes support for Kerbal Alarm Clock and Infernal Robotics. This functionality is provided to client programs via a Remote Procedure Call server, using protocol buffers for serialization. The server component sets up a TCP/IP server that remote scripts can connect to. This communication could be on the local machine only, over a local network, or even over the wider internet if configured correctly. The server is also extensible. Additional remote procedures (grouped into “services”) can be added to the server using the “Service API”.

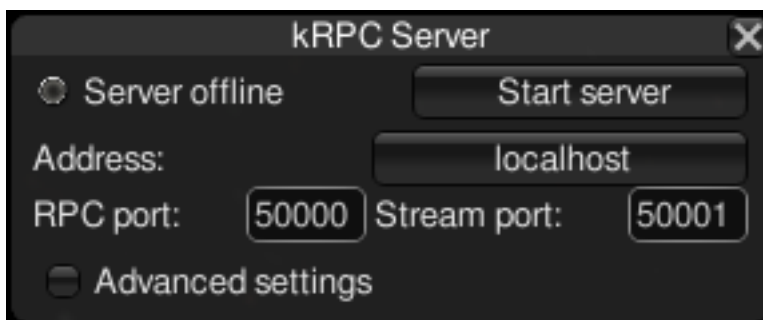
GETTING STARTED

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

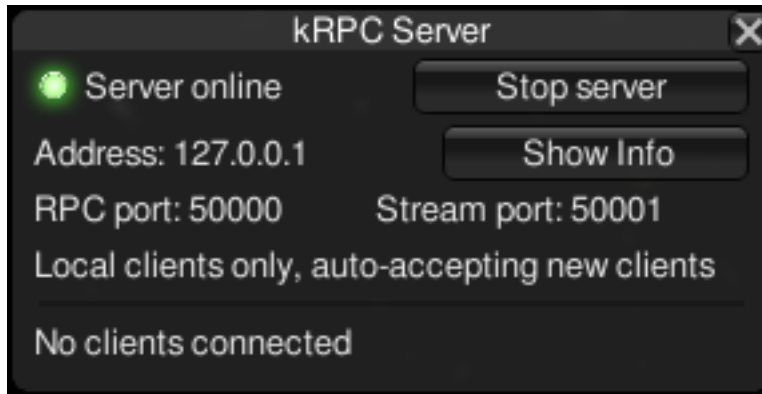
1.1 The Server Plugin

1.1.1 Installation

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



4. Click "Start server" to, erm... start the server! If all goes well, the light should turn a happy green color:



5. You can hide the window by clicking the close button in the top right. The window can also be shown/hidden by clicking on the icon in the top right:



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

1.1.2 Configuration

The server is configured using the window displayed in-game:

1. **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.
2. **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 the 'Advanced settings' box:

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 – as long as it can do network I/O.

TUTORIALS AND EXAMPLES

This collection of tutorials and example scripts explain how to use the features of kRPC. They are written for the Python client, although the concepts apply to all of the client languages.

2.1 Sub-Orbital Flight

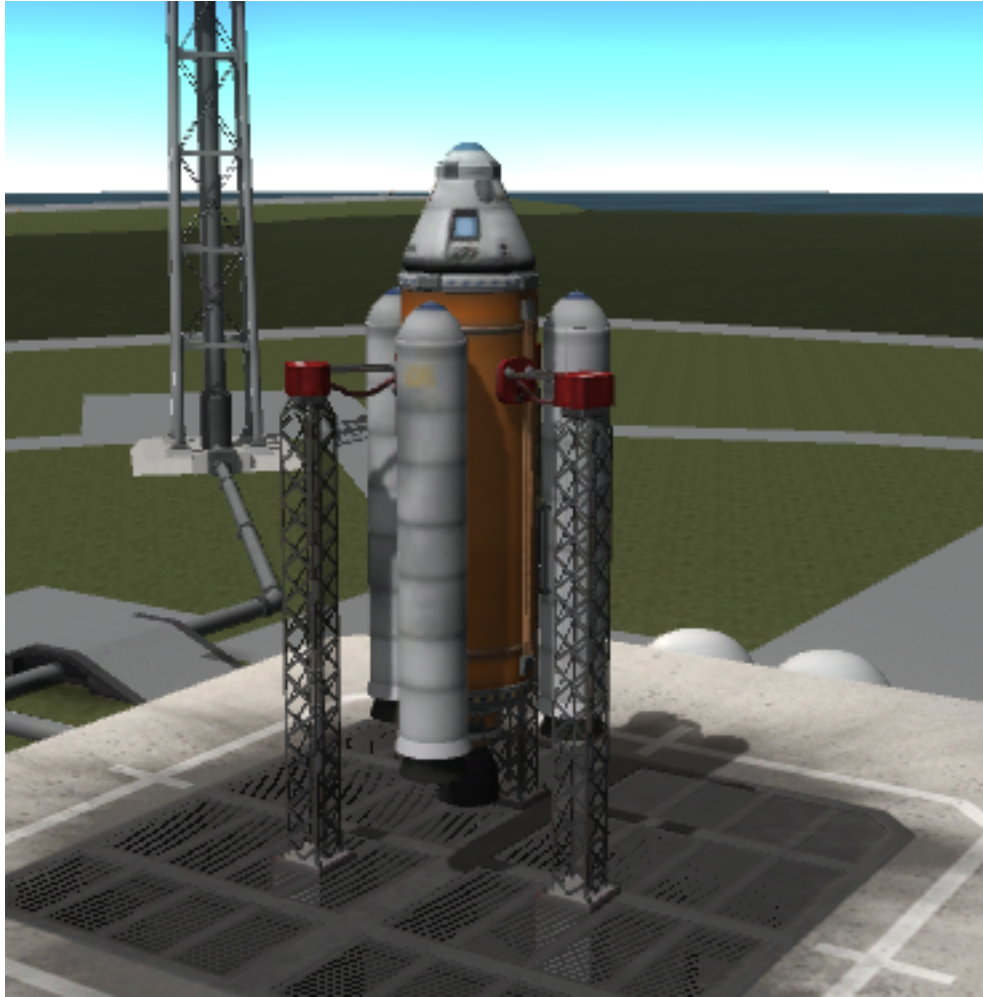
This introductory tutorial uses kRPC to send some Kerbals on a sub-orbital flight, and (hopefully) returns them safely back to Kerbin. It covers the following topics:

- Controlling a rocket (activating stages, setting the throttle)
- Using the auto pilot to point the vessel in a specific direction
- 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.

2.1.1 Part One: Preparing for Launch

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



The first thing we need to do is load the python client module and 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:

```
import krpc
conn = krpc.connect(name='Sub-orbital flight script')
```

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

```
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.

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

```
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 a while loop that repeatedly checks how much solid fuel there is left in the rocket. When the loop exits, we will activate the next stage to jettison the boosters:

```
while vessel.resources.amount('SolidFuel') > 0.1:
    time.sleep(1)
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.

2.1.3 Part Three: Reaching Apoapsis

Next we will execute a gravity turn when the rocket reaches a sufficiently high altitude. The following loop repeatedly checks the altitude and exits when the rocket reaches 10km:

```
while vessel.flight().mean_altitude < 10000:
    time.sleep(1)
```

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:

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

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

```
while vessel.orbit.apoapsis_altitude < 100000:
    time.sleep(1)
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.

```
while vessel.flight().surface_altitude > 1000:
    time.sleep(1)
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:

```
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

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

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)
- and 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:

```
1 import krpc
2 conn = krpc.connect()
3 vessel = conn.space_center.active_vessel
4 print(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

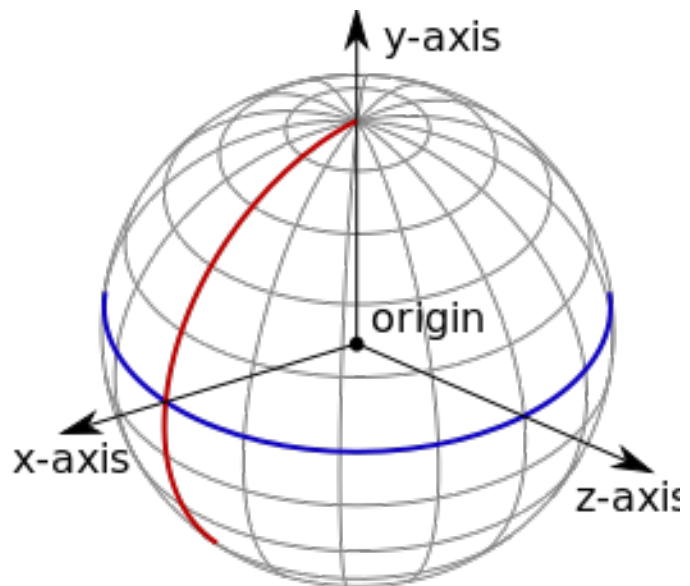


Fig. 2.1: The reference frame for a celestial body, such as Kerbin. The equator is shown in blue, and the prime meridian in red. The black arrows show the coordinate axes, and the origin is at the center of the planet.

Another example of the orbital reference frame for a vessel, obtained by calling

is or-

a

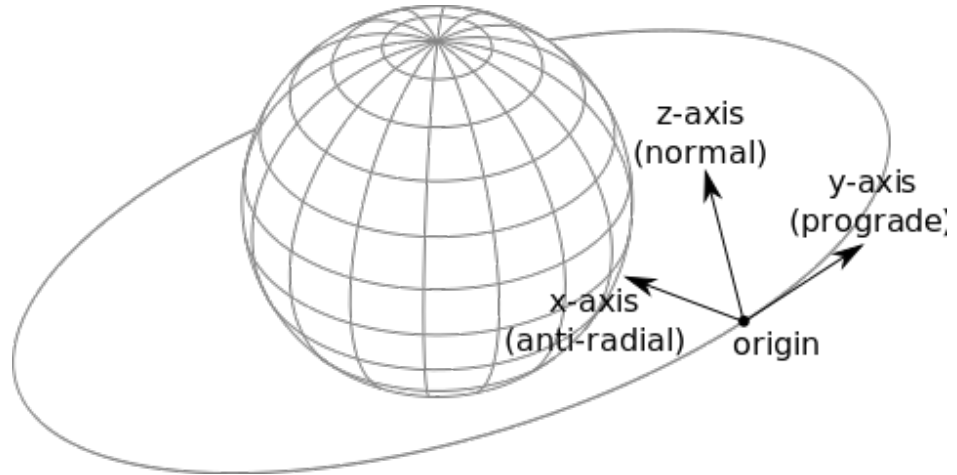


Fig. 2.2: The orbital reference frame for a vessel.

`Vessel.orbital_reference_frame`. This is fixed to the vessel (the origin moves with the vessel) and is orientated so that the axes point in the orbital prograde/normal/radial directions.

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

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:

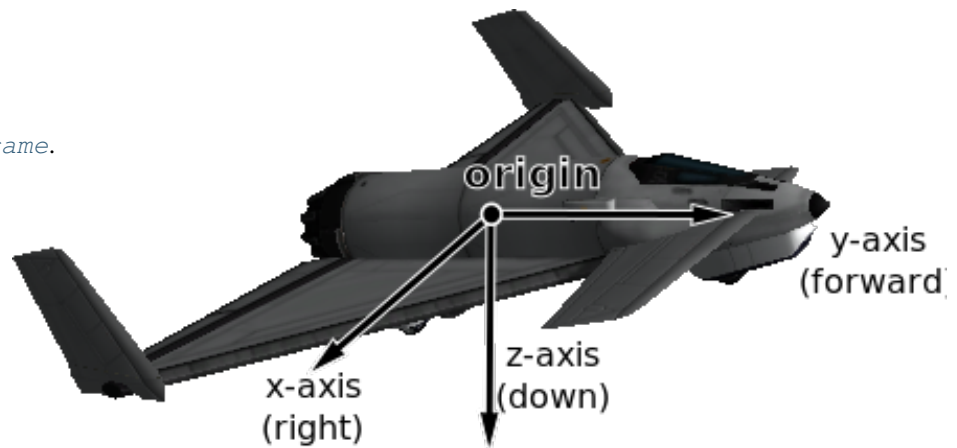


Fig. 2.3: The reference frame for an aircraft.

- 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.

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:

- `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`
- `DockingPort.reference_frame`

2.2.3 Converting Between Reference Frames

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

- `SpaceCenter.transform_position()`
- `SpaceCenter.transform_direction()`
- `SpaceCenter.transform_rotation()`

- `SpaceCenter.transform_velocity()`

2.2.4 Visual Debugging

Reference frames can be confusing, and choosing the correct one is a challenge in itself. To aid debugging, kRPC provides some methods with which you can draw direction vectors in-game.

`SpaceCenter.draw_direction()` will draw a direction vector, starting from the center of mass of the active vessel. For example, the following code draws the direction of the current vessel's velocity relative to the surface:

```
1 import krpc
2 conn = krpc.connect(name='Visual Debugging')
3 vessel = conn.space_center.active_vessel
4
5 ref_frame = vessel.orbit.body.reference_frame
6 velocity = vessel.flight(ref_frame).velocity
7 conn.space_center.draw_direction(velocity, ref_frame, (1,0,0))
8
9 while True:
10     pass
```

Note: The client must remain connected, otherwise kRPC will stop drawing the directions, hence the while loop at the end of this example.

2.2.5 Examples

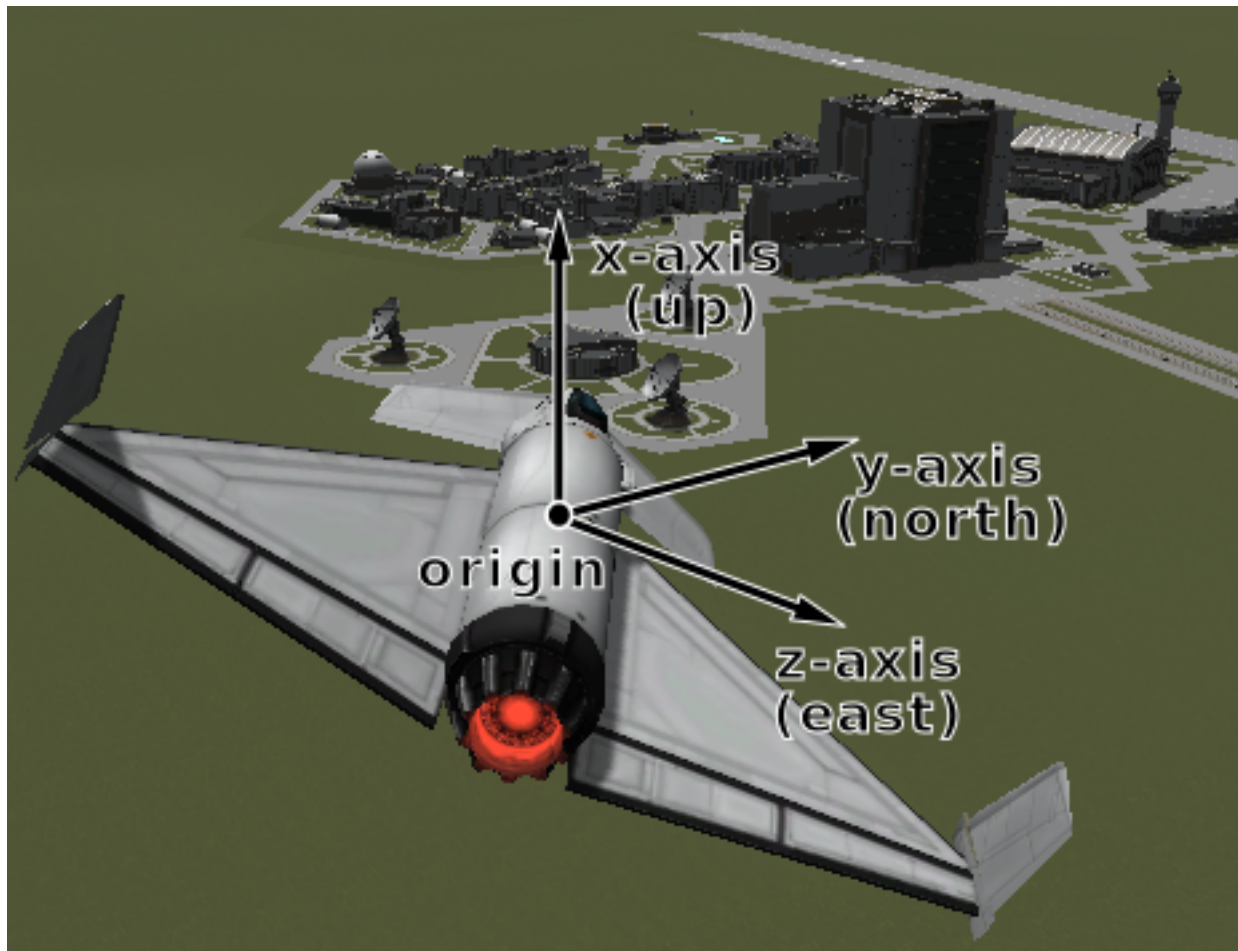
The following examples demonstrate various uses of reference frames.

Navball directions

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

```
1 import krpc
2 conn = krpc.connect(name='Navball directions')
3 vessel = conn.space_center.active_vessel
4 ap = vessel.auto_pilot
5 ap.reference_frame = vessel.surface_reference_frame
6 ap.engage()
7
8 # Point the vessel north on the navball, with a pitch of 0 degrees
9 ap.target_direction = (0,1,0)
10 ap.wait()
11
12 # Point the vessel vertically upwards on the navball
13 ap.target_direction = (1,0,0)
14 ap.wait()
15
16 # Point the vessel west (heading of 270 degrees), with a pitch of 0 degrees
17 ap.target_direction = (0,0,-1)
18 ap.wait()
19
20 ap.disengage()
```

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



Line 9 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.

Line 13 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.

Line 17 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`.

```

1 import krpc
2 conn = krpc.connect(name='Orbital directions')
3 vessel = conn.space_center.active_vessel
4 ap = vessel.auto_pilot
5 ap.reference_frame = vessel.orbital_reference_frame
6 ap.engage()
7
8 # Point the vessel in the prograde direction
9 ap.target_direction = (0,1,0)

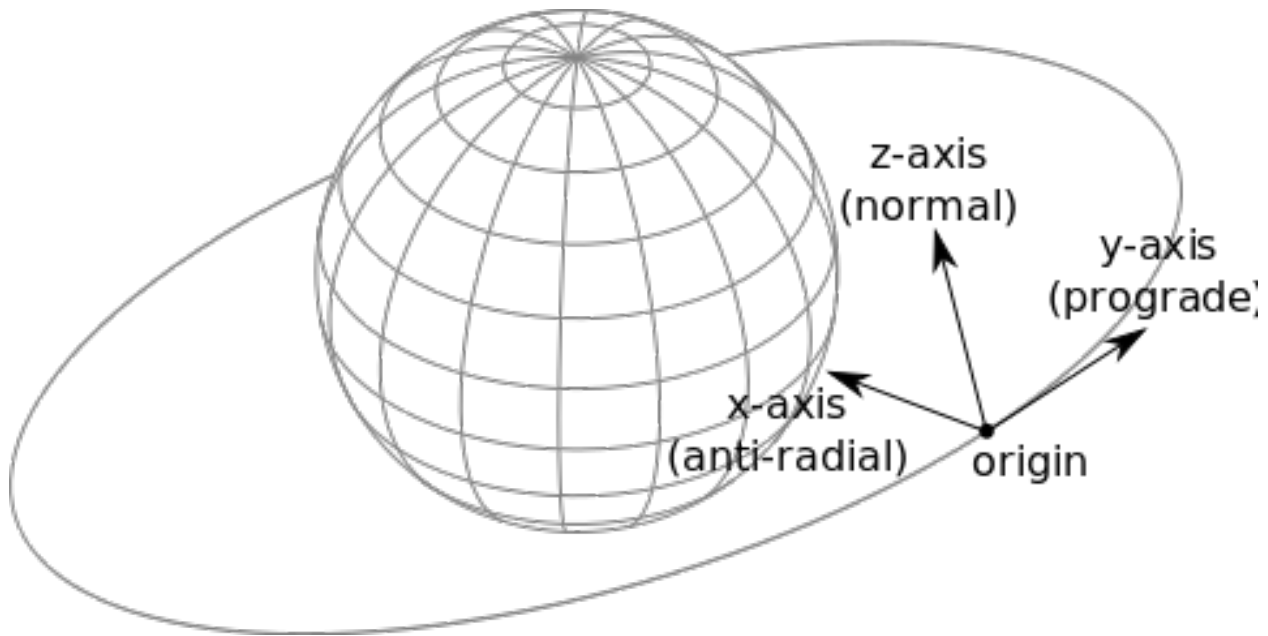
```

```

10 ap.wait()
11
12 # Point the vessel in the orbit normal direction
13 ap.target_direction = (0,0,1)
14 ap.wait()
15
16 # Point the vessel in the orbit radial direction
17 ap.target_direction = (-1,0,0)
18 ap.wait()
19
20 ap.disengage()

```

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



Surface 'prograde'

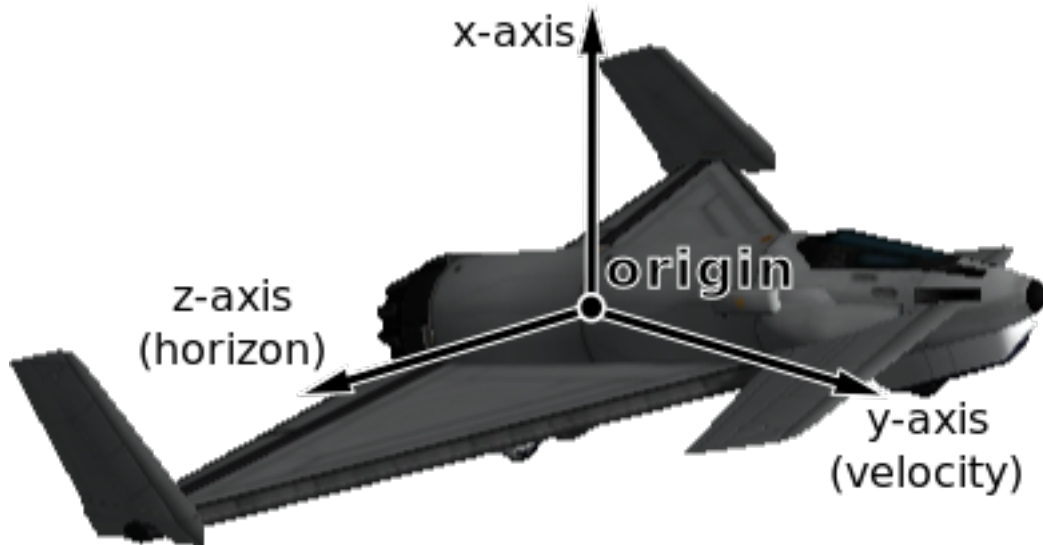
This example demonstrates how to point the vessel in the 'prograde' direction on the navball, when in 'surface' mode. This is the direction of the vessels velocity relative to the surface:

```

1 import krpc
2 conn = krpc.connect(name='Surface prograde')
3 vessel = conn.space_center.active_vessel
4 ap = vessel.auto_pilot
5
6 ap.reference_frame = vessel.surface_velocity_reference_frame
7 ap.target_direction = (0,1,0)
8 ap.engage()
9 ap.wait()
10 ap.disengage()

```

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



Orbital speed

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:

```

1 import krpc, time
2 conn = krpc.connect(name='Orbital speed')
3 vessel = conn.space_center.active_vessel
4
5 while True:
6
7     velocity = vessel.flight(vessel.orbit.body.non_rotating_reference_frame).velocity
8     print('Orbital velocity = (%.1f, %.1f, %.1f)' % velocity)
9
10    speed = vessel.flight(vessel.orbit.body.non_rotating_reference_frame).speed
11    print('Orbital speed = %.1f m/s' % speed)
12
13    time.sleep(1)

```

Surface speed

To compute the speed of a vessel relative to the surface of a planet/moon, you need to get the velocity relative to the planet's reference frame (`CelestialBody.reference_frame`). This reference frame rotates with the body, therefore the rotational velocity of the body is taken into account when computing the velocity of the vessel:

```

1 import krpc, time
2 conn = krpc.connect(name='Surface speed')
3 vessel = conn.space_center.active_vessel
4
5 while True:
6
7     velocity = vessel.flight(vessel.orbit.body.reference_frame).velocity
8     print('Surface velocity = (%.1f, %.1f, %.1f)' % velocity)
9
10    speed = vessel.flight(vessel.orbit.body.reference_frame).speed

```

```
11 print('Surface speed = %.1f m/s' % speed)
12
13 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):

```
1 import krpc, math, time
2 conn = krpc.connect(name='Angle of attack')
3 vessel = conn.space_center.active_vessel
4
5 while True:
6
7     d = vessel.direction(vessel.orbit.body.reference_frame)
8     v = vessel.velocity(vessel.orbit.body.reference_frame)
9
10    # Compute the dot product of d and v
11    dotprod = d[0]*v[0] + d[1]*v[1] + d[2]*v[2]
12
13    # Compute the magnitude of v
14    vmag = math.sqrt(v[0]**2 + v[1]**2 + v[2]**2)
15    # Note: don't need to magnitude of d as it is a unit vector
16
17    # Compute the angle between the vectors
18    if dotprod == 0:
19        angle = 0
20    else:
21        angle = abs(math.acos (dotprod / vmag) * (180. / math.pi))
22
23    print('Angle of attack = %.1f' % angle)
24
25    time.sleep(1)
```

Note that the orientation of the reference frame used to get the direction and velocity vectors (on lines 7 and 8) 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, line 8 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.

2.3 Launch into Orbit

This tutorial launches a two-stage rocket into a 150km circular orbit. The craft file for the rocket can be downloaded [here](#) and the entire python script from [here](#).

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.

```
import krpc, time, math

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')
periapsis = conn.add_stream(getattr, vessel.orbit, 'periapsis_altitude')
eccentricity = conn.add_stream(getattr, vessel.orbit, 'eccentricity')
stage_2_resources = vessel.resources_in_decouple_stage(stage=2, cumulative=False)
stage_3_resources = vessel.resources_in_decouple_stage(stage=3, cumulative=False)
srb_fuel = conn.add_stream(stage_3_resources.amount, 'SolidFuel')
launcher_fuel = conn.add_stream(stage_2_resources.amount, 'LiquidFuel')

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

# 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.

```

# 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()
            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.

```
# 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

# 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.

```
# 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.

```
# 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).

```
# 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
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
node.remove()

print('Launch complete')

```

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

2.4 Pitch, Heading and Roll

The following example calculates the pitch, heading and rolls angles of the active vessel once per second:

```

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

def cross_product(x, y):
    return (x[1]*y[2] - x[2]*y[1], x[2]*y[0] - x[0]*y[2], x[0]*y[1] - x[1]*y[0])

def dot_product(x, y):
    return x[0]*y[0] + x[1]*y[1] + x[2]*y[2]

def magnitude(x):
    return math.sqrt(x[0]**2 + x[1]**2 + x[2]**2)

def angle_between_vectors(x, y):
    """ Compute the angle between vector x and y """
    dp = dot_product(x, y)
    if dp == 0:
        return 0
    xm = magnitude(x)
    ym = magnitude(y)
    return math.acos(dp / (xm*ym)) * (180. / math.pi)

def angle_between_vector_and_plane(x, n):
    """ Compute the angle between a vector x and plane with normal vector n """
    dp = dot_product(x, n)
    if dp == 0:
        return 0
    xm = magnitude(x)
    nm = magnitude(n)
    return math.asin(dp / (xm*nm)) * (180. / math.pi)

while True:

    vessel_direction = vessel.direction(vessel.surface_reference_frame)

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

```

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

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

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

- *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:

```
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:

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

ports = vessel.parts.docking_ports
port = list(filter(lambda p: p.part.title == 'Clamp-O-Tron Docking Port', ports))[0]
part = port.part
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: http://wiki.kerbalspaceprogram.com/wiki/Specific_impulse#Multiple_engines

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

active_engines = filter(lambda e: e.active and e.has_fuel, vessel.parts.engines)

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

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

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

2.6 Docking Guidance

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

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

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

# Set up curses
stdscr = curses.initscr()
curses.nocbreak()
```

```
stdscr.keypad(1)
curses.noecho()

try:

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

    while True:

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

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

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

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

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

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

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

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

```

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

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

stdscr.refresh()
time.sleep(0.25)

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

```


3.1 C# Client

This client provides functionality to interact with a kRPC server from programs written in C#. The `KRPC.Client.dll` assembly can be [installed using NuGet](#) or [downloaded from GitHub](#).

3.1.1 Installing the Library

Install the client using [NuGet](#) or download the assembly from [GitHub](#) and reference it in your project. You also need to install [Google.Protobuf](#) using [NuGet](#).

Note: The copy of `Google.Protobuf.dll` in the GameData folder shipped with the kRPC server plugin should be *avoided*. It is a modified version to work within KSP. [See here for more details](#).

3.1.2 Connecting to the Server

To connect to a server, create a `Connection` object. For example to connect to a server running on the local machine:

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

class Program {
    public static void Main () {
        var connection = new Connection (name : "Example");
        var krpc = connection.KRPC ();
        System.Console.WriteLine (krpc.GetStatus ().Version);
    }
}
```

The class constructor also accepts arguments that specify what address and port numbers to connect to. For example:

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

class Program {
    public static void Main () {
        var connection = new Connection (
            name : "Example", address: IPAddress.Parse("10.0.2.2"), rpcPort: 1000, streamPort: 1001);
        var krpc = connection.KRPC ();
    }
}
```

```
        System.Console.WriteLine (krpc.GetStatus ().Version);  
    }  
}
```

3.1.3 Interacting with the Server

kRPC groups remote procedures into services. The functionality for the services are defined in namespace `KRPC.Client.Services.*`.

To interact with a service, you must first instantiate it. The following example connects to the server, instantiates the `SpaceCenter` service, and outputs the name of the active vessel:

```
using KRPC.Client;  
using KRPC.Client.Services.SpaceCenter;  
  
class Program {  
    public static void Main () {  
        var connection = new Connection (name : "Vessel Name");  
        var spaceCenter = connection.SpaceCenter ();  
        var vessel = spaceCenter.ActiveVessel;  
        System.Console.WriteLine (vessel.Name);  
    }  
}
```

3.1.4 Streaming Data from the Server

A stream repeatedly executes a function on the server, with a fixed set of argument values. It provides a more efficient way of repeatedly getting the result of a function, avoiding the network overhead of having to invoke it directly.

For example, consider the following loop that continuously prints out the position of the active vessel. This loop incurs significant communication overheads, as the `Vessel.Position` method is called repeatedly.

```
using KRPC.Client;  
using KRPC.Client.Services.SpaceCenter;  
using System;  
  
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.Out.WriteLine(vessel.Position(refframe));  
    }  
}
```

The following code achieves the same thing, but is far more efficient. It calls `Connection.AddStream` once at the start of the program to create a stream, and then repeatedly gets the position from the stream.

```
using KRPC.Client;  
using KRPC.Client.Services.SpaceCenter;  
using System;  
  
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 position = connection.AddStream(() => vessel.Position(refframe));
while (true)
    Console.Out.WriteLine(position.Get());
}

```

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

3.1.5 Client API Reference

class Connection

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

Connection (string name = "", IPAddress address = null, int rpcPort = 50000, int streamPort = 0)

Connect to a kRPC server on the specified IP address and port numbers. If streamPort is 0, does not connect to the stream server. Passes an optional name to the server to identify the client (up to 32 bytes of UTF-8 encoded text).

Stream<ReturnType> **AddStream**<ReturnType> (LambdaExpression expression)

Create a new stream from the given lambda expression. Returns a stream object that can be used to obtain the latest value of the stream.

class Stream<ReturnType>

Object representing a stream.

ReturnType **Get** ()

Get the most recent value of the stream.

void **Remove** ()

Remove the stream from the server.

3.2 KRPC API

class KRPC

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

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

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

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

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

GameScene **CurrentGameScene** { get; }

Get the current game scene.

uint **AddStream** (KRPC.Schema.KRPC.Request request)

Add a streaming request and return its identifier.

Parameters

Note: Do not call this method from client code. Use *streams* provided by the C# client library.

void **RemoveStream** (uint *id*)
 Remove a streaming request.

Parameters

Note: Do not call this method from client code. Use *streams* provided by the C# client library.

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.

3.3 SpaceCenter API

3.3.1 SpaceCenter

class SpaceCenter

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

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

The currently active vessel.

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

A list of all the vessels in the game.

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

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

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

The currently targeted celestial body.

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

The currently targeted vessel.

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

The currently targeted docking port.

void **ClearTarget** ()
Clears the current target.

void **LaunchVesselFromVAB** (string name)
Launch a new vessel from the VAB onto the launchpad.

Parameters

- **name** – Name of the vessel’s craft file.

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

Parameters

- **name** – Name of the vessel’s craft file.

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

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

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.

float **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.

float **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*.

int **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.

int **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.

bool **CanRailsWarpAt** (int 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.

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

void **WarpTo** (double UT, float maxRailsRate = 100000.0, float maxPhysicsRate = 2.0)
Uses time acceleration to warp forward to a time in the future, specified by universal time *UT*. This call

blocks until the desired time is reached. Uses regular “on-rails” or physical time warp as appropriate. For example, physical time warp is used when the active vessel is traveling through an atmosphere. When using regular “on-rails” time warp, the warp rate is limited by *maxRailsRate*, and when using physical time warp, the warp rate is limited by *maxPhysicsRate*.

Parameters

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

Returns When the time warp is complete.

`Tuple<double, double, double> TransformPosition (Tuple<double, double, double> position, ReferenceFrame from, ReferenceFrame to)`

Converts a position vector from one reference frame to another.

Parameters

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

Returns The corresponding position vector in reference frame *to*.

`Tuple<double, double, double> TransformDirection (Tuple<double, double, double> direction, ReferenceFrame from, ReferenceFrame to)`

Converts a direction vector from one reference frame to another.

Parameters

- **direction** – Direction vector in reference frame *from*.
- **from** – The reference frame that the direction vector is in.
- **to** – The reference frame to convert the direction vector to.

Returns The corresponding direction vector in reference frame *to*.

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

Converts a rotation from one reference frame to another.

Parameters

- **rotation** – Rotation in reference frame *from*.
- **from** – The reference frame that the rotation is in.
- **to** – The corresponding rotation in reference frame *to*.

Returns The corresponding rotation in reference frame *to*.

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

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

Parameters

- **position** – Position vector in reference frame *from*.

- **velocity** – Velocity vector in reference frame *from*.
- **from** – The reference frame that the position and velocity vectors are in.
- **to** – The reference frame to convert the velocity vector to.

Returns The corresponding velocity in reference frame *to*.

bool FARAvailable { get; }

Whether *Ferram Aerospace Research* is installed.

bool RemoteTechAvailable { get; }

Whether *RemoteTech* is installed.

void DrawDirection (*Tuple<double, double, double> direction, ReferenceFrame referenceFrame, Tuple<double, double, double> color, float length = 10.0*)

Draw a direction vector on the active vessel.

Parameters

- **direction** – Direction to draw the line in.
- **referenceFrame** – Reference frame that the direction is in.
- **color** – The color to use for the line, as an RGB color.
- **length** – The length of the line. Defaults to 10.

void DrawLine (*Tuple<double, double, double> start, Tuple<double, double, double> end, ReferenceFrame referenceFrame, Tuple<double, double, double> color*)

Draw a line.

Parameters

- **start** – Position of the start of the line.
- **end** – Position of the end of the line.
- **referenceFrame** – Reference frame that the position are in.
- **color** – The color to use for the line, as an RGB color.

void ClearDrawing ()

Remove all directions and lines currently being drawn.

enum WarpMode

Returned by *WarpMode*

Rails

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

Physics

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

None

Time warp is not active.

3.3.2 Vessel

class Vessel

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources.

string Name { get; set; }

The name of the vessel.

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

The type of the vessel.

VesselSituation **Situation** { **get**; }

The situation the vessel is in.

double **MET** { **get**; }

The mission elapsed time in seconds.

Flight **Flight** (*ReferenceFrame* *referenceFrame* = *None*)

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

Parameters

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

Note: When this is called with no arguments, the vessel's surface reference frame is used. This reference frame moves with the vessel, therefore velocities and speeds returned by the flight object will be zero. See the *reference frames tutorial* for examples of getting the *orbital speed* and *surface speed* of a vessel.

Vessel **Target** { **get**; **set**; }

The target vessel. `null` if there is no target. When setting the target, the target cannot be the current vessel.

Orbit **Orbit** { **get**; }

The current orbit of the vessel.

Control **Control** { **get**; }

Returns a *Control* object that can be used to manipulate the vessel's control inputs. For example, its pitch/yaw/roll controls, RCS and thrust.

AutoPilot **AutoPilot** { **get**; }

An *AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

Resources **Resources** { **get**; }

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

Resources **ResourcesInDecoupleStage** (*int* *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.

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.

Comms **Comms** { **get**; }

A *Comms* object, that can used to interact with RemoteTech for this vessel.

Note: Requires [RemoteTech](#) to be installed.

float **Mass** { **get**; }

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

float **DryMass** { **get**; }

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

float **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.

float **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.

float **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.

float **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.

float **SpecificImpulse** { **get**; }

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

float **VacuumSpecificImpulse** { **get**; }

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

float **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](#).

ReferenceFrame **ReferenceFrame** { **get**; }

The reference frame that is fixed relative to the vessel, and orientated with the vessel.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel.
- The x-axis points out to the right of the vessel.
- The y-axis points in the forward direction of the vessel.
- The z-axis points out of the bottom off the vessel.

ReferenceFrame **OrbitalReferenceFrame** { **get**; }

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.

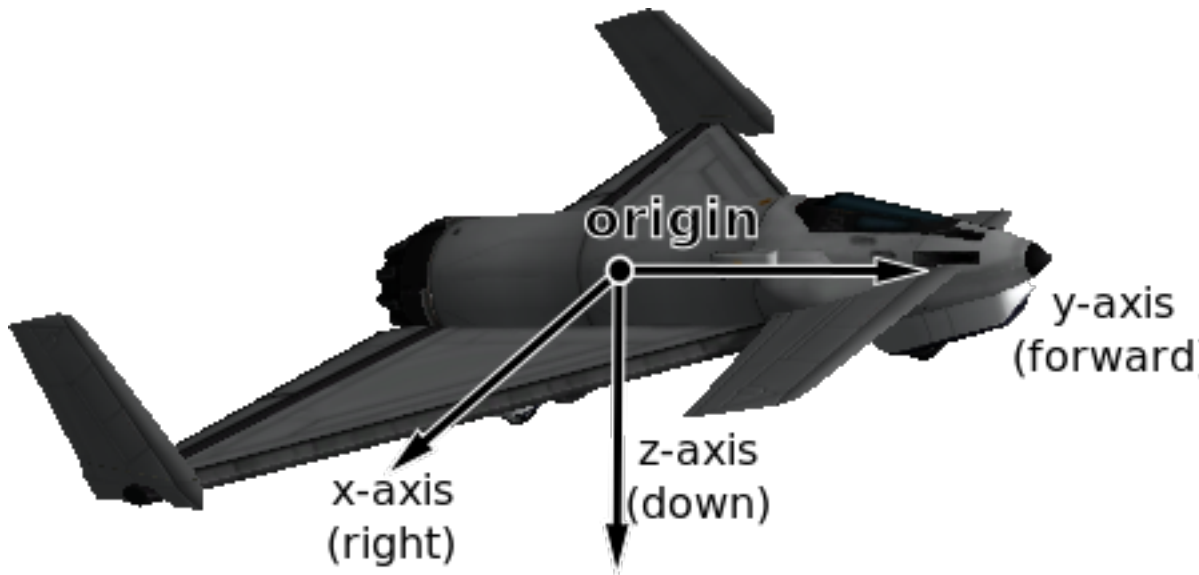


Fig. 3.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

- The z-axis points in the orbital normal direction.

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

ReferenceFrame **SurfaceReferenceFrame** { **get**; }

The reference frame that is fixed relative to the vessel, and orientated with the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the north and up directions on the surface of the body.
- The x-axis points in the [zenith](#) direction (upwards, normal to the body being orbited, from the center of the body towards the center of mass of the vessel).
- The y-axis points northwards towards the [astronomical horizon](#) (north, and tangential to the surface of the body – the direction in which a compass would point when on the surface).
- The z-axis points eastwards towards the [astronomical horizon](#) (east, and tangential to the surface of the body – east on a compass when on the surface).

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

ReferenceFrame **SurfaceVelocityReferenceFrame** { **get**; }

The reference frame that is fixed relative to the vessel, and orientated with the velocity vector of the vessel relative to the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel’s velocity vector.
- The y-axis points in the direction of the vessel’s velocity vector, relative to the surface of the body being orbited.
- The z-axis is in the plane of the [astronomical horizon](#).

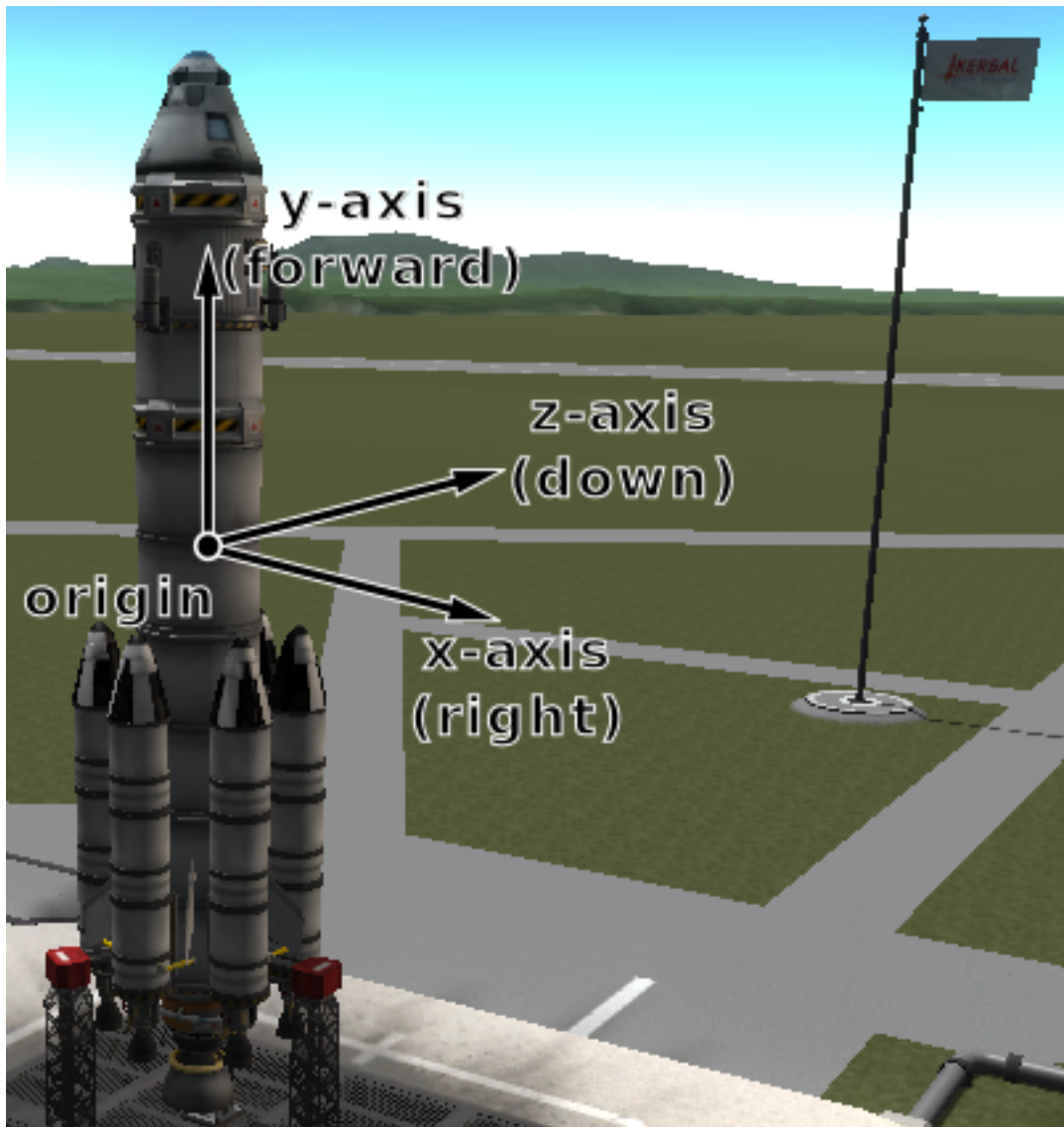


Fig. 3.2: Vessel reference frame origin and axes for the Kerbal-X rocket

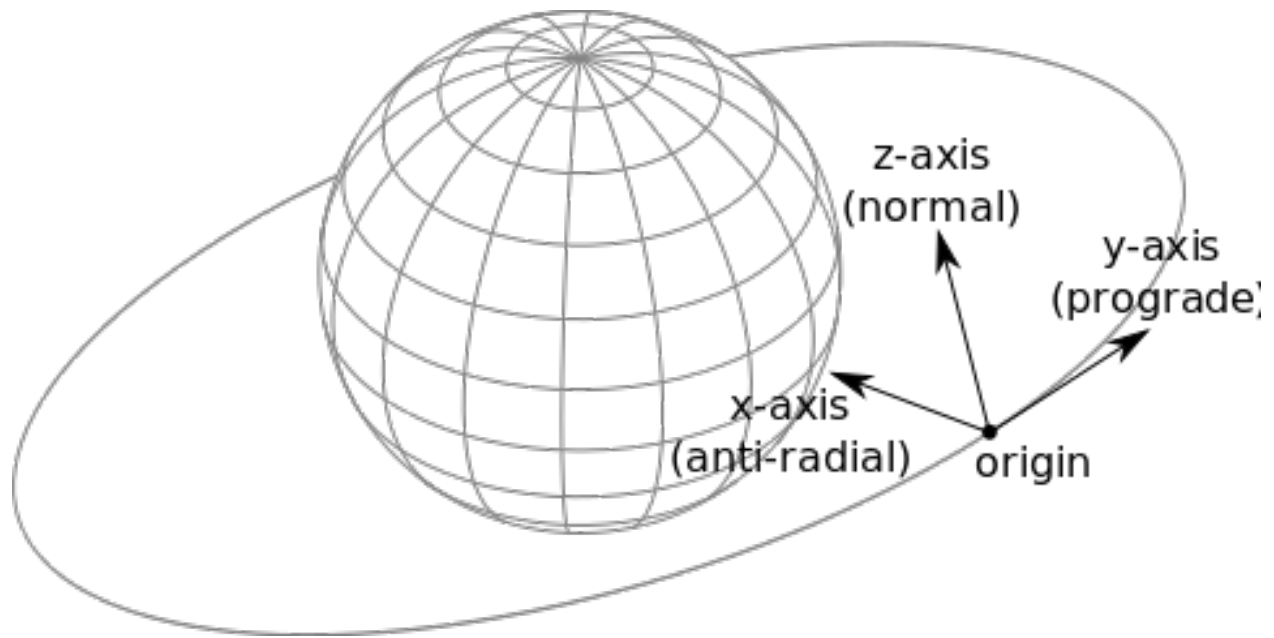


Fig. 3.3: Vessel orbital reference frame origin and axes

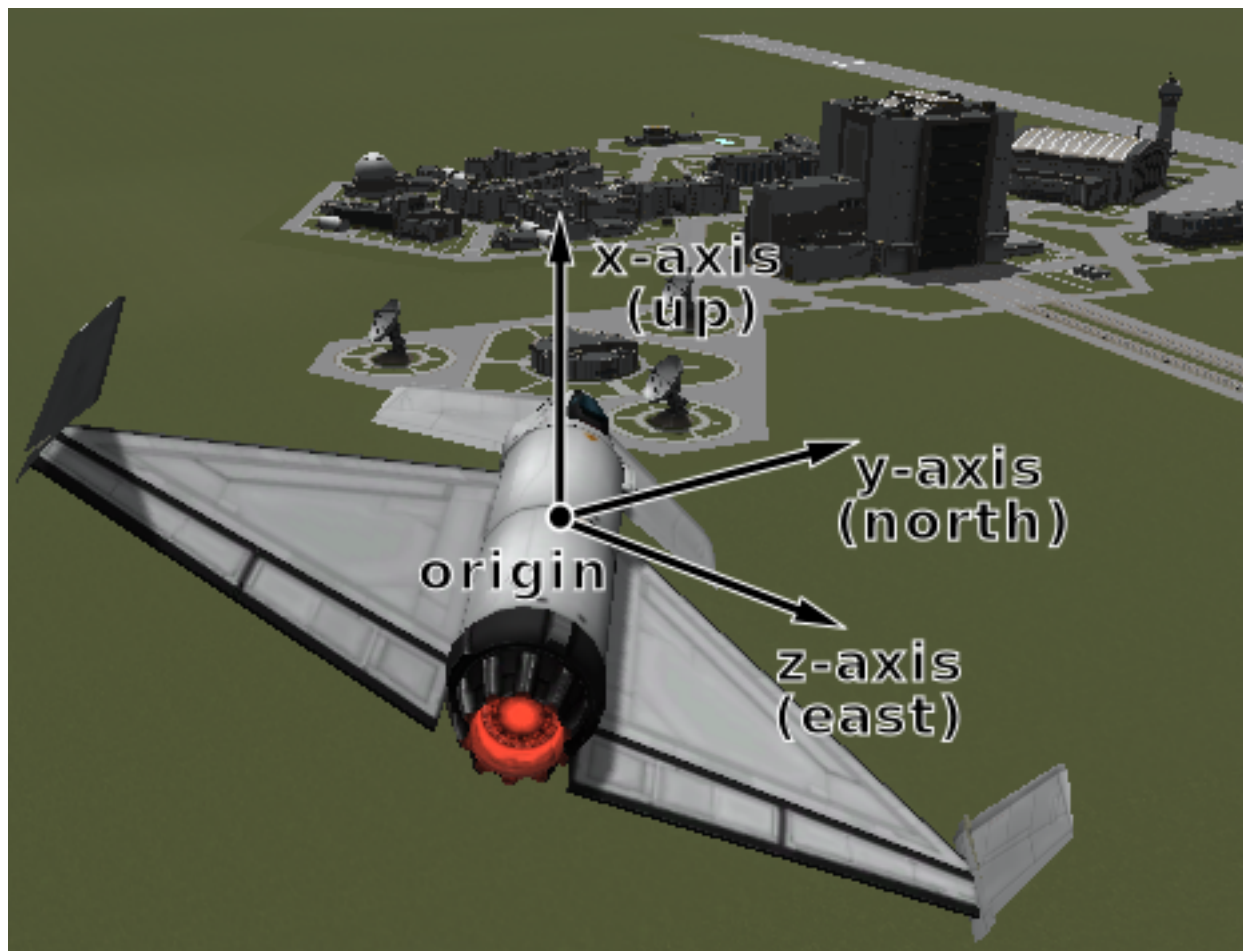


Fig. 3.4: Vessel surface reference frame origin and axes

- The x-axis is orthogonal to the other two axes.

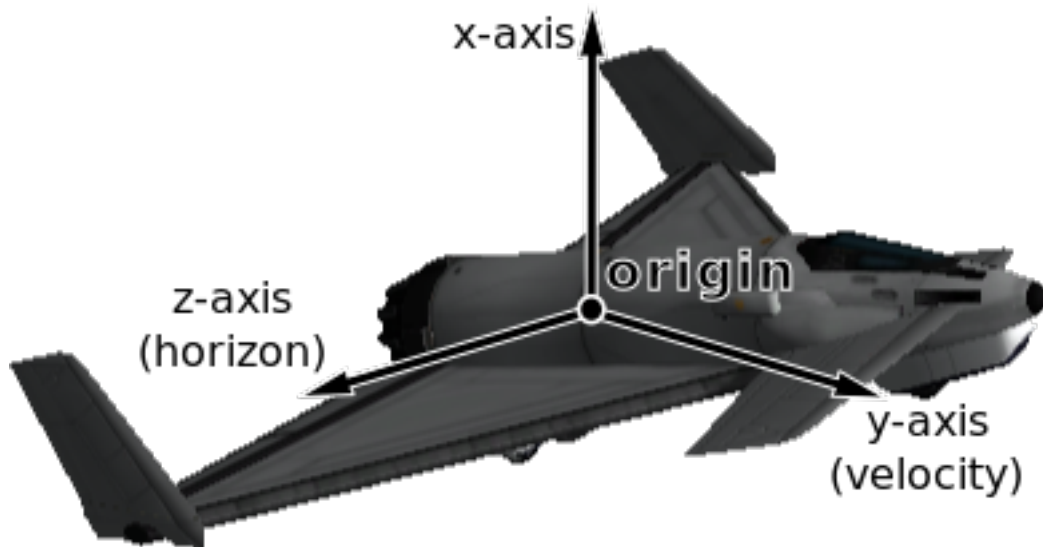


Fig. 3.5: Vessel surface velocity reference frame origin and axes

Position (*ReferenceFrame* *referenceFrame*)

Returns the position vector of the center of mass of the vessel in the given reference frame.

Parameters

Velocity (*ReferenceFrame* *referenceFrame*)

Returns the velocity vector of the center of mass of the vessel in the given reference frame.

Parameters

Rotation (*ReferenceFrame* *referenceFrame*)

Returns the rotation of the center of mass of the vessel in the given reference frame.

Parameters

Direction (*ReferenceFrame* *referenceFrame*)

Returns the direction in which the vessel is pointing, as a unit vector, in the given reference frame.

Parameters

AngularVelocity (*ReferenceFrame* *referenceFrame*)

Returns the angular velocity of the vessel in the given reference frame. The magnitude of the returned vector is the rotational speed in radians per second, and the direction of the vector indicates the axis of rotation (using the right hand rule).

Parameters

enum VesselType

See *Vessel.Type*.

Ship

Ship.

Station

Station.

Lander

Lander.

Probe
Probe.

Rover
Rover.

Base
Base.

Debris
Debris.

enum VesselSituation
See *Vessel.Situation*.

Docked
Vessel is docked to another.

Escaping
Escaping.

Flying
Vessel is flying through an atmosphere.

Landed
Vessel is landed on the surface of a body.

Orbiting
Vessel is orbiting a body.

PreLaunch
Vessel is awaiting launch.

Splashed
Vessel has splashed down in an ocean.

SubOrbital
Vessel is on a sub-orbital trajectory.

3.3.3 CelestialBody

class CelestialBody
Represents a celestial body (such as a planet or moon).

string Name { get; }
The name of the body.

IList<CelestialBody> Satellites { get; }
A list of celestial bodies that are in orbit around this celestial body.

Orbit Orbit { get; }
The orbit of the body.

float Mass { get; }
The mass of the body, in kilograms.

float GravitationalParameter { get; }
The standard gravitational parameter of the body in m^3s^{-2} .

float SurfaceGravity { get; }
The acceleration due to gravity at sea level (mean altitude) on the body, in m/s^2 .

`float RotationalPeriod { get; }`

The sidereal rotational period of the body, in seconds.

`float RotationalSpeed { get; }`

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

`float EquatorialRadius { get; }`

The equatorial radius of the body, in meters.

`double SurfaceHeight (double latitude, double longitude)`

The height of the surface relative to mean sea level at the given position, in meters. When over water this is equal to 0.

Parameters

- **latitude** – Latitude in degrees
- **longitude** – Longitude in degrees

`double BedrockHeight (double latitude, double longitude)`

The height of the surface relative to mean sea level at the given position, in meters. When over water, this is the height of the sea-bed and is therefore a negative value.

Parameters

- **latitude** – Latitude in degrees
- **longitude** – Longitude in degrees

`Tuple<double, double, double> MSLPosition (double latitude, double longitude, ReferenceFrame referenceFrame)`

The position at mean sea level at the given latitude and longitude, in the given reference frame.

Parameters

- **latitude** – Latitude in degrees
- **longitude** – Longitude in degrees
- **referenceFrame** – Reference frame for the returned position vector

`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

`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

`float SphereOfInfluence { get; }`

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

`bool HasAtmosphere { get; }`

true if the body has an atmosphere.

`float AtmosphereDepth { get; }`

The depth of the atmosphere, in meters.

`bool HasAtmosphericOxygen { get; }`

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

ReferenceFrame `ReferenceFrame { get; }`

The reference frame that is fixed relative to the celestial body.

- The origin is at the center of the body.
- The axes rotate with the body.
- The x-axis points from the center of the body towards the intersection of the prime meridian and equator (the position at 0° longitude, 0° latitude).
- The y-axis points from the center of the body towards the north pole.
- The z-axis points from the center of the body towards the equator at 90°E longitude.

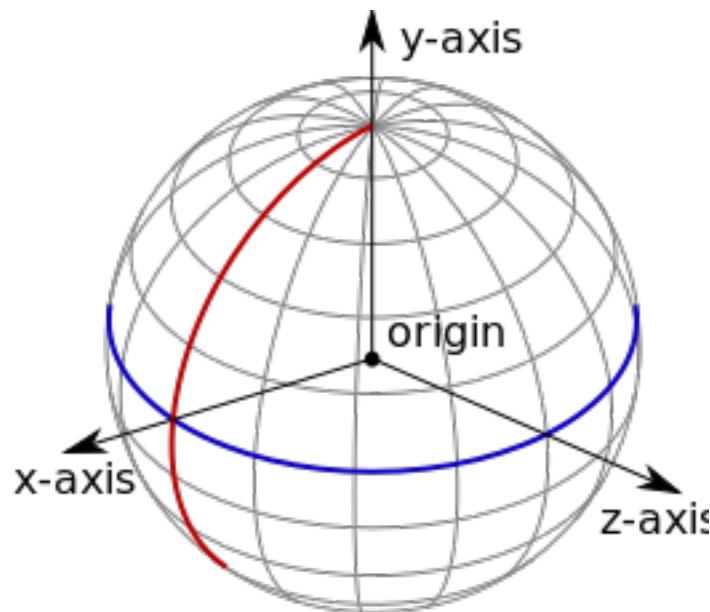


Fig. 3.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

ReferenceFrame `NonRotatingReferenceFrame { get; }`

The reference frame that is fixed relative to this celestial body, and orientated in a fixed direction (it does not rotate with the body).

- The origin is at the center of the body.
- The axes do not rotate.
- The x-axis points in an arbitrary direction through the equator.
- The y-axis points from the center of the body towards the north pole.
- The z-axis points in an arbitrary direction through the equator.

ReferenceFrame **OrbitalReferenceFrame** { **get**; }

Gets the reference frame that is fixed relative to this celestial body, but orientated with the body's orbital prograde/normal/radial directions.

- The origin is at the center of the body.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

Tuple<double, double, double> **Position** (*ReferenceFrame* *referenceFrame*)

Returns the position vector of the center of the body in the specified reference frame.

Parameters

Tuple<double, double, double> **Velocity** (*ReferenceFrame* *referenceFrame*)

Returns the velocity vector of the body in the specified reference frame.

Parameters

Tuple<double, double, double, double> **Rotation** (*ReferenceFrame* *referenceFrame*)

Returns the rotation of the body in the specified reference frame.

Parameters

Tuple<double, double, double> **Direction** (*ReferenceFrame* *referenceFrame*)

Returns the direction in which the north pole of the celestial body is pointing, as a unit vector, in the specified reference frame.

Parameters

Tuple<double, double, double> **AngularVelocity** (*ReferenceFrame* *referenceFrame*)

Returns the angular velocity of the body in the specified reference frame. The magnitude of the vector is the rotational speed of the body, in radians per second, and the direction of the vector indicates the axis of rotation, using the right-hand rule.

Parameters

3.3.4 Flight

class **Flight**

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

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

float **GForce** { **get**; }

The current G force acting on the vessel in m/s^2 .

double **MeanAltitude** { **get**; }

The altitude above sea level, in meters.

double **SurfaceAltitude** { **get**; }

The altitude above the surface of the body or sea level, whichever is closer, in meters.

`double BedrockAltitude { get; }`

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor.

`double Elevation { get; }`

The elevation of the terrain under the vessel, in meters. This is the height of the terrain above sea level, and is negative when the vessel is over the sea.

`double Latitude { get; }`

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

`double Longitude { get; }`

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

`Tuple<double, double, double> Velocity { get; }`

The velocity vector of the vessel. The magnitude of the vector is the speed of the vessel in meters per second. The direction of the vector is the direction of the vessels motion.

`double Speed { get; }`

The speed of the vessel in meters per second.

`double HorizontalSpeed { get; }`

The horizontal speed of the vessel in meters per second.

`double VerticalSpeed { get; }`

The vertical speed of the vessel in meters per second.

`Tuple<double, double, double> CenterOfMass { get; }`

The position of the center of mass of the vessel.

`Tuple<double, double, double, double> Rotation { get; }`

The rotation of the vessel.

`Tuple<double, double, double> Direction { get; }`

The direction vector that the vessel is pointing in.

`float Pitch { get; }`

The pitch angle of the vessel relative to the horizon, in degrees. A value between -90° and $+90^\circ$.

`float Heading { get; }`

The heading angle of the vessel relative to north, in degrees. A value between 0° and 360° .

`float Roll { get; }`

The roll angle of the vessel relative to the horizon, in degrees. A value between -180° and $+180^\circ$.

`Tuple<double, double, double> Prograde { get; }`

The unit direction vector pointing in the prograde direction.

`Tuple<double, double, double> Retrograde { get; }`

The unit direction vector pointing in the retrograde direction.

`Tuple<double, double, double> Normal { get; }`

The unit direction vector pointing in the normal direction.

`Tuple<double, double, double> AntiNormal { get; }`

The unit direction vector pointing in the anti-normal direction.

`Tuple<double, double, double> Radial { get; }`

The unit direction vector pointing in the radial direction.

`Tuple<double, double, double> AntiRadial { get; }`

The unit direction vector pointing in the anti-radial direction.

float AtmosphereDensity { get; }

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

float DynamicPressure { get; }

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

Note: Calculated using [KSPs stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

float StaticPressure { get; }

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

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

Tuple<double, double, double> AerodynamicForce { get; }

The total aerodynamic forces acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

Tuple<double, double, double> Lift { get; }

The [aerodynamic lift](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

Tuple<double, double, double> Drag { get; }

The [aerodynamic drag](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

float SpeedOfSound { get; }

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

Note: Not available when [Ferram Aerospace Research](#) is installed.

float Mach { get; }

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

Note: Not available when [Ferram Aerospace Research](#) is installed.

`float EquivalentAirSpeed { get; }`

The equivalent air speed of the vessel, in m/s .

Note: Not available when [Ferram Aerospace Research](#) is installed.

`float TerminalVelocity { get; }`

An estimate of the current terminal velocity of the vessel, in m/s . This is the speed at which the drag forces cancel out the force of gravity.

Note: Calculated using [KSPs stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

`float AngleOfAttack { get; }`

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

`float SideslipAngle { get; }`

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

`float TotalAirTemperature { get; }`

The total air temperature of the atmosphere around the vessel, in Kelvin. This temperature includes the *Flight.StaticAirTemperature* and the vessel's kinetic energy.

`float StaticAirTemperature { get; }`

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

`float StallFraction { get; }`

Gets the current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

Note: Requires [Ferram Aerospace Research](#).

`float DragCoefficient { get; }`

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

Note: Requires [Ferram Aerospace Research](#).

`float LiftCoefficient { get; }`

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

Note: Requires [Ferram Aerospace Research](#).

`float BallisticCoefficient { get; }`

Gets the ballistic coefficient.

Note: Requires [Ferram Aerospace Research](#).

`float ThrustSpecificFuelConsumption { get; }`

Gets 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 product one newton of thrust.

Note: Requires [Ferram Aerospace Research](#).

3.3.5 Orbit

class Orbit

Describes an orbit. For example, the orbit of a vessel, obtained by calling *Vessel.Orbit*, or a celestial body, obtained by calling *CelestialBody.Orbit*.

CelestialBody Body { get; }

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

double Apoapsis { get; }

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

Note: For the apoapsis altitude reported on the in-game map view, use *Orbit.ApoapsisAltitude*.

double Periapsis { get; }

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

Note: For the periapsis altitude reported on the in-game map view, use *Orbit.PeriapsisAltitude*.

double ApoapsisAltitude { get; }

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

Note: This is equal to *Orbit.Apoapsis* minus the equatorial radius of the body.

double PeriapsisAltitude { get; }

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

Note: This is equal to *Orbit.Periapsis* minus the equatorial radius of the body.

double SemiMajorAxis { get; }

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

double SemiMinorAxis { get; }

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

double Radius { get; }

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

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

double Speed { get; }

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

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

`double Period { get; }`

The orbital period, in seconds.

`double TimeToApoapsis { get; }`

The time until the object reaches apoapsis, in seconds.

`double TimeToPeriapsis { get; }`

The time until the object reaches periapsis, in seconds.

`double Eccentricity { get; }`

The *eccentricity* of the orbit.

`double Inclination { get; }`

The *inclination* of the orbit, in radians.

`double LongitudeOfAscendingNode { get; }`

The *longitude of the ascending node*, in radians.

`double ArgumentOfPeriapsis { get; }`

The *argument of periapsis*, in radians.

`double MeanAnomalyAtEpoch { get; }`

The *mean anomaly at epoch*.

`double Epoch { get; }`

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

`double MeanAnomaly { get; }`

The *mean anomaly*.

`double EccentricAnomaly { get; }`

The *eccentric anomaly*.

`Tuple<double, double, double> ReferencePlaneNormal (ReferenceFrame referenceFrame)`

The unit direction vector 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

`Tuple<double, double, double> ReferencePlaneDirection (ReferenceFrame referenceFrame)`

The unit direction vector from which the orbits longitude of ascending node is measured, in the given reference frame.

Parameters

`double TimeToSOIChange { get; }`

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

Orbit `NextOrbit { get; }`

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

3.3.6 Control

class Control

Used to manipulate the controls of a vessel. This includes adjusting the throttle, enabling/disabling systems such as SAS and RCS, or altering the direction in which the vessel is pointing.

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.

bool SAS { get; set; }
The state of SAS.

Note: Equivalent to *AutoPilot.SAS*

SASMode SASMode { get; set; }
The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

Note: Equivalent to *AutoPilot.SASMode*

SpeedMode SpeedMode { get; set; }
The current *SpeedMode* of the navball. This is the mode displayed next to the speed at the top of the navball.

bool RCS { get; set; }
The state of RCS.

bool Gear { get; set; }
The state of the landing gear/legs.

bool Lights { get; set; }
The state of the lights.

bool Brakes { get; set; }
The state of the wheel brakes.

bool Abort { get; set; }
The state of the abort action group.

float Throttle { get; set; }
The state of the throttle. A value between 0 and 1.

float Pitch { get; set; }
The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

float Yaw { get; set; }
The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

float Roll { get; set; }
The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

float Forward { get; set; }
The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

float Up { get; set; }
The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

float Right { get; set; }
The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

float 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.

float 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.

int CurrentStage { get; }

The current stage of the vessel. Corresponds to the stage number in the in-game UI.

IList<Vessel> ActivateNextStage ()

Activates the next stage. Equivalent to pressing the space bar in-game.

Returns A list of vessel objects that are jettisoned from the active vessel.

bool GetActionGroup (uint group)

Returns `true` if the given action group is enabled.

Parameters

- **group** – A number between 0 and 9 inclusive.

void SetActionGroup (uint group, bool state)

Sets the state of the given action group (a value between 0 and 9 inclusive).

Parameters

- **group** – A number between 0 and 9 inclusive.

void ToggleActionGroup (uint group)

Toggles the state of the given action group.

Parameters

- **group** – A number between 0 and 9 inclusive.

Node AddNode (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.

IList<Node> Nodes { get; }

Returns a list of all existing maneuver nodes, ordered by time from first to last.

void RemoveNodes ()

Remove all maneuver nodes.

enum 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

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.

3.3.7 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*
 - *Cargo Bay*
 - *Decoupler*
 - *Docking Port*
 - *Engine*
 - *Fairing*
 - *Intake*
 - *Landing Gear*
 - *Landing Leg*
 - *Launch Clamp*
 - *Light*
 - *Parachute*
 - *Radiator*
 - *Resource Converter*
 - *Resource Harvester*
 - *Reaction Wheel*
 - *Sensor*
 - *Solar Panel*
- *Trees of Parts*
 - *Traversing the Tree*
 - *Attachment Modes*
- *Fuel Lines*
- *Staging*

Parts

class Parts

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling *Vessel.Parts*.

ICollection<Part> **All** { **get**; }

A list of all of the vessels parts.

Part **Root** { **get**; }

The vessels root part.

Note: See the discussion on *Trees of Parts*.

Part **Controlling** { **get**; **set**; }

The part from which the vessel is controlled.

ICollection<Part> **WithName** (string *name*)

A list of parts whose *Part.Name* is *name*.

Parameters

ICollection<Part> **WithTitle** (string *title*)

A list of all parts whose *Part.Title* is *title*.

Parameters

IList<Part> WithModule (string moduleName)

A list of all parts that contain a *Module* whose *Module.Name* is *moduleName*.

Parameters

IList<Part> InStage (int stage)

A list of all parts that are activated in the given *stage*.

Parameters

Note: See the discussion on *Staging*.

IList<Part> InDecoupleStage (int stage)

A list of all parts that are decoupled in the given *stage*.

Parameters

Note: See the discussion on *Staging*.

IList<Module> ModulesWithName (string moduleName)

A list of modules (combined across all parts in the vessel) whose *Module.Name* is *moduleName*.

Parameters

IList<CargoBay> CargoBays { get; }

A list of all cargo bays in the vessel.

IList<Decoupler> Decouplers { get; }

A list of all decouplers in the vessel.

IList<DockingPort> DockingPorts { get; }

A list of all docking ports in the vessel.

DockingPort DockingPortWithName (string name)

The first docking port in the vessel with the given port name, as returned by *DockingPort.Name*. Returns `null` if there are no such docking ports.

Parameters

IList<Engine> Engines { get; }

A list of all engines in the vessel.

IList<Fairing> Fairings { get; }

A list of all fairings in the vessel.

IList<Intake> Intakes { get; }

A list of all intakes in the vessel.

IList<LandingGear> LandingGear { get; }

A list of all landing gear attached to the vessel.

IList<LandingLeg> LandingLegs { get; }

A list of all landing legs attached to the vessel.

IList<LaunchClamp> LaunchClamps { get; }

A list of all launch clamps attached to the vessel.

IList<Light> Lights { get; }

A list of all lights in the vessel.

`IList<Parachute> Parachutes { get; }`
A list of all parachutes in the vessel.

`IList<Radiator> Radiators { get; }`
A list of all radiators in the vessel.

`IList<ReactionWheel> ReactionWheels { get; }`
A list of all reaction wheels in the vessel.

`IList<ResourceConverter> ResourceConverters { get; }`
A list of all resource converters in the vessel.

`IList<ResourceHarvester> ResourceHarvesters { get; }`
A list of all resource harvesters in the vessel.

`IList<Sensor> Sensors { get; }`
A list of all sensors in the vessel.

`IList<SolarPanel> SolarPanels { get; }`
A list of all solar panels in the vessel.

Part

class Part

Instances of this class represents a part. A vessel is made of multiple parts. Instances can be obtained by various methods in *Parts*.

`string Name { get; }`
Internal name of the part, as used in *part cfg files*. For example “Mark1-2Pod”.

`string Title { get; }`
Title of the part, as shown when the part is right clicked in-game. For example “Mk1-2 Command Pod”.

`double Cost { get; }`
The cost of the part, in units of funds.

`Vessel Vessel { get; }`
The vessel that contains this part.

`Part Parent { get; }`
The parts parent. Returns `null` if the part does not have a parent. This, in combination with *Part.Children*, can be used to traverse the vessels parts tree.

Note: See the discussion on *Trees of Parts*.

`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.

Note: See the discussion on *Trees of Parts*.

`bool AxiallyAttached { get; }`
Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns `false`.

Note: See the discussion on [Attachment Modes](#).

bool RadiallyAttached { get; }

Whether the part is radially attached to its parent, i.e. on the side of its parent. If the part has no parent, returns `false`.

Note: See the discussion on [Attachment Modes](#).

int Stage { get; }

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

Note: See the discussion on [Staging](#).

int DecoupleStage { get; }

The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

Note: See the discussion on [Staging](#).

bool Massless { get; }

Whether the part is `massless`.

double Mass { get; }

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

double DryMass { get; }

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

double ImpactTolerance { get; }

The impact tolerance of the part, in meters per second.

double Temperature { get; }

Temperature of the part, in Kelvin.

double SkinTemperature { get; }

Temperature of the skin of the part, in Kelvin.

double MaxTemperature { get; }

Maximum temperature that the part can survive, in Kelvin.

double MaxSkinTemperature { get; }

Maximum temperature that the skin of the part can survive, in Kelvin.

float ThermalMass { get; }

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

float ThermalSkinMass { get; }

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

float 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.

float 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.

float 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.

float 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.

float 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.

float ThermalSkinToInternalFlux { get; }

The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

Resources Resources { get; }

A *Resources* object for the part.

bool Crossfeed { get; }

Whether this part is crossfeed capable.

bool IsFuelLine { get; }

Whether this part is a fuel line.

IList<Part> FuelLinesFrom { get; }

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

Note: See the discussion on *Fuel Lines*.

IList<Part> FuelLinesTo { get; }

The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

Note: See the discussion on *Fuel Lines*.

IList<Module> Modules { get; }

The modules for this part.

CargoBay CargoBay { get; }

A *CargoBay* if the part is a cargo bay, otherwise null.

Decoupler Decoupler { get; }

A *Decoupler* if the part is a decoupler, otherwise null.

DockingPort DockingPort { get; }

A *DockingPort* if the part is a docking port, otherwise null.

Engine Engine { get; }

An *Engine* if the part is an engine, otherwise null.

Fairing **Fairing** { get; }
 A *Fairing* if the part is a fairing, otherwise null.

Intake **Intake** { get; }
 An *Intake* if the part is an intake, otherwise null.

LandingGear **LandingGear** { get; }
 A *LandingGear* if the part is a landing gear , otherwise null.

LandingLeg **LandingLeg** { get; }
 A *LandingLeg* if the part is a landing leg, otherwise null.

LaunchClamp **LaunchClamp** { get; }
 A *LaunchClamp* if the part is a launch clamp, otherwise null.

Light **Light** { get; }
 A *Light* if the part is a light, otherwise null.

Parachute **Parachute** { get; }
 A *Parachute* if the part is a parachute, otherwise null.

Radiator **Radiator** { get; }
 A *Radiator* if the part is a radiator, otherwise null.

ReactionWheel **ReactionWheel** { get; }
 A *ReactionWheel* if the part is a reaction wheel, otherwise null.

ResourceConverter **ResourceConverter** { get; }
 A *ResourceConverter* if the part is a resource converter, otherwise null.

ResourceHarvester **ResourceHarvester** { get; }
 A *ResourceHarvester* if the part is a resource harvester, otherwise null.

Sensor **Sensor** { get; }
 A *Sensor* if the part is a sensor, otherwise null.

SolarPanel **SolarPanel** { get; }
 A *SolarPanel* if the part is a solar panel, otherwise null.

Tuple<double, double, double> Position (*ReferenceFrame* referenceFrame)
 The position of the part in the given reference frame.

Parameters

Tuple<double, double, double> Direction (*ReferenceFrame* referenceFrame)
 The direction of the part in the given reference frame.

Parameters

Tuple<double, double, double> Velocity (*ReferenceFrame* referenceFrame)
 The velocity of the part in the given reference frame.

Parameters

Tuple<double, double, double, double> Rotation (*ReferenceFrame* referenceFrame)
 The rotation of the part in the given reference frame.

Parameters

ReferenceFrame **ReferenceFrame** { get; }
 The reference frame that is fixed relative to this part.

- The origin is at the position of the part.
- The axes rotate with the part.

- The x, y and z axis directions depend on the design of the part.

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by `DockingPort.ReferenceFrame`.

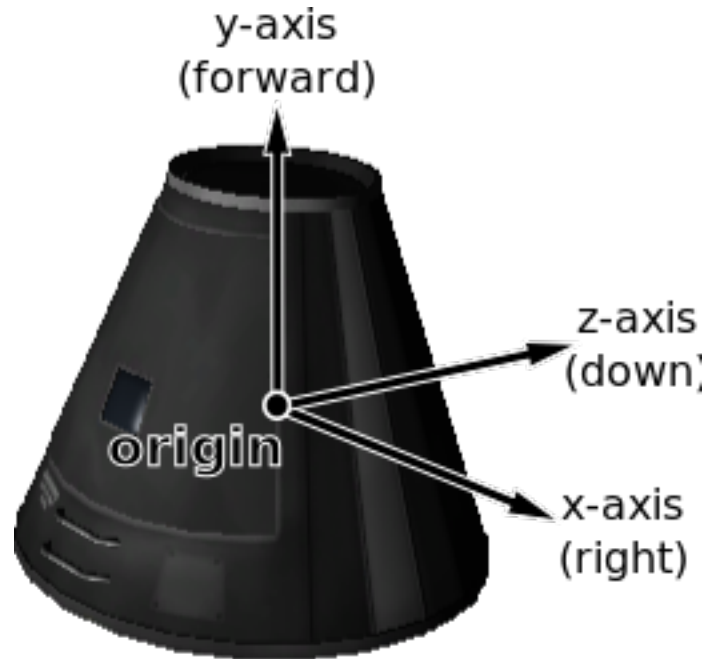


Fig. 3.7: Mk1 Command Pod reference frame origin and axes

Module

class Module

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” `PartModule` that contains all the functionality of an engine. This class allows you to interact with KSPs `PartModules`, and any `PartModules` that have been added by other mods.

`string Name { get; }`

Name of the `PartModule`. For example, “ModuleEngines”.

`Part Part { get; }`

The part that contains this module.

`IDictionary<string, string> Fields { get; }`

The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

`bool HasField (string name)`

Returns `true` if the module has a field with the given name.

Parameters

- **name** – Name of the field.

`string GetField (string name)`

Returns the value of a field.

Parameters

- **name** – Name of the field.

`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.

`bool HasEvent (string name)`

true if the module has an event with the given name.

Parameters

`void TriggerEvent (string name)`

Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters

`IList<string> Actions { get; }`

A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

`bool HasAction (string name)`

true if the part has an action with the given name.

Parameters

`void SetAction (string name, bool value = True)`

Set the value of an action with the given name.

Parameters**Specific Types of Part**

The following classes provide functionality for specific types of part.

- *Cargo Bay*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Fairing*
- *Intake*
- *Landing Gear*
- *Landing Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *Sensor*
- *Solar Panel*

Cargo Bay

class CargoBay

Obtained by calling *Part.CargoBay*.

Part **Part** { **get**; }

The part object for this cargo bay.

CargoBayState **State** { **get**; }

The state of the cargo bay.

bool **Open** { **get**; **set**; }

Whether the cargo bay is open.

enum CargoBayState

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.

Decoupler

class Decoupler

Obtained by calling *Part.Decoupler*

Part **Part** { **get**; }

The part object for this decoupler.

void **Decouple** ()

Fires the decoupler. Has no effect if the decoupler has already fired.

bool **Decoupled** { **get**; }

Whether the decoupler has fired.

float **Impulse** { **get**; }

The impulse that the decoupler imparts when it is fired, in Newton seconds.

Docking Port

class DockingPort

Obtained by calling *Part.DockingPort*

Part **Part** { **get**; }

The part object for this docking port.

string **Name** { **get**; **set**; }

The port name of the docking port. This is the name of the port that can be set in the right click menu, when the **Docking Port Alignment Indicator** mod is installed. If this mod is not installed, returns the title of the part (*Part.Title*).

DockingPortState **State** { **get**; }

The current state of the docking port.

Part **DockedPart** { **get**; }

The part that this docking port is docked to. Returns `null` if this docking port is not docked to anything.

Vessel **Undock** ()

Undocks the docking port and returns the vessel that was undocked from. After undocking, the active vessel may change (*SpaceCenter.ActiveVessel*). This method can be called for either docking port in a docked pair - both calls will have the same effect. Returns `null` if the docking port is not docked to anything.

float **ReengageDistance** { **get**; }

The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

bool **HasShield** { **get**; }

Whether the docking port has a shield.

bool **Shielded** { **get**; **set**; }

The state of the docking ports shield, if it has one. Returns `true` if the docking port has a shield, and the shield is closed. Otherwise returns `false`. When set to `true`, the shield is closed, and when set to `false` the shield is opened. If the docking port does not have a shield, setting this attribute has no effect.

Tuple<double, double, double> **Position** (*ReferenceFrame* *referenceFrame*)

The position of the docking port in the given reference frame.

Parameters

Tuple<double, double, double> **Direction** (*ReferenceFrame* *referenceFrame*)

The direction that docking port points in, in the given reference frame.

Parameters

Tuple<double, double, double, double> **Rotation** (*ReferenceFrame* *referenceFrame*)

The rotation of the docking port, in the given reference frame.

Parameters

ReferenceFrame **ReferenceFrame** { **get**; }

The reference frame that is fixed relative to this docking port, and oriented with the port.

- The origin is at the position of the docking port.
- The axes rotate with the docking port.
- The x-axis points out to the right side of the docking port.
- The y-axis points in the direction the docking port is facing.
- The z-axis points out of the bottom off the docking port.

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by *Part.ReferenceFrame*.

enum **DockingPortState**

See *DockingPort.State*.

Ready

The docking port is ready to dock to another docking port.

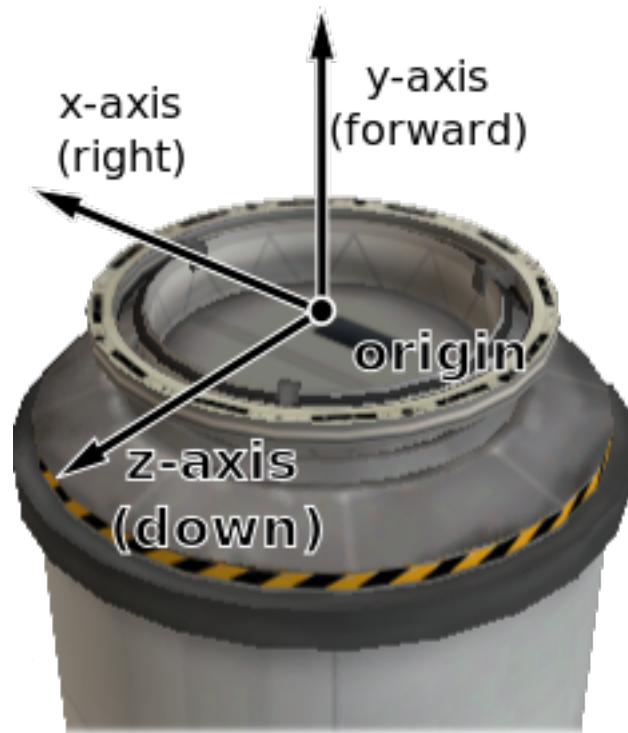


Fig. 3.8: Docking port reference frame origin and axes

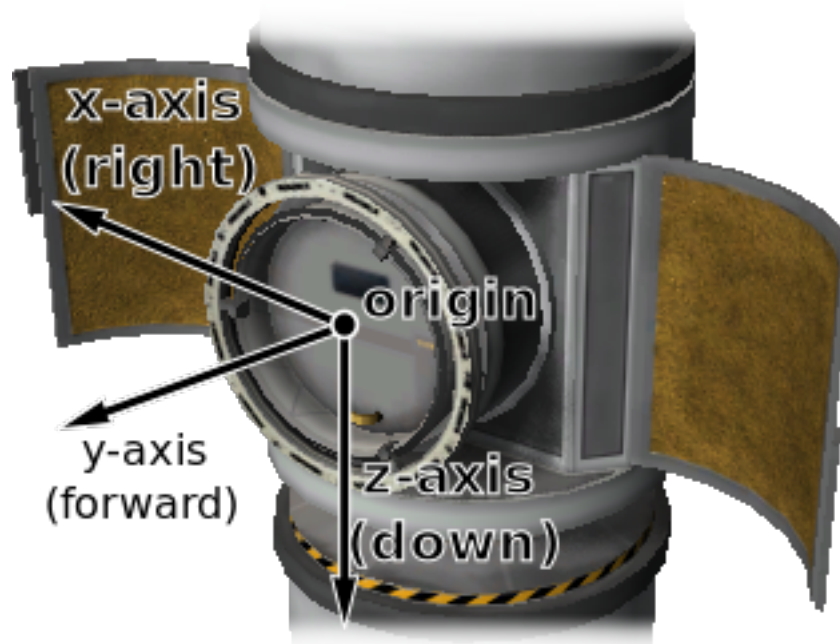


Fig. 3.9: Inline docking port reference frame origin and axes

Docked

The docking port is docked to another docking port, or docked to another part (from the VAB/SPH).

Docking

The docking port is very close to another docking port, but has not docked. It is using magnetic force to acquire a solid dock.

Undocking

The docking port has just been undocked from another docking port, and is disabled until it moves away by a sufficient distance (*DockingPort.ReengageDistance*).

Shielded

The docking port has a shield, and the shield is closed.

Moving

The docking ports shield is currently opening/closing.

Engine

class Engine

Obtained by calling *Part.Engine*.

Part **Part** { **get**; }

The part object for this engine.

bool **Active** { **get**; **set**; }

Whether the engine is active. Setting this attribute may have no effect, depending on *Engine.CanShutdown* and *Engine.CanRestart*.

float **Thrust** { **get**; }

The current amount of thrust being produced by the engine, in Newtons. Returns zero if the engine is not active or if it has no fuel.

float **AvailableThrust** { **get**; }

The maximum available amount of thrust that can be produced by the engine, in Newtons. This takes *Engine.ThrustLimit* into account, and is the amount of thrust produced by the engine when activated and the main throttle is set to 100%. Returns zero if the engine does not have any fuel.

float **MaxThrust** { **get**; }

Gets the maximum amount of thrust that can be produced by the engine, in Newtons. This is the amount of thrust produced by the engine when activated, *Engine.ThrustLimit* is set to 100% and the main vessel's throttle is set to 100%.

float **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.

float **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.

float **SpecificImpulse** { **get**; }

The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

float **VacuumSpecificImpulse** { **get**; }

The vacuum specific impulse of the engine, in seconds.

float **KerbinSeaLevelSpecificImpulse** { **get**; }

The specific impulse of the engine at sea level on Kerbin, in seconds.

IList<string> Propellants { get; }

The names of resources that the engine consumes.

IDictionary<string, float> PropellantRatios { get; }

The ratios of resources that the engine consumes. A dictionary mapping resource names to the ratios at which they are consumed by the engine.

bool HasFuel { get; }

Whether the engine has run out of fuel (or flamed out).

float 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).

bool ThrottleLocked { get; }

Whether the *Control.Throttle* affects the engine. For example, this is `true` for liquid fueled rockets, and `false` for solid rocket boosters.

bool 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.

bool CanShutdown { get; }

Gets whether the engine can be shutdown once activated. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

bool HasModes { get; }

Whether the engine has multiple modes of operation.

string Mode { get; set; }

The name of the current engine mode.

IDictionary<string, Engine> Modes { get; }

The available modes for the engine. A dictionary mapping mode names to *Engine* objects.

void ToggleMode ()

Toggle the current engine mode.

bool AutoModeSwitch { get; set; }

Whether the engine will automatically switch modes.

bool Gimballed { get; }

Whether the engine nozzle is gimballed, i.e. can provide a turning force.

float GimbalRange { get; }

The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

bool GimbalLocked { get; set; }

Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

float GimbalLimit { get; set; }

The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

Fairing

class Fairing

Obtained by calling *Part.Fairing*.

Part Part { get; }

The part object for this fairing.

```
void Jettison ()
    Jettison the fairing. Has no effect if it has already been jettisoned.

bool Jettisoned { get; }
    Whether the fairing has been jettisoned.
```

Intake

```
class Intake
    Obtained by calling Part.Intake.

    Part Part { get; }
        The part object for this intake.

    bool Open { get; set; }
        Whether the intake is open.

    float Speed { get; }
        Speed of the flow into the intake, in m/s.

    float Flow { get; }
        The rate of flow into the intake, in units of resource per second.

    float Area { get; }
        The area of the intake's opening, in square meters.
```

Landing Gear

```
class LandingGear
    Obtained by calling Part.LandingGear.

    Part Part { get; }
        The part object for this landing gear.

    LandingGearState State { get; }
        Gets the current state of the landing gear.
```

Note: Fixed landing gear are always deployed.

```
bool Deployable { get; }
    Whether the landing gear is deployable.

bool Deployed { get; set; }
    Whether the landing gear is deployed.
```

Note: Fixed landing gear are always deployed. Returns an error if you try to deploy fixed landing gear.

```
enum LandingGearState
    See LandingGear.State.

    Deployed
        Landing gear is fully deployed.

    Retracted
        Landing gear is fully retracted.
```

Deploying

Landing gear is being deployed.

Retracting

Landing gear is being retracted.

Landing Leg**class LandingLeg**

Obtained by calling *Part.LandingLeg*.

Part **Part** { **get**; }

The part object for this landing leg.

LandingLegState **State** { **get**; }

The current state of the landing leg.

bool **Deployed** { **get**; **set**; }

Whether the landing leg is deployed.

enum LandingLegState

See *LandingLeg.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.

Repairing

Landing leg is being repaired.

Launch Clamp**class LaunchClamp**

Obtained by calling *Part.LaunchClamp*.

Part **Part** { **get**; }

The part object for this launch clamp.

void **Release** ()

Releases the docking clamp. Has no effect if the clamp has already been released.

Light**class Light**

Obtained by calling *Part.Light*.

Part **Part** { **get**; }
 The part object for this light.

bool Active { **get**; **set**; }
 Whether the light is switched on.

float PowerUsage { **get**; }
 The current power usage, in units of charge per second.

Parachute

class Parachute

Obtained by calling *Part.Parachute*.

Part **Part** { **get**; }
 The part object for this parachute.

void Deploy ()
 Deploys the parachute. This has no effect if the parachute has already been deployed.

bool Deployed { **get**; }
 Whether the parachute has been deployed.

ParachuteState **State** { **get**; }
 The current state of the parachute.

float DeployAltitude { **get**; **set**; }
 The altitude at which the parachute will full deploy, in meters.

float DeployMinPressure { **get**; **set**; }
 The minimum pressure at which the parachute will semi-deploy, in atmospheres.

enum ParachuteState

See *Parachute.State*.

Stowed
 The parachute is safely tucked away inside its housing.

Active
 The parachute is still stowed, but ready to semi-deploy.

SemiDeployed
 The parachute has been deployed and is providing some drag, but is not fully deployed yet.

Deployed
 The parachute is fully deployed.

Cut
 The parachute has been cut.

Radiator

class Radiator

Obtained by calling *Part.Radiator*.

Part **Part** { **get**; }
 The part object for this radiator.

bool Deployable { **get**; }
 Whether the radiator is deployable.

bool Deployed { get; set; }

For a deployable radiator, `true` if the radiator is extended. If the radiator is not deployable, this is always `true`.

RadiatorState **State { get; }**

The current state of the radiator.

Note: A fixed radiator is always *RadiatorState.Extended*.

enum RadiatorState

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

Obtained by calling *Part.ResourceConverter*.

Part **Part { get; }**

The part object for this converter.

int Count { get; }

The number of converters in the part.

string Name (int index)

The name of the specified converter.

Parameters

- **index** – Index of the converter.

bool Active (int index)

True if the specified converter is active.

Parameters

- **index** – Index of the converter.

void Start (int index)

Start the specified converter.

Parameters

- **index** – Index of the converter.

void Stop (int index)

Stop the specified converter.

Parameters

- **index** – Index of the converter.

ResourceConverterState **State** (*int index*)

The state of the specified converter.

Parameters

- **index** – Index of the converter.

string **StatusInfo** (*int index*)

Status information for the specified converter. This is the full status message shown in the in-game UI.

Parameters

- **index** – Index of the converter.

IList<string> **Inputs** (*int index*)

List of the names of resources consumed by the specified converter.

Parameters

- **index** – Index of the converter.

IList<string> **Outputs** (*int index*)

List of the names of resources produced by the specified converter.

Parameters

- **index** – Index of the converter.

enum ResourceConverterState

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

Obtained by calling *Part.ResourceHarvester*.

Part **Part** { **get**; }

The part object for this harvester.

ResourceHarvesterState **State** { **get**; }

The state of the harvester.

bool Deployed { get; set; }
Whether the harvester is deployed.

bool Active { get; set; }
Whether the harvester is actively drilling.

float ExtractionRate { get; }
The rate at which the drill is extracting ore, in units per second.

float ThermalEfficiency { get; }
The thermal efficiency of the drill, as a percentage of its maximum.

float CoreTemperature { get; }
The core temperature of the drill, in Kelvin.

float OptimumCoreTemperature { get; }
The core temperature at which the drill will operate with peak efficiency, in Kelvin.

enum ResourceHarvesterState

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

Obtained by calling *Part.ReactionWheel*.

Part Part { get; }
The part object for this reaction wheel.

bool Active { get; set; }
Whether the reaction wheel is active.

bool Broken { get; }
Whether the reaction wheel is broken.

float PitchTorque { get; }
The torque in the pitch axis, in Newton meters.

float YawTorque { get; }
The torque in the yaw axis, in Newton meters.

float RollTorque { get; }
The torque in the roll axis, in Newton meters.

Sensor

class Sensor

Obtained by calling *Part.Sensor*.

Part **Part** { **get**; }

The part object for this sensor.

bool **Active** { **get**; **set**; }

Whether the sensor is active.

string **Value** { **get**; }

The current value of the sensor.

float **PowerUsage** { **get**; }

The current power usage of the sensor, in units of charge per second.

Solar Panel

class SolarPanel

Obtained by calling *Part.SolarPanel*.

Part **Part** { **get**; }

The part object for this solar panel.

bool **Deployed** { **get**; **set**; }

Whether the solar panel is extended.

SolarPanelState **State** { **get**; }

The current state of the solar panel.

float **EnergyFlow** { **get**; }

The current amount of energy being generated by the solar panel, in units of charge per second.

float **SunExposure** { **get**; }

The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

enum SolarPanelState

See *SolarPanel.State*.

Extended

Solar panel is fully extended.

Retracted

Solar panel is fully retracted.

Extending

Solar panel is being extended.

Retracting

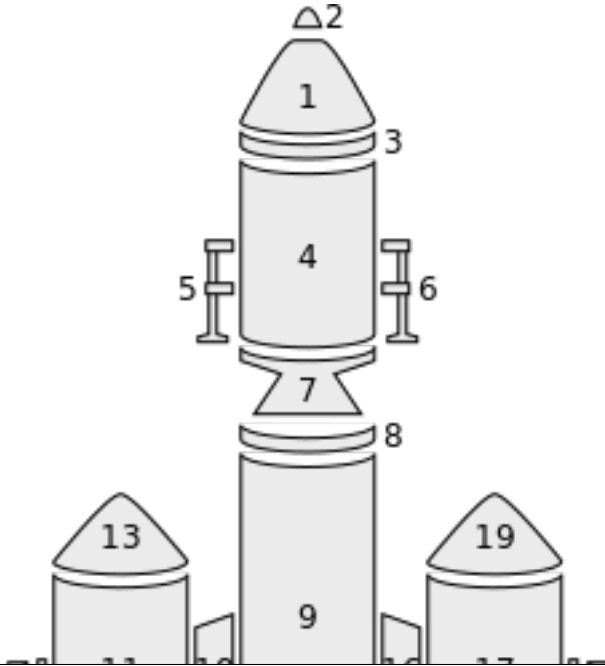
Solar panel is being retracted.

Broken

Solar panel is broken.

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 vessel's *root part* (part number 1 in the example above) and can be obtained by calling `Parts.Root`. A part's children can be obtained by calling `Part.Children`. If the part does not have any children, `Part.Children` returns an empty list. A part's 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 KRPC.Client;
using KRPC.Client.Services.SpaceCenter;
using System;
using System.Collections.Generic;
using System.Net;

class AttachmentModes {
    public static void Main () {
        var connection = new Connection ();
        var vessel = connection.SpaceCenter ().GetVessel ();
        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;
            Console.WriteLine (new String (' ', depth) + part.name);
            foreach (var child in part.Children) {
                stack.Push (new Tuple<Part,int> (child, depth + 1));
            }
        }
    }
}
```

When this code is executed using the craft file for the example vessel pictured above, the following is printed out:

```

Command Pod Mk1
TR-18A Stack Decoupler
FL-T400 Fuel Tank
LV-909 Liquid Fuel Engine
TR-18A Stack Decoupler
FL-T800 Fuel Tank
LV-909 Liquid Fuel Engine
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launch Stability Enhancer
FTX-2 External Fuel Duct
LV-909 Liquid Fuel Engine
Aerodynamic Nose Cone
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launch Stability Enhancer
FTX-2 External Fuel Duct
LV-909 Liquid Fuel Engine
Aerodynamic Nose Cone
LT-1 Landing Struts
LT-1 Landing Struts
Mk16 Parachute

```

Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

The root part of a vessel (for example the command pod – part 1) does not have a parent part,

so does not have an attachment mode. However, the part is 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 KRPC.Client;
using KRPC.Client.Services.SpaceCenter;
using System;
using System.Collections.Generic;
using System.Net;

class AttachmentModes
{
    public static void Main ()
    {
        var connection = new Connection ();
        var vessel = connection.SpaceCenter ().ActiveVessel;
        var root = vessel.Parts.Root;
        var stack = new Stack<Tuple<Part,int>> ();
    }
}

```



```
stack.Push (new Tuple<Part,int> (root, 0));
while (stack.Count > 0) {
    var item = stack.Pop ();
    Part part = item.Item1;
    int depth = item.Item2;
    string attachMode = (part.AxiallyAttached ? "axial" : "radial");
    Console.WriteLine (new String (' ', depth) + part.Title + " - " + attachMode);
    foreach (var child in part.Children)
        stack.Push (new Tuple<Part,int> (child, depth + 1));
}
}
```

When this code is execute using the craft file
for the example vessel pictured above, the fol-
lowing is printed out:

```
Command Pod Mk1 - axial
TR-18A Stack Decoupler - axial
FL-T400 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TR-18A Stack Decoupler - axial
FL-T800 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
LT-1 Landing Struts - radial
LT-1 Landing Struts - radial
Mk16 Parachute - axial
```

Fuel Lines

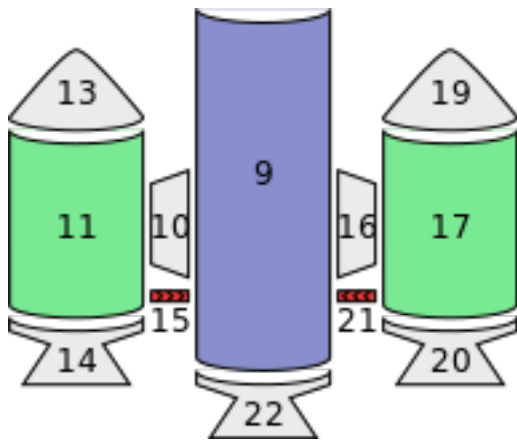


Fig. 3.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.

The attributes `Part.FuelLinesFrom` and `Part.FuelLinesTo` can be used to discover these connections. In the example in Figure 5, when `Part.FuelLinesTo` is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When `Part.FuelLinesFrom` is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).

Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using `Part.Stage` and `Part.DecoupleStage` respectively. For parts that are not activated by staging, `Part.Stage` returns -1. For parts that are never decoupled, `Part.DecoupleStage` returns a value of -1.

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *acti-*

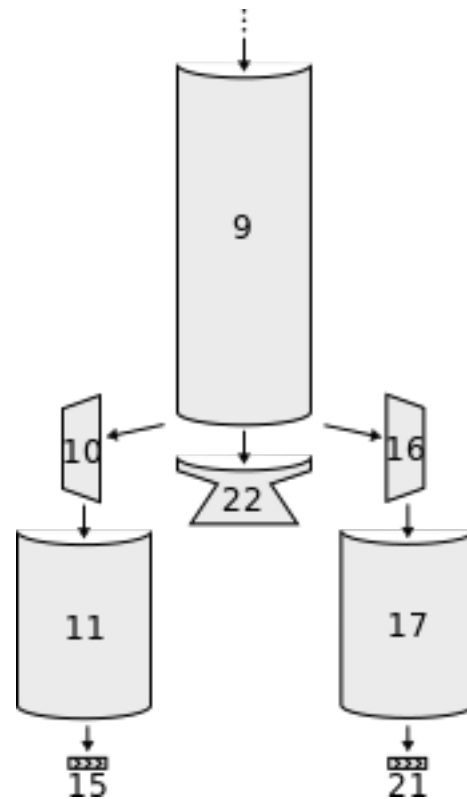


Fig. 3.13: **Figure 4** – A subset of the parts tree from Figure 2 above.

vated. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

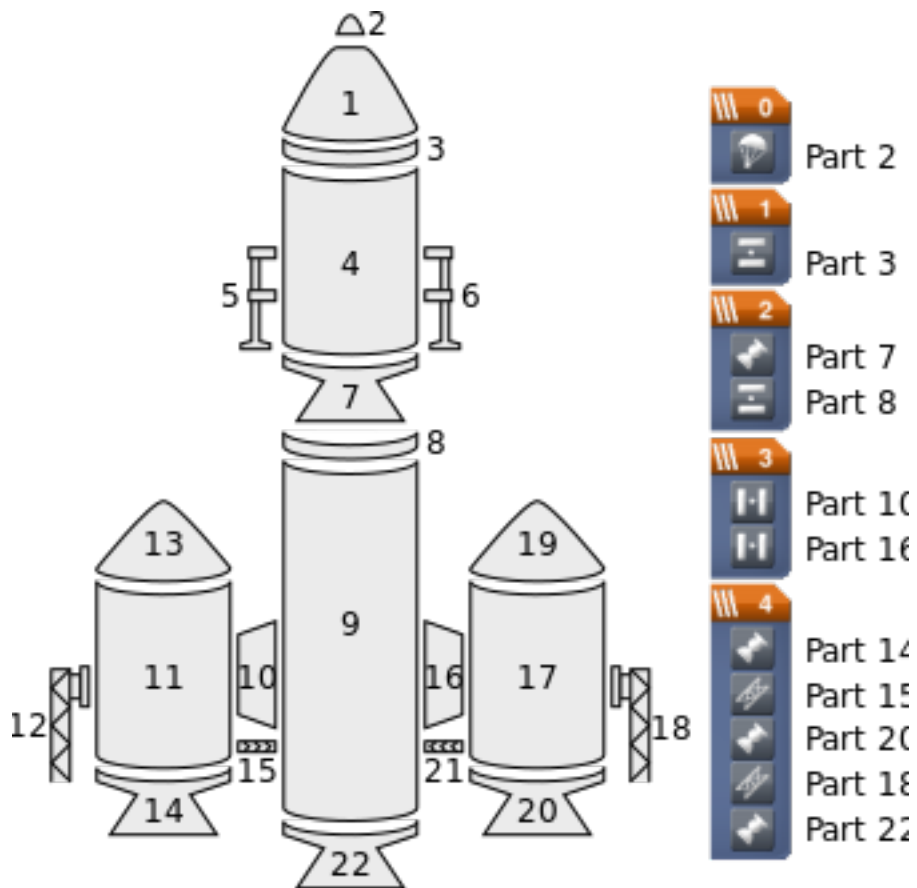


Fig. 3.14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

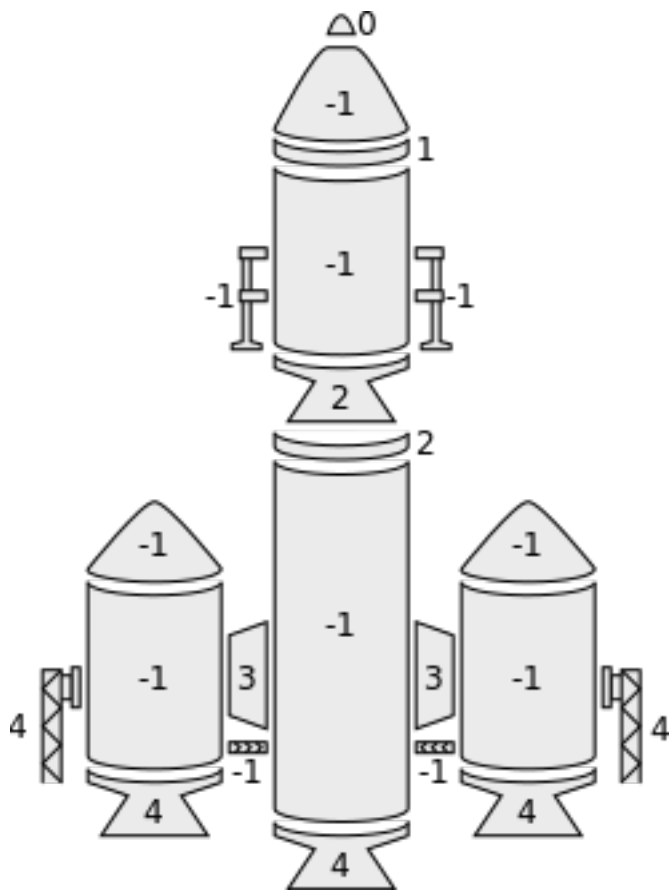


Fig. 3.15: **Figure 7** – The stage in which each part is *activated*.

3.3.8 Resources

class Resources
Created by calling `Vessel.Resources`,
`Vessel.ResourcesInDecoupleStage`
or `Part.Resources`.

`IList<string> Names { get; }`
A list of resource names that can be stored.

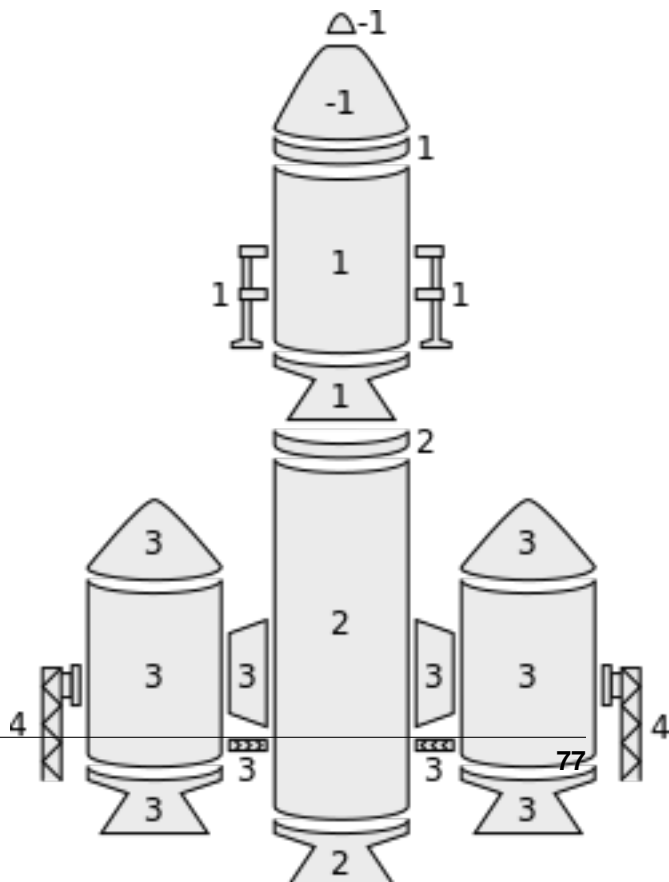
`bool HasResource (string name)`
Check whether the named resource can be stored.

Parameters
• **name** – The name of the resource.

`float Max (string name)`
Returns the amount of a resource that can be stored.

Parameters
• **name** – The name of the resource.

`float Amount (string name)`
Returns the amount of a resource that is currently



stored.

Parameters

- **name** – The name of the resource.

`float` **Density** (*string name*)

Returns the density of a resource, in kg/l.

Parameters

- **name** – The name of the resource.

ResourceFlowMode **FlowMode** (*string name*)

Returns the flow mode of a resource.

Parameters

- **name** – The name of the resource.

`enum` **ResourceFlowMode**

See *Resources.FlowMode*.

Vessel

The resource flows to any part in the vessel. For example, electric charge.

Stage

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

Adjacent

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

None

The resource does not flow. For example, solid fuel.

3.3.9 Node

`class` **Node**

Represents a maneuver node. Can be created using *Control.AddNode*.

`float` **Prograde** { **get**; **set**; }

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

`float` **Normal** { **get**; **set**; }

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

`float` **Radial** { **get**; **set**; }

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

`float` **DeltaV** { **get**; **set**; }

The delta-v of the maneuver node, in meters per second.

Note: Does not change when executing the maneuver node. See [Node.RemainingDeltaV](#).

float RemainingDeltaV { get; }

Gets the remaining delta-v of the maneuver node, in meters per second. Changes as the node is executed. This is equivalent to the delta-v reported in-game.

Tuple<double, double, double> BurnVector (*ReferenceFrame* referenceFrame = None)

Returns a vector whose direction is the direction of the maneuver node burn, and whose magnitude is the delta-v of the burn in m/s.

Parameters

Note: Does not change when executing the maneuver node. See [Node.RemainingBurnVector](#).

Tuple<double, double, double> RemainingBurnVector (*ReferenceFrame* referenceFrame = None)

Returns a vector whose direction is the direction of the maneuver node burn, and whose magnitude is the delta-v of the burn in m/s. The direction and magnitude change as the burn is executed.

Parameters

double UT { get; set; }

The universal time at which the maneuver will occur, in seconds.

double TimeTo { get; }

The time until the maneuver node will be encountered, in seconds.

Orbit Orbit { get; }

The orbit that results from executing the maneuver node.

void Remove ()

Removes the maneuver node.

ReferenceFrame ReferenceFrame { get; }

Gets the reference frame that is fixed relative to the maneuver node's burn.

- The origin is at the position of the maneuver node.
- The y-axis points in the direction of the burn.
- The x-axis and z-axis point in arbitrary but fixed directions.

ReferenceFrame OrbitalReferenceFrame { get; }

Gets the reference frame that is fixed relative to the maneuver node, and orientated with the orbital prograde/normal/radial directions of the original orbit at the maneuver node's position.

- The origin is at the position of the maneuver node.
- The x-axis points in the orbital anti-radial direction of the original orbit, at the position of the maneuver node.
- The y-axis points in the orbital prograde direction of the original orbit, at the position of the maneuver node.
- The z-axis points in the orbital normal direction of the original orbit, at the position of the maneuver node.

Tuple<double, double, double> Position (*ReferenceFrame* referenceFrame)
Returns the position vector of the maneuver node in the given reference frame.

Parameters

Tuple<double, double, double> Direction (*ReferenceFrame* referenceFrame)
Returns the unit direction vector of the maneuver nodes burn in the given reference frame.

Parameters

3.3.10 Comms

class Comms

Used to interact with RemoteTech. Created using a call to *Vessel.Comms*.

Note: This class requires *RemoteTech* to be installed.

bool HasLocalControl { **get**; }
Whether the vessel can be controlled locally.

bool HasFlightComputer { **get**; }
Whether the vessel has a RemoteTech flight computer on board.

bool HasConnection { **get**; }
Whether the vessel can receive commands from the KSC or a command station.

bool HasConnectionToGroundStation { **get**; }
Whether the vessel can transmit science data to a ground station.

double SignalDelay { **get**; }
The signal delay when sending commands to the vessel, in seconds.

double SignalDelayToGroundStation { **get**; }
The signal delay between the vessel and the closest ground station, in seconds.

`double SignalDelayToVessel (Vessel other)`

Returns the signal delay between the current vessel and another vessel, in seconds.

Parameters

3.3.11 ReferenceFrame

class ReferenceFrame

Represents a reference frame for positions, rotations and velocities. Contains:

- The position of the origin.
- The directions of the x, y and z axes.
- The linear velocity of the frame.
- The angular velocity of the frame.

Note: This class does not contain any properties or methods. It is only used as a parameter to other functions.

3.3.12 AutoPilot

class AutoPilot

Provides basic auto-piloting utilities for a vessel. Created by calling *Vessel.AutoPilot*.

`void Engage ()`

Engage the auto-pilot.

`void Disengage ()`

Disengage the auto-pilot.

`void Wait ()`

Blocks until the vessel is pointing in the target direction (if set) and has the target roll (if set).

`float Error { get; }`

The error, in degrees, between the direction the ship has been asked to point in and the direction it is pointing in. Returns zero if the auto-pilot has not been engaged, SAS is not enabled, SAS is in stability assist mode, or no target direction is set.

`float RollError { get; }`

The error, in degrees, between the roll the ship has been asked to be in and the actual roll. Returns zero if the auto-pilot has not been engaged or no target roll is set.

`ReferenceFrame ReferenceFrame { get; set; }`

The reference frame for the target direction (*AutoPilot.TargetDirection*).

`Tuple<double, double, double> TargetDirection { get; set; }`
The target direction. `null` if no target direction is set.

`void TargetPitchAndHeading (float pitch, float heading)`
Set (*AutoPilot.TargetDirection*) from a pitch and heading angle.

Parameters

- **pitch** – Target pitch angle, in degrees between -90° and +90°.
- **heading** – Target heading angle, in degrees between 0° and 360°.

`float TargetRoll { get; set; }`
The target roll, in degrees. `NaN` if no target roll is set.

`bool SAS { get; set; }`
The state of SAS.

Note: Equivalent to *Control.SAS*

`SASMode SASMode { get; set; }`
The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

Note: Equivalent to *Control.SASMode*

`float RotationSpeedMultiplier { get; set; }`
Target rotation speed multiplier. Defaults to 1.

`float MaxRotationSpeed { get; set; }`
Maximum target rotation speed. Defaults to 1.

`float RollSpeedMultiplier { get; set; }`
Target roll speed multiplier. Defaults to 1.

`float MaxRollSpeed { get; set; }`
Maximum target roll speed. Defaults to 1.

`void SetPIDParameters (float Kp = 1.0, float Ki = 0.0, float Kd = 0.0)`
Sets the gains for the rotation rate PID controller.

Parameters

- **Kp** – Proportional gain.
- **Ki** – Integral gain.
- **Kd** – Derivative gain.

3.3.13 Geometry Types

class Vector3

3-dimensional vectors are represented as a 3-tuple.

For example:

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

class VectorExample {
    public static void Main () {
        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);
    }
}
```

class Quaternion

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

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

class QuaternionExample {
    public static void Main () {
        var connection = new Connection ();
        var spaceCenter = connection.SpaceCenter ();
        var vessel = spaceCenter.ActiveVessel;
        Tuple<double, double, double, double> q = vessel.Flight ().Rotation;
        Console.WriteLine (q.Item1 + ", " + q.Item2 + ", " + q.Item3 + ", " + q.Item4);
    }
}
```

3.4 InfernalRobotics API

Provides RPCs to interact with the [InfernalRobotics](#) mod. Provides the following classes:

3.4.1 InfernalRobotics

class InfernalRobotics

This service provides functionality to interact with the [InfernalRobotics](#) mod.

`IList<ControlGroup> ServoGroups { get; }`

A list of all the servo groups in the active vessel.

ControlGroup **ServoGroupWithName** (*string name*)

Returns the servo group with the given *name* or `null` if none exists. If multiple servo groups have the same name, only one of them is returned.

Parameters

- **name** – Name of servo group to find.

Servo **ServoWithName** (*string name*)

Returns the servo with the given *name*, from all servo groups, or `null` if none exists. If multiple servos have the same name, only one of them is returned.

Parameters

- **name** – Name of the servo to find.

3.4.2 ControlGroup

class ControlGroup

A group of servos, obtained by calling *InfernalRobotics.ServoGroups* or *InfernalRobotics.ServoGroupWithName*. Represents the “Servo Groups” in the Infernal-Robotics UI.

string **Name** { **get**; **set**; }

The name of the group.

string **ForwardKey** { **get**; **set**; }

The key assigned to be the “forward” key for the group.

string **ReverseKey** { **get**; **set**; }

The key assigned to be the “reverse” key for the group.

float **Speed** { **get**; **set**; }

The speed multiplier for the group.

bool **Expanded** { **get**; **set**; }

Whether the group is expanded in the Infernal-Robotics UI.

ICollection<Servo> **Servos** { **get**; }

The servos that are in the group.

Servo **ServoWithName** (*string name*)

Returns the servo with the given *name* from this group, or `null` if none exists.

Parameters

- **name** – Name of servo to find.

void **MoveRight** ()

Moves all of the servos in the group to the right.


```
void MoveLeft ()
    Moves all of the servos in the group to the left.

void MoveCenter ()
    Moves all of the servos in the group to the center.

void MoveNextPreset ()
    Moves all of the servos in the group to the next
    preset.

void MovePrevPreset ()
    Moves all of the servos in the group to the previous
    preset.

void Stop ()
    Stops the servos in the group.
```

3.4.3 Servo

```
class Servo
    Represents a servo. Obtained using
    ControlGroup.Servos, ControlGroup.ServoWithName or
    InfernalRobotics.ServoWithName.

string Name { get; set; }
    The name of the servo.

bool Highlight { set; }
    Whether the servo should be highlighted in-game.

float Position { get; }
    The position of the servo.

float MinConfigPosition { get; }
    The minimum position of the servo, specified by the
    part configuration.

float MaxConfigPosition { get; }
    The maximum position of the servo, specified by
    the part configuration.

float MinPosition { get; set; }
    The minimum position of the servo, specified by the
    in-game tweak menu.

float MaxPosition { get; set; }
    The maximum position of the servo, specified by
    the in-game tweak menu.

float ConfigSpeed { get; }
    The speed multiplier of the servo, specified by the
    part configuration.

float Speed { get; set; }
    The speed multiplier of the servo, specified by the
    in-game tweak menu.

float CurrentSpeed { get; set; }
    The current speed at which the servo is moving.
```

`float Acceleration { get; set; }`
The current speed multiplier set in the UI.

`bool IsMoving { get; }`
Whether the servo is moving.

`bool IsFreeMoving { get; }`
Whether the servo is freely moving.

`bool IsLocked { get; set; }`
Whether the servo is locked.

`bool IsAxisInverted { get; set; }`
Whether the servos axis is inverted.

`void MoveRight ()`
Moves the servo to the right.

`void MoveLeft ()`
Moves the servo to the left.

`void MoveCenter ()`
Moves the servo to the center.

`void MoveNextPreset ()`
Moves the servo to the next preset.

`void MovePrevPreset ()`
Moves the servo to the previous preset.

`void MoveTo (float position, float speed)`
Moves the servo to *position* and sets the speed multiplier to *speed*.

Parameters

- **position** – The position to move the servo to.
- **speed** – Speed multiplier for the movement.

`void Stop ()`
Stops the servo.

3.4.4 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 KRPC.Client;
using KRPC.Client.Services.InfernalRobotics;
using System;
using System.Threading;
using System.Net;

class IR {
    public static void Main () {
        var connection = new Connection (name: "InfernalRobotics Example");
        var ir = connection.InfernalRobotics ();
```

```

var group = ir.ServoGroupWithName ("MyGroup");
if (group == null) {
    Console.WriteLine ("Group not found");
    return;
}

foreach (var servo in group.Servos)
    Console.WriteLine (servo.Name + " " + servo.Position);

group.MoveRight ();
Thread.Sleep (1000);
group.Stop ();
}

```

3.5 Kerbal Alarm Clock API

Provides RPCs to interact with the [Kerbal Alarm Clock](#) mod. Provides the following classes:

3.5.1 KerbalAlarmClock

class KerbalAlarmClock

This service provides functionality to interact with the [Kerbal Alarm Clock](#) mod.

[IList<Alarm>](#) **Alarms** { get; }

A list of all the alarms.

[Alarm](#) **AlarmWithName** (*string name*)

Get the alarm with the given *name*, or null if no alarms have that name. If more than one alarm has the name, only returns one of them.

Parameters

- **name** – Name of the alarm to search for.

[IList<Alarm>](#) **AlarmsWithType** (*AlarmType type*)

Get a list of alarms of the specified *type*.

Parameters

- **type** – Type of alarm to return.

[Alarm](#) **CreateAlarm** (*AlarmType type, string name, double ut*)

Create a new alarm and return it.

Parameters

- **type** – Type of the new alarm.
- **name** – Name of the new alarm.
- **ut** – Time at which the new alarm should trigger.

3.5.2 Alarm

class Alarm

Represents an alarm. Obtained by calling `KerbAlArAlmClock.AlArms`, `KerbAlArAlmClock.AlArmAWithNAme` or `KerbAlArAlmClock.AlArmsWithType`.

AlarmAction **Action** { **get**; **set**; }

The action that the alarm triggers.

double Margin { **get**; **set**; }

The number of seconds before the event that the alarm will fire.

double Time { **get**; **set**; }

The time at which the alarm will fire.

AlarmType **Type** { **get**; }

The type of the alarm.

string ID { **get**; }

The unique identifier for the alarm.

string Name { **get**; **set**; }

The short name of the alarm.

string Notes { **get**; **set**; }

The long description of the alarm.

double Remaining { **get**; }

The number of seconds until the alarm will fire.

bool Repeat { **get**; **set**; }

Whether the alarm will be repeated after it has fired.

double RepeatPeriod { **get**; **set**; }

The time delay to automatically create an alarm after it has fired.

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

The vessel that the alarm is attached to.

CelestialBody **XferOriginBody** { **get**; **set**; }

The celestial body the vessel is departing from.

CelestialBody **XferTargetBody** { **get**; **set**; }

The celestial body the vessel is arriving at.

void Remove ()

Removes the alarm.

3.5.3 AlarmType

enum AlarmType

The type of an alarm.

Raw

An alarm for a specific date/time or a specific period in the future.

Maneuver

An alarm based on the next maneuver node on the current ships flight path. This node will be stored and can be restored when you come back to the ship.

ManeuverAuto

See *AlarmType.Maneuver*.

Apoapsis

An alarm for furthest part of the orbit from the planet.

Periapsis

An alarm for nearest part of the orbit from the planet.

AscendingNode

Ascending node for the targeted object, or equatorial ascending node.

DescendingNode

Descending node for the targeted object, or equatorial descending node.

Closest

An alarm based on the closest approach of this vessel to the targeted vessel, some number of orbits into the future.

Contract

An alarm based on the expiry or deadline of contracts in career modes.

ContractAuto

See *AlarmType.Contract*.

Crew

An alarm that is attached to a crew member.

Distance

An alarm that is triggered when a selected target comes within a chosen distance.

EarthTime

An alarm based on the time in the “Earth” alternative Universe (aka the Real World).

LaunchRendezvous

An alarm that fires as your landed craft passes under the orbit of your target.

SOIChange

An alarm manually based on when the next SOI point is on the flight path or set to continually monitor the active flight path and add alarms as it detects SOI changes.

SOIChangeAuto

See *AlarmType.SOIChange*.

Transfer

An alarm based on Interplanetary Transfer Phase

Angles, i.e. when should I launch to planet X?
Based on Kosmo Not's post and used in Olex's
Calculator.

TransferModelled

See *AlarmType.Transfer*.

3.5.4 AlarmAction

enum AlarmAction

The action performed by an alarm when it fires.

DoNothing

Don't do anything at all...

DoNothingDeleteWhenPassed

Don't do anything, and delete the alarm.

KillWarp

Drop out of time warp.

KillWarpOnly

Drop out of time warp.

MessageOnly

Display a message.

PauseGame

Pause the game.

3.5.5 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 KRPC.Client;
using KRPC.Client.Services.SpaceCenter;
using KRPC.Client.Services.KerbalAlarmClock;
using System;
using System.Net;

class KAC {
    public static void Main () {
        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;
    }
}
```

4.1 C++ Client

This client provides functionality to interact with a kRPC server from programs written in C++. It can be [downloaded from GitHub](#).

4.1.1 Installing the Library

Installing 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 [via the ASIO website](#).

Protobuf version 3 is also required, and can be [downloaded from GitHub](#). Installation instructions [can be found here](#).

Note: The version of protobuf currently provided in Ubuntu's apt repositories is version 2. This will *not* work with kRPC.

Install 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
```

Install 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
```

Install manually

The library is fairly simple to build manually if you can't use the configure script or CMake. The headers are in the `include` folder and the source files are in `src`.

4.1.2 Using the Library

kRPC programs need to be compiled with C++ 2011 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 <krpc.hpp>
#include <krpc/services/krpc.hpp>
#include <iostream>

int main() {
    krpc::Client 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 to force GCC to use the correct version of the library.

Connecting to the Server

To connect to a server, use the `krpc::connect()` function. This returns a client object 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, including a descriptive name for the client, as follows:

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

int main() {
    krpc::Client conn = krpc::connect("Remote example", "my.domain.name", 1000, 1001);
    krpc::services::KRPC krpc(&conn);
    std::cout << krpc.get_status().version() << std::endl;
}
```


Interacting with the Server

kRPC groups remote procedures into services. 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 connects to the server, instantiates the SpaceCenter service and outputs the name of the active vessel:

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

using namespace krpc::services;

int main() {
    krpc::Client conn = krpc::connect("Vessel Name");
    SpaceCenter sc(&conn);
    SpaceCenter::Vessel vessel = sc.active_vessel();
    std::cout << vessel.name() << std::endl;
}
```

Streaming Data from the Server

A stream repeatedly executes a function on the server, with a fixed set of argument values. It provides a more efficient way of repeatedly getting the result of a function, avoiding the network overhead of having to invoke it directly.

For example, consider the following loop that continuously prints out the position of the active vessel. This loop incurs significant communication overheads, as the `vessel.position()` function is called repeatedly.

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

using namespace krpc::services;

int main() {
    krpc::Client conn = krpc::connect();
    KRPC krpc(&conn);
    SpaceCenter sc(&conn);
    SpaceCenter::Vessel vessel = sc.active_vessel();
    SpaceCenter::ReferenceFrame refframe = vessel.orbit().body().reference_frame();
    while (true) {
        std::tuple<double, double, double> pos = vessel.position(refframe);
        std::cout << std::get<0>(pos) << ", "
                  << std::get<1>(pos) << ", "
                  << std::get<2>(pos) << std::endl;
    }
}
```

The following code achieves the same thing, but is far more efficient. It calls `vessel.position_stream()` once at the start of the program to create a stream, and then repeatedly gets the position from the stream.

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

```

#include <iostream>

using namespace krpc::services;

int main() {
    krpc::Client conn = krpc::connect();
    KRPC krpc(&conn);
    SpaceCenter sc(&conn);
    SpaceCenter::Vessel vessel = sc.active_vessel();
    SpaceCenter::ReferenceFrame refframe = vessel.orbit().body().reference_frame();
    krpc::Stream<std::tuple<double, double, double>> pos_stream = vessel.position_stream(refframe);
    while (true) {
        std::tuple<double, double, double> pos = pos_stream();
        std::cout << std::get<0>(pos) << ", "
                  << std::get<1>(pos) << ", "
                  << std::get<2>(pos) << std::endl;
    }
}

```

A stream can be created for any function call (except property setters) by adding `_stream` to the end of the function's name. This returns a stream object of type `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<T>::operator()()`. A stream can be stopped and removed from the server by calling `krpc::Stream<T>::remove()` on the stream object. All of a client's streams are automatically stopped when it disconnects.

4.1.3 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, for example, in the client connection dialog on 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.
- **stream_port** (`unsigned int`) – The port number of the Stream Server. Defaults to 50001. Set it to 0 to disable connection to the stream server.

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()`.

class KRPC

This class provides access to the basic server functionality provided by the KRPC service. Most of this functionality is used internally by the client (for example to create and remove streams) and therefore does not need to be used directly from application code. The only exception that may be useful is `KRPC::get_status()`.

KRPC (`krpc::Client *client`)

Construct an instance of this service from the given `krpc::Client` object.

`krpc::schema::Status` **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:

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

int main() {
    krpc::Client conn = krpc::connect("Remote example", "my.domain.name", 1000, 1001);
    krpc::services::KRPC krpc(&conn);
    std::cout << krpc.get_status().version() << std::endl;
}
```

Or to get the rate at which the server is sending and receiving data over the network:

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

int main() {
    krpc::Client conn = krpc::connect();
    krpc::services::KRPC krpc(&conn);
    krpc::schema::Status status = krpc.get_status();
    std::cout << "Data in = " << (status.bytes_read_rate()/1024.0) << " KB/s" << std::endl;
    std::cout << "Data out = " << (status.bytes_written_rate()/1024.0) << " KB/s" << std::endl;
}
```

class `Stream`<T>

A stream object. Streams are created by calling a function with `_stream` appended to its name.

T operator () ()

Get the most recently received value from the stream.

void remove ()

Remove the stream from the server.

4.2 KRPC API

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.

`krpc::schema::Status` **get_status** ()

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

`krpc::schema::Services` **get_services** ()

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

`KRPC::GameScene` **current_game_scene** ()

Get the current game scene.

`uint32_t` **add_stream** (`krpc::schema::Request request`)

Add a streaming request and return its identifier.

Parameters

Note: Do not call this method from client code. Use *streams* provided by the C++ client library.

void **remove_stream** (uint32_t *id*)

Remove a streaming request.

Parameters

Note: Do not call this method from client code. Use *streams* provided by the C++ client library.

enum struct GameScene

The game scene. See *KRPC::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.

4.3 SpaceCenter API

4.3.1 SpaceCenter

class **SpaceCenter** : public *krcp::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 (*krcp::Client* **client*)

Construct an instance of this service.

SpaceCenter::Vessel **active_vessel** ()

void **set_active_vessel** (*SpaceCenter::Vessel* *value*)

The currently active vessel.

std::vector<*SpaceCenter::Vessel*> **vessels** ()

A list of all the vessels in the game.

std::map<std::string, *SpaceCenter::CelestialBody*> **bodies** ()

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

SpaceCenter::CelestialBody **target_body** ()

void **set_target_body** (*SpaceCenter::CelestialBody* *value*)

The currently targeted celestial body.

SpaceCenter::Vessel **target_vessel** ()

void **set_target_vessel** (*SpaceCenter::Vessel value*)
The currently targeted vessel.

SpaceCenter::DockingPort **target_docking_port** ()

void **set_target_docking_port** (*SpaceCenter::DockingPort value*)
The currently targeted docking port.

void **clear_target** ()
Clears the current target.

void **launch_vessel_from_vab** (std::string *name*)
Launch a new vessel from the VAB onto the launchpad.

Parameters

- **name** – Name of the vessel’s craft file.

void **launch_vessel_from_sph** (std::string *name*)
Launch a new vessel from the SPH onto the runway.

Parameters

- **name** – Name of the vessel’s craft file.

double **ut** ()
The current universal time in seconds.

float **g** ()
The value of the [gravitational constant](#) G in $N(m/kg)^2$.

SpaceCenter::WarpMode **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.

float **warp_rate** ()
The current warp rate. This is the rate at which time is passing for either on-rails or physical time warp. For example, a value of 10 means time is passing 10x faster than normal. Returns 1 if time warp is not active.

float **warp_factor** ()
The current warp factor. This is the index of the rate at which time is passing for either regular “on-rails” or physical time warp. Returns 0 if time warp is not active. When in on-rails time warp, this is equal to *SpaceCenter::rails_warp_factor()*, and in physics time warp, this is equal to *SpaceCenter::physics_warp_factor()*.

int32_t **rails_warp_factor** ()

void **set_rails_warp_factor** (int32_t *value*)
The time warp rate, using regular “on-rails” time warp. A value between 0 and 7 inclusive. 0 means no time warp. Returns 0 if physical time warp is active. If requested time warp factor cannot be set, it will be set to the next lowest possible value. For example, if the vessel is too close to a planet. See [the KSP wiki](#) for details.

int32_t **physics_warp_factor** ()

void **set_physics_warp_factor** (int32_t *value*)
The physical time warp rate. A value between 0 and 3 inclusive. 0 means no time warp. Returns 0 if regular “on-rails” time warp is active.

bool **can_rails_warp_at** (int32_t *factor* = 1)
Returns `true` if regular “on-rails” time warp can be used, at the specified warp *factor*. The maximum

time warp rate is limited by various things, including how close the active vessel is to a planet. See [the KSP wiki](#) for details.

Parameters

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

`int32_t maximum_rails_warp_factor()`

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

`void warp_to(double ut, float max_rails_rate = 100000.0, float max_physics_rate = 2.0)`

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

Parameters

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

Returns When the time warp is complete.

`std::tuple<double, double, double> transform_position` (`std::tuple<double, double, double> position`, `SpaceCenter::ReferenceFrame from`, `SpaceCenter::ReferenceFrame to`)

Converts a position vector from one reference frame to another.

Parameters

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

Returns The corresponding position vector in reference frame *to*.

`std::tuple<double, double, double> transform_direction` (`std::tuple<double, double, double> direction`, `SpaceCenter::ReferenceFrame from`, `SpaceCenter::ReferenceFrame to`)

Converts a direction vector from one reference frame to another.

Parameters

- **direction** – Direction vector in reference frame *from*.
- **from** – The reference frame that the direction vector is in.
- **to** – The reference frame to convert the direction vector to.

Returns The corresponding direction vector in reference frame *to*.

`std::tuple<double, double, double, double> transform_rotation` (`std::tuple<double, double, double, double> rotation`, `SpaceCenter::ReferenceFrame from`, `SpaceCenter::ReferenceFrame to`)

Converts a rotation from one reference frame to another.

Parameters

- **rotation** – Rotation in reference frame *from*.
- **from** – The reference frame that the rotation is in.
- **to** – The corresponding rotation in reference frame *to*.

Returns The corresponding rotation in reference frame *to*.

std::tuple<double, double, double> **transform_velocity** (std::tuple<double, double, double> *position*, std::tuple<double, double, double> *velocity*, *SpaceCenter::ReferenceFrame from, SpaceCenter::ReferenceFrame to*)

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

Parameters

- **position** – Position vector in reference frame *from*.
- **velocity** – Velocity vector in reference frame *from*.
- **from** – The reference frame that the position and velocity vectors are in.
- **to** – The reference frame to convert the velocity vector to.

Returns The corresponding velocity in reference frame *to*.

bool **far_available** ()
Whether *Ferram Aerospace Research* is installed.

bool **remote_tech_available** ()
Whether *RemoteTech* is installed.

void **draw_direction** (std::tuple<double, double, double> *direction*, *SpaceCenter::ReferenceFrame reference_frame*, std::tuple<double, double, double> *color*, float *length* = 10.0)
Draw a direction vector on the active vessel.

Parameters

- **direction** – Direction to draw the line in.
- **reference_frame** – Reference frame that the direction is in.
- **color** – The color to use for the line, as an RGB color.
- **length** – The length of the line. Defaults to 10.

void **draw_line** (std::tuple<double, double, double> *start*, std::tuple<double, double, double> *end*, *SpaceCenter::ReferenceFrame reference_frame*, std::tuple<double, double, double> *color*)
Draw a line.

Parameters

- **start** – Position of the start of the line.
- **end** – Position of the end of the line.
- **reference_frame** – Reference frame that the position are in.
- **color** – The color to use for the line, as an RGB color.

void **clear_drawing** ()
Remove all directions and lines currently being drawn.

enum struct WarpMode

Returned by *SpaceCenter::WarpMode*

enumerator rails

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

enumerator physics

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

enumerator none

Time warp is not active.

4.3.2 Vessel

class Vessel

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources.

std::string **name** ()

void **set_name** (std::string value)

The name of the vessel.

SpaceCenter::VesselType **type** ()

void **set_type** (*SpaceCenter::VesselType* value)

The type of the vessel.

SpaceCenter::VesselSituation **situation** ()

The situation the vessel is in.

double **met** ()

The mission elapsed time in seconds.

SpaceCenter::Flight **flight** (*SpaceCenter::ReferenceFrame* 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** – Reference frame. Defaults to the vessel’s surface reference frame (*SpaceCenter::Vessel::surface_reference_frame*()).

Note: When this is called with no arguments, the vessel’s surface reference frame is used. This reference frame moves with the vessel, therefore velocities and speeds returned by the flight object will be zero. See the *reference frames tutorial* for examples of getting the *orbital speed* and *surface speed* of a vessel.

SpaceCenter::Vessel **target** ()

void **set_target** (*SpaceCenter::Vessel* value)

The target vessel. NULL if there is no target. When setting the target, the target cannot be the current vessel.

SpaceCenter::Orbit **orbit** ()

The current orbit of the vessel.

SpaceCenter::Control **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.

SpaceCenter::AutoPilot **auto_pilot** ()

An *SpaceCenter::AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

SpaceCenter::Resources **resources** ()

A *SpaceCenter::Resources* object, that can be used to get information about resources stored in the vessel.

SpaceCenter::Resources **resources_in_decouple_stage** (int32_t *stage*, bool *cumulative* = True)

Returns a *SpaceCenter::Resources* object, that can be used to get information about resources stored in a given *stage*.

Parameters

- **stage** – Get resources for parts that are decoupled in this stage.
- **cumulative** – When `false`, returns the resources for parts decoupled in just the given stage. When `true` returns the resources decoupled in the given stage and all subsequent stages combined.

Note: For details on stage numbering, see the discussion on [Staging](#).

SpaceCenter::Parts **parts** ()

A *SpaceCenter::Parts* object, that can be used to interact with the parts that make up this vessel.

SpaceCenter::Comms **comms** ()

A *SpaceCenter::Comms* object, that can be used to interact with RemoteTech for this vessel.

Note: Requires [RemoteTech](#) to be installed.

float **mass** ()

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

float **dry_mass** ()

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

float **thrust** ()

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

float **available_thrust** ()

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

float **max_thrust** ()

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

float **max_vacuum_thrust** ()

The total maximum thrust that can be produced by the vessel's active engines when the vessel is in a vacuum, in Newtons. This is computed by summing *SpaceCenter::Engine::max_vacuum_thrust* () for every active engine.

float **specific_impulse** ()

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

float **vacuum_specific_impulse** ()

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

float **kerbin_sea_level_specific_impulse** ()

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

SpaceCenter::ReferenceFrame **reference_frame** ()

The reference frame that is fixed relative to the vessel, and orientated with the vessel.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel.
- The x-axis points out to the right of the vessel.
- The y-axis points in the forward direction of the vessel.
- The z-axis points out of the bottom off the vessel.

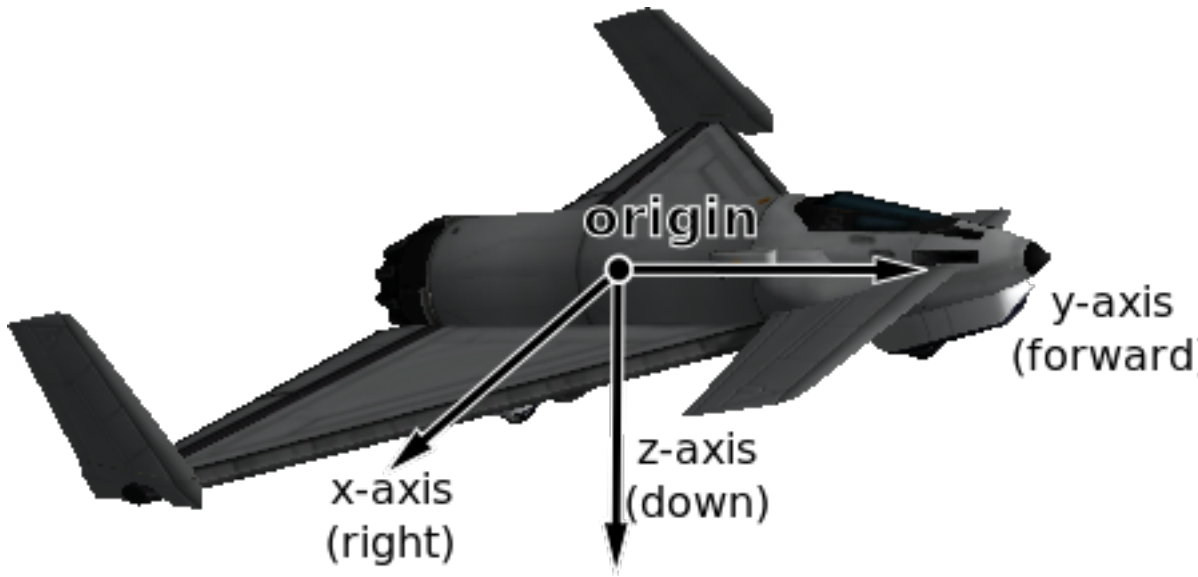


Fig. 4.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

SpaceCenter::ReferenceFrame **orbital_reference_frame** ()

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

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

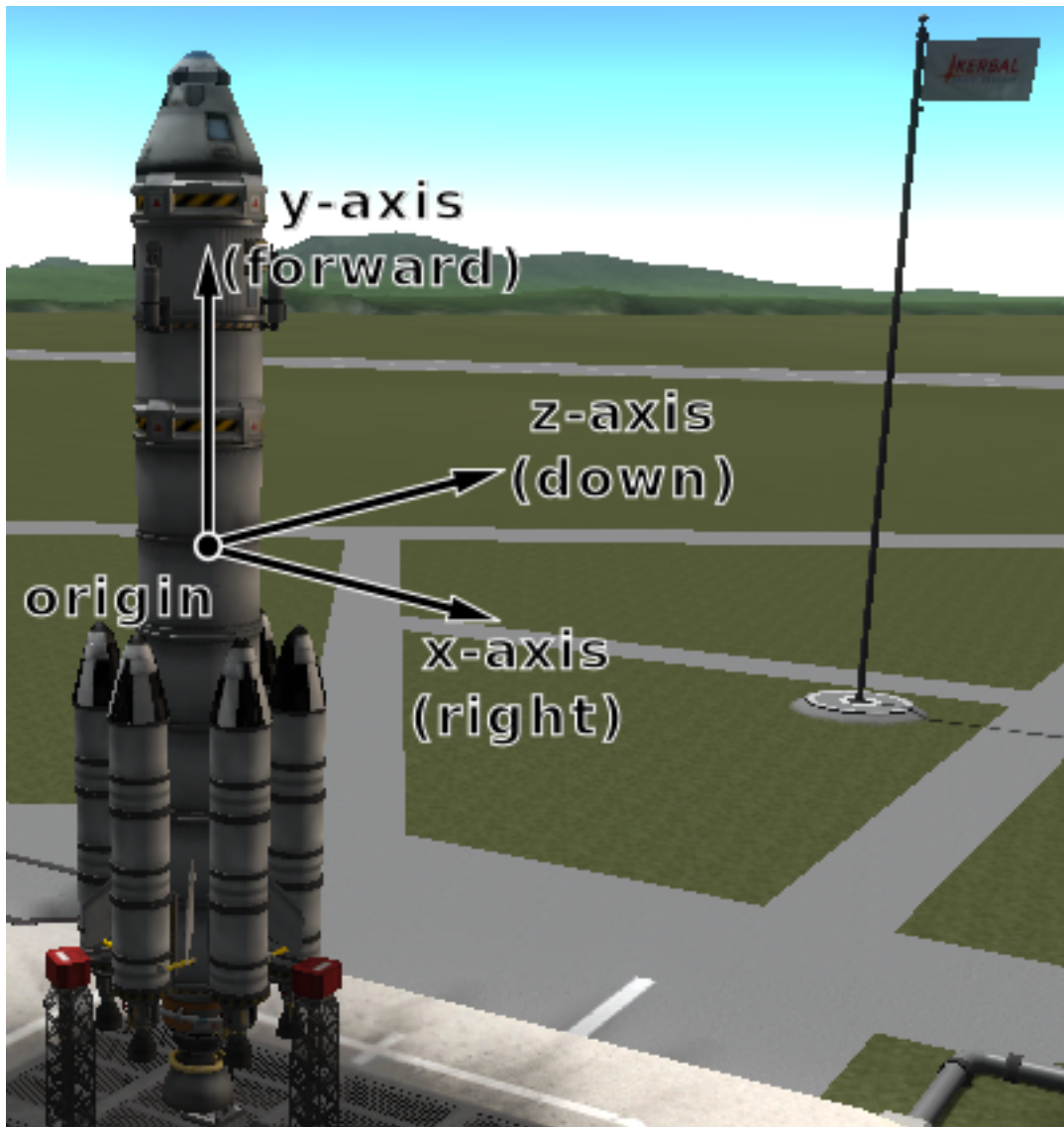


Fig. 4.2: Vessel reference frame origin and axes for the Kerbal-X rocket

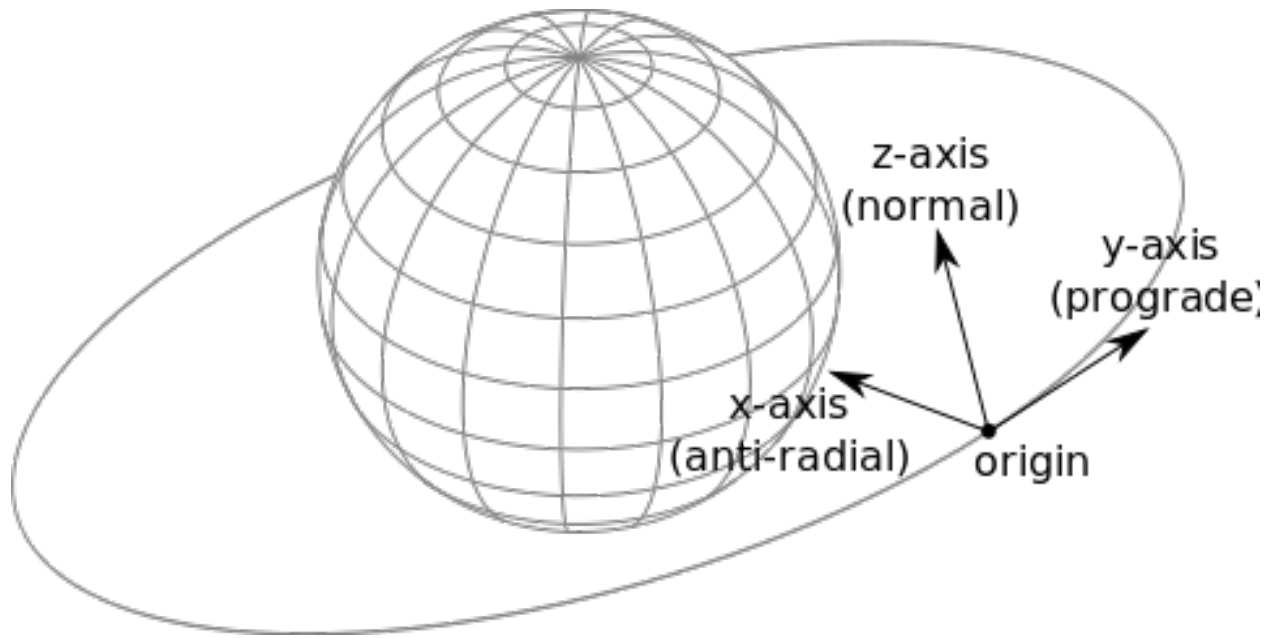


Fig. 4.3: Vessel orbital reference frame origin and axes

SpaceCenter::ReferenceFrame **surface_reference_frame** ()

The reference frame that is fixed relative to the vessel, and orientated with the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the north and up directions on the surface of the body.
- The x-axis points in the [zenith](#) direction (upwards, normal to the body being orbited, from the center of the body towards the center of mass of the vessel).
- The y-axis points northwards towards the [astronomical horizon](#) (north, and tangential to the surface of the body – the direction in which a compass would point when on the surface).
- The z-axis points eastwards towards the [astronomical horizon](#) (east, and tangential to the surface of the body – east on a compass when on the surface).

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

SpaceCenter::ReferenceFrame **surface_velocity_reference_frame** ()

The reference frame that is fixed relative to the vessel, and orientated with the velocity vector of the vessel relative to the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel’s velocity vector.
- The y-axis points in the direction of the vessel’s velocity vector, relative to the surface of the body being orbited.
- The z-axis is in the plane of the [astronomical horizon](#).
- The x-axis is orthogonal to the other two axes.

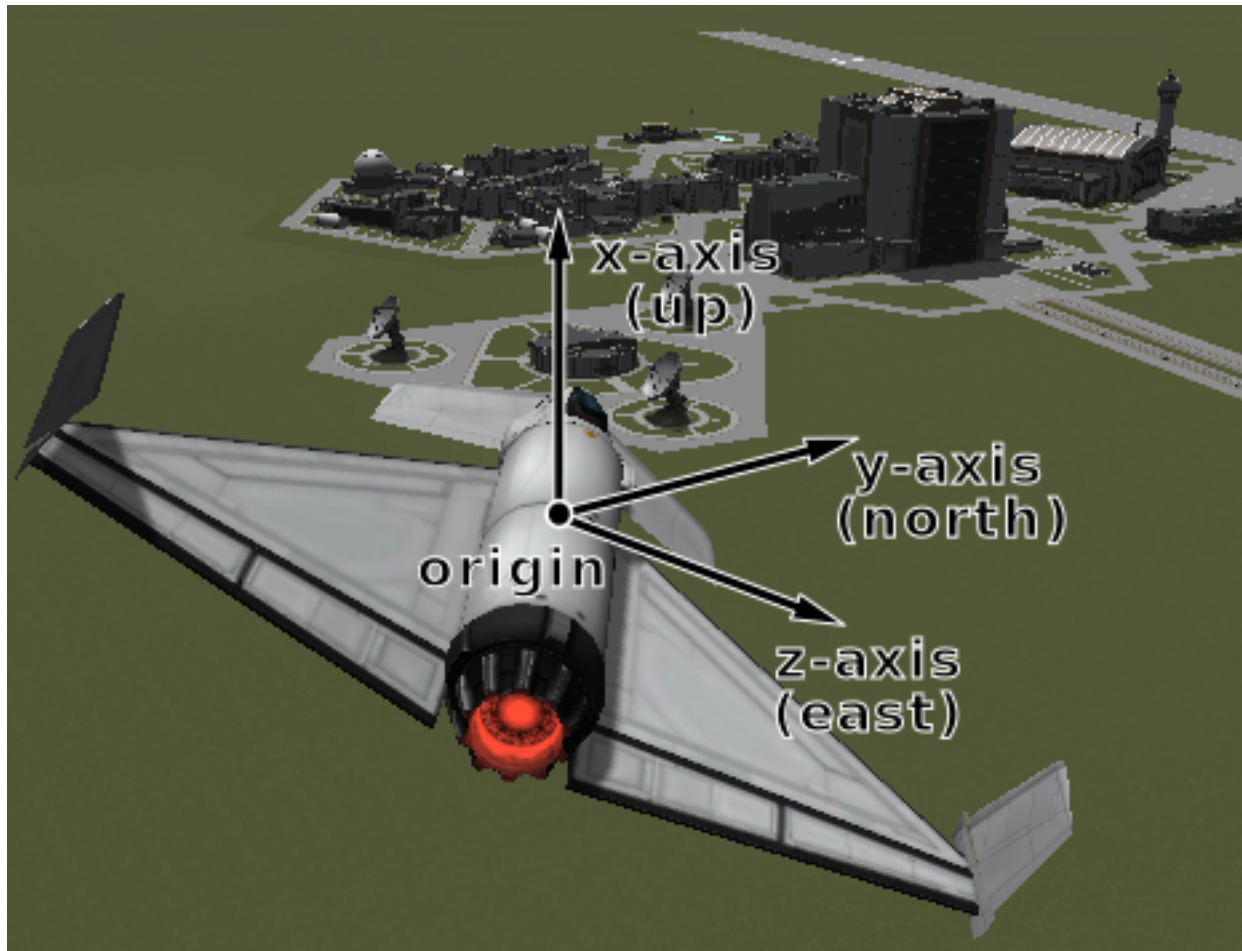


Fig. 4.4: Vessel surface reference frame origin and axes

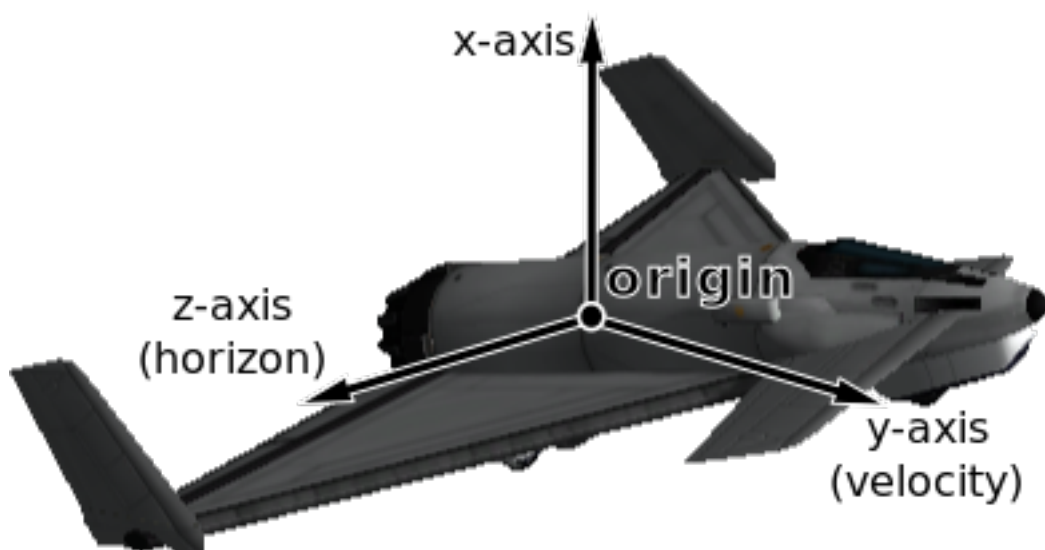


Fig. 4.5: Vessel surface velocity reference frame origin and axes

`std::tuple<double, double, double> position (SpaceCenter::ReferenceFrame reference_frame)`

Returns the position vector of the center of mass of the vessel in the given reference frame.

Parameters

`std::tuple<double, double, double> velocity (SpaceCenter::ReferenceFrame reference_frame)`

Returns the velocity vector of the center of mass of the vessel in the given reference frame.

Parameters

`std::tuple<double, double, double, double> rotation (SpaceCenter::ReferenceFrame reference_frame)`

Returns the rotation of the center of mass of the vessel in the given reference frame.

Parameters

`std::tuple<double, double, double> direction (SpaceCenter::ReferenceFrame reference_frame)`

Returns the direction in which the vessel is pointing, as a unit vector, in the given reference frame.

Parameters

`std::tuple<double, double, double> angular_velocity (SpaceCenter::ReferenceFrame reference_frame)`

Returns the angular velocity of the vessel in the given reference frame. The magnitude of the returned vector is the rotational speed in radians per second, and the direction of the vector indicates the axis of rotation (using the right hand rule).

Parameters

enum struct VesselType

See `SpaceCenter::Vessel::type()`.

enumerator ship

Ship.

enumerator station

Station.

enumerator lander

Lander.

enumerator probe

Probe.

enumerator rover

Rover.

enumerator base

Base.

enumerator debris

Debris.

enum struct VesselSituation

See `SpaceCenter::Vessel::situation()`.

enumerator docked

Vessel is docked to another.

enumerator escaping

Escaping.

enumerator flying

Vessel is flying through an atmosphere.

enumerator landed

Vessel is landed on the surface of a body.

enumerator orbiting

Vessel is orbiting a body.

enumerator pre_launch

Vessel is awaiting launch.

enumerator splashed

Vessel has splashed down in an ocean.

enumerator sub_orbital

Vessel is on a sub-orbital trajectory.

4.3.3 CelestialBody

class CelestialBody

Represents a celestial body (such as a planet or moon).

std::string **name** ()

The name of the body.

std::vector<SpaceCenter::CelestialBody> **satellites** ()

A list of celestial bodies that are in orbit around this celestial body.

SpaceCenter::Orbit **orbit** ()

The orbit of the body.

float **mass** ()

The mass of the body, in kilograms.

float **gravitational_parameter** ()

The *standard gravitational parameter* of the body in m^3s^{-2} .

float **surface_gravity** ()

The acceleration due to gravity at sea level (mean altitude) on the body, in m/s^2 .

float **rotational_period** ()

The sidereal rotational period of the body, in seconds.

float **rotational_speed** ()

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

float **equatorial_radius** ()

The equatorial radius of the body, in meters.

double **surface_height** (double *latitude*, double *longitude*)

The height of the surface relative to mean sea level at the given position, in meters. When over water this is equal to 0.

Parameters

- **latitude** – Latitude in degrees
- **longitude** – Longitude in degrees

double **bedrock_height** (double *latitude*, double *longitude*)

The height of the surface relative to mean sea level at the given position, in meters. When over water, this is the height of the sea-bed and is therefore a negative value.

Parameters

- **latitude** – Latitude in degrees
- **longitude** – Longitude in degrees

std::tuple<double, double, double> **msl_position** (double *latitude*, double *longitude*, *SpaceCenter::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

std::tuple<double, double, double> **surface_position** (double *latitude*, double *longitude*, *SpaceCenter::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

std::tuple<double, double, double> **bedrock_position** (double *latitude*, double *longitude*, *SpaceCenter::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

float **sphere_of_influence** ()

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

bool **has_atmosphere** ()

true if the body has an atmosphere.

float **atmosphere_depth** ()

The depth of the atmosphere, in meters.

bool **has_atmospheric_oxygen** ()

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

SpaceCenter::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.

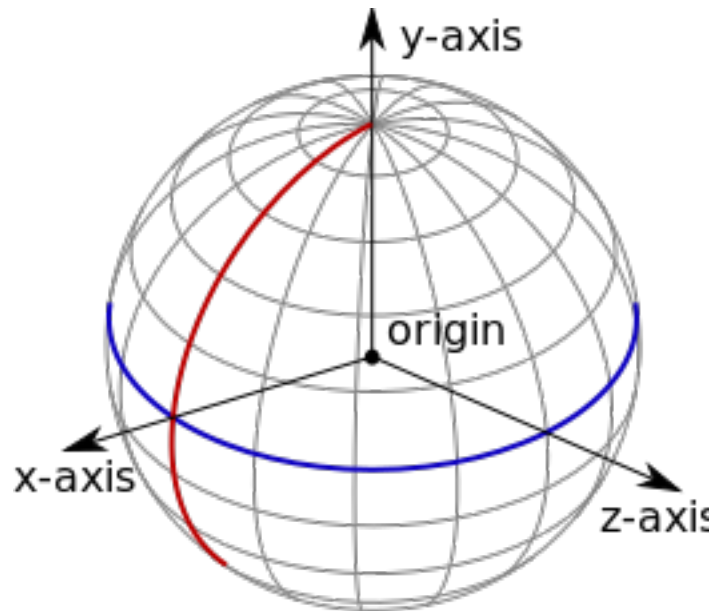


Fig. 4.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

SpaceCenter::ReferenceFrame **non_rotating_reference_frame** ()

The reference frame that is fixed relative to this celestial body, and orientated in a fixed direction (it does not rotate with the body).

- The origin is at the center of the body.
- The axes do not rotate.
- The x-axis points in an arbitrary direction through the equator.
- The y-axis points from the center of the body towards the north pole.
- The z-axis points in an arbitrary direction through the equator.

SpaceCenter::ReferenceFrame **orbital_reference_frame** ()

Gets the reference frame that is fixed relative to this celestial body, but orientated with the body's orbital prograde/normal/radial directions.

- The origin is at the center of the body.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

`std::tuple<double, double, double> position (SpaceCenter::ReferenceFrame reference_frame)`

Returns the position vector of the center of the body in the specified reference frame.

Parameters

`std::tuple<double, double, double> velocity (SpaceCenter::ReferenceFrame reference_frame)`

Returns the velocity vector of the body in the specified reference frame.

Parameters

std::tuple<double, double, double, double> **rotation** (*SpaceCenter::ReferenceFrame* *reference_frame*)

Returns the rotation of the body in the specified reference frame.

Parameters

std::tuple<double, double, double> **direction** (*SpaceCenter::ReferenceFrame* *reference_frame*)

Returns the direction in which the north pole of the celestial body is pointing, as a unit vector, in the specified reference frame.

Parameters

std::tuple<double, double, double> **angular_velocity** (*SpaceCenter::ReferenceFrame* *reference_frame*)

Returns the angular velocity of the body in the specified reference frame. The magnitude of the vector is the rotational speed of the body, in radians per second, and the direction of the vector indicates the axis of rotation, using the right-hand rule.

Parameters

4.3.4 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.

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

float **g_force** ()

The current G force acting on the vessel in m/s^2 .

double **mean_altitude** ()

The altitude above sea level, in meters.

double **surface_altitude** ()

The altitude above the surface of the body or sea level, whichever is closer, in meters.

double **bedrock_altitude** ()

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor.

double **elevation** ()

The elevation of the terrain under the vessel, in meters. This is the height of the terrain above sea level, and is negative when the vessel is over the sea.

double **latitude** ()

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

double **longitude** ()

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

std::tuple<double, double, double> **velocity** ()

The velocity vector of the vessel. The magnitude of the vector is the speed of the vessel in meters per second. The direction of the vector is the direction of the vessels motion.

double **speed** ()

The speed of the vessel in meters per second.

double **horizontal_speed** ()

The horizontal speed of the vessel in meters per second.

double **vertical_speed** ()

The vertical speed of the vessel in meters per second.

std::tuple<double, double, double> **center_of_mass** ()

The position of the center of mass of the vessel.

std::tuple<double, double, double, double> **rotation** ()

The rotation of the vessel.

std::tuple<double, double, double> **direction** ()

The direction vector that the vessel is pointing in.

float **pitch** ()

The pitch angle of the vessel relative to the horizon, in degrees. A value between -90° and $+90^\circ$.

float **heading** ()

The heading angle of the vessel relative to north, in degrees. A value between 0° and 360° .

float **roll** ()

The roll angle of the vessel relative to the horizon, in degrees. A value between -180° and $+180^\circ$.

std::tuple<double, double, double> **prograde** ()

The unit direction vector pointing in the prograde direction.

std::tuple<double, double, double> **retrograde** ()

The unit direction vector pointing in the retrograde direction.

std::tuple<double, double, double> **normal** ()

The unit direction vector pointing in the normal direction.

std::tuple<double, double, double> **anti_normal** ()

The unit direction vector pointing in the anti-normal direction.

std::tuple<double, double, double> **radial** ()

The unit direction vector pointing in the radial direction.

std::tuple<double, double, double> **anti_radial** ()

The unit direction vector pointing in the anti-radial direction.

float **atmosphere_density** ()

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

float **dynamic_pressure** ()

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

Note: Calculated using [KSPs stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

float **static_pressure** ()

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

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

std::tuple<double, double, double> **aerodynamic_force** ()

The total aerodynamic forces acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

std::tuple<double, double, double> **lift** ()

The [aerodynamic lift](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

std::tuple<double, double, double> **drag** ()

The [aerodynamic drag](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

float **speed_of_sound** ()

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

Note: Not available when [Ferram Aerospace Research](#) is installed.

float **mach** ()

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

Note: Not available when [Ferram Aerospace Research](#) is installed.

float **equivalent_air_speed** ()

The [equivalent air speed](#) of the vessel, in *m/s*.

Note: Not available when [Ferram Aerospace Research](#) is installed.

float **terminal_velocity** ()

An estimate of the current terminal velocity of the vessel, in *m/s*. This is the speed at which the drag forces cancel out the force of gravity.

Note: Calculated using [KSPs stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

float **angle_of_attack** ()

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

float **sideslip_angle** ()

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

float **total_air_temperature** ()

The **total air temperature** of the atmosphere around the vessel, in Kelvin. This temperature includes the `SpaceCenter::Flight::static_air_temperature()` and the vessel's kinetic energy.

float **static_air_temperature** ()

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

float **stall_fraction** ()

Gets the current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

Note: Requires [Ferram Aerospace Research](#).

float **drag_coefficient** ()

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

Note: Requires [Ferram Aerospace Research](#).

float **lift_coefficient** ()

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

Note: Requires [Ferram Aerospace Research](#).

float **ballistic_coefficient** ()

Gets the **ballistic coefficient**.

Note: Requires [Ferram Aerospace Research](#).

float **thrust_specific_fuel_consumption** ()

Gets 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 product one newton of thrust.

Note: Requires [Ferram Aerospace Research](#).

4.3.5 Orbit

class **Orbit**

Describes an orbit. For example, the orbit of a vessel, obtained by calling `SpaceCenter::Vessel::orbit()`, or a celestial body, obtained by calling `SpaceCenter::CelestialBody::orbit()`.

`SpaceCenter::CelestialBody` **body** ()

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

double **apoapsis** ()

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

Note: For the apoapsis altitude reported on the in-game map view, use `SpaceCenter::Orbit::apoapsis_altitude()`.

double **periapsis** ()

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

Note: For the periapsis altitude reported on the in-game map view, use `SpaceCenter::Orbit::periapsis_altitude()`.

double **apoapsis_altitude** ()

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

Note: This is equal to `SpaceCenter::Orbit::apoapsis()` minus the equatorial radius of the body.

double **periapsis_altitude** ()

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

Note: This is equal to `SpaceCenter::Orbit::periapsis()` minus the equatorial radius of the body.

double **semi_major_axis** ()

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

double **semi_minor_axis** ()

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

double **radius** ()

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

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

double **speed** ()

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

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

double **period** ()

The orbital period, in seconds.

double **time_to_apoapsis** ()

The time until the object reaches apoapsis, in seconds.

double **time_to_periapsis** ()

The time until the object reaches periapsis, in seconds.

double **eccentricity** ()

The *eccentricity* of the orbit.

double **inclination** ()
 The *inclination* of the orbit, in radians.

double **longitude_of_ascending_node** ()
 The *longitude of the ascending node*, in radians.

double **argument_of_periapsis** ()
 The *argument of periapsis*, in radians.

double **mean_anomaly_at_epoch** ()
 The *mean anomaly at epoch*.

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

double **mean_anomaly** ()
 The *mean anomaly*.

double **eccentric_anomaly** ()
 The *eccentric anomaly*.

static std::tuple<double, double, double> **reference_plane_normal** (*SpaceCenter::ReferenceFrame*
reference_frame)
 The unit direction vector 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

static std::tuple<double, double, double> **reference_plane_direction** (*SpaceCenter::ReferenceFrame*
reference_frame)
 The unit direction vector from which the orbits longitude of ascending node is measured, in the given reference frame.

Parameters

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.

SpaceCenter::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.

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.

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.

bool **sas** ()

void **set_sas** (bool *value*)
 The state of SAS.

Note: Equivalent to `SpaceCenter::AutoPilot::sas()`

SpaceCenter::SASMode **sas_mode** ()

void **set_sas_mode** (*SpaceCenter::SASMode value*)

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

Note: Equivalent to `SpaceCenter::AutoPilot::sas_mode()`

SpaceCenter::SpeedMode **speed_mode** ()

void **set_speed_mode** (*SpaceCenter::SpeedMode value*)

The current *SpaceCenter::SpeedMode* of the navball. This is the mode displayed next to the speed at the top of the navball.

bool **rcs** ()

void **set_rcs** (bool *value*)

The state of RCS.

bool **gear** ()

void **set_gear** (bool *value*)

The state of the landing gear/legs.

bool **lights** ()

void **set_lights** (bool *value*)

The state of the lights.

bool **brakes** ()

void **set_brakes** (bool *value*)

The state of the wheel brakes.

bool **abort** ()

void **set_abort** (bool *value*)

The state of the abort action group.

float **throttle** ()

void **set_throttle** (float *value*)

The state of the throttle. A value between 0 and 1.

float **pitch** ()

void **set_pitch** (float *value*)

The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

float **yaw** ()

void **set_yaw** (float *value*)

The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

float **roll** ()

void **set_roll** (float *value*)

The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

float **forward** ()

void **set_forward** (float *value*)

The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

float **up** ()

void **set_up** (float *value*)

The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

float **right** ()

void **set_right** (float *value*)

The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

float **wheel_throttle** ()

void **set_wheel_throttle** (float *value*)

The state of the wheel throttle. A value between -1 and 1. A value of 1 rotates the wheels forwards, a value of -1 rotates the wheels backwards.

float **wheel_steering** ()

void **set_wheel_steering** (float *value*)

The state of the wheel steering. A value between -1 and 1. A value of 1 steers to the left, and a value of -1 steers to the right.

int32_t **current_stage** ()

The current stage of the vessel. Corresponds to the stage number in the in-game UI.

std::vector<*SpaceCenter::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.

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.

void **set_action_group** (uint32_t *group*, bool *state*)

Sets the state of the given action group (a value between 0 and 9 inclusive).

Parameters

- **group** – A number between 0 and 9 inclusive.

void **toggle_action_group** (uint32_t *group*)

Toggles the state of the given action group.

Parameters

- **group** – A number between 0 and 9 inclusive.

SpaceCenter::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 *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** – Universal time of the maneuver node.
- **prograde** – Delta-v in the prograde direction.
- **normal** – Delta-v in the normal direction.

- **radial** – Delta-v in the radial direction.

`std::vector<SpaceCenter::Node> nodes ()`

Returns a list of all existing maneuver nodes, ordered by time from first to last.

`void remove_nodes ()`

Remove all maneuver nodes.

enum struct SASMode

The behavior of the SAS auto-pilot. See `SpaceCenter::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

See `SpaceCenter::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.

4.3.7 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*
 - *Cargo Bay*
 - *Decoupler*
 - *Docking Port*
 - *Engine*
 - *Fairing*
 - *Intake*
 - *Landing Gear*
 - *Landing Leg*
 - *Launch Clamp*
 - *Light*
 - *Parachute*
 - *Radiator*
 - *Resource Converter*
 - *Resource Harvester*
 - *Reaction Wheel*
 - *Sensor*
 - *Solar Panel*
- *Trees of Parts*
 - *Traversing the Tree*
 - *Attachment Modes*
- *Fuel Lines*
- *Staging*

Parts

class **Parts**

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling `SpaceCenter::Vessel::parts()`.

`std::vector<SpaceCenter::Part> all ()`
 A list of all of the vessels parts.

`SpaceCenter::Part root ()`
 The vessels root part.

Note: See the discussion on *Trees of Parts*.

`SpaceCenter::Part controlling ()`

void `set_controlling (SpaceCenter::Part value)`
 The part from which the vessel is controlled.

`std::vector<SpaceCenter::Part> with_name (std::string name)`
 A list of parts whose `SpaceCenter::Part::name ()` is *name*.

Parameters

`std::vector<SpaceCenter::Part> with_title (std::string title)`
 A list of all parts whose `SpaceCenter::Part::title ()` is *title*.

Parameters

`std::vector<SpaceCenter::Part> with_module (std::string module_name)`

A list of all parts that contain a `SpaceCenter::Module` whose `SpaceCenter::Module::name()` is `module_name`.

Parameters

`std::vector<SpaceCenter::Part> in_stage (int32_t stage)`

A list of all parts that are activated in the given `stage`.

Parameters

Note: See the discussion on [Staging](#).

`std::vector<SpaceCenter::Part> in_decouple_stage (int32_t stage)`

A list of all parts that are decoupled in the given `stage`.

Parameters

Note: See the discussion on [Staging](#).

`std::vector<SpaceCenter::Module> modules_with_name (std::string module_name)`

A list of modules (combined across all parts in the vessel) whose `SpaceCenter::Module::name()` is `module_name`.

Parameters

`std::vector<SpaceCenter::CargoBay> cargo_bays ()`

A list of all cargo bays in the vessel.

`std::vector<SpaceCenter::Decoupler> decouplers ()`

A list of all decouplers in the vessel.

`std::vector<SpaceCenter::DockingPort> docking_ports ()`

A list of all docking ports in the vessel.

`SpaceCenter::DockingPort docking_port_with_name (std::string name)`

The first docking port in the vessel with the given port name, as returned by `SpaceCenter::DockingPort::name()`. Returns NULL if there are no such docking ports.

Parameters

`std::vector<SpaceCenter::Engine> engines ()`

A list of all engines in the vessel.

`std::vector<SpaceCenter::Fairing> fairings ()`

A list of all fairings in the vessel.

`std::vector<SpaceCenter::Intake> intakes ()`

A list of all intakes in the vessel.

`std::vector<SpaceCenter::LandingGear> landing_gear ()`

A list of all landing gear attached to the vessel.

`std::vector<SpaceCenter::LandingLeg> landing_legs ()`

A list of all landing legs attached to the vessel.

`std::vector<SpaceCenter::LaunchClamp> launch_clamps ()`

A list of all launch clamps attached to the vessel.

`std::vector<SpaceCenter::Light> lights ()`
 A list of all lights in the vessel.

`std::vector<SpaceCenter::Parachute> parachutes ()`
 A list of all parachutes in the vessel.

`std::vector<SpaceCenter::Radiator> radiators ()`
 A list of all radiators in the vessel.

`std::vector<SpaceCenter::ReactionWheel> reaction_wheels ()`
 A list of all reaction wheels in the vessel.

`std::vector<SpaceCenter::ResourceConverter> resource_converters ()`
 A list of all resource converters in the vessel.

`std::vector<SpaceCenter::ResourceHarvester> resource_harvesters ()`
 A list of all resource harvesters in the vessel.

`std::vector<SpaceCenter::Sensor> sensors ()`
 A list of all sensors in the vessel.

`std::vector<SpaceCenter::SolarPanel> solar_panels ()`
 A list of all solar panels in the vessel.

Part

class Part

Instances of this class represents a part. A vessel is made of multiple parts. Instances can be obtained by various methods in `SpaceCenter::Parts`.

`std::string name ()`
 Internal name of the part, as used in `part cfg files`. For example “Mark1-2Pod”.

`std::string title ()`
 Title of the part, as shown when the part is right clicked in-game. For example “Mk1-2 Command Pod”.

`double cost ()`
 The cost of the part, in units of funds.

`SpaceCenter::Vessel vessel ()`
 The vessel that contains this part.

`SpaceCenter::Part parent ()`
 The parts parent. Returns NULL if the part does not have a parent. This, in combination with `SpaceCenter::Part::children ()`, can be used to traverse the vessels parts tree.

Note: See the discussion on *Trees of Parts*.

`std::vector<SpaceCenter::Part> 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.

Note: See the discussion on *Trees of Parts*.

`bool axially_attached ()`
 Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns `false`.

Note: See the discussion on [Attachment Modes](#).

bool **radially_attached** ()

Whether the part is radially attached to its parent, i.e. on the side of its parent. If the part has no parent, returns `false`.

Note: See the discussion on [Attachment Modes](#).

int32_t **stage** ()

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

Note: See the discussion on [Staging](#).

int32_t **decouple_stage** ()

The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

Note: See the discussion on [Staging](#).

bool **massless** ()

Whether the part is `massless`.

double **mass** ()

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

double **dry_mass** ()

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

double **impact_tolerance** ()

The impact tolerance of the part, in meters per second.

double **temperature** ()

Temperature of the part, in Kelvin.

double **skin_temperature** ()

Temperature of the skin of the part, in Kelvin.

double **max_temperature** ()

Maximum temperature that the part can survive, in Kelvin.

double **max_skin_temperature** ()

Maximum temperature that the skin of the part can survive, in Kelvin.

float **thermal_mass** ()

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

float **thermal_skin_mass** ()

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

float **thermal_resource_mass** ()

A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

float **thermal_conduction_flux** ()

The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **thermal_convection_flux** ()

The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **thermal_radiation_flux** ()

The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **thermal_internal_flux** ()

The rate at which heat energy is being generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **thermal_skin_to_internal_flux** ()

The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

SpaceCenter::Resources **resources** ()

A *SpaceCenter::Resources* object for the part.

bool **crossfeed** ()

Whether this part is crossfeed capable.

bool **is_fuel_line** ()

Whether this part is a fuel line.

std::vector<*SpaceCenter::Part*> **fuel_lines_from** ()

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

Note: See the discussion on *Fuel Lines*.

std::vector<*SpaceCenter::Part*> **fuel_lines_to** ()

The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

Note: See the discussion on *Fuel Lines*.

std::vector<*SpaceCenter::Module*> **modules** ()

The modules for this part.

SpaceCenter::CargoBay **cargo_bay** ()

A *SpaceCenter::CargoBay* if the part is a cargo bay, otherwise NULL.

SpaceCenter::Decoupler **decoupler** ()

A *SpaceCenter::Decoupler* if the part is a decoupler, otherwise NULL.

SpaceCenter::DockingPort **docking_port** ()

A *SpaceCenter::DockingPort* if the part is a docking port, otherwise NULL.

SpaceCenter::Engine **engine** ()

An *SpaceCenter::Engine* if the part is an engine, otherwise NULL.

SpaceCenter::Fairing **fairing** ()

A *SpaceCenter::Fairing* if the part is a fairing, otherwise NULL.

SpaceCenter::Intake **intake** ()

An *SpaceCenter::Intake* if the part is an intake, otherwise NULL.

SpaceCenter::LandingGear **landing_gear** ()

A *SpaceCenter::LandingGear* if the part is a landing gear , otherwise NULL.

SpaceCenter::LandingLeg **landing_leg** ()

A *SpaceCenter::LandingLeg* if the part is a landing leg, otherwise NULL.

SpaceCenter::LaunchClamp **launch_clamp** ()

A *SpaceCenter::LaunchClamp* if the part is a launch clamp, otherwise NULL.

SpaceCenter::Light **light** ()

A *SpaceCenter::Light* if the part is a light, otherwise NULL.

SpaceCenter::Parachute **parachute** ()

A *SpaceCenter::Parachute* if the part is a parachute, otherwise NULL.

SpaceCenter::Radiator **radiator** ()

A *SpaceCenter::Radiator* if the part is a radiator, otherwise NULL.

SpaceCenter::ReactionWheel **reaction_wheel** ()

A *SpaceCenter::ReactionWheel* if the part is a reaction wheel, otherwise NULL.

SpaceCenter::ResourceConverter **resource_converter** ()

A *SpaceCenter::ResourceConverter* if the part is a resource converter, otherwise NULL.

SpaceCenter::ResourceHarvester **resource_harvester** ()

A *SpaceCenter::ResourceHarvester* if the part is a resource harvester, otherwise NULL.

SpaceCenter::Sensor **sensor** ()

A *SpaceCenter::Sensor* if the part is a sensor, otherwise NULL.

SpaceCenter::SolarPanel **solar_panel** ()

A *SpaceCenter::SolarPanel* if the part is a solar panel, otherwise NULL.

`std::tuple<double, double, double>` **position** (*SpaceCenter::ReferenceFrame* *reference_frame*)

The position of the part in the given reference frame.

Parameters

`std::tuple<double, double, double>` **direction** (*SpaceCenter::ReferenceFrame* *reference_frame*)

The direction of the part in the given reference frame.

Parameters

`std::tuple<double, double, double>` **velocity** (*SpaceCenter::ReferenceFrame* *reference_frame*)

The velocity of the part in the given reference frame.

Parameters

`std::tuple<double, double, double, double>` **rotation** (*SpaceCenter::ReferenceFrame* *reference_frame*) *reference_frame*

The rotation of the part in the given reference frame.

Parameters

SpaceCenter::ReferenceFrame **reference_frame** ()

The reference frame that is fixed relative to this part.

- The origin is at the position of the part.
- The axes rotate with the part.

- The x, y and z axis directions depend on the design of the part.

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by `SpaceCenter::DockingPort::reference_frame()`.

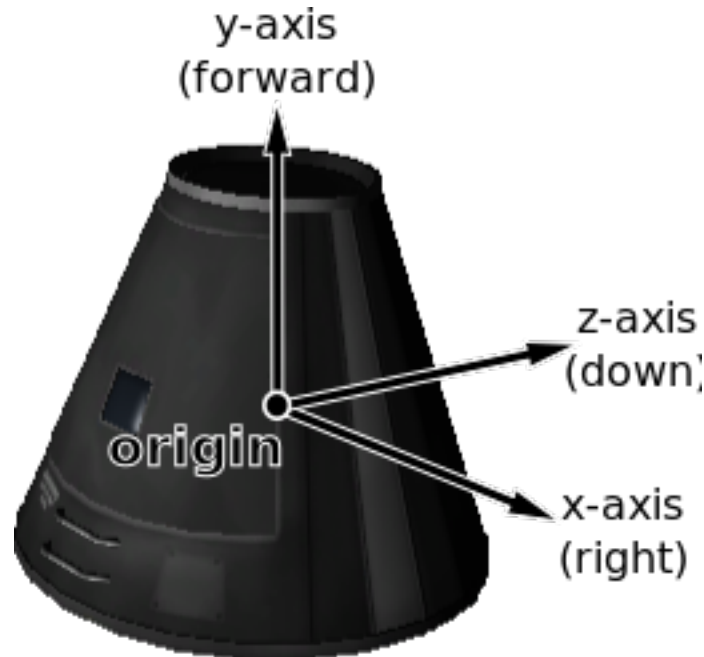


Fig. 4.7: Mk1 Command Pod reference frame origin and axes

Module

class Module

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” PartModule that contains all the functionality of an engine. This class allows you to interact with KSPs PartModules, and any PartModules that have been added by other mods.

`std::string name()`

Name of the PartModule. For example, “ModuleEngines”.

`SpaceCenter::Part part()`

The part that contains this module.

`std::map<std::string, std::string> fields()`

The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

`bool has_field(std::string name)`

Returns `true` if the module has a field with the given name.

Parameters

- **name** – Name of the field.

`std::string get_field(std::string name)`

Returns the value of a field.

Parameters

- **name** – Name of the field.

std::vector<std::string> **events** ()

A list of the names of all of the modules events. Events are the clickable buttons visible in the right-click menu of the part.

bool **has_event** (std::string *name*)

true if the module has an event with the given name.

Parameters

void **trigger_event** (std::string *name*)

Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters

std::vector<std::string> **actions** ()

A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

bool **has_action** (std::string *name*)

true if the part has an action with the given name.

Parameters

void **set_action** (std::string *name*, bool *value* = True)

Set the value of an action with the given name.

Parameters

Specific Types of Part

The following classes provide functionality for specific types of part.

- *Cargo Bay*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Fairing*
- *Intake*
- *Landing Gear*
- *Landing Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *Sensor*
- *Solar Panel*

Cargo Bay

class CargoBay

Obtained by calling *SpaceCenter::Part::cargo_bay()*.

SpaceCenter::Part **part** ()

The part object for this cargo bay.

SpaceCenter::CargoBayState **state** ()

The state of the cargo bay.

bool **open** ()

void **set_open** (bool *value*)

Whether the cargo bay is open.

enum struct CargoBayState

See *SpaceCenter::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.

Decoupler

class Decoupler

Obtained by calling *SpaceCenter::Part::decoupler()*

SpaceCenter::Part **part** ()

The part object for this decoupler.

void **decouple** ()

Fires the decoupler. Has no effect if the decoupler has already fired.

bool **decoupled** ()

Whether the decoupler has fired.

float **impulse** ()

The impulse that the decoupler imparts when it is fired, in Newton seconds.

Docking Port

class DockingPort

Obtained by calling *SpaceCenter::Part::docking_port()*

SpaceCenter::Part **part** ()

The part object for this docking port.

std::string **name** ()

void **set_name** (std::string *value*)

The port name of the docking port. This is the name of the port that can be set in the right click menu, when the [Docking Port Alignment Indicator](#) mod is installed. If this mod is not installed, returns the title of the part ([SpaceCenter::Part::title\(\)](#)).

[SpaceCenter::DockingPortState](#) **state** ()

The current state of the docking port.

[SpaceCenter::Part](#) **docked_part** ()

The part that this docking port is docked to. Returns NULL if this docking port is not docked to anything.

[SpaceCenter::Vessel](#) **undock** ()

Undocks the docking port and returns the vessel that was undocked from. After undocking, the active vessel may change ([SpaceCenter::active_vessel\(\)](#)). This method can be called for either docking port in a docked pair - both calls will have the same effect. Returns NULL if the docking port is not docked to anything.

float **reengage_distance** ()

The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

bool **has_shield** ()

Whether the docking port has a shield.

bool **shielded** ()

void **set_shielded** (bool *value*)

The state of the docking ports shield, if it has one. Returns true if the docking port has a shield, and the shield is closed. Otherwise returns false. When set to true, the shield is closed, and when set to false the shield is opened. If the docking port does not have a shield, setting this attribute has no effect.

std::tuple<double, double, double> **position** ([SpaceCenter::ReferenceFrame](#) *reference_frame*)

The position of the docking port in the given reference frame.

Parameters

std::tuple<double, double, double> **direction** ([SpaceCenter::ReferenceFrame](#) *reference_frame*)

The direction that docking port points in, in the given reference frame.

Parameters

std::tuple<double, double, double, double> **rotation** ([SpaceCenter::ReferenceFrame](#) *reference_frame*)

The rotation of the docking port, in the given reference frame.

Parameters

[SpaceCenter::ReferenceFrame](#) **reference_frame** ()

The reference frame that is fixed relative to this docking port, and oriented with the port.

- The origin is at the position of the docking port.
- The axes rotate with the docking port.
- The x-axis points out to the right side of the docking port.
- The y-axis points in the direction the docking port is facing.
- The z-axis points out of the bottom off the docking port.

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by [SpaceCenter::Part::reference_frame\(\)](#).

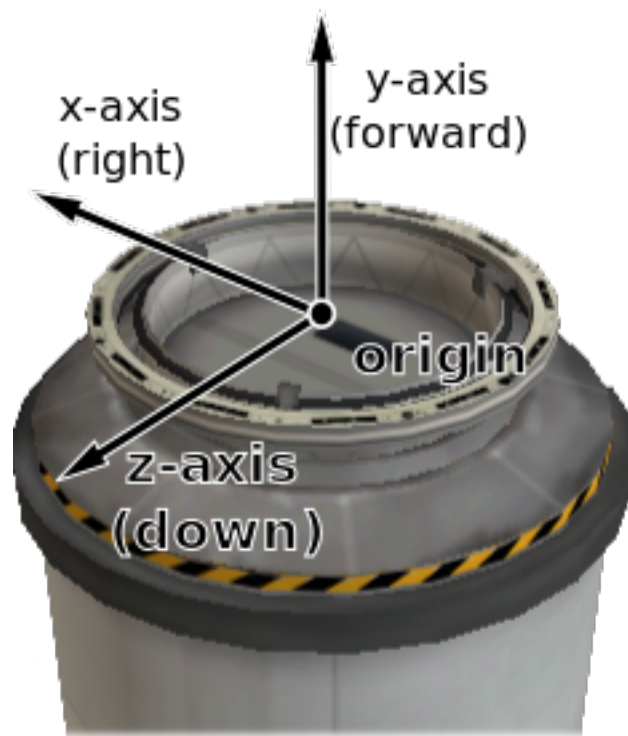


Fig. 4.8: Docking port reference frame origin and axes

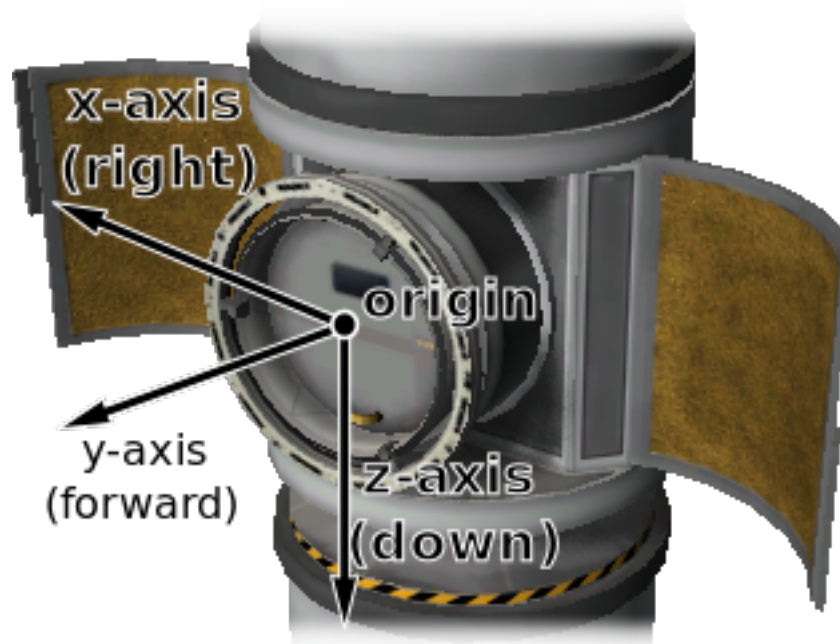


Fig. 4.9: Inline docking port reference frame origin and axes

enum struct DockingPortState

See `SpaceCenter::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 (`SpaceCenter::DockingPort::reengage_distance()`).

enumerator shielded

The docking port has a shield, and the shield is closed.

enumerator moving

The docking ports shield is currently opening/closing.

Engine

class Engine

Obtained by calling `SpaceCenter::Part::engine()`.

`SpaceCenter::Part` **part** ()

The part object for this engine.

bool **active** ()

void **set_active** (bool *value*)

Whether the engine is active. Setting this attribute may have no effect, depending on `SpaceCenter::Engine::can_shutdown()` and `SpaceCenter::Engine::can_restart()`.

float **thrust** ()

The current amount of thrust being produced by the engine, in Newtons. Returns zero if the engine is not active or if it has no fuel.

float **available_thrust** ()

The maximum available amount of thrust that can be produced by the engine, in Newtons. This takes `SpaceCenter::Engine::thrust_limit()` into account, and is the amount of thrust produced by the engine when activated and the main throttle is set to 100%. Returns zero if the engine does not have any fuel.

float **max_thrust** ()

Gets the maximum amount of thrust that can be produced by the engine, in Newtons. This is the amount of thrust produced by the engine when activated, `SpaceCenter::Engine::thrust_limit()` is set to 100% and the main vessel's throttle is set to 100%.

float **max_vacuum_thrust** ()

The maximum amount of thrust that can be produced by the engine in a vacuum, in Newtons. This is the amount of thrust produced by the engine when activated, `SpaceCenter::Engine::thrust_limit()` is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

float **thrust_limit** ()

void **set_thrust_limit** (float *value*)
The thrust limiter of the engine. A value between 0 and 1. Setting this attribute may have no effect, for example the thrust limit for a solid rocket booster cannot be changed in flight.

float **specific_impulse** ()
The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

float **vacuum_specific_impulse** ()
The vacuum specific impulse of the engine, in seconds.

float **kerbin_sea_level_specific_impulse** ()
The specific impulse of the engine at sea level on Kerbin, in seconds.

std::vector<std::string> **propellants** ()
The names of resources that the engine consumes.

std::map<std::string, float> **propellant_ratios** ()
The ratios of resources that the engine consumes. A dictionary mapping resource names to the ratios at which they are consumed by the engine.

bool **has_fuel** ()
Whether the engine has run out of fuel (or flamed out).

float **throttle** ()
The current throttle setting for the engine. A value between 0 and 1. This is not necessarily the same as the vessel's main throttle setting, as some engines take time to adjust their throttle (such as jet engines).

bool **throttle_locked** ()
Whether the `SpaceCenter::Control::throttle()` affects the engine. For example, this is `true` for liquid fueled rockets, and `false` for solid rocket boosters.

bool **can_restart** ()
Whether the engine can be restarted once shutdown. If the engine cannot be shutdown, returns `false`. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

bool **can_shutdown** ()
Gets whether the engine can be shutdown once activated. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

bool **has_modes** ()
Whether the engine has multiple modes of operation.

std::string **mode** ()
The name of the current engine mode.

void **set_mode** (std::string *value*)
The name of the current engine mode.

std::map<std::string, `SpaceCenter::Engine`> **modes** ()
The available modes for the engine. A dictionary mapping mode names to `SpaceCenter::Engine` objects.

void **toggle_mode** ()
Toggle the current engine mode.

bool **auto_mode_switch** ()
Whether the engine will automatically switch modes.

bool **gimballed** ()
Whether the engine nozzle is gimballed, i.e. can provide a turning force.

float **gimbal_range** ()
The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

bool **gimbal_locked** ()

void **set_gimbal_locked** (bool *value*)
Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

float **gimbal_limit** ()

void **set_gimbal_limit** (float *value*)
The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

Fairing

class **Fairing**

Obtained by calling *SpaceCenter::Part::fairing()*.

SpaceCenter::Part **part** ()
The part object for this fairing.

void **jettison** ()
Jettison the fairing. Has no effect if it has already been jettisoned.

bool **jettisoned** ()
Whether the fairing has been jettisoned.

Intake

class **Intake**

Obtained by calling *SpaceCenter::Part::intake()*.

SpaceCenter::Part **part** ()
The part object for this intake.

bool **open** ()

void **set_open** (bool *value*)
Whether the intake is open.

float **speed** ()
Speed of the flow into the intake, in *m/s*.

float **flow** ()
The rate of flow into the intake, in units of resource per second.

float **area** ()
The area of the intake's opening, in square meters.

Landing Gear

class **LandingGear**

Obtained by calling *SpaceCenter::Part::landing_gear()*.

SpaceCenter::Part **part** ()
The part object for this landing gear.

SpaceCenter::LandingGearState **state** ()
Gets the current state of the landing gear.

Note: Fixed landing gear are always deployed.

bool **deployable** ()
Whether the landing gear is deployable.

bool **deployed** ()

void **set_deployed** (bool *value*)
Whether the landing gear is deployed.

Note: Fixed landing gear are always deployed. Returns an error if you try to deploy fixed landing gear.

enum struct LandingGearState

See *SpaceCenter::LandingGear::state* ().

enumerator deployed
Landing gear is fully deployed.

enumerator retracted
Landing gear is fully retracted.

enumerator deploying
Landing gear is being deployed.

enumerator retracting
Landing gear is being retracted.

Landing Leg

class LandingLeg

Obtained by calling *SpaceCenter::Part::landing_leg* ().

SpaceCenter::Part **part** ()
The part object for this landing leg.

SpaceCenter::LandingLegState **state** ()
The current state of the landing leg.

bool **deployed** ()

void **set_deployed** (bool *value*)
Whether the landing leg is deployed.

enum struct LandingLegState

See *SpaceCenter::LandingLeg::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.

enumerator repairing

Landing leg is being repaired.

Launch Clamp**class LaunchClamp**

Obtained by calling *SpaceCenter::Part::launch_clamp()*.

SpaceCenter::Part **part** ()

The part object for this launch clamp.

void **release** ()

Releases the docking clamp. Has no effect if the clamp has already been released.

Light**class Light**

Obtained by calling *SpaceCenter::Part::light()*.

SpaceCenter::Part **part** ()

The part object for this light.

bool **active** ()

void **set_active** (bool *value*)

Whether the light is switched on.

float **power_usage** ()

The current power usage, in units of charge per second.

Parachute**class Parachute**

Obtained by calling *SpaceCenter::Part::parachute()*.

SpaceCenter::Part **part** ()

The part object for this parachute.

void **deploy** ()

Deploys the parachute. This has no effect if the parachute has already been deployed.

bool **deployed** ()

Whether the parachute has been deployed.

SpaceCenter::ParachuteState **state** ()

The current state of the parachute.

float **deploy_altitude** ()

void **set_deploy_altitude** (float *value*)

The altitude at which the parachute will full deploy, in meters.

float **deploy_min_pressure** ()

void **set_deploy_min_pressure** (float *value*)

The minimum pressure at which the parachute will semi-deploy, in atmospheres.

enum struct ParachuteState

See *SpaceCenter::Parachute::state()*.

enumerator stowed

The parachute is safely tucked away inside its housing.

enumerator active

The parachute is still stowed, but ready to semi-deploy.

enumerator semi_deployed

The parachute has been deployed and is providing some drag, but is not fully deployed yet.

enumerator deployed

The parachute is fully deployed.

enumerator cut

The parachute has been cut.

Radiator

class Radiator

Obtained by calling *SpaceCenter::Part::radiator()*.

SpaceCenter::Part **part** ()

The part object for this radiator.

bool **deployable** ()

Whether the radiator is deployable.

bool **deployed** ()

void **set_deployed** (bool *value*)

For a deployable radiator, `true` if the radiator is extended. If the radiator is not deployable, this is always `true`.

SpaceCenter::RadiatorState **state** ()

The current state of the radiator.

Note: A fixed radiator is always *SpaceCenter::RadiatorState::extended*.

enum struct RadiatorState

SpaceCenter::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**

Obtained by calling `SpaceCenter::Part::resource_converter()`.

`SpaceCenter::Part` **part** ()

The part object for this converter.

`int32_t` **count** ()

The number of converters in the part.

`std::string` **name** (`int32_t index`)

The name of the specified converter.

Parameters

- **index** – Index of the converter.

`bool` **active** (`int32_t index`)

True if the specified converter is active.

Parameters

- **index** – Index of the converter.

`void` **start** (`int32_t index`)

Start the specified converter.

Parameters

- **index** – Index of the converter.

`void` **stop** (`int32_t index`)

Stop the specified converter.

Parameters

- **index** – Index of the converter.

`SpaceCenter::ResourceConverterState` **state** (`int32_t index`)

The state of the specified converter.

Parameters

- **index** – Index of the converter.

`std::string` **status_info** (`int32_t index`)

Status information for the specified converter. This is the full status message shown in the in-game UI.

Parameters

- **index** – Index of the converter.

`std::vector<std::string>` **inputs** (`int32_t index`)

List of the names of resources consumed by the specified converter.

Parameters

- **index** – Index of the converter.

`std::vector<std::string>` **outputs** (`int32_t index`)

List of the names of resources produced by the specified converter.

Parameters

- **index** – Index of the converter.

enum struct ResourceConverterState

See *SpaceCenter::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 *SpaceCenter::ResourceConverter::status_info()* for more information.

Resource Harvester**class ResourceHarvester**

Obtained by calling *SpaceCenter::Part::resource_harvester()*.

SpaceCenter::Part **part** ()

The part object for this harvester.

SpaceCenter::ResourceHarvesterState **state** ()

The state of the harvester.

bool **deployed** ()

void **set_deployed** (bool *value*)

Whether the harvester is deployed.

bool **active** ()

void **set_active** (bool *value*)

Whether the harvester is actively drilling.

float **extraction_rate** ()

The rate at which the drill is extracting ore, in units per second.

float **thermal_efficiency** ()

The thermal efficiency of the drill, as a percentage of its maximum.

float **core_temperature** ()

The core temperature of the drill, in Kelvin.

float **optimum_core_temperature** ()

The core temperature at which the drill will operate with peak efficiency, in Kelvin.

enum struct ResourceHarvesterState

See *SpaceCenter::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

Obtained by calling *SpaceCenter::Part::reaction_wheel()*.

SpaceCenter::Part **part** ()

The part object for this reaction wheel.

bool **active** ()

void **set_active** (bool *value*)

Whether the reaction wheel is active.

bool **broken** ()

Whether the reaction wheel is broken.

float **pitch_torque** ()

The torque in the pitch axis, in Newton meters.

float **yaw_torque** ()

The torque in the yaw axis, in Newton meters.

float **roll_torque** ()

The torque in the roll axis, in Newton meters.

Sensor

class Sensor

Obtained by calling *SpaceCenter::Part::sensor()*.

SpaceCenter::Part **part** ()

The part object for this sensor.

bool **active** ()

void **set_active** (bool *value*)

Whether the sensor is active.

std::string **value** ()

The current value of the sensor.

float **power_usage** ()

The current power usage of the sensor, in units of charge per second.

Solar Panel

class SolarPanel

Obtained by calling `SpaceCenter::Part::solar_panel()`.

`SpaceCenter::Part` **part** ()

The part object for this solar panel.

bool **deployed** ()

void **set_deployed** (bool *value*)

Whether the solar panel is extended.

`SpaceCenter::SolarPanelState` **state** ()

The current state of the solar panel.

float **energy_flow** ()

The current amount of energy being generated by the solar panel, in units of charge per second.

float **sun_exposure** ()

The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

enum struct SolarPanelState

See `SpaceCenter::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**

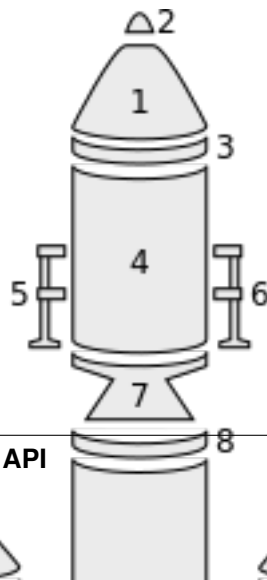
Solar panel is being retracted.

enumerator **broken**

Solar panel is broken.

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 `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 NULL.

The following C++ example uses these attributes to perform a depth-first traversal over all of the parts in a vessel:

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

using namespace krpc::services;

int main() {
    krpc::Client conn = krpc::connect("");
    SpaceCenter sc(&conn);
    auto vessel = sc.active_vessel();

    auto root = vessel.parts().root();
    std::stack<std::pair<SpaceCenter::Part, int> >
    stack.push(std::pair<SpaceCenter::Part, int>(root, 0));
    while (stack.size() > 0) {
        auto part = stack.top().first;
        auto depth = stack.top().second;
        stack.pop();
        std::cout << std::string(depth, ' ') << part->name() << "\n";
        auto children = part->children();
        for (std::vector<SpaceCenter::Part>::iterator it = children.begin(); it != children.end(); ++it)
            stack.push(std::pair<SpaceCenter::Part, int>(*it, 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
TR-18A Stack Decoupler
FL-T400 Fuel Tank
LV-909 Liquid Fuel Engine
TR-18A Stack Decoupler
FL-T800 Fuel Tank
LV-909 Liquid Fuel Engine
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launch Stability Enhancer
```



```

FTX-2 External Fuel Duct
LV-909 Liquid Fuel Engine
Aerodynamic Nose Cone
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launch Stability Enhancer
FTX-2 External Fuel Duct
LV-909 Liquid Fuel Engine
Aerodynamic Nose Cone
LT-1 Landing Struts
LT-1 Landing Struts
Mk16 Parachute

```

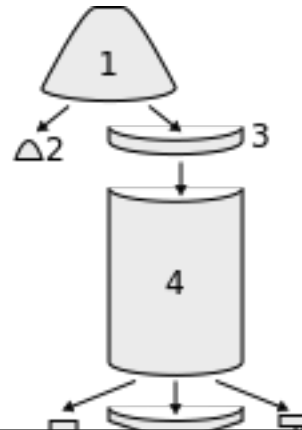
Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

The root part of a vessel (for example the command pod – part 1) does not have a parent part, so does not have an attachment mode. However, the part is considered to be *axially* attached to nothing.

The following C++ example does a depth-first traversal as before, but also prints out the attachment mode used by the part:



```

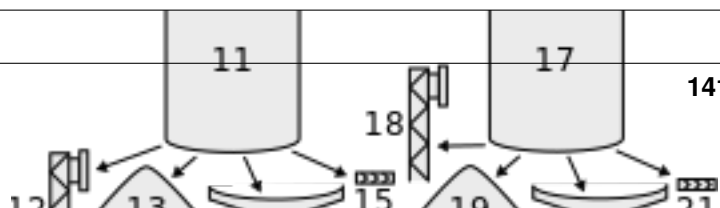
#include <krpc.hpp>
#include <krpc/services/space_center.hpp>
#include <iostream>
#include <stack>

using namespace krpc::services;

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.size() > 0) {
        auto part = stack.top().first;
        auto depth = stack.top().second;
        stack.pop();
        std::string attach_mode;
        if (part.axially_attached()) {
            attach_mode = "axial";

```



```
    } else { // radially_attached
        attach_mode = "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));
    }
}
}
```

When this code is execute using the craft file
for the example vessel pictured above, the fol-
lowing is printed out:

```
Command Pod Mk1 - axial
TR-18A Stack Decoupler - axial
FL-T400 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TR-18A Stack Decoupler - axial
FL-T800 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
LT-1 Landing Struts - radial
LT-1 Landing Struts - radial
Mk16 Parachute - axial
```

Fuel Lines

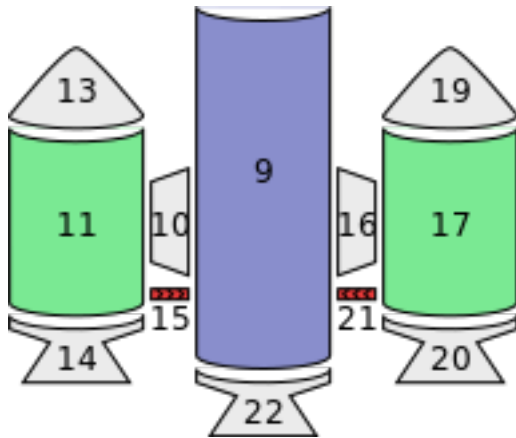


Fig. 4.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.

The attributes `SpaceCenter::Part::fuel_lines_from()` and `SpaceCenter::Part::fuel_lines_to()` can be used to discover these connections. In the example in Figure 5, when `SpaceCenter::Part::fuel_lines_to()` is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When `SpaceCenter::Part::fuel_lines_from()` is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).

Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using `SpaceCenter::Part::stage()` and `SpaceCenter::Part::decouple_stage()` respectively. For parts that are not activated by staging, `SpaceCenter::Part::stage()` returns -1. For parts that are never decoupled,

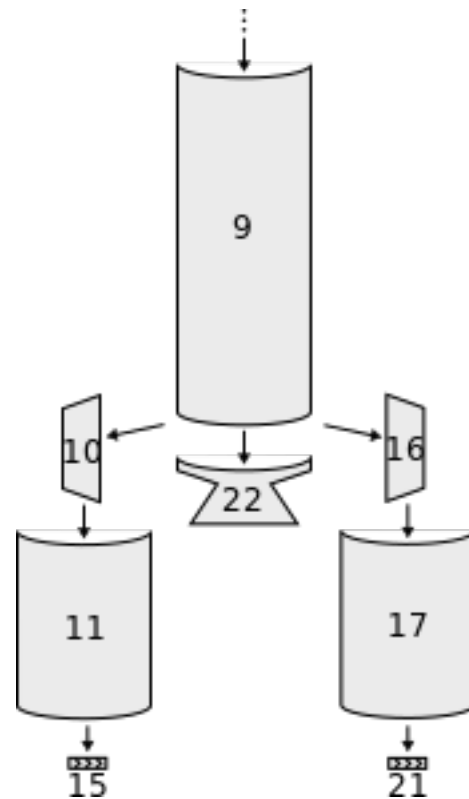


Fig. 4.13: **Figure 4** – A subset of the parts tree from Figure 2 above.

`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*.

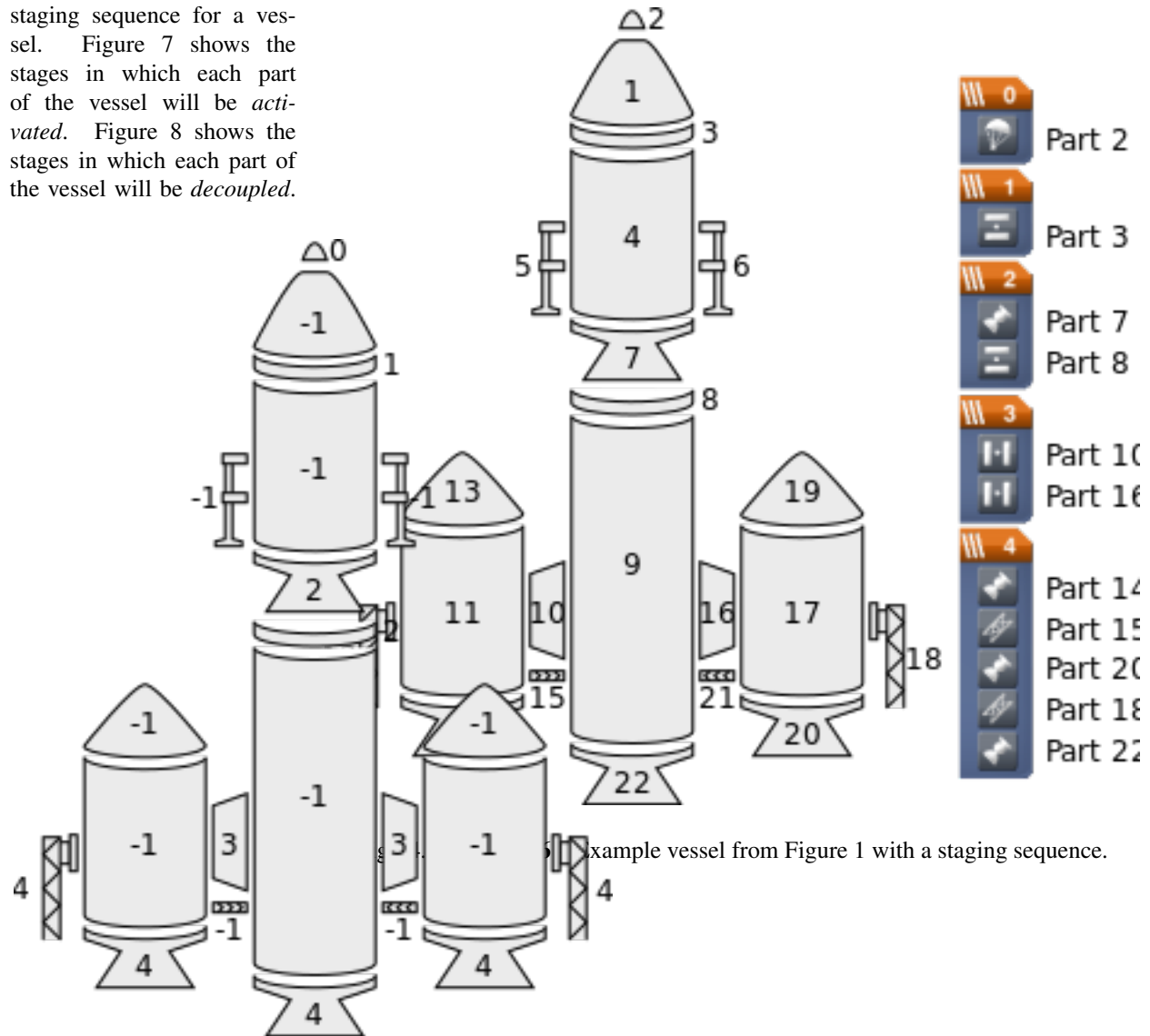


Fig. 4.15: **Figure 7** – The stage in which each part is *activated*.

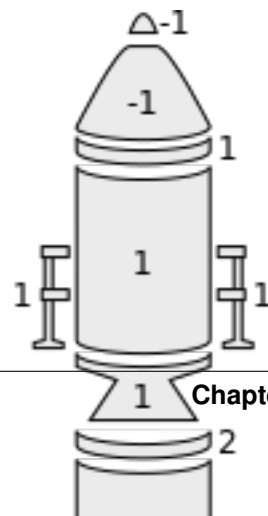
4.3.8 Resources

class **Resources**

Created by calling
`SpaceCenter::Vessel::resources()`,
`SpaceCenter::Vessel::resources_in_decouple_stage()`
or `SpaceCenter::Part::resources()`.

`std::vector<std::string> names()`

A list of resource names that can be stored.



bool **has_resource** (std::string *name*)
 Check whether the named resource can be stored.

Parameters

- **name** – The name of the resource.

float **max** (std::string *name*)
 Returns the amount of a resource that can be stored.

Parameters

- **name** – The name of the resource.

float **amount** (std::string *name*)
 Returns the amount of a resource that is currently stored.

Parameters

- **name** – The name of the resource.

static float **density** (std::string *name*)
 Returns the density of a resource, in kg/l.

Parameters

- **name** – The name of the resource.

static *SpaceCenter::ResourceFlowMode* **flow_mode** (std::string *name*)
 Returns the flow mode of a resource.

Parameters

- **name** – The name of the resource.

enum struct **ResourceFlowMode**
 See *SpaceCenter::Resources::flow_mode()*.

enumerator vessel
 The resource flows to any part in the vessel. For example, electric charge.

enumerator stage
 The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

enumerator adjacent
 The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

enumerator none
 The resource does not flow. For example, solid fuel.

4.3.9 Node

class **Node**
 Represents a maneuver node. Can be created using *SpaceCenter::Control::add_node()*.

float **prograde** ()

void **set_prograde** (float *value*)

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

float **normal** ()

void **set_normal** (float *value*)

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

float **radial** ()

void **set_radial** (float *value*)

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

float **delta_v** ()

void **set_delta_v** (float *value*)

The delta-v of the maneuver node, in meters per second.

Note:	Does	not
change	when	exe-
cuting	the	maneu-
ver	node.	See

SpaceCenter::Node::remaining_delta_v().

float **remaining_delta_v** ()

Gets the remaining delta-v of the maneuver node, in meters per second. Changes as the node is executed. This is equivalent to the delta-v reported in-game.

std::tuple<double, double, double> **burn_vector** (*SpaceCenter::ReferenceFrame* *reference_frame* = None)

Returns a vector whose direction the direction of the maneuver node burn, and whose magnitude is the delta-v of the burn in m/s.

Parameters

Note:	Does	not
change	when	exe-
cuting	the	maneu-
ver	node.	See

SpaceCenter::Node::remaining_burn_vector().

std::tuple<double, double, double> **remaining_burn_vector** (*SpaceCenter::ReferenceFrame* *reference_frame* = None)

Returns a vector whose direction the direction of the maneuver node burn, and whose magnitude is

the delta-v of the burn in m/s. The direction and magnitude change as the burn is executed.

Parameters

double **ut** ()

void **set_ut** (double *value*)

The universal time at which the maneuver will occur, in seconds.

double **time_to** ()

The time until the maneuver node will be encountered, in seconds.

SpaceCenter::Orbit **orbit** ()

The orbit that results from executing the maneuver node.

void **remove** ()

Removes the maneuver node.

SpaceCenter::ReferenceFrame **reference_frame** ()

Gets 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.

SpaceCenter::ReferenceFrame **orbital_reference_frame** ()

Gets the reference frame that is fixed relative to the maneuver node, and orientated with the orbital prograde/normal/radial directions of the original orbit at the maneuver node's position.

- The origin is at the position of the maneuver node.
- The x-axis points in the orbital anti-radial direction of the original orbit, at the position of the maneuver node.
- The y-axis points in the orbital prograde direction of the original orbit, at the position of the maneuver node.
- The z-axis points in the orbital normal direction of the original orbit, at the position of the maneuver node.

std::tuple<double, double, double> **position** (*SpaceCenter::ReferenceFrame* *reference_frame*)

Returns the position vector of the maneuver node in the given reference frame.

Parameters

std::tuple<double, double, double> **direction** (*SpaceCenter::ReferenceFrame* *reference_frame*)

Returns the unit direction vector of the maneuver nodes burn in the given reference frame.

Parameters

4.3.10 Comms

class **Comms**

Used to interact with RemoteTech. Created using a call to `SpaceCenter::Vessel::comms()`.

Note: This class requires `RemoteTech` to be installed.

bool **has_local_control**()
Whether the vessel can be controlled locally.

bool **has_flight_computer**()
Whether the vessel has a RemoteTech flight computer on board.

bool **has_connection**()
Whether the vessel can receive commands from the KSC or a command station.

bool **has_connection_to_ground_station**()
Whether the vessel can transmit science data to a ground station.

double **signal_delay**()
The signal delay when sending commands to the vessel, in seconds.

double **signal_delay_to_ground_station**()
The signal delay between the vessel and the closest ground station, in seconds.

double **signal_delay_to_vessel**(*SpaceCenter::Vessel other*)
Returns the signal delay between the current vessel and another vessel, in seconds.

Parameters

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.

4.3.12 AutoPilot

class **AutoPilot**

Provides basic auto-piloting utilities for a vessel. Created by calling `SpaceCenter::Vessel::auto_pilot()`.

void **engage**()

Engage the auto-pilot.

void **disengage**()

Disengage the auto-pilot.

void **wait**()

Blocks until the vessel is pointing in the target direction (if set) and has the target roll (if set).

float **error**()

The error, in degrees, between the direction the ship has been asked to point in and the direction it is pointing in. Returns zero if the auto-pilot has not been engaged, SAS is not enabled, SAS is in stability assist mode, or no target direction is set.

float **roll_error**()

The error, in degrees, between the roll the ship has been asked to be in and the actual roll. Returns zero if the auto-pilot has not been engaged or no target roll is set.

`SpaceCenter::ReferenceFrame` **reference_frame**()

void **set_reference_frame**(`SpaceCenter::ReferenceFrame` value)

The reference frame for the target direction (`SpaceCenter::AutoPilot::target_direction()`).

std::tuple<double, double, double> **target_direction**()

void **set_target_direction**(std::tuple<double, double, double> value)

The target direction. NULL if no target direction is set.

void **target_pitch_and_heading**(float pitch, float heading)

Set (`SpaceCenter::AutoPilot::target_direction()`) from a pitch and heading angle.

Parameters

- **pitch** – Target pitch angle, in degrees between -90° and +90°.

- **heading** – Target heading angle, in degrees between 0° and 360°.

float **target_roll** ()

void **set_target_roll** (float *value*)

The target roll, in degrees. NaN if no target roll is set.

bool **sas** ()

void **set_sas** (bool *value*)

The state of SAS.

Note: Equivalent to
SpaceCenter::Control::sas()

SpaceCenter::SASMode **sas_mode** ()

void **set_sas_mode** (*SpaceCenter::SASMode* *value*)

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

Note: Equivalent to
SpaceCenter::Control::sas_mode()

float **rotation_speed_multiplier** ()

void **set_rotation_speed_multiplier** (float *value*)

Target rotation speed multiplier. Defaults to 1.

float **max_rotation_speed** ()

void **set_max_rotation_speed** (float *value*)

Maximum target rotation speed. Defaults to 1.

float **roll_speed_multiplier** ()

void **set_roll_speed_multiplier** (float *value*)

Target roll speed multiplier. Defaults to 1.

float **max_roll_speed** ()

void **set_max_roll_speed** (float *value*)

Maximum target roll speed. Defaults to 1.

void **set_pid_parameters** (float *kp* = 1.0, float *ki* = 0.0, float *kd* = 0.0)

Sets the gains for the rotation rate PID controller.

Parameters

- **kp** – Proportional gain.
- **ki** – Integral gain.
- **kd** – Derivative gain.

4.3.13 Geometry Types

class **Vector3**

3-dimensional vectors are represented as a 3-tuple.

For example:

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

using namespace krpc::services;

int main() {
    krpc::Client conn = krpc::connect();
    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;
}
```

class **Quaternion**

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

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

using namespace krpc::services;

int main() {
    krpc::Client conn = krpc::connect();
    SpaceCenter sc(&conn);
    std::tuple<double, double, double, double> q = sc.active_vessel().flight().rotation();
    std::cout << std::get<0>(q) << " "
              << std::get<1>(q) << " "
              << std::get<2>(q) << " "
              << std::get<3>(q) << std::endl;
}
```

4.4 InfernalRobotics API

Provides RPCs to interact with the [InfernalRobotics](#) mod. Provides the following classes:

4.4.1 InfernalRobotics

class InfernalRobotics : public `kRPC::Service`

This service provides functionality to interact with the `InfernalRobotics` mod.

InfernalRobotics (`kRPC::Client *client`)

Construct an instance of this service.

`std::vector<InfernalRobotics::ControlGroup> servo_groups ()`

A list of all the servo groups in the active vessel.

`InfernalRobotics::ControlGroup servo_group_with_name (std::string name)`

Returns the servo group with the given *name* or NULL if none exists. If multiple servo groups have the same name, only one of them is returned.

Parameters

- **name** – Name of servo group to find.

`InfernalRobotics::Servo servo_with_name (std::string name)`

Returns the servo with the given *name*, from all servo groups, or NULL if none exists. If multiple servos have the same name, only one of them is returned.

Parameters

- **name** – Name of the servo to find.

4.4.2 ControlGroup

class ControlGroup

A group of servos, obtained by calling

`InfernalRobotics::servo_groups ()`
or `InfernalRobotics::servo_group_with_name ()`.

Represents the “Servo Groups” in the Infernal-Robotics UI.

`std::string name ()`

`void set_name (std::string value)`

The name of the group.

`std::string forward_key ()`

`void set_forward_key (std::string value)`

The key assigned to be the “forward” key for the group.

`std::string reverse_key ()`

`void set_reverse_key (std::string value)`

The key assigned to be the “reverse” key for the group.

float **speed** ()

void **set_speed** (float *value*)
The speed multiplier for the group.

bool **expanded** ()

void **set_expanded** (bool *value*)
Whether the group is expanded in the Infernal-Robotics UI.

std::vector<*InfernalRobotics::Servo*> **servos** ()
The servos that are in the group.

InfernalRobotics::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.

void **move_right** ()
Moves all of the servos in the group to the right.

void **move_left** ()
Moves all of the servos in the group to the left.

void **move_center** ()
Moves all of the servos in the group to the center.

void **move_next_preset** ()
Moves all of the servos in the group to the next preset.

void **move_prev_preset** ()
Moves all of the servos in the group to the previous preset.

void **stop** ()
Stops the servos in the group.

4.4.3 Servo

class Servo
Represents a servo.
Obtained using *InfernalRobotics::ControlGroup::servos()*,
InfernalRobotics::ControlGroup::servo_with_name()
or *InfernalRobotics::servo_with_name()*.

std::string **name** ()

void **set_name** (std::string *value*)
The name of the servo.

void **set_highlight** (bool *value*)
Whether the servo should be highlighted in-game.

float **position**()
The position of the servo.

float **min_config_position**()
The minimum position of the servo, specified by the part configuration.

float **max_config_position**()
The maximum position of the servo, specified by the part configuration.

float **min_position**()

void **set_min_position**(float *value*)
The minimum position of the servo, specified by the in-game tweak menu.

float **max_position**()

void **set_max_position**(float *value*)
The maximum position of the servo, specified by the in-game tweak menu.

float **config_speed**()
The speed multiplier of the servo, specified by the part configuration.

float **speed**()

void **set_speed**(float *value*)
The speed multiplier of the servo, specified by the in-game tweak menu.

float **current_speed**()

void **set_current_speed**(float *value*)
The current speed at which the servo is moving.

float **acceleration**()

void **set_acceleration**(float *value*)
The current speed multiplier set in the UI.

bool **is_moving**()
Whether the servo is moving.

bool **is_free_moving**()
Whether the servo is freely moving.

bool **is_locked**()

void **set_is_locked**(bool *value*)
Whether the servo is locked.

bool **is_axis_inverted**()

```
void set_is_axis_inverted (bool value)
    Whether the servos axis is inverted.

void move_right ()
    Moves the servo to the right.

void move_left ()
    Moves the servo to the left.

void move_center ()
    Moves the servo to the center.

void move_next_preset ()
    Moves the servo to the next preset.

void move_prev_preset ()
    Moves the servo to the previous preset.

void move_to (float position, float speed)
    Moves the servo to position and sets the speed multiplier to speed.
```

Parameters

- **position** – The position to move the servo to.
- **speed** – Speed multiplier for the movement.

```
void stop ()
    Stops the servo.
```

4.4.4 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 <krpc.hpp>
#include <krpc/services/infernal_robotics.hpp>
#include <iostream>
#include <vector>

using namespace krpc::services;

int main() {
    auto conn = krpc::connect("InfernalRobotics Example");
    InfernalRobotics infernal_robotics(&conn);

    InfernalRobotics::ControlGroup group = infernal_robotics.servo_group_with_name("MyGroup");
    if (group == InfernalRobotics::ControlGroup())
        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();
    sleep(1);
}
```

```
group.stop();  
}
```

4.5 Kerbal Alarm Clock API

Provides RPCs to interact with the [Kerbal Alarm Clock](#) mod. Provides the following classes:

4.5.1 KerbalAlarmClock

class KerbalAlarmClock : public `krpc::Service`

This service provides functionality to interact with the [Kerbal Alarm Clock](#) mod.

KerbalAlarmClock (`krpc::Client *client`)

Construct an instance of this service.

`std::vector<KerbalAlarmClock::Alarm> alarms()`

A list of all the alarms.

`KerbalAlarmClock::Alarm alarm_with_name` (`std::string name`)

Get the alarm with the given *name*, or NULL if no alarms have that name. If more than one alarm has the name, only returns one of them.

Parameters

- **name** – Name of the alarm to search for.

`std::vector<KerbalAlarmClock::Alarm> alarms_with_type` (`KerbalAlarmClock::AlarmType type`)

Get a list of alarms of the specified *type*.

Parameters

- **type** – Type of alarm to return.

`KerbalAlarmClock::Alarm create_alarm` (`KerbalAlarmClock::AlarmType type`, `std::string name`, `double ut`)

Create a new alarm and return it.

Parameters

- **type** – Type of the new alarm.
- **name** – Name of the new alarm.
- **ut** – Time at which the new alarm should trigger.

4.5.2 Alarm

class Alarm

Represents an alarm.
Obtained by calling

`KerbalAlarmClock::alarms()`,
`KerbalAlarmClock::alarm_with_name()`
or `KerbalAlarmClock::alarms_with_type()`.

KerbalAlarmClock::AlarmAction **action** ()

void **set_action** (*KerbalAlarmClock::AlarmAction value*)

The action that the alarm triggers.

double **margin** ()

void **set_margin** (double *value*)

The number of seconds before the event that the alarm will fire.

double **time** ()

void **set_time** (double *value*)

The time at which the alarm will fire.

KerbalAlarmClock::AlarmType **type** ()

The type of the alarm.

std::string **id** ()

The unique identifier for the alarm.

std::string **name** ()

void **set_name** (std::string *value*)

The short name of the alarm.

std::string **notes** ()

void **set_notes** (std::string *value*)

The long description of the alarm.

double **remaining** ()

The number of seconds until the alarm will fire.

bool **repeat** ()

void **set_repeat** (bool *value*)

Whether the alarm will be repeated after it has fired.

double **repeat_period** ()

void **set_repeat_period** (double *value*)

The time delay to automatically create an alarm after it has fired.

SpaceCenter::Vessel **vessel** ()

void **set_vessel** (*SpaceCenter::Vessel value*)

The vessel that the alarm is attached to.

SpaceCenter::CelestialBody **xfer_origin_body** ()

void **set_xfer_origin_body** (*SpaceCenter::CelestialBody value*)

The celestial body the vessel is departing from.

SpaceCenter::CelestialBody **xfer_target_body** ()

void **set_xfer_target_body** (*SpaceCenter::CelestialBody* value)

The celestial body the vessel is arriving at.

void **remove** ()

Removes the alarm.

4.5.3 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 *KerbalAlarmClock::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 *KerbalAlarmClock::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 `KerbAlarmClock::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 `KerbAlarmClock::AlarmType::transfer`.

4.5.4 AlarmAction

enum struct AlarmAction

The action performed by an alarm when it fires.

enumerator do_nothing

Don’t do anything at all...

enumerator do_nothing_delete_when_passed

Don’t do anything, and delete the alarm.

enumerator kill_warp

Drop out of time warp.

enumerator kill_warp_only

Drop out of time warp.

enumerator message_only

Display a message.

enumerator pause_game

Pause the game.

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

```
#include <iostream>

using namespace krpc::services;

int main() {
    krpc::Client conn = krpc::connect("Kerbal Alarm Clock Example");
    SpaceCenter sc(&conn);
    KerbalAlarmClock kac(&conn);

    auto alarm = kac.create_alarm(KerbalAlarmClock::AlarmType::raw,
                                   "My New Alarm",
                                   sc.ut()+10);

    alarm.set_notes("10 seconds have now passed since the alarm was created.");
    alarm.set_action(KerbalAlarmClock::AlarmAction::message_only);
}
```

5.1 Java Client

This client provides functionality to interact with a kRPC server from programs written in Java. A jar containing the `krpc.client` package can be [downloaded from GitHub](#). It requires Java version 1.7.

5.1.1 Using the Library

The kRPC client library depends on the [protobuf](#) and [javatuples](#) libraries. A prebuilt jar for protobuf is available via [Maven](#). Note that you need protobuf version 3. Version 2 is not compatible with kRPC.

The following example program connects to the server, queries it for its version and prints it out:

```
import java.io.IOException;
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KRPC;

public class Basic {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        KRPC krpc = KRPC.newInstance(connection);
        System.out.println("Connected to kRPC version " + krpc.getStatus().getVersion());
    }
}
```

To compile this program using `javac` on the command line, save the source as `Example.java` and run the following:

```
javac -cp libkrpc-0.2.2.jar:protobuf-java-3.0.0-beta-2.jar:javatuples-1.2.jar Example.java
```

You may need to change the paths to the JAR files.

5.1.2 Connecting to the Server

To connect to a server, use the `Connection.newInstance()` function. This returns a connection object 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, including a descriptive name for the client, as follows:

```
import java.io.IOException;
import krpc.client.Connection;
import krpc.client.RPCException;
```

```
import krpc.client.services.KRPC;

public class Connecting {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("Remote example", "my.domain.name", 1000, 1000);
        System.out.println(KRPC.newInstance(connection).getStatus().getVersion());
    }
}
```

5.1.3 Interacting with the Server

Interaction with the server is performed via a connection object. Functionality for services are defined in the packages `krpc.client.services.*`. Before a service can be used it must first be instantiated. The following example connects to the server, instantiates the `SpaceCenter` service, and outputs the name of the active vessel:

```
import java.io.IOException;
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;

public class Interacting {
    public static void main (String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("Vessel Name");
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        System.out.println(vessel.getName());
    }
}
```

5.1.4 Streaming Data from the Server

A stream repeatedly executes a function on the server, with a fixed set of argument values. It provides a more efficient way of repeatedly getting the result of a function, avoiding the network overhead of having to invoke it directly.

For example, consider the following loop that continuously prints out the position of the active vessel. This loop incurs significant communication overheads, as the `vessel.position()` function is called repeatedly.

```
import java.io.IOException;
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KRPC;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;
import krpc.client.services.SpaceCenter.ReferenceFrame;

public class Streaming {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        ReferenceFrame refframe = vessel.getOrbit().getBody().getReferenceFrame();
        while (true)
            System.out.println(vessel.position(refframe));
    }
}
```

The following code achieves the same thing, but is far more efficient. It calls `Connection.addStream` once at the start of the program to create a stream, and then repeatedly gets the position from the stream.

```
import java.io.IOException;
import org.javatuples.Triplet;
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.Vessel;
import krpc.client.services.SpaceCenter.ReferenceFrame;

public class Streaming2 {
    public static void main(String[] args) throws IOException, RPCException, StreamException {
        Connection connection = Connection.newInstance();
        SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
        Vessel vessel = spaceCenter.getActiveVessel();
        ReferenceFrame refframe = vessel.getOrbit().getBody().getReferenceFrame();
        Stream<Triplet<Double,Double,Double>> vessel_stream = connection.addStream(vessel, "position");
        while (true)
            System.out.println(vessel_stream.get());
    }
}
```

Streams are created 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). The most recent value for the stream can be obtained by calling `Stream.get`.

Streams can also be added for static methods as follows:

```
Stream<Double> time_stream = connection.addStream(SpaceCenter.class, "getUt");
```

A stream can be removed by calling `Stream.remove()`. All of a clients streams are automatically stopped when it disconnects.

5.1.5 Client API Reference

class `Connection`

This class provides the interface for communicating with the server.

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, 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, 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.
- **stream_port** (*int*) – The port number of the Stream Server. Defaults to 50001.

void **close** ()

Close the connection.

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.

class **Stream**<T>

A stream object.

T **get** ()

Get the most recent value for the stream.

void **remove** ()

Remove the stream from the server.

abstract class **RemoteObject**

The abstract base class for all remote objects.

5.2 KRPC API

public class **KRPC**

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

`krpc.schema.KRPC.Status` **getStatus** ()

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

`krpc.schema.KRPC.Services` **getServices** ()

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

GameScene **getCurrentGameScene** ()

Get the current game scene.

int **addStream** (`krpc.schema.KRPC.Request request`)

Add a streaming request and return its identifier.

Parameters

- **request** (`krpc.schema.KRPC.Request`) –

Note: Do not call this method from client code. Use *streams* provided by the Java client library.

void **removeStream** (int *id*)

Remove a streaming request.

Parameters

- **id** (*int*) –

Note: Do not call this method from client code. Use *streams* provided by the Java client library.

```
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.
```

5.3 SpaceCenter API

5.3.1 SpaceCenter

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

    Vessel getActiveVessel ()
        The currently active vessel.

    void setActiveVessel (Vessel value)
        The currently active vessel.

    java.util.List<Vessel> getVessels ()
        A list of all the vessels in the game.

    java.util.Map<String, CelestialBody> getBodies ()
        A dictionary of all celestial bodies (planets, moons, etc.) in the game, keyed by the name of the body.

    CelestialBody getTargetBody ()
        The currently targeted celestial body.

    void setTargetBody (CelestialBody value)
        The currently targeted celestial body.

    Vessel getTargetVessel ()
        The currently targeted vessel.

    void setTargetVessel (Vessel value)
        The currently targeted vessel.

    DockingPort getTargetDockingPort ()
        The currently targeted docking port.

    void setTargetDockingPort (DockingPort value)
        The currently targeted docking port.

    void clearTarget ()
        Clears the current target.

    void launchVesselFromVAB (String name)
        Launch a new vessel from the VAB onto the launchpad.
```

Parameters

- **name** (*String*) – Name of the vessel’s craft file.

void **launchVesselFromSPH** (*String name*)

Launch a new vessel from the SPH onto the runway.

Parameters

- **name** (*String*) – Name of the vessel’s craft file.

double **getUT** ()

The current universal time in seconds.

float **getG** ()

The value of the [gravitational constant](#) G in $N(m/kg)^2$.

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.

float **getWarpRate** ()

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

float **getWarpFactor** ()

The current warp factor. This is the index of the rate at which time is passing for either regular “on-rails” or physical time warp. Returns 0 if time warp is not active. When in on-rails time warp, this is equal to *getRailsWarpFactor()*, and in physics time warp, this is equal to *getPhysicsWarpFactor()*.

int **getRailsWarpFactor** ()

void **setRailsWarpFactor** (int *value*)

The time warp rate, using regular “on-rails” time warp. A value between 0 and 7 inclusive. 0 means no time warp. Returns 0 if physical time warp is active. If requested time warp factor cannot be set, it will be set to the next lowest possible value. For example, if the vessel is too close to a planet. See [the KSP wiki](#) for details.

int **getPhysicsWarpFactor** ()

void **setPhysicsWarpFactor** (int *value*)

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

boolean **canRailsWarpAt** (int *factor*)

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

Parameters

- **factor** (*int*) – The warp factor to check.

int **getMaximumRailsWarpFactor** ()

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

void **warpTo** (double *UT*, float *maxRailsRate*, float *maxPhysicsRate*)

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

using regular “on-rails” time warp, the warp rate is limited by *maxRailsRate*, and when using physical time warp, the warp rate is limited by *maxPhysicsRate*.

Parameters

- **UT** (*double*) – The universal time to warp to, in seconds.
- **maxRailsRate** (*float*) – The maximum warp rate in regular “on-rails” time warp.
- **maxPhysicsRate** (*float*) – The maximum warp rate in physical time warp.

Returns When the time warp is complete.

org.javatuples.Triplet<Double, Double, Double> **transformPosition** (org.javatuples.Triplet<Double, Double, Double> *position*, *ReferenceFrame* *from*, *ReferenceFrame* *to*)

Converts a position vector from one reference frame to another.

Parameters

- **position** (*org.javatuples.Triplet<Double, Double, Double>*) – Position vector in reference frame *from*.
- **from** (*ReferenceFrame*) – The reference frame that the position vector is in.
- **to** (*ReferenceFrame*) – The reference frame to convert the position vector to.

Returns The corresponding position vector in reference frame *to*.

org.javatuples.Triplet<Double, Double, Double> **transformDirection** (org.javatuples.Triplet<Double, Double, Double> *direction*, *ReferenceFrame* *from*, *ReferenceFrame* *to*)

Converts a direction vector from one reference frame to another.

Parameters

- **direction** (*org.javatuples.Triplet<Double, Double, Double>*) – Direction vector in reference frame *from*.
- **from** (*ReferenceFrame*) – The reference frame that the direction vector is in.
- **to** (*ReferenceFrame*) – The reference frame to convert the direction vector to.

Returns The corresponding direction vector in reference frame *to*.

org.javatuples.Quartet<Double, Double, Double, Double> **transformRotation** (org.javatuples.Quartet<Double, Double, Double, Double> *rotation*, *ReferenceFrame* *from*, *ReferenceFrame* *to*)

Converts a rotation from one reference frame to another.

Parameters

- **rotation** (*org.javatuples.Quartet<Double, Double, Double, Double>*) – Rotation in reference frame *from*.
- **from** (*ReferenceFrame*) – The reference frame that the rotation is in.
- **to** (*ReferenceFrame*) – The corresponding rotation in reference frame *to*.

Returns The corresponding rotation in reference frame *to*.

`org.javatuples.Triplet<Double, Double, Double> transformVelocity` (`org.javatuples.Triplet<Double, Double, Double> position`, `org.javatuples.Triplet<Double, Double, Double> velocity`, `ReferenceFrame from`, `ReferenceFrame to`)

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

Parameters

- **position** (`org.javatuples.Triplet<Double, Double, Double>`) – Position vector in reference frame *from*.
- **velocity** (`org.javatuples.Triplet<Double, Double, Double>`) – Velocity vector in reference frame *from*.
- **from** (`ReferenceFrame`) – The reference frame that the position and velocity vectors are in.
- **to** (`ReferenceFrame`) – The reference frame to convert the velocity vector to.

Returns The corresponding velocity in reference frame *to*.

boolean **getFARAvailable** ()

Whether `Ferram Aerospace Research` is installed.

boolean **getRemoteTechAvailable** ()

Whether `RemoteTech` is installed.

void **drawDirection** (`org.javatuples.Triplet<Double, Double, Double> direction`, `ReferenceFrame referenceFrame`, `org.javatuples.Triplet<Double, Double, Double> color`, float *length*)

Draw a direction vector on the active vessel.

Parameters

- **direction** (`org.javatuples.Triplet<Double, Double, Double>`) – Direction to draw the line in.
- **referenceFrame** (`ReferenceFrame`) – Reference frame that the direction is in.
- **color** (`org.javatuples.Triplet<Double, Double, Double>`) – The color to use for the line, as an RGB color.
- **length** (`float`) – The length of the line. Defaults to 10.

void **drawLine** (`org.javatuples.Triplet<Double, Double, Double> start`, `org.javatuples.Triplet<Double, Double, Double> end`, `ReferenceFrame referenceFrame`, `org.javatuples.Triplet<Double, Double, Double> color`)

Draw a line.

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** (`ReferenceFrame`) – Reference frame that the position are in.
- **color** (`org.javatuples.Triplet<Double, Double, Double>`) – The color to use for the line, as an RGB color.

void **clearDrawing** ()
Remove all directions and lines currently being drawn.

public enum **WarpMode**
Returned by *WarpMode*

public *WarpMode* **RAILS**
Time warp is active, and in regular “on-rails” mode.

public *WarpMode* **PHYSICS**
Time warp is active, and in physical time warp mode.

public *WarpMode* **NONE**
Time warp is not active.

5.3.2 Vessel

public class **Vessel**

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources.

String **getName** ()

void **setName** (*String* value)
The name of the vessel.

VesselType **getType** ()

void **setType** (*VesselType* value)
The type of the vessel.

VesselSituation **getSituation** ()
The situation the vessel is in.

double **getMET** ()
The mission elapsed time in seconds.

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

Parameters

- **referenceFrame** (*ReferenceFrame*) – Reference frame. Defaults to the vessel’s surface reference frame (*Vessel.getSurfaceReferenceFrame*()).

Note: When this is called with no arguments, the vessel’s surface reference frame is used. This reference frame moves with the vessel, therefore velocities and speeds returned by the flight object will be zero. See the *reference frames tutorial* for examples of getting the *orbital speed* and *surface speed* of a vessel.

Vessel **getTarget** ()

void **setTarget** (*Vessel* value)
The target vessel. `null` if there is no target. When setting the target, the target cannot be the current vessel.

Orbit **getOrbit** ()
The current orbit of the vessel.

Control `getControl ()`

Returns a *Control* object that can be used to manipulate the vessel's control inputs. For example, its pitch/yaw/roll controls, RCS and thrust.

AutoPilot `getAutoPilot ()`

An *AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

Resources `getResources ()`

A *Resources* object, that can be used to get information about resources stored in the vessel.

Resources `resourcesInDecoupleStage (int stage, boolean cumulative)`

Returns a *Resources* object, that can be used to get information about resources stored in a given *stage*.

Parameters

- **stage** (*int*) – Get resources for parts that are decoupled in this stage.
- **cumulative** (*boolean*) – When *false*, returns the resources for parts decoupled in just the given stage. When *true* returns the resources decoupled in the given stage and all subsequent stages combined.

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

Parts `getParts ()`

A *Parts* object, that can be used to interact with the parts that make up this vessel.

Comms `getComms ()`

A *Comms* object, that can be used to interact with RemoteTech for this vessel.

Note: Requires *RemoteTech* to be installed.

float `getMass ()`

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

float `getDryMass ()`

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

float `getThrust ()`

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

float `getAvailableThrust ()`

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

float `getMaxThrust ()`

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

float `getMaxVacuumThrust ()`

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

float `getSpecificImpulse ()`

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

float **getVacuumSpecificImpulse** ()

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

float **getKerbinSeaLevelSpecificImpulse** ()

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

ReferenceFrame **getReferenceFrame** ()

The reference frame that is fixed relative to the vessel, and orientated with the vessel.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel.
- The x-axis points out to the right of the vessel.
- The y-axis points in the forward direction of the vessel.
- The z-axis points out of the bottom off the vessel.

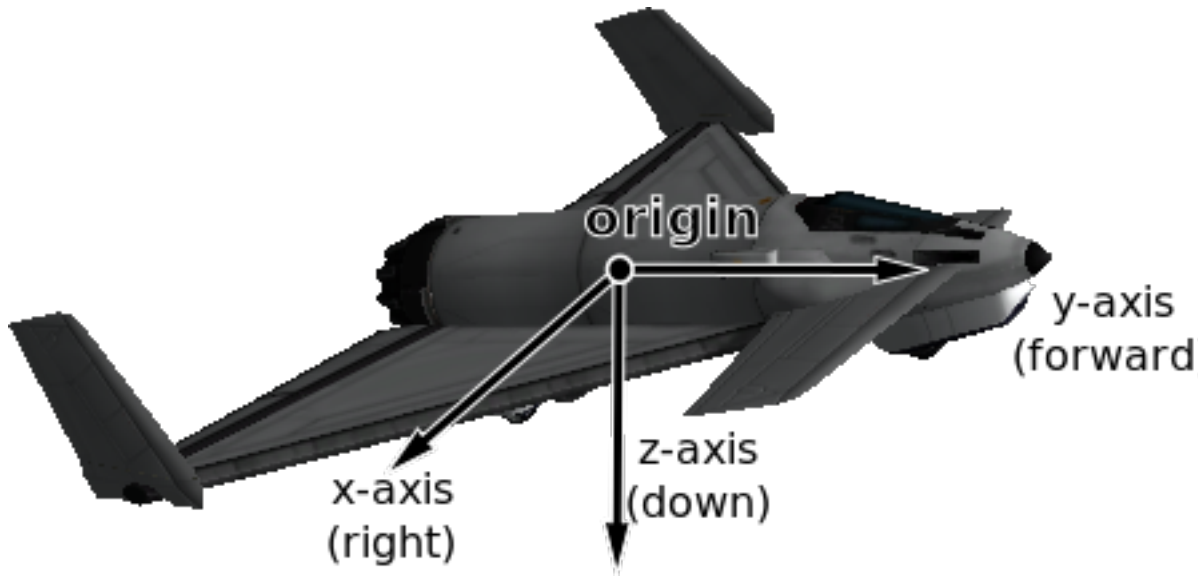


Fig. 5.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

ReferenceFrame **getOrbitalReferenceFrame** ()

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

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

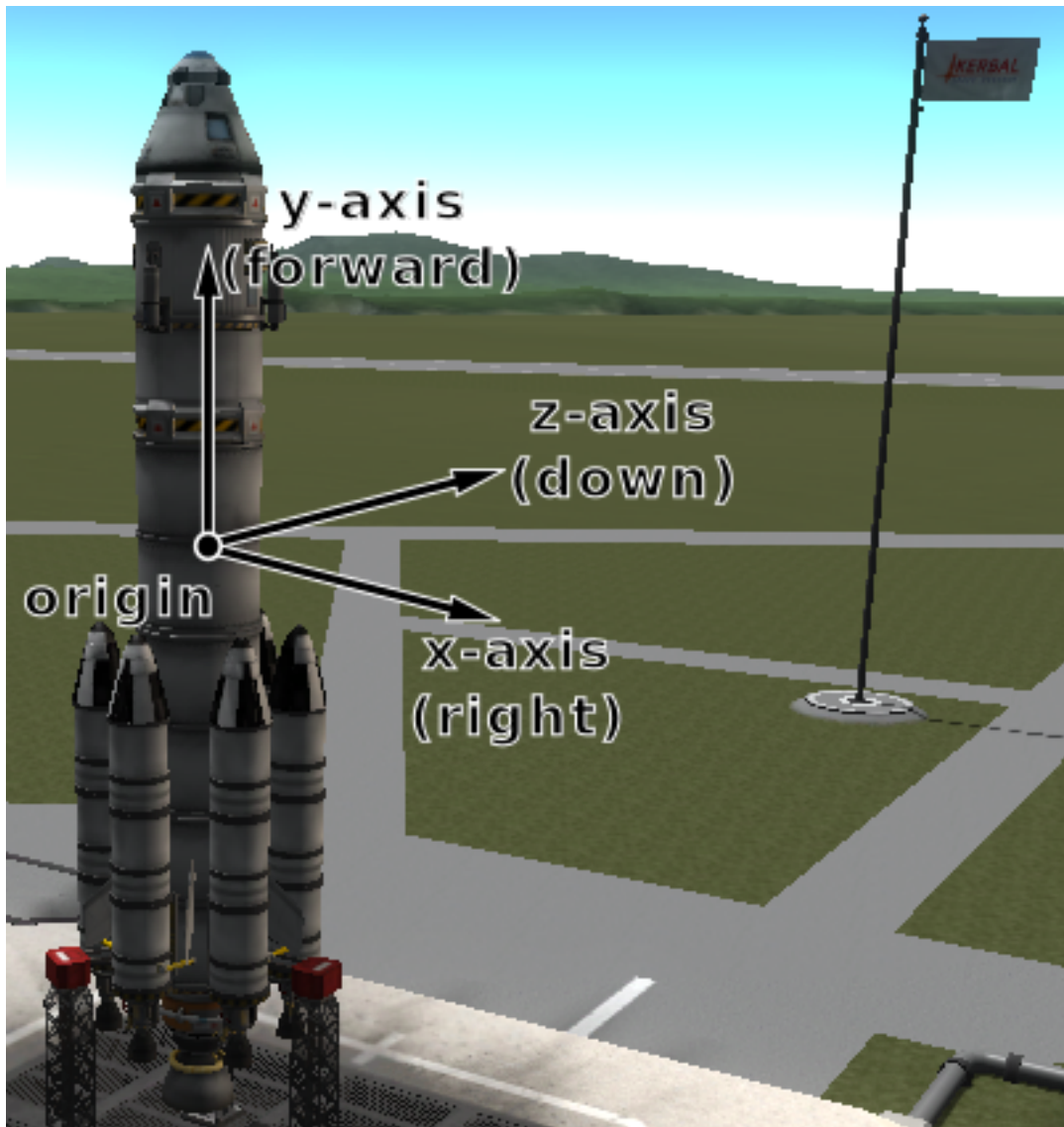


Fig. 5.2: Vessel reference frame origin and axes for the Kerbal-X rocket

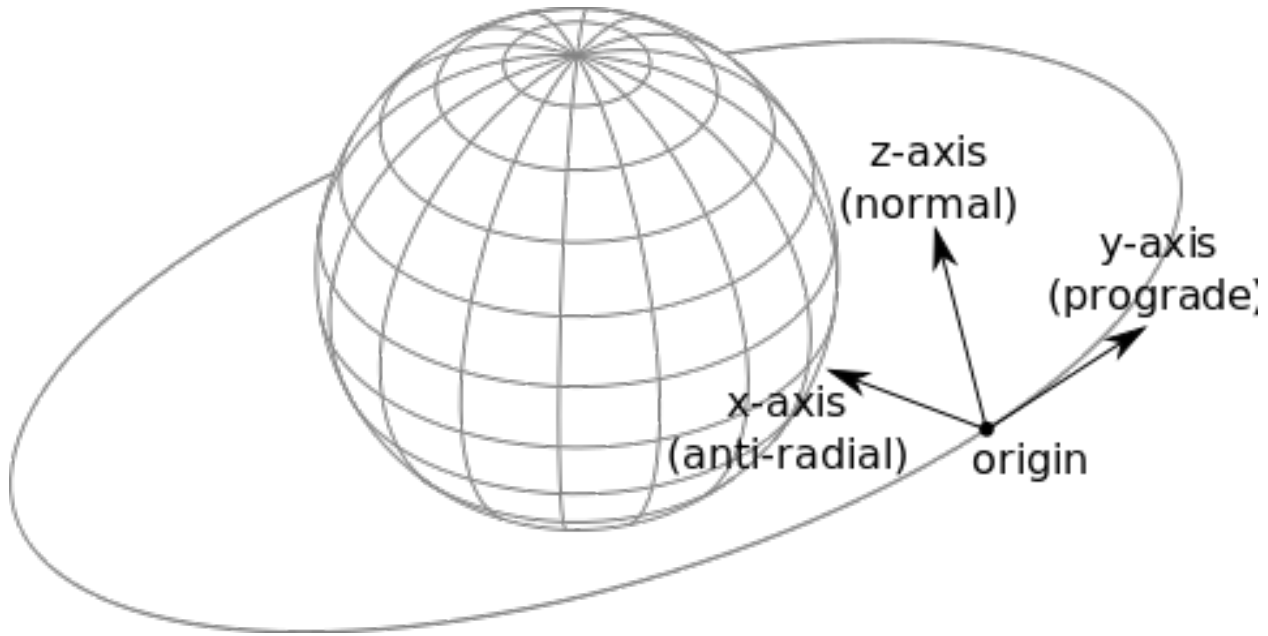


Fig. 5.3: Vessel orbital reference frame origin and axes

ReferenceFrame **getSurfaceReferenceFrame** ()

The reference frame that is fixed relative to the vessel, and orientated with the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the north and up directions on the surface of the body.
- The x-axis points in the [zenith](#) direction (upwards, normal to the body being orbited, from the center of the body towards the center of mass of the vessel).
- The y-axis points northwards towards the [astronomical horizon](#) (north, and tangential to the surface of the body – the direction in which a compass would point when on the surface).
- The z-axis points eastwards towards the [astronomical horizon](#) (east, and tangential to the surface of the body – east on a compass when on the surface).

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

ReferenceFrame **getSurfaceVelocityReferenceFrame** ()

The reference frame that is fixed relative to the vessel, and orientated with the velocity vector of the vessel relative to the surface of the body being orbited.

- The origin is at the center of mass of the vessel.
- The axes rotate with the vessel’s velocity vector.
- The y-axis points in the direction of the vessel’s velocity vector, relative to the surface of the body being orbited.
- The z-axis is in the plane of the [astronomical horizon](#).
- The x-axis is orthogonal to the other two axes.

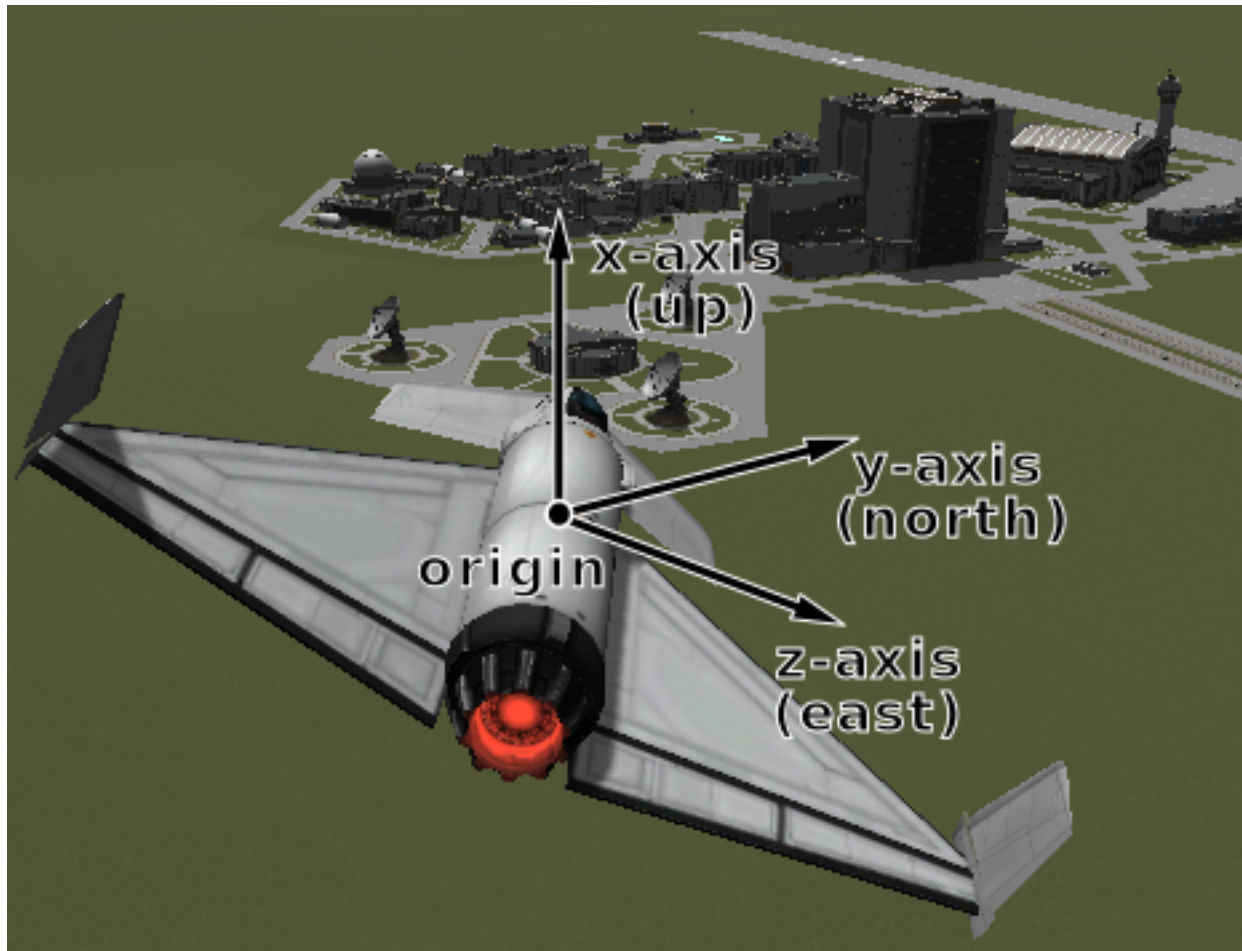


Fig. 5.4: Vessel surface reference frame origin and axes

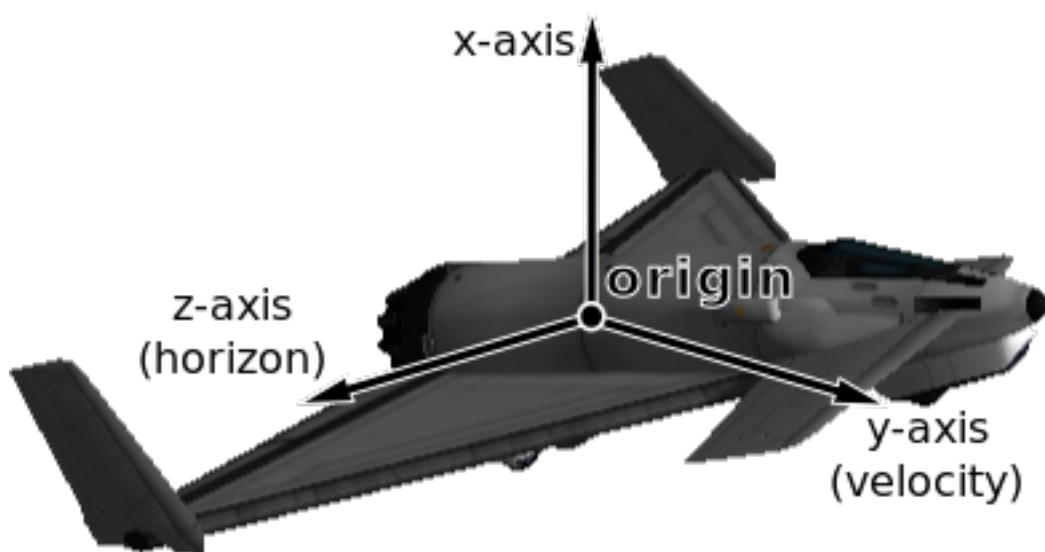


Fig. 5.5: Vessel surface velocity reference frame origin and axes

org.javatuples.Triplet<Double, Double, Double> **position** (*ReferenceFrame* referenceFrame)

Returns the position vector of the center of mass of the vessel in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **velocity** (*ReferenceFrame* referenceFrame)

Returns the velocity vector of the center of mass of the vessel in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Quartet<Double, Double, Double, Double> **rotation** (*ReferenceFrame* referenceFrame)

Returns the rotation of the center of mass of the vessel in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **direction** (*ReferenceFrame* referenceFrame)

Returns the direction in which the vessel is pointing, as a unit vector, in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **angularVelocity** (*ReferenceFrame* referenceFrame)

Returns the angular velocity of the vessel in the given reference frame. The magnitude of the returned vector is the rotational speed in radians per second, and the direction of the vector indicates the axis of rotation (using the right hand rule).

Parameters

- **referenceFrame** (*ReferenceFrame*) –

public enum **VesselType**

See *Vessel.getType()*.

public *VesselType* **SHIP**

Ship.

public *VesselType* **STATION**

Station.

public *VesselType* **LANDER**

Lander.

public *VesselType* **PROBE**

Probe.

public *VesselType* **ROVER**

Rover.

public *VesselType* **BASE**

Base.

public *VesselType* **DEBRIS**

Debris.

public enum **VesselSituation**

See *Vessel.getSituation()*.

public *VesselSituation* **DOCKED**
Vessel is docked to another.

public *VesselSituation* **ESCAPING**
Escaping.

public *VesselSituation* **FLYING**
Vessel is flying through an atmosphere.

public *VesselSituation* **LANDED**
Vessel is landed on the surface of a body.

public *VesselSituation* **ORBITING**
Vessel is orbiting a body.

public *VesselSituation* **PRE_LAUNCH**
Vessel is awaiting launch.

public *VesselSituation* **SPLASHED**
Vessel has splashed down in an ocean.

public *VesselSituation* **SUB_ORBITAL**
Vessel is on a sub-orbital trajectory.

5.3.3 CelestialBody

public class **CelestialBody**
Represents a celestial body (such as a planet or moon).

String **getName** ()
The name of the body.

java.util.List<*CelestialBody*> **getSatellites** ()
A list of celestial bodies that are in orbit around this celestial body.

Orbit **getOrbit** ()
The orbit of the body.

float **getMass** ()
The mass of the body, in kilograms.

float **getGravitationalParameter** ()
The *standard gravitational parameter* of the body in m^3s^{-2} .

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

float **getRotationalPeriod** ()
The sidereal rotational period of the body, in seconds.

float **getRotationalSpeed** ()
The rotational speed of the body, in radians per second.

float **getEquatorialRadius** ()
The equatorial radius of the body, in meters.

double **surfaceHeight** (double *latitude*, double *longitude*)
The height of the surface relative to mean sea level at the given position, in meters. When over water this is equal to 0.

Parameters

- **latitude** (*double*) – Latitude in degrees

- **longitude** (*double*) – Longitude in degrees

double bedrockHeight (*double latitude*, *double longitude*)

The height of the surface relative to mean sea level at the given position, in meters. When over water, this is the height of the sea-bed and is therefore a negative value.

Parameters

- **latitude** (*double*) – Latitude in degrees
- **longitude** (*double*) – Longitude in degrees

org.javatuples.Triplet<Double, Double, Double> mSLPosition (*double latitude*, *double longitude*, *ReferenceFrame referenceFrame*)

The position at mean sea level at the given latitude and longitude, in the given reference frame.

Parameters

- **latitude** (*double*) – Latitude in degrees
- **longitude** (*double*) – Longitude in degrees
- **referenceFrame** (*ReferenceFrame*) – Reference frame for the returned position vector

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

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

float getSphereOfInfluence ()

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

boolean getHasAtmosphere ()

true if the body has an atmosphere.

float getAtmosphereDepth ()

The depth of the atmosphere, in meters.

boolean getHasAtmosphericOxygen ()

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

ReferenceFrame **getReferenceFrame** ()

The reference frame that is fixed relative to the celestial body.

- The origin is at the center of the body.
- The axes rotate with the body.
- The x-axis points from the center of the body towards the intersection of the prime meridian and equator (the position at 0° longitude, 0° latitude).
- The y-axis points from the center of the body towards the north pole.
- The z-axis points from the center of the body towards the equator at 90°E longitude.

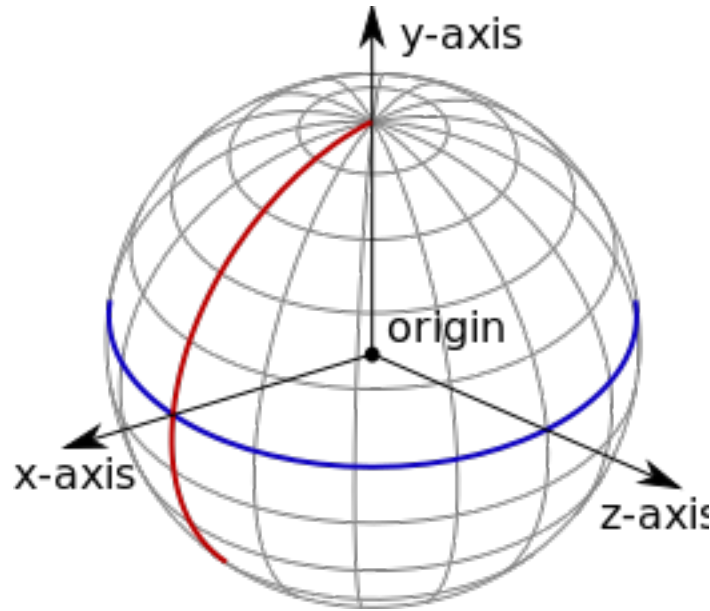


Fig. 5.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

ReferenceFrame **getNonRotatingReferenceFrame** ()

The reference frame that is fixed relative to this celestial body, and orientated in a fixed direction (it does not rotate with the body).

- The origin is at the center of the body.
- The axes do not rotate.
- The x-axis points in an arbitrary direction through the equator.
- The y-axis points from the center of the body towards the north pole.
- The z-axis points in an arbitrary direction through the equator.

ReferenceFrame **getOrbitalReferenceFrame** ()

Gets the reference frame that is fixed relative to this celestial body, but orientated with the body's orbital prograde/normal/radial directions.

- The origin is at the center of the body.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.

- The z-axis points in the orbital normal direction.

org.javatuples.Triplet<Double, Double, Double> **position** (*ReferenceFrame* referenceFrame)
Returns the position vector of the center of the body in the specified reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **velocity** (*ReferenceFrame* referenceFrame)
Returns the velocity vector of the body in the specified reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Quartet<Double, Double, Double, Double> **rotation** (*ReferenceFrame* referenceFrame)
Returns the rotation of the body in the specified reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **direction** (*ReferenceFrame* referenceFrame)
Returns the direction in which the north pole of the celestial body is pointing, as a unit vector, in the specified reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **angularVelocity** (*ReferenceFrame* referenceFrame)
Returns the angular velocity of the body in the specified reference frame. The magnitude of the vector is the rotational speed of the body, in radians per second, and the direction of the vector indicates the axis of rotation, using the right-hand rule.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

5.3.4 Flight

public class **Flight**

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

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

float **getGForce** ()

The current G force acting on the vessel in m/s^2 .

double **getMeanAltitude** ()

The altitude above sea level, in meters.

double **getSurfaceAltitude** ()

The altitude above the surface of the body or sea level, whichever is closer, in meters.

double **getBedrockAltitude** ()

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor.

double **getElevation** ()

The elevation of the terrain under the vessel, in meters. This is the height of the terrain above sea level, and is negative when the vessel is over the sea.

double **getLatitude** ()

The [latitude](#) of the vessel for the body being orbited, in degrees.

double **getLongitude** ()

The [longitude](#) of the vessel for the body being orbited, in degrees.

org.javatuples.Triplet<Double, Double, Double> **getVelocity** ()

The velocity vector of the vessel. The magnitude of the vector is the speed of the vessel in meters per second. The direction of the vector is the direction of the vessels motion.

double **getSpeed** ()

The speed of the vessel in meters per second.

double **getHorizontalSpeed** ()

The horizontal speed of the vessel in meters per second.

double **getVerticalSpeed** ()

The vertical speed of the vessel in meters per second.

org.javatuples.Triplet<Double, Double, Double> **getCenterOfMass** ()

The position of the center of mass of the vessel.

org.javatuples.Quartet<Double, Double, Double, Double> **getRotation** ()

The rotation of the vessel.

org.javatuples.Triplet<Double, Double, Double> **getDirection** ()

The direction vector that the vessel is pointing in.

float **getPitch** ()

The pitch angle of the vessel relative to the horizon, in degrees. A value between -90° and +90°.

float **getHeading** ()

The heading angle of the vessel relative to north, in degrees. A value between 0° and 360°.

float **getRoll** ()

The roll angle of the vessel relative to the horizon, in degrees. A value between -180° and +180°.

org.javatuples.Triplet<Double, Double, Double> **getPrograde** ()

The unit direction vector pointing in the prograde direction.

org.javatuples.Triplet<Double, Double, Double> **getRetrograde** ()

The unit direction vector pointing in the retrograde direction.

org.javatuples.Triplet<Double, Double, Double> **getNormal** ()

The unit direction vector pointing in the normal direction.

org.javatuples.Triplet<Double, Double, Double> **getAntiNormal** ()

The unit direction vector pointing in the anti-normal direction.

org.javatuples.Triplet<Double, Double, Double> **getRadial** ()

The unit direction vector pointing in the radial direction.

org.javatuples.Triplet<Double, Double, Double> **getAntiRadial** ()

The unit direction vector pointing in the anti-radial direction.

float **getAtmosphereDensity** ()

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

float **getDynamicPressure** ()

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

Note: Calculated using [KSPs stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

float **getStaticPressure** ()

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

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

org.javatuples.Triplet<Double, Double, Double> **getAerodynamicForce** ()

The total aerodynamic forces acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

org.javatuples.Triplet<Double, Double, Double> **getLift** ()

The [aerodynamic lift](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

org.javatuples.Triplet<Double, Double, Double> **getDrag** ()

The [aerodynamic drag](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

float **getSpeedOfSound** ()

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

Note: Not available when [Ferram Aerospace Research](#) is installed.

float **getMach** ()

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

Note: Not available when [Ferram Aerospace Research](#) is installed.

float **getEquivalentAirSpeed** ()

The [equivalent air speed](#) of the vessel, in m/s .

Note: Not available when [Ferram Aerospace Research](#) is installed.

float **getTerminalVelocity** ()

An estimate of the current terminal velocity of the vessel, in m/s . This is the speed at which the drag forces cancel out the force of gravity.

Note: Calculated using [KSPs stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

float **getAngleOfAttack** ()

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

float **getSideslipAngle** ()

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

float **getTotalAirTemperature** ()

The [total air temperature](#) of the atmosphere around the vessel, in Kelvin. This temperature includes the [Flight.getStaticAirTemperature\(\)](#) and the vessel's kinetic energy.

float **getStaticAirTemperature** ()

The [static \(ambient\) temperature](#) of the atmosphere around the vessel, in Kelvin.

float **getStallFraction** ()

Gets the current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

Note: Requires [Ferram Aerospace Research](#).

float **getDragCoefficient** ()

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

Note: Requires [Ferram Aerospace Research](#).

float **getLiftCoefficient** ()

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

Note: Requires [Ferram Aerospace Research](#).

float **getBallisticCoefficient** ()

Gets the [ballistic coefficient](#).

Note: Requires [Ferram Aerospace Research](#).

float **getThrustSpecificFuelConsumption** ()

Gets 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 product one newton of thrust.

Note: Requires [Ferram Aerospace Research](#).

5.3.5 Orbit

public class **Orbit**

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

CelestialBody **getBody()**

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

double **getApoapsis()**

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

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

double **getPeriapsis()**

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

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

double **getApoapsisAltitude()**

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

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

double **getPeriapsisAltitude()**

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

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

double **getSemiMajorAxis()**

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

double **getSemiMinorAxis()**

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

double **getRadius()**

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

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

double **getSpeed()**

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

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

double **getPeriod** ()

The orbital period, in seconds.

double **getTimeToApoapsis** ()

The time until the object reaches apoapsis, in seconds.

double **getTimeToPeriapsis** ()

The time until the object reaches periapsis, in seconds.

double **getEccentricity** ()

The *eccentricity* of the orbit.

double **getInclination** ()

The *inclination* of the orbit, in radians.

double **getLongitudeOfAscendingNode** ()

The *longitude of the ascending node*, in radians.

double **getArgumentOfPeriapsis** ()

The *argument of periapsis*, in radians.

double **getMeanAnomalyAtEpoch** ()

The *mean anomaly at epoch*.

double **getEpoch** ()

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

double **getMeanAnomaly** ()

The *mean anomaly*.

double **getEccentricAnomaly** ()

The *eccentric anomaly*.

org.javatuples.Triplet<Double, Double, Double> **referencePlaneNormal** (*ReferenceFrame* *referenceFrame*)

The unit direction vector 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*) –

org.javatuples.Triplet<Double, Double, Double> **referencePlaneDirection** (*ReferenceFrame* *referenceFrame*)

The unit direction vector from which the orbits longitude of ascending node is measured, in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

double **getTimeToSOIChange** ()

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

Orbit **getNextOrbit** ()

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

5.3.6 Control

public class **Control**

Used to manipulate the controls of a vessel. This includes adjusting the throttle, enabling/disabling systems such as SAS and RCS, or altering the direction in which the vessel is pointing.

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.

boolean **getSAS** ()

void **setSAS** (boolean *value*)
The state of SAS.

Note: Equivalent to *AutoPilot.getSAS()*

SASMode **getSASMode** ()

void **setSASMode** (*SASMode* *value*)
The current *SASMode*. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

Note: Equivalent to *AutoPilot.getSASMode()*

SpeedMode **getSpeedMode** ()

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

boolean **getRCS** ()

void **setRCS** (boolean *value*)
The state of RCS.

boolean **getGear** ()

void **setGear** (boolean *value*)
The state of the landing gear/legs.

boolean **getLights** ()

void **setLights** (boolean *value*)
The state of the lights.

boolean **getBrakes** ()

void **setBrakes** (boolean *value*)
The state of the wheel brakes.

boolean **getAbort** ()

void **setAbort** (boolean *value*)
The state of the abort action group.

float **getThrottle** ()

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

float **getPitch** ()

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

float **getYaw** ()

void **setYaw** (float *value*)
The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

float **getRoll** ()

void **setRoll** (float *value*)
The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

float **getForward** ()

void **setForward** (float *value*)
The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

float **getUp** ()

void **setUp** (float *value*)
The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

float **getRight** ()

void **setRight** (float *value*)
The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

float **getWheelThrottle** ()

void **setWheelThrottle** (float *value*)
The state of the wheel throttle. A value between -1 and 1. A value of 1 rotates the wheels forwards, a value of -1 rotates the wheels backwards.

float **getWheelSteering** ()

void **setWheelSteering** (float *value*)
The state of the wheel steering. A value between -1 and 1. A value of 1 steers to the left, and a value of -1 steers to the right.

int **getCurrentStage** ()
The current stage of the vessel. Corresponds to the stage number in the in-game UI.

java.util.List<[Vessel](#)> **activateNextStage** ()
Activates the next stage. Equivalent to pressing the space bar in-game.

Returns A list of vessel objects that are jettisoned from the active vessel.

boolean **getActionGroup** (int *group*)
Returns `true` if the given action group is enabled.

Parameters

- **group** (*int*) – A number between 0 and 9 inclusive.

void **setActionGroup** (int *group*, boolean *state*)
Sets the state of the given action group (a value between 0 and 9 inclusive).

Parameters

- **group** (*int*) – A number between 0 and 9 inclusive.

- **state** (*boolean*) –

void **toggleActionGroup** (int *group*)
Toggles the state of the given action group.

Parameters

- **group** (*int*) – A number between 0 and 9 inclusive.

Node **addNode** (double *UT*, float *prograde*, float *normal*, float *radial*)

Creates a maneuver node at the given universal time, and returns a *Node* object that can be used to modify it. Optionally sets the magnitude of the delta-v for the maneuver node in the prograde, normal and radial directions.

Parameters

- **UT** (*double*) – Universal time of the maneuver node.
- **prograde** (*float*) – Delta-v in the prograde direction.
- **normal** (*float*) – Delta-v in the normal direction.
- **radial** (*float*) – Delta-v in the radial direction.

java.util.List<*Node*> **getNodes** ()

Returns a list of all existing maneuver nodes, ordered by time from first to last.

void **removeNodes** ()

Remove all maneuver nodes.

public enum **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**

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.

5.3.7 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*
 - *Cargo Bay*
 - *Decoupler*
 - *Docking Port*
 - *Engine*
 - *Fairing*
 - *Intake*
 - *Landing Gear*
 - *Landing Leg*
 - *Launch Clamp*
 - *Light*
 - *Parachute*
 - *Radiator*
 - *Resource Converter*
 - *Resource Harvester*
 - *Reaction Wheel*
 - *Sensor*
 - *Solar Panel*
- *Trees of Parts*
 - *Traversing the Tree*
 - *Attachment Modes*
- *Fuel Lines*
- *Staging*

Parts

public class **Parts**

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling *Vessel.getParts()*.

java.util.List<*Part*> **getAll** ()
A list of all of the vessels parts.

Part **getRoot** ()
The vessels root part.

Note: See the discussion on *Trees of Parts*.

Part **getControlling** ()

void **setControlling** (*Part* value)

The part from which the vessel is controlled.

java.util.List<*Part*> **withName** (String name)

A list of parts whose *Part.getName()* is name.

Parameters

- **name** (String) –

java.util.List<*Part*> **withTitle** (String title)

A list of all parts whose *Part.getTitle()* is title.

Parameters

- **title** (String) –

java.util.List<*Part*> **withModule** (String moduleName)

A list of all parts that contain a *Module* whose *Module.getName()* is moduleName.

Parameters

- **moduleName** (String) –

java.util.List<*Part*> **inStage** (int stage)

A list of all parts that are activated in the given stage.

Parameters

- **stage** (int) –

Note: See the discussion on *Staging*.

java.util.List<*Part*> **inDecoupleStage** (int stage)

A list of all parts that are decoupled in the given stage.

Parameters

- **stage** (int) –

Note: See the discussion on *Staging*.

java.util.List<*Module*> **modulesWithName** (String moduleName)

A list of modules (combined across all parts in the vessel) whose *Module.getName()* is moduleName.

Parameters

- **moduleName** (String) –

java.util.List<*CargoBay*> **getCargoBays** ()

A list of all cargo bays in the vessel.

java.util.List<*Decoupler*> **getDecouplers** ()

A list of all decouplers in the vessel.

`java.util.List<DockingPort> getDockingPorts ()`

A list of all docking ports in the vessel.

`DockingPort dockingPortWithName (String name)`

The first docking port in the vessel with the given port name, as returned by `DockingPort.getName()`. Returns null if there are no such docking ports.

Parameters

- `name (String)` –

`java.util.List<Engine> getEngines ()`

A list of all engines in the vessel.

`java.util.List<Fairing> getFairings ()`

A list of all fairings in the vessel.

`java.util.List<Intake> getIntakes ()`

A list of all intakes in the vessel.

`java.util.List<LandingGear> getLandingGear ()`

A list of all landing gear attached to the vessel.

`java.util.List<LandingLeg> getLandingLegs ()`

A list of all landing legs attached to the vessel.

`java.util.List<LaunchClamp> getLaunchClamps ()`

A list of all launch clamps attached to the vessel.

`java.util.List<Light> getLights ()`

A list of all lights in the vessel.

`java.util.List<Parachute> getParachutes ()`

A list of all parachutes in the vessel.

`java.util.List<Radiator> getRadiators ()`

A list of all radiators in the vessel.

`java.util.List<ReactionWheel> getReactionWheels ()`

A list of all reaction wheels in the vessel.

`java.util.List<ResourceConverter> getResourceConverters ()`

A list of all resource converters in the vessel.

`java.util.List<ResourceHarvester> getResourceHarvesters ()`

A list of all resource harvesters in the vessel.

`java.util.List<Sensor> getSensors ()`

A list of all sensors in the vessel.

`java.util.List<SolarPanel> getSolarPanels ()`

A list of all solar panels in the vessel.

Part

public class **Part**

Instances of this class represents a part. A vessel is made of multiple parts. Instances can be obtained by various methods in *Parts*.

`String getName ()`

Internal name of the part, as used in *part* *cfg* files. For example “Mark1-2Pod”.

String `getTitle ()`

Title of the part, as shown when the part is right clicked in-game. For example “Mk1-2 Command Pod”.

double `getCost ()`

The cost of the part, in units of funds.

Vessel `getVessel ()`

The vessel that contains this part.

Part `getParent ()`

The parts parent. Returns `null` if the part does not have a parent. This, in combination with `Part.getChildren()`, can be used to traverse the vessels parts tree.

Note: See the discussion on *Trees of Parts*.

java.util.List<Part> `getChildren ()`

The parts children. Returns an empty list if the part has no children. This, in combination with `Part.getParent()`, can be used to traverse the vessels parts tree.

Note: See the discussion on *Trees of Parts*.

boolean `getAxiallyAttached ()`

Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns `false`.

Note: See the discussion on *Attachment Modes*.

boolean `getRadiallyAttached ()`

Whether the part is radially attached to its parent, i.e. on the side of its parent. If the part has no parent, returns `false`.

Note: See the discussion on *Attachment Modes*.

int `getStage ()`

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

Note: See the discussion on *Staging*.

int `getDecoupleStage ()`

The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

Note: See the discussion on *Staging*.

boolean `getMassless ()`

Whether the part is *massless*.

double `getMass ()`

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

double **getDryMass** ()

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

double **getImpactTolerance** ()

The impact tolerance of the part, in meters per second.

double **getTemperature** ()

Temperature of the part, in Kelvin.

double **getSkinTemperature** ()

Temperature of the skin of the part, in Kelvin.

double **getMaxTemperature** ()

Maximum temperature that the part can survive, in Kelvin.

double **getMaxSkinTemperature** ()

Maximum temperature that the skin of the part can survive, in Kelvin.

float **getThermalMass** ()

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

float **getThermalSkinMass** ()

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

float **getThermalResourceMass** ()

A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

float **getThermalConductionFlux** ()

The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalConvectionFlux** ()

The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalRadiationFlux** ()

The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalInternalFlux** ()

The rate at which heat energy is begin generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalSkinToInternalFlux** ()

The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

Resources **getResources** ()

A *Resources* object for the part.

boolean **getCrossfeed** ()

Whether this part is crossfeed capable.

boolean **getIsFuelLine** ()

Whether this part is a fuel line.

java.util.List<*Part*> **getFuelLinesFrom** ()

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

Note: See the discussion on *Fuel Lines*.

java.util.List<*Part*> **getFuelLinesTo** ()

The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

Note: See the discussion on *Fuel Lines*.

java.util.List<*Module*> **getModules** ()

The modules for this part.

CargoBay **getCargoBay** ()

A *CargoBay* if the part is a cargo bay, otherwise null.

Decoupler **getDecoupler** ()

A *Decoupler* if the part is a decoupler, otherwise null.

DockingPort **getDockingPort** ()

A *DockingPort* if the part is a docking port, otherwise null.

Engine **getEngine** ()

An *Engine* if the part is an engine, otherwise null.

Fairing **getFairing** ()

A *Fairing* if the part is a fairing, otherwise null.

Intake **getIntake** ()

An *Intake* if the part is an intake, otherwise null.

LandingGear **getLandingGear** ()

A *LandingGear* if the part is a landing gear , otherwise null.

LandingLeg **getLandingLeg** ()

A *LandingLeg* if the part is a landing leg, otherwise null.

LaunchClamp **getLaunchClamp** ()

A *LaunchClamp* if the part is a launch clamp, otherwise null.

Light **getLight** ()

A *Light* if the part is a light, otherwise null.

Parachute **getParachute** ()

A *Parachute* if the part is a parachute, otherwise null.

Radiator **getRadiator** ()

A *Radiator* if the part is a radiator, otherwise null.

ReactionWheel **getReactionWheel** ()

A *ReactionWheel* if the part is a reaction wheel, otherwise null.

ResourceConverter **getResourceConverter** ()

A *ResourceConverter* if the part is a resource converter, otherwise null.

ResourceHarvester **getResourceHarvester** ()

A *ResourceHarvester* if the part is a resource harvester, otherwise null.

Sensor **getSensor** ()

A *Sensor* if the part is a sensor, otherwise null.

SolarPanel **getSolarPanel** ()

A *SolarPanel* if the part is a solar panel, otherwise null.

org.javatuples.Triplet<Double, Double, Double> **position** (*ReferenceFrame* referenceFrame)

The position of the part in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **direction** (*ReferenceFrame* referenceFrame)

The direction of the part in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **velocity** (*ReferenceFrame* referenceFrame)

The velocity of the part in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Quartet<Double, Double, Double, Double> **rotation** (*ReferenceFrame* referenceFrame)

The rotation of the part in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

ReferenceFrame **getReferenceFrame** ()

The reference frame that is fixed relative to this part.

- The origin is at the position of the part.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *DockingPort*.*getReferenceFrame* ().

Module

public class **Module**

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” *PartModule* that contains all the functionality of an engine. This class allows you to interact with KSPs *PartModules*, and any *PartModules* that have been added by other mods.

String **getName** ()

Name of the *PartModule*. For example, “ModuleEngines”.

Part **getPart** ()

The part that contains this module.

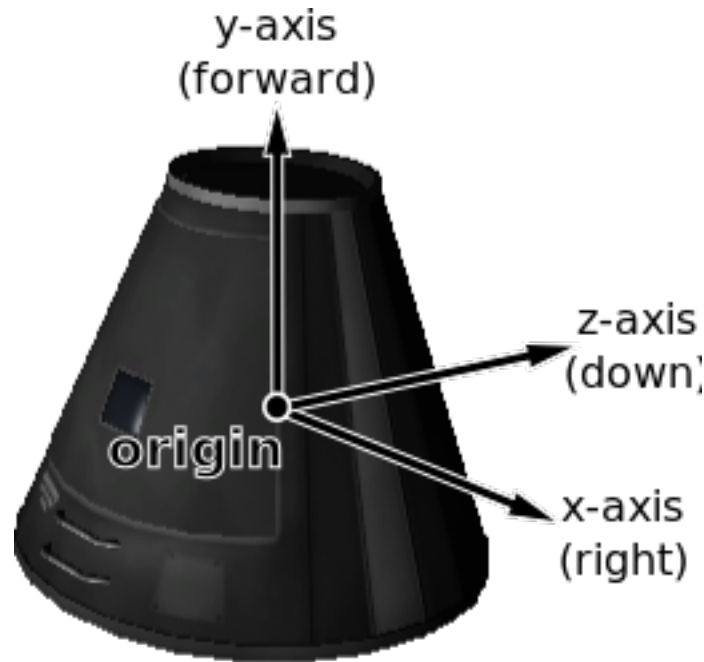


Fig. 5.7: Mk1 Command Pod reference frame origin and axes

`java.util.Map<String, String> getFields ()`

The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

`boolean hasField (String name)`

Returns `true` if the module has a field with the given name.

Parameters

- `name` (*String*) – Name of the field.

`String getField (String name)`

Returns the value of a field.

Parameters

- `name` (*String*) – Name of the field.

`java.util.List<String> getEvents ()`

A list of the names of all of the modules events. Events are the clickable buttons visible in the right-click menu of the part.

`boolean hasEvent (String name)`

`true` if the module has an event with the given name.

Parameters

- `name` (*String*) –

`void triggerEvent (String name)`

Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters

- `name` (*String*) –

`java.util.List<String> getActions ()`

A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

`boolean hasAction (String name)`

true if the part has an action with the given name.

Parameters

- **name** (*String*) –

`void setAction (String name, boolean value)`

Set the value of an action with the given name.

Parameters

- **name** (*String*) –
- **value** (*boolean*) –

Specific Types of Part

The following classes provide functionality for specific types of part.

- *Cargo Bay*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Fairing*
- *Intake*
- *Landing Gear*
- *Landing Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *Sensor*
- *Solar Panel*

Cargo Bay

public class **CargoBay**

Obtained by calling `Part.getCargoBay()`.

Part **getPart** ()

The part object for this cargo bay.

CargoBayState **getState** ()

The state of the cargo bay.

boolean **getOpen** ()


```

    void setOpen (boolean value)
        Whether the cargo bay is open.

public enum CargoBayState
    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.

```

Decoupler

```

public class Decoupler
    Obtained by calling Part.getDecoupler()

    Part getPart ()
        The part object for this decoupler.

    void decouple ()
        Fires the decoupler. Has no effect if the decoupler has already fired.

    boolean getDecoupled ()
        Whether the decoupler has fired.

    float getImpulse ()
        The impulse that the decoupler imparts when it is fired, in Newton seconds.

```

Docking Port

```

public class DockingPort
    Obtained by calling Part.getDockingPort()

    Part getPart ()
        The part object for this docking port.

    String getName ()

    void setName (String value)
        The port name of the docking port. This is the name of the port that can be set in the right click menu,
        when the Docking Port Alignment Indicator mod is installed. If this mod is not installed, returns the title
        of the part (Part.getTitle()).

    DockingPortState getState ()
        The current state of the docking port.

    Part getDockedPart ()
        The part that this docking port is docked to. Returns null if this docking port is not docked to anything.

    Vessel undock ()
        Undocks the docking port and returns the vessel that was undocked from. After undocking, the active
        vessel may change (getActiveVessel()). This method can be called for either docking port in a

```

docked pair - both calls will have the same effect. Returns `null` if the docking port is not docked to anything.

float **getReengageDistance** ()

The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

boolean **getHasShield** ()

Whether the docking port has a shield.

boolean **getShielded** ()

void **setShielded** (boolean *value*)

The state of the docking ports shield, if it has one. Returns `true` if the docking port has a shield, and the shield is closed. Otherwise returns `false`. When set to `true`, the shield is closed, and when set to `false` the shield is opened. If the docking port does not have a shield, setting this attribute has no effect.

org.javatuples.Triplet<Double, Double, Double> **position** (*ReferenceFrame* *referenceFrame*)

The position of the docking port in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **direction** (*ReferenceFrame* *referenceFrame*)

The direction that docking port points in, in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Quartet<Double, Double, Double, Double> **rotation** (*ReferenceFrame* *referenceFrame*, *referenceFrame*)

The rotation of the docking port, in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

ReferenceFrame **getReferenceFrame** ()

The reference frame that is fixed relative to this docking port, and oriented with the port.

- The origin is at the position of the docking port.
- The axes rotate with the docking port.
- The x-axis points out to the right side of the docking port.
- The y-axis points in the direction the docking port is facing.
- The z-axis points out of the bottom off the docking port.

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by *Part*.*getReferenceFrame* ().

public enum **DockingPortState**

See *DockingPort*.*getState* ().

public *DockingPortState* **READY**

The docking port is ready to dock to another docking port.

public *DockingPortState* **DOCKED**

The docking port is docked to another docking port, or docked to another part (from the VAB/SPH).

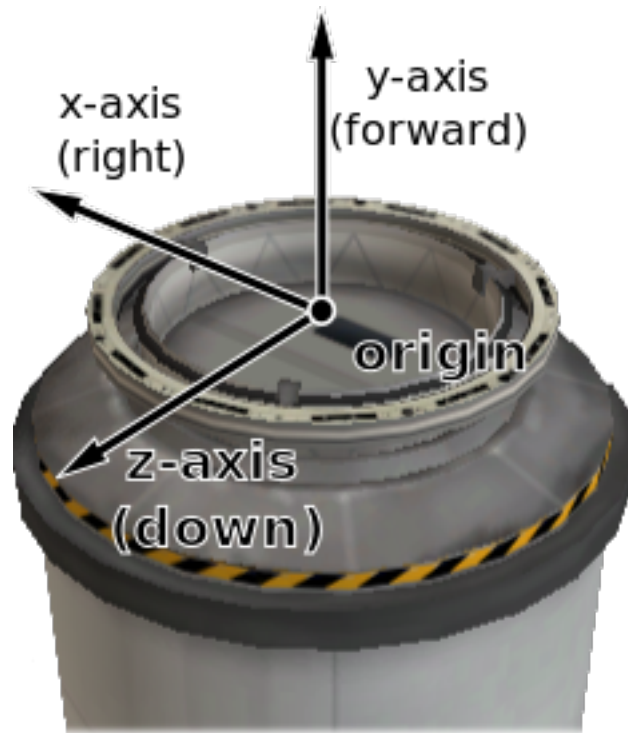


Fig. 5.8: Docking port reference frame origin and axes

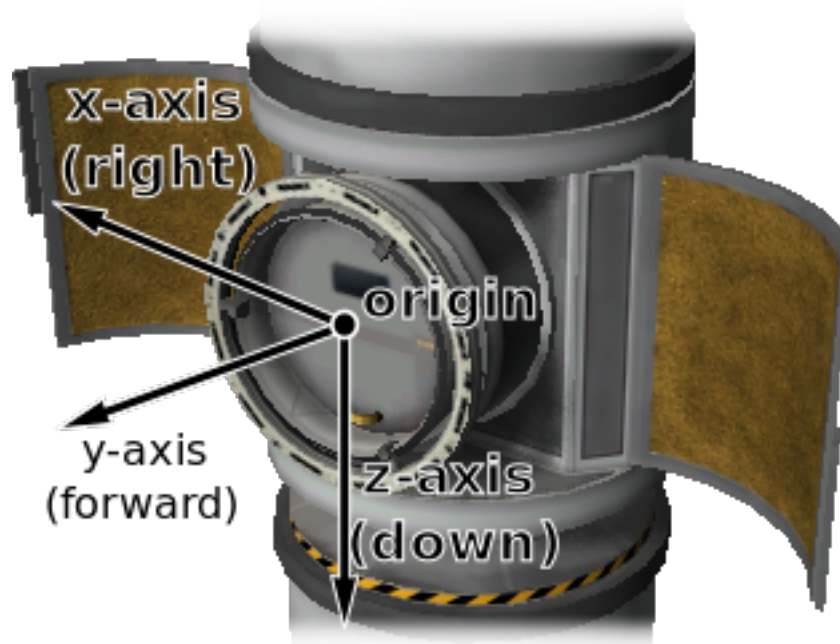


Fig. 5.9: Inline docking port reference frame origin and axes

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**

Obtained by calling *Part.getEngine()*.

Part **getPart()**

The part object for this engine.

boolean **getActive()**

void **setActive** (boolean *value*)

Whether the engine is active. Setting this attribute may have no effect, depending on *Engine.getCanShutdown()* and *Engine.getCanRestart()*.

float **getThrust()**

The current amount of thrust being produced by the engine, in Newtons. Returns zero if the engine is not active or if it has no fuel.

float **getAvailableThrust()**

The maximum available amount of thrust that can be produced by the engine, in Newtons. This takes *Engine.getThrustLimit()* into account, and is the amount of thrust produced by the engine when activated and the main throttle is set to 100%. Returns zero if the engine does not have any fuel.

float **getMaxThrust()**

Gets the maximum amount of thrust that can be produced by the engine, in Newtons. This is the amount of thrust produced by the engine when activated, *Engine.getThrustLimit()* is set to 100% and the main vessel's throttle is set to 100%.

float **getMaxVacuumThrust()**

The maximum amount of thrust that can be produced by the engine in a vacuum, in Newtons. This is the amount of thrust produced by the engine when activated, *Engine.getThrustLimit()* is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

float **getThrustLimit()**

void **setThrustLimit** (float *value*)

The thrust limiter of the engine. A value between 0 and 1. Setting this attribute may have no effect, for example the thrust limit for a solid rocket booster cannot be changed in flight.

float **getSpecificImpulse()**

The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

float **getVacuumSpecificImpulse()**

The vacuum specific impulse of the engine, in seconds.

float **getKerbinSeaLevelSpecificImpulse()**

The specific impulse of the engine at sea level on Kerbin, in seconds.

`java.util.List<String> getPropellants ()`
 The names of resources that the engine consumes.

`java.util.Map<String, Single> getPropellantRatios ()`
 The ratios of resources that the engine consumes. A dictionary mapping resource names to the ratios at which they are consumed by the engine.

`boolean getHasFuel ()`
 Whether the engine has run out of fuel (or flamed out).

`float getThrottle ()`
 The current throttle setting for the engine. A value between 0 and 1. This is not necessarily the same as the vessel's main throttle setting, as some engines take time to adjust their throttle (such as jet engines).

`boolean getThrottleLocked ()`
 Whether the `Control.getThrottle()` affects the engine. For example, this is `true` for liquid fueled rockets, and `false` for solid rocket boosters.

`boolean getCanRestart ()`
 Whether the engine can be restarted once shutdown. If the engine cannot be shutdown, returns `false`. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

`boolean getCanShutdown ()`
 Gets whether the engine can be shutdown once activated. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

`boolean getHasModes ()`
 Whether the engine has multiple modes of operation.

`String getMode ()`

`void setMode (String value)`
 The name of the current engine mode.

`java.util.Map<String, Engine> getModes ()`
 The available modes for the engine. A dictionary mapping mode names to `Engine` objects.

`void toggleMode ()`
 Toggle the current engine mode.

`boolean getAutoModeSwitch ()`

`void setAutoModeSwitch (boolean value)`
 Whether the engine will automatically switch modes.

`boolean getGimballed ()`
 Whether the engine nozzle is gimbaled, i.e. can provide a turning force.

`float getGimbalRange ()`
 The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimbaled.

`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 gimbaled.

`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.

Fairing

public class **Fairing**

Obtained by calling *Part.getFairing()*.

Part **getPart** ()

The part object for this fairing.

void **jettison** ()

Jettison the fairing. Has no effect if it has already been jettisoned.

boolean **getJettisoned** ()

Whether the fairing has been jettisoned.

Intake

public class **Intake**

Obtained by calling *Part.getIntake()*.

Part **getPart** ()

The part object for this intake.

boolean **getOpen** ()

void **setOpen** (boolean *value*)

Whether the intake is open.

float **getSpeed** ()

Speed of the flow into the intake, in *m/s*.

float **getFlow** ()

The rate of flow into the intake, in units of resource per second.

float **getArea** ()

The area of the intake's opening, in square meters.

Landing Gear

public class **LandingGear**

Obtained by calling *Part.getLandingGear()*.

Part **getPart** ()

The part object for this landing gear.

LandingGearState **getState** ()

Gets the current state of the landing gear.

Note: Fixed landing gear are always deployed.

boolean **getDeployable** ()

Whether the landing gear is deployable.

boolean **getDeployed** ()

void **setDeployed** (boolean *value*)

Whether the landing gear is deployed.

Note: Fixed landing gear are always deployed. Returns an error if you try to deploy fixed landing gear.

```
public enum LandingGearState
    See LandingGear.getState().

    public LandingGearState DEPLOYED
        Landing gear is fully deployed.

    public LandingGearState RETRACTED
        Landing gear is fully retracted.

    public LandingGearState DEPLOYING
        Landing gear is being deployed.

    public LandingGearState RETRACTING
        Landing gear is being retracted.
```

Landing Leg

```
public class LandingLeg
    Obtained by calling Part.getLandingLeg().

    Part getPart ()
        The part object for this landing leg.

    LandingLegState getState ()
        The current state of the landing leg.

    boolean getDeployed ()

    void setDeployed (boolean value)
        Whether the landing leg is deployed.

public enum LandingLegState
    See LandingLeg.getState().

    public LandingLegState DEPLOYED
        Landing leg is fully deployed.

    public LandingLegState RETRACTED
        Landing leg is fully retracted.

    public LandingLegState DEPLOYING
        Landing leg is being deployed.

    public LandingLegState RETRACTING
        Landing leg is being retracted.

    public LandingLegState BROKEN
        Landing leg is broken.

    public LandingLegState REPAIRING
        Landing leg is being repaired.
```

Launch Clamp

```
public class LaunchClamp
    Obtained by calling Part.getLaunchClamp().
```

Part **getPart** ()

The part object for this launch clamp.

void **release** ()

Releases the docking clamp. Has no effect if the clamp has already been released.

Light

public class **Light**

Obtained by calling *Part.getLight* ().

Part **getPart** ()

The part object for this light.

boolean **getActive** ()

void **setActive** (boolean *value*)

Whether the light is switched on.

float **getPowerUsage** ()

The current power usage, in units of charge per second.

Parachute

public class **Parachute**

Obtained by calling *Part.getParachute* ().

Part **getPart** ()

The part object for this parachute.

void **deploy** ()

Deploys the parachute. This has no effect if the parachute has already been deployed.

boolean **getDeployed** ()

Whether the parachute has been deployed.

ParachuteState **getState** ()

The current state of the parachute.

float **getDeployAltitude** ()

void **setDeployAltitude** (float *value*)

The altitude at which the parachute will full deploy, in meters.

float **getDeployMinPressure** ()

void **setDeployMinPressure** (float *value*)

The minimum pressure at which the parachute will semi-deploy, in atmospheres.

public enum **ParachuteState**

See *Parachute.getState* ().

public *ParachuteState* **STOWED**

The parachute is safely tucked away inside its housing.

public *ParachuteState* **ACTIVE**

The parachute is still stowed, but ready to semi-deploy.

public *ParachuteState* **SEMI_DEPLOYED**

The parachute has been deployed and is providing some drag, but is not fully deployed yet.


```
public ParachuteState DEPLOYED
    The parachute is fully deployed.

public ParachuteState CUT
    The parachute has been cut.
```

Radiator

```
public class Radiator
    Obtained by calling Part.getRadiator().

    Part getPart ()
        The part object for this radiator.

    boolean getDeployable ()
        Whether the radiator is deployable.

    boolean getDeployed ()

    void setDeployed (boolean value)
        For a deployable radiator, true if the radiator is extended. If the radiator is not deployable, this is always
        true.

    RadiatorState getState ()
        The current state of the radiator.
```

Note: A fixed radiator is always *RadiatorState.EXTENDED*.

```
public enum RadiatorState
    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
    Obtained by calling Part.getResourceConverter().

    Part getPart ()
        The part object for this converter.

    int getCount ()
        The number of converters in the part.
```

String **name** (int *index*)

The name of the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

boolean **active** (int *index*)

True if the specified converter is active.

Parameters

- **index** (*int*) – Index of the converter.

void **start** (int *index*)

Start the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

void **stop** (int *index*)

Stop the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

ResourceConverterState **state** (int *index*)

The state of the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

String **statusInfo** (int *index*)

Status information for the specified converter. This is the full status message shown in the in-game UI.

Parameters

- **index** (*int*) – Index of the converter.

java.util.List<*String*> **inputs** (int *index*)

List of the names of resources consumed by the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

java.util.List<*String*> **outputs** (int *index*)

List of the names of resources produced by the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

public enum **ResourceConverterState**

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
    Obtained by calling Part.getResourceHarvester().

    Part getPart ()
        The part object for this harvester.

    ResourceHarvesterState getState ()
        The state of the harvester.

    boolean getDeployed ()

    void setDeployed (boolean value)
        Whether the harvester is deployed.

    boolean getActive ()

    void setActive (boolean value)
        Whether the harvester is actively drilling.

    float getExtractionRate ()
        The rate at which the drill is extracting ore, in units per second.

    float getThermalEfficiency ()
        The thermal efficiency of the drill, as a percentage of its maximum.

    float getCoreTemperature ()
        The core temperature of the drill, in Kelvin.

    float getOptimumCoreTemperature ()
        The core temperature at which the drill will operate with peak efficiency, in Kelvin.

public enum ResourceHarvesterState
    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
    Obtained by calling Part.getReactionWheel().

    Part getPart ()
        The part object for this reaction wheel.

    boolean getActive ()

    void setActive (boolean value)
        Whether the reaction wheel is active.

    boolean getBroken ()
        Whether the reaction wheel is broken.

    float getPitchTorque ()
        The torque in the pitch axis, in Newton meters.

    float getYawTorque ()
        The torque in the yaw axis, in Newton meters.

    float getRollTorque ()
        The torque in the roll axis, in Newton meters.
```

Sensor

```
public class Sensor
    Obtained by calling Part.getSensor().

    Part getPart ()
        The part object for this sensor.

    boolean getActive ()

    void setActive (boolean value)
        Whether the sensor is active.

    String getValue ()
        The current value of the sensor.

    float getPowerUsage ()
        The current power usage of the sensor, in units of charge per second.
```

Solar Panel

```
public class SolarPanel
    Obtained by calling Part.getSolarPanel().

    Part getPart ()
        The part object for this solar panel.

    boolean getDeployed ()

    void setDeployed (boolean value)
        Whether the solar panel is extended.

    SolarPanelState getState ()
        The current state of the solar panel.
```

```
float getEnergyFlow ()
```

The current amount of energy being generated by the solar panel, in units of charge per second.

```
float getSunExposure ()
```

The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

```
public enum SolarPanelState
```

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
```

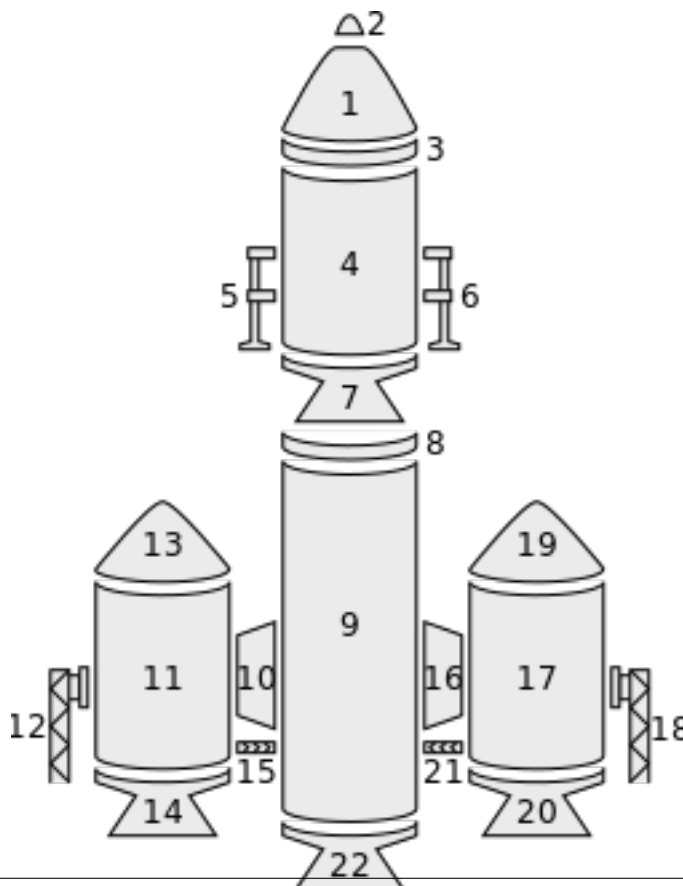
Solar panel is being retracted.

```
public SolarPanelState BROKEN
```

Solar panel is broken.

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.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 java.io.IOException;
import java.util.ArrayDeque;
import java.util.Deque;
import org.javatuples.Pair;
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;

public class TreeTraversal {
    public static void main(String[] args) throws Exception {
        Connection connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.newInstance(connection).getVessel();
        Part root = vessel.getParts().getRoot();
        Deque<Pair<Part, Integer>> stack = new ArrayDeque<>();
        stack.push(new Pair<Part, Integer>(root, 0));
        while (stack.size() > 0) {
            Pair<Part, Integer> item = stack.pop();
            Part part = item.getValue0();
            int depth = item.getValue1();
            String prefix = "";
            for (int i = 0; i < depth; i++)
                prefix += " ";
            System.out.println(prefix + part.getName());
            for (Part child : part.getChildren())
                stack.push(new Pair<Part, Integer>(child, depth + 1));
        }
    }
}

```

When this code is executed using the craft file for the example vessel pictured above, the following is printed out:

```

Command Pod Mk1
TR-18A Stack Decoupler
FL-T400 Fuel Tank
LV-909 Liquid Fuel Engine
TR-18A Stack Decoupler
FL-T800 Fuel Tank
LV-909 Liquid Fuel Engine
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launch Stability Enhancer
FTX-2 External Fuel Duct
LV-909 Liquid Fuel Engine
Aerodynamic Nose Cone
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launch Stability Enhancer
FTX-2 External Fuel Duct
LV-909 Liquid Fuel Engine
Aerodynamic Nose Cone
LT-1 Landing Struts
LT-1 Landing Struts
Mk16 Parachute

```

Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its

parent (the fuel tank – part 4).

The root part of a vessel (for example the command pod – part 1) does not have a parent part, so does not have an attachment mode. However, the part is 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 java.io.IOException;
import java.util.ArrayDeque;
import java.util.Deque;
import org.javatuples.Pair;
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;

public class AttachmentModes {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.newInstance(connection).getActiveVessel();
        Part root = vessel.getParts().getRoot();
        Deque<Pair<Part, Integer>> stack = new ArrayDeque<Pair<Part, Integer>>();
        stack.push(new Pair<Part, Integer>(root, 0));
        while (stack.size() > 0) {
            Pair<Part, Integer> item = stack.pop();
            Part part = item.getValue0();
            int depth = item.getValue1();
            String prefix = "";
            for (int i = 0; i < depth; i++)
                prefix += " ";
            String attachMode = part.getAxiallyAttached() ? "axial" : "radial";
            System.out.println(prefix + part.getTitle() + " - " + attachMode);
            for (Part child : part.getChildren())
                stack.push(new Pair<Part, Integer>(child, depth + 1));
        }
    }
}
```

When this code is execute using the craft file for the example vessel pictured above, the following is printed out:

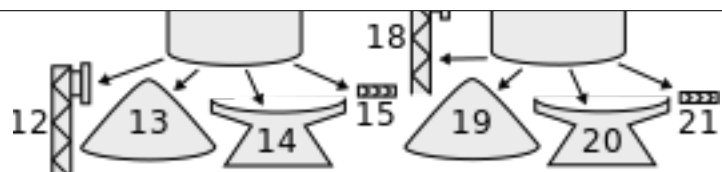


Fig. 5.11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the child part.

```

Command Pod Mk1 - axial
TR-18A Stack Decoupler - axial
FL-T400 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TR-18A Stack Decoupler - axial
FL-T800 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
LT-1 Landing Struts - radial
LT-1 Landing Struts - radial
Mk16 Parachute - axial

```

Fuel Lines

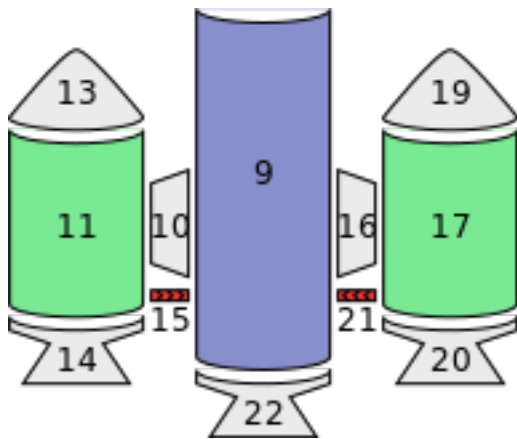
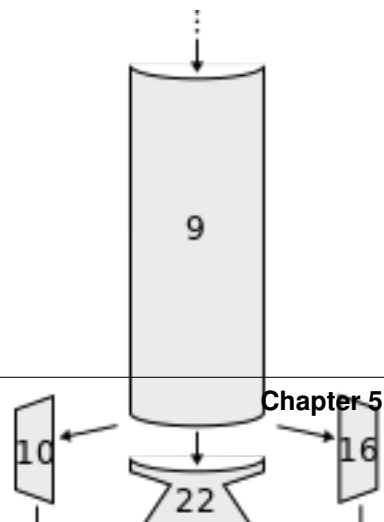


Fig. 5.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.



The attributes `Part.getFuelLinesFrom()` and `Part.getFuelLinesTo()` can be used to discover these connections. In the example in Figure 5, when `Part.getFuelLinesTo()` is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When `Part.getFuelLinesFrom()` is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).

Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using `Part.getStage()` and `Part.getDecoupleStage()` respectively. For parts that are not activated by staging, `Part.getStage()` returns -1. For parts that are never decoupled, `Part.getDecoupleStage()` returns a value of -1.

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *activated*. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

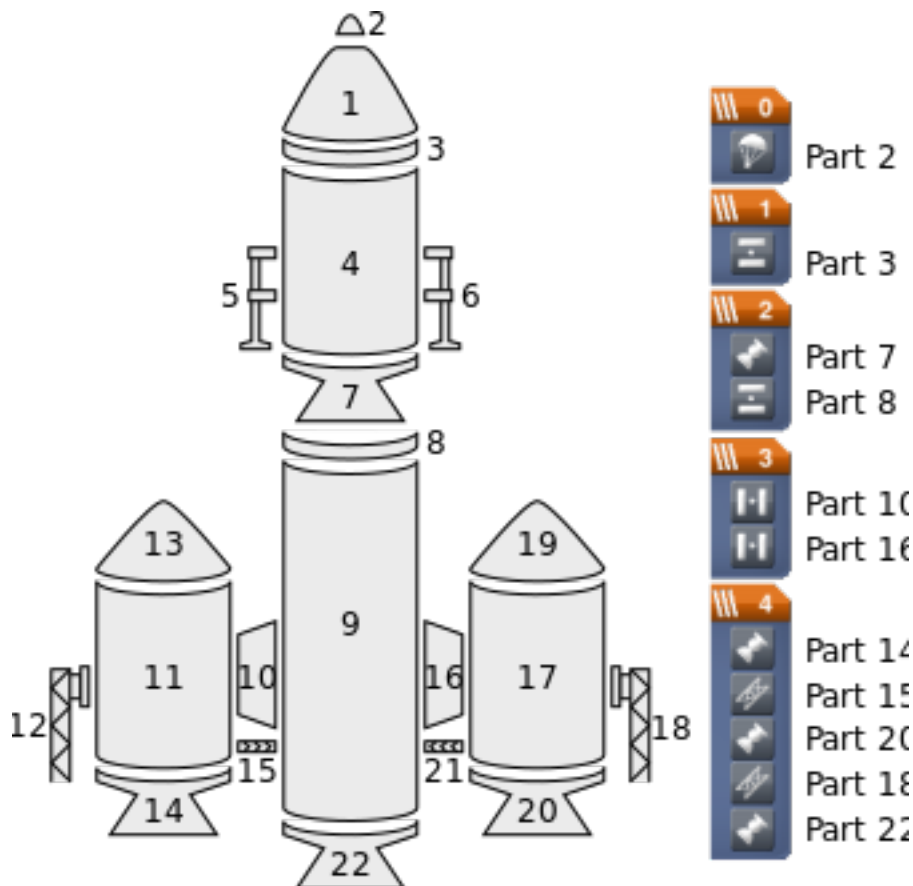


Fig. 5.14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

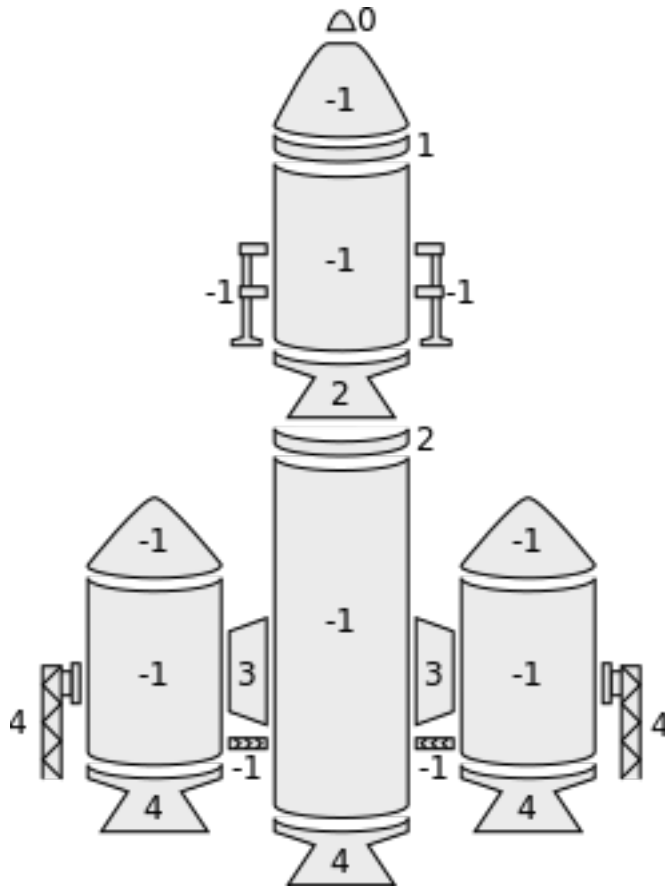


Fig. 5.15: **Figure 7** – The stage in which each part is *activated*.

5.3.8 Resources

public class **Resources**

Created by calling
`Vessel.getResources()`,
`Vessel.resourcesInDecoupleStage(int, boolean)` or `Part.getResources()`.

`java.util.List<String> getNames ()`

A list of resource names that can be stored.

boolean **hasResource** (`String name`)

Check whether the named resource can be stored.

Parameters

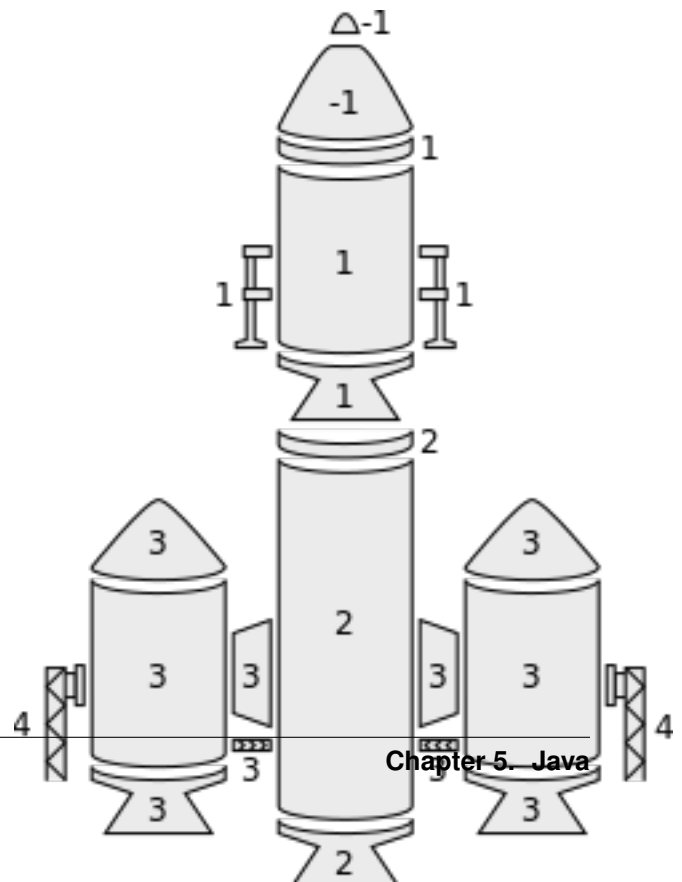
- **name** (`String`) – The name of the resource.

float **max** (`String name`)

Returns the amount of a resource that can be stored.

Parameters

- **name** (`String`) – The name of the resource.



float **amount** (*String name*)

Returns the amount of a resource that is currently stored.

Parameters

- **name** (*String*) – The name of the resource.

float **density** (*String name*)

Returns the density of a resource, in kg/l.

Parameters

- **name** (*String*) – The name of the resource.

ResourceFlowMode **flowMode** (*String name*)

Returns the flow mode of a resource.

Parameters

- **name** (*String*) – The name of the resource.

public enum **ResourceFlowMode**

See *Resources.flowMode* (*String*).

public *ResourceFlowMode* **VESSEL**

The resource flows to any part in the vessel. For example, electric charge.

public *ResourceFlowMode* **STAGE**

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

public *ResourceFlowMode* **ADJACENT**

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

public *ResourceFlowMode* **NONE**

The resource does not flow. For example, solid fuel.

5.3.9 Node

public class **Node**

Represents a maneuver node. Can be created using *Control.addNode* (*double*, *float*, *float*, *float*).

float **getPrograde** ()

void **setPrograde** (*float value*)

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

float **getNormal** ()

void **setNormal** (*float value*)

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

float **getRadial** ()

void **setRadial** (float *value*)

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

float **getDeltaV** ()

void **setDeltaV** (float *value*)

The delta-v of the maneuver node, in meters per second.

Note: Does not change when executing the maneuver node. See [Node.getRemainingDeltaV\(\)](#).

float **getRemainingDeltaV** ()

Gets the remaining delta-v of the maneuver node, in meters per second. Changes as the node is executed. This is equivalent to the delta-v reported in-game.

org.javatuples.Triplet<Double, Double, Double> **burnVector** (*ReferenceFrame* *referenceFrame*)

Returns a vector whose direction the direction of the maneuver node burn, and whose magnitude is the delta-v of the burn in m/s.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

Note:	Does	not
change	when	exe-
cuting	the	maneu-
ver	node.	See

[Node.remainingBurnVector \(ReferenceFrame\)](#).

org.javatuples.Triplet<Double, Double, Double> **remainingBurnVector** (*ReferenceFrame* *referenceFrame*)

Returns a vector whose direction the direction of the maneuver node burn, and whose magnitude is the delta-v of the burn in m/s. The direction and magnitude change as the burn is executed.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

double **getUT** ()

void **setUT** (double *value*)

The universal time at which the maneuver will occur, in seconds.

double **getTimeTo** ()

The time until the maneuver node will be encountered, in seconds.

Orbit **getOrbit** ()

The orbit that results from executing the maneuver node.

void **remove** ()

Removes the maneuver node.

ReferenceFrame **getReferenceFrame** ()

Gets the reference frame that is fixed relative to the maneuver node's burn.

- The origin is at the position of the maneuver node.
- The y-axis points in the direction of the burn.
- The x-axis and z-axis point in arbitrary but fixed directions.

ReferenceFrame **getOrbitalReferenceFrame** ()

Gets the reference frame that is fixed relative to the maneuver node, and orientated with the orbital prograde/normal/radial directions of the original orbit at the maneuver node's position.

- The origin is at the position of the maneuver node.
- The x-axis points in the orbital anti-radial direction of the original orbit, at the position of the maneuver node.
- The y-axis points in the orbital prograde direction of the original orbit, at the position of the maneuver node.
- The z-axis points in the orbital normal direction of the original orbit, at the position of the maneuver node.

org.javatuples.Triplet<Double, Double, Double> **position** (*ReferenceFrame* referenceFrame)

Returns the position vector of the maneuver node in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

org.javatuples.Triplet<Double, Double, Double> **direction** (*ReferenceFrame* referenceFrame)

Returns the unit direction vector of the maneuver nodes burn in the given reference frame.

Parameters

- **referenceFrame** (*ReferenceFrame*) –

5.3.10 Comms

public class **Comms**

Used to interact with RemoteTech. Created using a call to *Vessel.getComms* ().

Note: This class requires [RemoteTech](#) to be installed.

boolean **getHasLocalControl** ()
Whether the vessel can be controlled locally.

boolean **getHasFlightComputer** ()
Whether the vessel has a RemoteTech flight computer on board.

boolean **getHasConnection** ()
Whether the vessel can receive commands from the KSC or a command station.

boolean **getHasConnectionToGroundStation** ()
Whether the vessel can transmit science data to a ground station.

double **getSignalDelay** ()
The signal delay when sending commands to the vessel, in seconds.

double **getSignalDelayToGroundStation** ()
The signal delay between the vessel and the closest ground station, in seconds.

double **signalDelayToVessel** (*Vessel other*)
Returns the signal delay between the current vessel and another vessel, in seconds.

Parameters

- **other** (*Vessel*) –

5.3.11 ReferenceFrame

public class **ReferenceFrame**
Represents a reference frame for positions, rotations and velocities. Contains:

- The position of the origin.
- The directions of the x, y and z axes.
- The linear velocity of the frame.
- The angular velocity of the frame.

Note: This class does not contain any properties or methods. It is only used as a parameter to other functions.

5.3.12 AutoPilot

public class **AutoPilot**

Provides basic auto-piloting utilities for a vessel.

Created by calling *Vessel.getAutoPilot()*.

void **engage** ()

Engage the auto-pilot.

void **disengage** ()

Disengage the auto-pilot.

void **wait** ()

Blocks until the vessel is pointing in the target direction (if set) and has the target roll (if set).

float **getError** ()

The error, in degrees, between the direction the ship has been asked to point in and the direction it is pointing in. Returns zero if the auto-pilot has not been engaged, SAS is not enabled, SAS is in stability assist mode, or no target direction is set.

float **getRollError** ()

The error, in degrees, between the roll the ship has been asked to be in and the actual roll. Returns zero if the auto-pilot has not been engaged or no target roll is set.

ReferenceFrame **getReferenceFrame** ()

void **setReferenceFrame** (*ReferenceFrame* value)

The reference frame for the target direction (*AutoPilot.getTargetDirection()*).

org.javatuples.Triplet<Double, Double, Double> **getTargetDirection** ()

void **setTargetDirection** (org.javatuples.Triplet<Double, Double, Double> value)

The target direction. null if no target direction is set.

void **targetPitchAndHeading** (float pitch, float heading)

Set (*AutoPilot.getTargetDirection()*) from a pitch and heading angle.

Parameters

- **pitch** (*float*) – Target pitch angle, in degrees between -90° and +90°.
- **heading** (*float*) – Target heading angle, in degrees between 0° and 360°.

float **getTargetRoll** ()

void **setTargetRoll** (float value)

The target roll, in degrees. NaN if no target roll is set.

boolean **getSAS** ()

void **setSAS** (boolean *value*)

The state of SAS.

Note: Equivalent to *Control.getSAS()*

SASMode **getSASMode** ()

void **setSASMode** (*SASMode value*)

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

Note: Equivalent to *Control.getSASMode()*

float **getRotationSpeedMultiplier** ()

void **setRotationSpeedMultiplier** (float *value*)

Target rotation speed multiplier. Defaults to 1.

float **getMaxRotationSpeed** ()

void **setMaxRotationSpeed** (float *value*)

Maximum target rotation speed. Defaults to 1.

float **getRollSpeedMultiplier** ()

void **setRollSpeedMultiplier** (float *value*)

Target roll speed multiplier. Defaults to 1.

float **getMaxRollSpeed** ()

void **setMaxRollSpeed** (float *value*)

Maximum target roll speed. Defaults to 1.

void **setPIDParameters** (float *Kp*, float *Ki*, float *Kd*)

Sets the gains for the rotation rate PID controller.

Parameters

- **Kp** (*float*) – Proportional gain.
- **Ki** (*float*) – Integral gain.
- **Kd** (*float*) – Derivative gain.

5.3.13 Geometry Types

3-dimensional vectors are represented as a 3-tuple.
For example:


```
import java.io.IOException;
import org.javatuples.Triplet;
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;

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());
    }
}
```

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

```
import java.io.IOException;
import org.javatuples.Quartet;
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;

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());
    }
}
```

5.4 InfernalRobotics API

Provides RPCs to interact with the `InfernalRobotics` mod. Provides the following classes:

5.4.1 InfernalRobotics

public class **InfernalRobotics**

This service provides functionality to interact with the `InfernalRobotics` mod.

java.util.List<*ControlGroup*> **getServoGroups**()

A list of all the servo groups in the active vessel.

ControlGroup **servoGroupWithName**(String name)

Returns the servo group with the given *name* or `null` if none exists. If multiple servo groups have the same name, only one of them is returned.

Parameters

- **name** (*String*) – Name of servo group to find.

Servo **servoWithName** (*String name*)

Returns the servo with the given *name*, from all servo groups, or `null` if none exists. If multiple servos have the same name, only one of them is returned.

Parameters

- **name** (*String*) – Name of the servo to find.

5.4.2 ControlGroup

public class **ControlGroup**

A group of servos, obtained by calling `getServoGroups()` or `servoGroupWithName(String)`. Represents the “Servo Groups” in the InfernalRobotics UI.

String **getName** ()

void **setName** (*String value*)

The name of the group.

String **getForwardKey** ()

void **setForwardKey** (*String value*)

The key assigned to be the “forward” key for the group.

String **getReverseKey** ()

void **setReverseKey** (*String value*)

The key assigned to be the “reverse” key for the group.

float **getSpeed** ()

void **setSpeed** (float *value*)

The speed multiplier for the group.

boolean **getExpanded** ()

void **setExpanded** (boolean *value*)

Whether the group is expanded in the Infernal-Robotics UI.

java.util.List<*Servo*> **getServos** ()

The servos that are in the group.

Servo **servoWithName** (*String name*)

Returns the servo with the given *name* from this group, or null if none exists.

Parameters

- **name** (*String*) – Name of servo to find.

void **moveRight** ()

Moves all of the servos in the group to the right.

void **moveLeft** ()

Moves all of the servos in the group to the left.

void **moveCenter** ()

Moves all of the servos in the group to the center.

void **moveNextPreset** ()

Moves all of the servos in the group to the next preset.

void **movePrevPreset** ()

Moves all of the servos in the group to the previous preset.

void **stop** ()

Stops the servos in the group.

5.4.3 Servo

public class **Servo**

Represents a servo. Obtained using *ControlGroup.getServos()*, *ControlGroup.servoWithName(String)* or *servoWithName(String)*.

String **getName** ()

void **setName** (*String value*)

The name of the servo.

void **setHighlight** (boolean *value*)

Whether the servo should be highlighted in-game.

float **getPosition** ()

The position of the servo.

float **getMinConfigPosition** ()

The minimum position of the servo, specified by the part configuration.

float **getMaxConfigPosition** ()

The maximum position of the servo, specified by the part configuration.

float **getMinPosition** ()

void **setMinPosition** (float *value*)
The minimum position of the servo, specified by the in-game tweak menu.

float **getMaxPosition** ()

void **setMaxPosition** (float *value*)
The maximum position of the servo, specified by the in-game tweak menu.

float **getConfigSpeed** ()
The speed multiplier of the servo, specified by the part configuration.

float **getSpeed** ()

void **setSpeed** (float *value*)
The speed multiplier of the servo, specified by the in-game tweak menu.

float **getCurrentSpeed** ()

void **setCurrentSpeed** (float *value*)
The current speed at which the servo is moving.

float **getAcceleration** ()

void **setAcceleration** (float *value*)
The current speed multiplier set in the UI.

boolean **getIsMoving** ()
Whether the servo is moving.

boolean **getIsFreeMoving** ()
Whether the servo is freely moving.

boolean **getIsLocked** ()

void **setIsLocked** (boolean *value*)
Whether the servo is locked.

boolean **getIsAxisInverted** ()

void **setIsAxisInverted** (boolean *value*)
Whether the servos axis is inverted.

void **moveRight** ()
Moves the servo to the right.

void **moveLeft** ()
Moves the servo to the left.

void **moveCenter** ()
Moves the servo to the center.

void **moveNextPreset** ()
Moves the servo to the next preset.

void **movePrevPreset** ()

Moves the servo to the previous preset.

void **moveTo** (float *position*, float *speed*)

Moves the servo to *position* and sets the speed multiplier to *speed*.

Parameters

- **position** (*float*) – The position to move the servo to.
- **speed** (*float*) – Speed multiplier for the movement.

void **stop** ()

Stops the servo.

5.4.4 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 java.io.IOException;
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.InfernalRobotics;
import krpc.client.services.InfernalRobotics.ControlGroup;
import krpc.client.services.InfernalRobotics.Servo;

public class IR {
    public static void main(String[] args) throws IOException, RPCException, InterruptedException {
        Connection connection = Connection.newInstance("InfernalRobotics Example");
        InfernalRobotics ir = InfernalRobotics.newInstance(connection);

        ControlGroup group = ir.servoGroupWithName("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();
    }
}
```

5.5 Kerbal Alarm Clock API

Provides RPCs to interact with the [Kerbal Alarm Clock](#) mod. Provides the following classes:

5.5.1 KerbalAlarmClock

public class **KerbalAlarmClock**

This service provides functionality to interact with the *Kerbal Alarm Clock* mod.

java.util.List<*Alarm*> **getAlarms** ()

A list of all the alarms.

Alarm **alarmWithName** (*String* name)

Get the alarm with the given *name*, or null if no alarms have that name. If more than one alarm has the name, only returns one of them.

Parameters

- **name** (*String*) – Name of the alarm to search for.

java.util.List<*Alarm*> **alarmsWithType** (*AlarmType* type)

Get a list of alarms of the specified *type*.

Parameters

- **type** (*AlarmType*) – Type of alarm to return.

Alarm **createAlarm** (*AlarmType* type, *String* name, double ut)

Create a new alarm and return it.

Parameters

- **type** (*AlarmType*) – Type of the new alarm.
- **name** (*String*) – Name of the new alarm.
- **ut** (*double*) – Time at which the new alarm should trigger.

5.5.2 Alarm

public class **Alarm**

Represents an alarm. Obtained by calling *getAlarms* (), *alarmWithName* (*String*) or *alarmsWithType* (*AlarmType*).

AlarmAction **getAction** ()

void **setAction** (*AlarmAction* value)

The action that the alarm triggers.

double **getMargin** ()

void **setMargin** (double value)

The number of seconds before the event that the alarm will fire.

double **getTime** ()

void **setTime** (double value)

The time at which the alarm will fire.

AlarmType **getType** ()

The type of the alarm.

String **getID** ()

The unique identifier for the alarm.

String **getName** ()

void **setName** (*String value*)

The short name of the alarm.

String **getNotes** ()

void **setNotes** (*String value*)

The long description of the alarm.

double **getRemaining** ()

The number of seconds until the alarm will fire.

boolean **getRepeat** ()

void **setRepeat** (boolean *value*)

Whether the alarm will be repeated after it has fired.

double **getRepeatPeriod** ()

void **setRepeatPeriod** (double *value*)

The time delay to automatically create an alarm after it has fired.

Vessel **getVessel** ()

void **setVessel** (*Vessel value*)

The vessel that the alarm is attached to.

CelestialBody **getXferOriginBody** ()

void **setXferOriginBody** (*CelestialBody value*)

The celestial body the vessel is departing from.

CelestialBody **getXferTargetBody** ()

void **setXferTargetBody** (*CelestialBody value*)

The celestial body the vessel is arriving at.

void **remove** ()

Removes the alarm.

5.5.3 AlarmType

public enum **AlarmType**

The type of an alarm.

public *AlarmType* **RAW**

An alarm for a specific date/time or a specific period in the future.

public *AlarmType* **MANEUVER**

An alarm based on the next maneuver node on the current ships flight path. This node will be stored and can be restored when you come back to the ship.

public *AlarmType* **MANEUVER_AUTO**

See *AlarmType.MANEUVER*.

public *AlarmType* **APOAPSIS**

An alarm for furthest part of the orbit from the planet.

public *AlarmType* **PERIAPSIS**

An alarm for nearest part of the orbit from the planet.

public *AlarmType* **ASCENDING_NODE**

Ascending node for the targeted object, or equatorial ascending node.

public *AlarmType* **DESCENDING_NODE**

Descending node for the targeted object, or equatorial descending node.

public *AlarmType* **CLOSEST**

An alarm based on the closest approach of this vessel to the targeted vessel, some number of orbits into the future.

public *AlarmType* **CONTRACT**

An alarm based on the expiry or deadline of contracts in career modes.

public *AlarmType* **CONTRACT_AUTO**

See *AlarmType.CONTRACT*.

public *AlarmType* **CREW**

An alarm that is attached to a crew member.

public *AlarmType* **DISTANCE**

An alarm that is triggered when a selected target comes within a chosen distance.

public *AlarmType* **EARTH_TIME**

An alarm based on the time in the “Earth” alternative Universe (aka the Real World).

public *AlarmType* **LAUNCH_RENDEVOUS**

An alarm that fires as your landed craft passes under the orbit of your target.

public *AlarmType* **SOI_CHANGE**

An alarm manually based on when the next SOI point is on the flight path or set to continually monitor the active flight path and add alarms as it detects SOI changes.

public *AlarmType* **SOI_CHANGE_AUTO**
See *AlarmType.SOI_CHANGE*.

public *AlarmType* **TRANSFER**
An alarm based on Interplanetary Transfer Phase
Angles, i.e. when should I launch to planet X?
Based on Kosmo Not's post and used in Olex's
Calculator.

public *AlarmType* **TRANSFER_MODELLED**
See *AlarmType.TRANSFER*.

5.5.4 AlarmAction

public enum **AlarmAction**
The action performed by an alarm when it fires.

public *AlarmAction* **DO_NOTHING**
Don't do anything at all...

public *AlarmAction* **DO_NOTHING_DELETE_WHEN_PASSED**
Don't do anything, and delete the alarm.

public *AlarmAction* **KILL_WARP**
Drop out of time warp.

public *AlarmAction* **KILL_WARP_ONLY**
Drop out of time warp.

public *AlarmAction* **MESSAGE_ONLY**
Display a message.

public *AlarmAction* **PAUSE_GAME**
Pause the game.

5.5.5 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 java.io.IOException;
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.KerbalAlarmClock;
import krpc.client.services.KerbalAlarmClock.Alarm;
import krpc.client.services.KerbalAlarmClock.AlarmAction;
import krpc.client.services.KerbalAlarmClock.AlarmType;

public class KAC {
    public static void main(String[] args) throws IOException, RPCException {
        Connection connection = Connection.newInstance("Kerbal Alarm Clock Example", "10.0.2.2");
        KerbalAlarmClock kac = KerbalAlarmClock.newInstance(connection);
        Alarm alarm = kac.createAlarm(AlarmType.RAW, "My New Alarm", SpaceCenter.newInstance(connection).getActiveVessel());
        alarm.setNotes("10 seconds have now passed since the alarm was created.");
        alarm.setAction(AlarmAction.MESSAGE_ONLY);
    }
}
```

```
}  
}
```

6.1 Lua Client

This client provides functionality to interact with a kRPC server from programs written in Lua. It can be [installed](#) using [LuaRocks](#) or downloaded from [GitHub](#).

6.1.1 Installing the Library

The Lua client and all of its dependencies can be installed using `luarocks` with a single command:

```
luarocks install krpc
```

6.1.2 Using the Library

Once it's installed, simply `require 'krpc'` and you are good to go!

6.1.3 Connecting to the Server

To connect to a server, use the `krpc.connect()` function. This returns a connection object through which you can interact with the server. For example to connect to a server running on the local machine:

```
local krpc = require 'krpc'
local conn = krpc.connect('Example')
print(conn.krpc:get_status().version)
```

This function also accepts arguments that specify what address and port numbers to connect to. For example:

```
local krpc = require 'krpc'
local conn = krpc.connect('Remote example', 'my.domain.name', 1000, 1001)
print(conn.krpc:get_status().version)
```

6.1.4 Interacting with the Server

Interaction with the server is performed via the client object (of type `krpc.Client`) returned when connecting to the server using `krpc.connect()`.

Upon connecting, the client interrogates the server to find out what functionality it provides and dynamically adds all of the classes, methods, properties to the client object.

For example, all of the functionality provided by the SpaceCenter service is accessible via `conn.space_center` and the functionality provided by the InfernalRobotics service is accessible via `conn.infernal_robotics`.

Calling methods, getting or setting properties, etc. are mapped to remote procedure calls and passed to the server by the lua client.

6.1.5 Streaming Data from the Server

Streams are not yet supported by the Lua client.

6.1.6 Reference

connect (`[name=nil]` `[, address='127.0.0.1']` `[, rpc_port=50000]` `[, stream_port=50001]`)

This function creates a connection to a kRPC server. It returns a `krpc.Client` object, through which the server can be communicated with.

Parameters

- **name** (*string*) – A descriptive name for the connection. This is passed to the server and appears, for example, in the client connection dialog on the in-game server window.
- **address** (*string*) – The address of the server to connect to. Can either be a hostname or an IP address in dotted decimal notation. Defaults to '127.0.0.1'.
- **rpc_port** (*number*) – The port number of the RPC Server. Defaults to 50000.
- **stream_port** (*number*) – The port number of the Stream Server. Defaults to 50001.

class Client

This class provides the interface for communicating with the server. It is dynamically populated with all the functionality provided by the server. Instances of this class should be obtained by calling `krpc.connect()`.

close()

Closes the connection to the server.

krpc

The built-in KRPC class, providing basic interactions with the server.

Return type `krpc.KRPC`

class KRPC

This class provides access to the basic server functionality provided by the KRPC service. An instance can be obtained by calling `krpc.Client.krpc`. Most of this functionality is used internally by the lua client and therefore does not need to be used directly from application code. The only exception that may be useful is:

get_status()

Gets a status message from the server containing information including the server's version string and performance statistics.

For example, the following prints out the version string for the server:

```
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 status = conn.krpc:get_status()
print('Data in = ' .. (status.bytes_read_rate/1024) .. ' KB/s')
print('Data out = ' .. (status.bytes_written_rate/1024) .. ' KB/s')
```

6.2 KRPC API

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

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`

static `add_stream(request)`

Add a streaming request and return its identifier.

Parameters `request` (`krpc.schema.KRPC.Request`) –

Return type number

Note: Streams are not supported by the Lua client.

static `remove_stream(id)`

Remove a streaming request.

Parameters `id` (`number`) –

Note: Streams are not supported by the Lua client.

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.

6.3 SpaceCenter API

6.3.1 SpaceCenter

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

active_vessel

The currently active vessel.

Attribute Can be read or written

Return type *SpaceCenter.Vessel*

vessels

A list of all the vessels in the game.

Attribute Read-only, cannot be set

Return type List of *SpaceCenter.Vessel*

bodies

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

Attribute Read-only, cannot be set

Return type Map from string to *SpaceCenter.CelestialBody*

target_body

The currently targeted celestial body.

Attribute Can be read or written

Return type *SpaceCenter.CelestialBody*

target_vessel

The currently targeted vessel.

Attribute Can be read or written

Return type *SpaceCenter.Vessel*

target_docking_port

The currently targeted docking port.

Attribute Can be read or written

Return type *SpaceCenter.DockingPort*

static clear_target ()

Clears the current target.

static launch_vessel_from_vab (name)

Launch a new vessel from the VAB onto the launchpad.

Parameters **name** (*string*) – Name of the vessel's craft file.

static launch_vessel_from_sph (name)

Launch a new vessel from the SPH onto the runway.

Parameters **name** (*string*) – Name of the vessel's craft file.

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_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*

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 vector from one reference frame to another.

Parameters

- **position** (*Tuple*) – Position vector in reference frame *from*.
- **from** (*SpaceCenter.ReferenceFrame*) – The reference frame that the position vector is in.
- **to** (*SpaceCenter.ReferenceFrame*) – The reference frame to convert the position vector to.

Returns The corresponding position vector in reference frame *to*.

Return type Tuple of (number, number, number)

static transform_direction (*direction*, *from*, *to*)

Converts a direction vector from one reference frame to another.

Parameters

- **direction** (*Tuple*) – Direction vector in reference frame *from*.
- **from** (*SpaceCenter.ReferenceFrame*) – The reference frame that the direction vector is in.
- **to** (*SpaceCenter.ReferenceFrame*) – The reference frame to convert the direction vector to.

Returns The corresponding direction 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 in reference frame *from*.
- **from** (*SpaceCenter.ReferenceFrame*) – The reference frame that the rotation is in.

- **to** ([SpaceCenter.ReferenceFrame](#)) – The corresponding rotation in reference frame *to*.

Returns The corresponding rotation in reference frame *to*.

Return type Tuple of (number, number, number, number)

static transform_velocity (*position, velocity, from, to*)

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

Parameters

- **position** (*Tuple*) – Position vector in reference frame *from*.
- **velocity** (*Tuple*) – Velocity vector in reference frame *from*.
- **from** ([SpaceCenter.ReferenceFrame](#)) – The reference frame that the position and velocity vectors are in.
- **to** ([SpaceCenter.ReferenceFrame](#)) – The reference frame to convert the velocity vector to.

Returns The corresponding velocity in reference frame *to*.

Return type Tuple of (number, number, number)

far_available

Whether [Ferram Aerospace Research](#) is installed.

Attribute Read-only, cannot be set

Return type boolean

remote_tech_available

Whether [RemoteTech](#) is installed.

Attribute Read-only, cannot be set

Return type boolean

static draw_direction (*direction, reference_frame, color* [, *length = 10.0*])

Draw a direction vector on the active vessel.

Parameters

- **direction** (*Tuple*) – Direction to draw the line in.
- **reference_frame** ([SpaceCenter.ReferenceFrame](#)) – Reference frame that the direction is in.
- **color** (*Tuple*) – The color to use for the line, as an RGB color.
- **length** (*number*) – The length of the line. Defaults to 10.

static draw_line (*start, end, reference_frame, color*)

Draw a line.

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 position are in.
- **color** (*Tuple*) – The color to use for the line, as an RGB color.

static clear_drawing ()

Remove all directions and lines currently being drawn.

class WarpMode

Returned by *SpaceCenter.WarpMode*

rails

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

physics

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

none

Time warp is not active.

6.3.2 Vessel

class Vessel

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources.

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*

met

The mission elapsed time in seconds.

Attribute Read-only, cannot be set

Return type number

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 speed* and *surface speed* of a vessel.

target

The target vessel. `nil` if there is no target. When setting the target, the target cannot be the current vessel.

Attribute Can be read or written

Return type `SpaceCenter.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`

auto_pilot

An `SpaceCenter.AutoPilot` object, that can be used to perform simple auto-piloting of the vessel.

Attribute Read-only, cannot be set

Return type `SpaceCenter.AutoPilot`

resources

A `SpaceCenter.Resources` object, that can 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*

comms

A *SpaceCenter.Comms* object, that can used to interact with RemoteTech for this vessel.

Attribute Read-only, cannot be set

Return type *SpaceCenter.Comms*

Note: Requires [RemoteTech](#) to be installed.

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

`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*

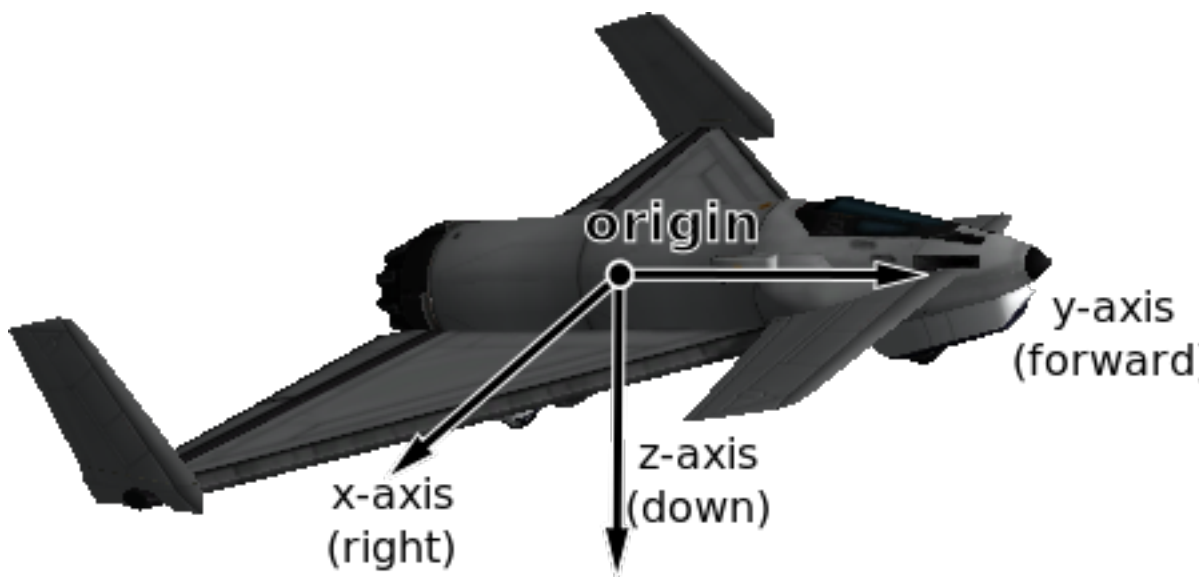


Fig. 6.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

`orbital_reference_frame`

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital pro-grade/normal/radial directions.

- The origin is at the center of mass of the vessel.

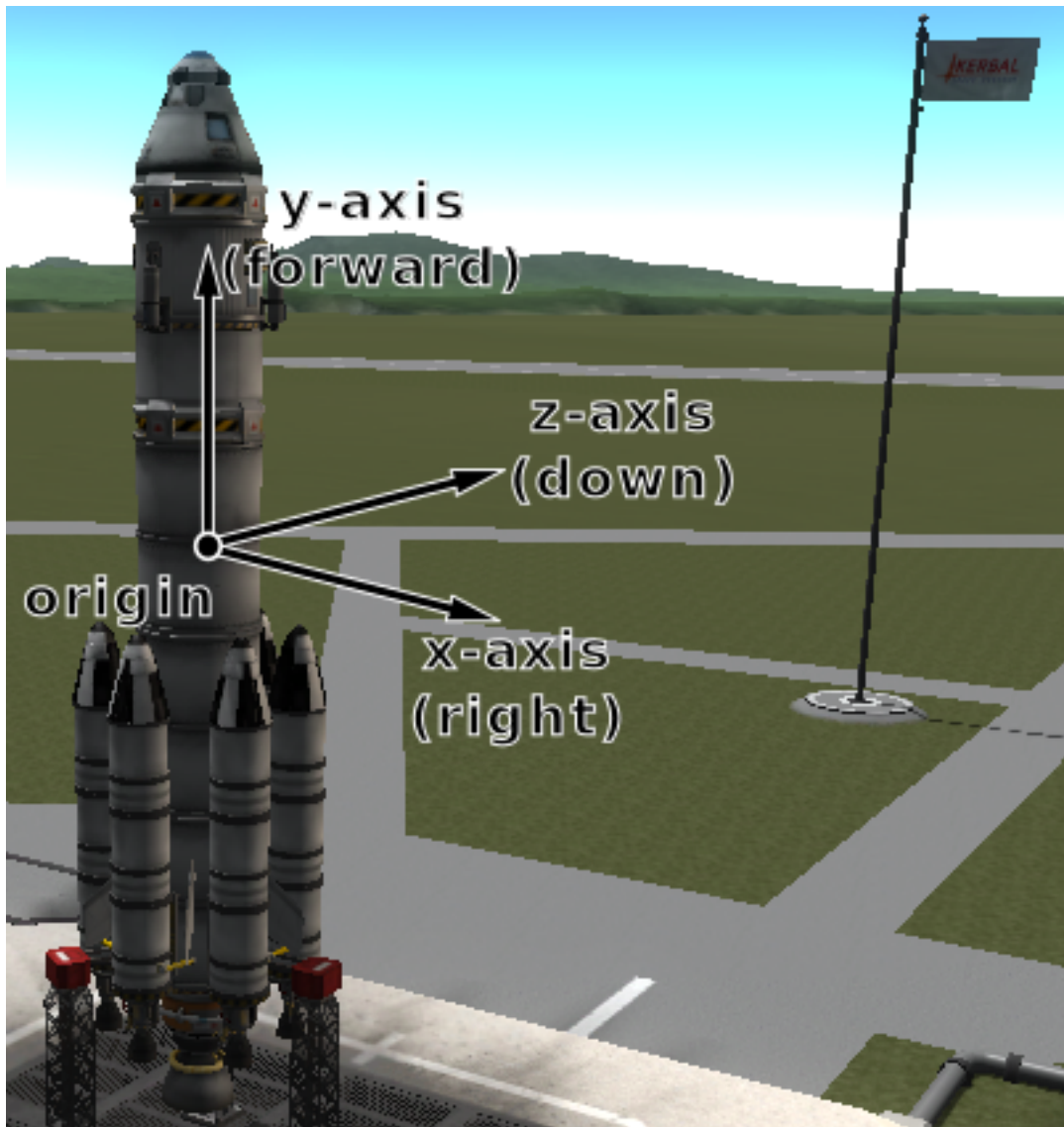


Fig. 6.2: Vessel reference frame origin and axes for the Kerbal-X rocket

- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

Attribute Read-only, cannot be set

Return type *SpaceCenter.ReferenceFrame*

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

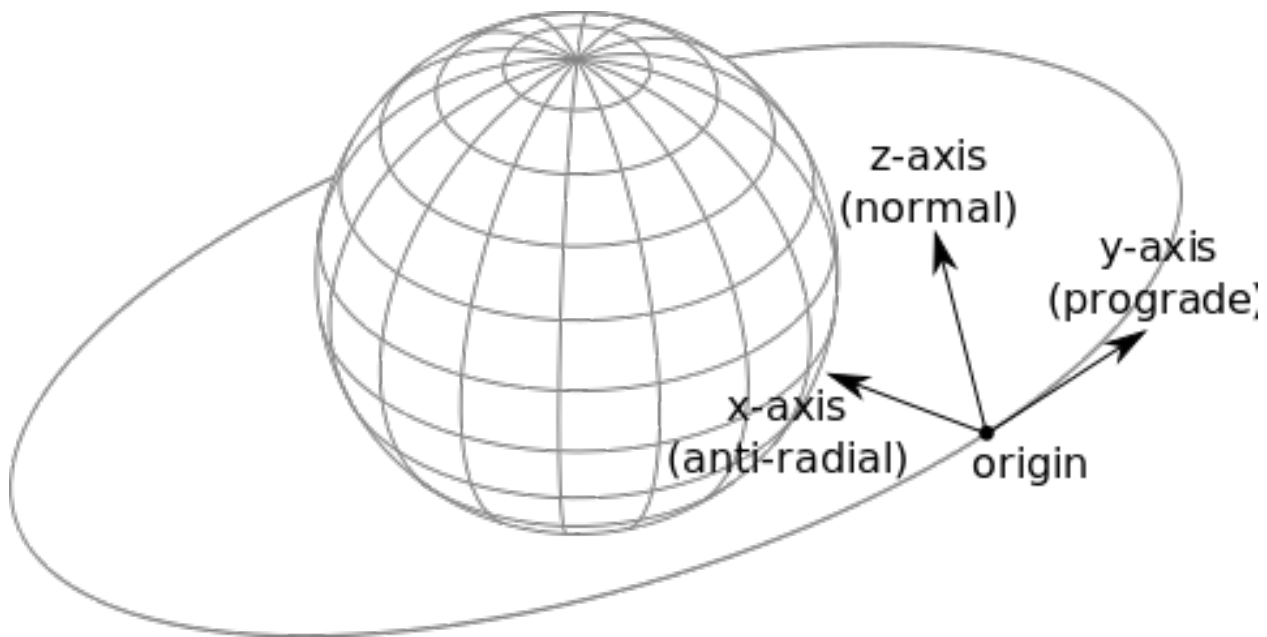


Fig. 6.3: Vessel orbital reference frame origin and axes

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.

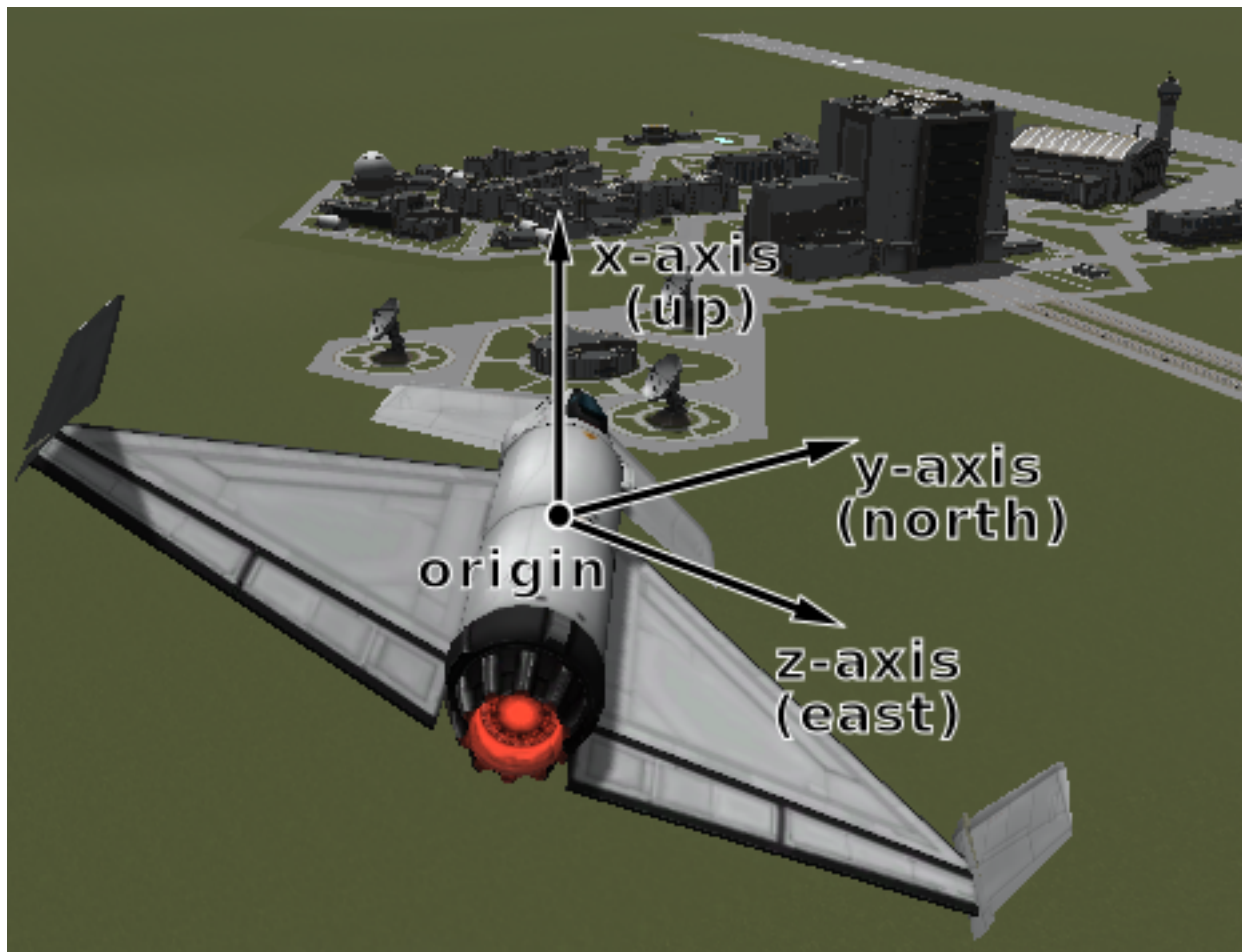


Fig. 6.4: Vessel surface reference frame origin and axes

`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 *SpaceCenter.ReferenceFrame*

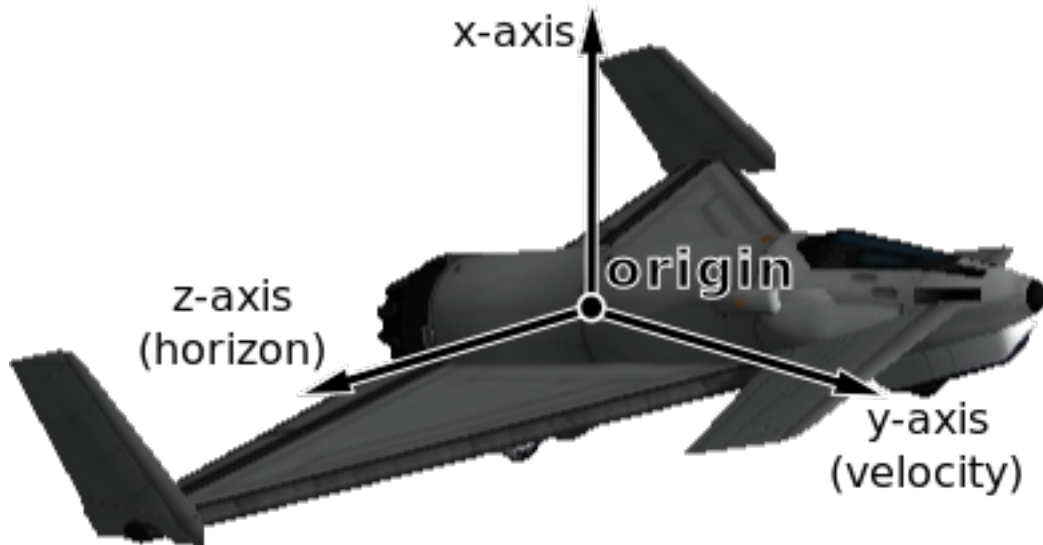


Fig. 6.5: Vessel surface velocity reference frame origin and axes

position (*reference_frame*)

Returns the position vector of the center of mass of the vessel in the given reference frame.

Parameters **reference_frame** (`SpaceCenter.ReferenceFrame`) –

Return type Tuple of (number, number, number)

velocity (*reference_frame*)

Returns the velocity vector of the center of mass of the vessel in the given reference frame.

Parameters **reference_frame** (`SpaceCenter.ReferenceFrame`) –

Return type Tuple of (number, number, number)

rotation (*reference_frame*)

Returns the rotation of the center of mass of the vessel in the given reference frame.

Parameters **reference_frame** (`SpaceCenter.ReferenceFrame`) –

Return type Tuple of (number, number, number, number)

direction (*reference_frame*)

Returns the direction in which the vessel is pointing, as a unit vector, in the given reference frame.

Parameters **reference_frame** (`SpaceCenter.ReferenceFrame`) –

Return type Tuple of (number, number, number)

angular_velocity (*reference_frame*)

Returns the angular velocity of the vessel in the given reference frame. The magnitude of the returned vector is the rotational speed in radians per second, and the direction of the vector indicates the axis of rotation (using the right hand rule).

Parameters **reference_frame** (`SpaceCenter.ReferenceFrame`) –

Return type Tuple of (number, number, number)

class VesselType

See `SpaceCenter.Vessel.type`.

ship
Ship.

station
Station.

lander
Lander.

probe
Probe.

rover
Rover.

base
Base.

debris
Debris.

class VesselSituation
See *SpaceCenter.Vessel.situation*.

docked
Vessel is docked to another.

escaping
Escaping.

flying
Vessel is flying through an atmosphere.

landed
Vessel is landed on the surface of a body.

orbiting
Vessel is orbiting a body.

pre_launch
Vessel is awaiting launch.

splashed
Vessel has splashed down in an ocean.

sub_orbital
Vessel is on a sub-orbital trajectory.

6.3.3 CelestialBody

class CelestialBody
Represents a celestial body (such as a planet or moon).

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 $m^3 s^{-2}$.

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

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 at the given position, in meters. 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 at the given position, in meters. When over water, this is the height of the sea-bed and is therefore a 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

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

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

Return type Tuple of (number, number, 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

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

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*

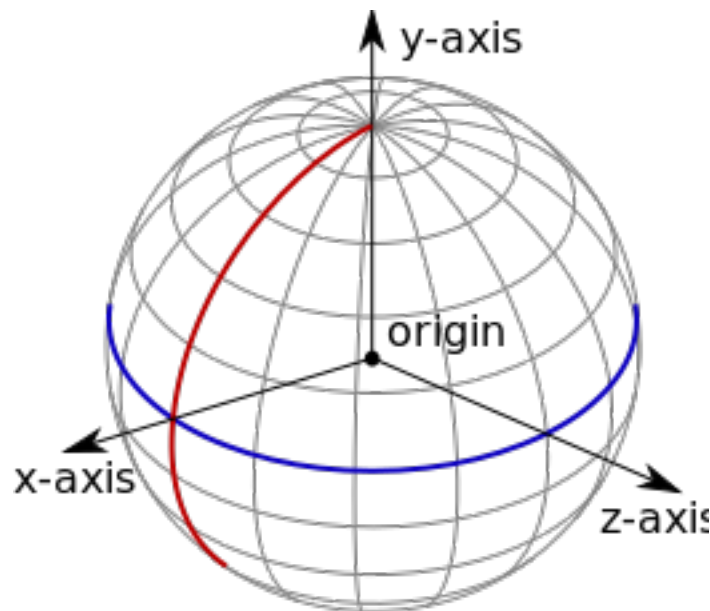


Fig. 6.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

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

Gets 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*)

Returns the position vector of the center of the body in the specified reference frame.

Parameters **reference_frame** (*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number)

velocity (*reference_frame*)

Returns the velocity vector of the body in the specified reference frame.

Parameters **reference_frame** (*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number)

rotation (*reference_frame*)

Returns the rotation of the body in the specified reference frame.

Parameters **reference_frame** (*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number, number)

direction (*reference_frame*)

Returns the direction in which the north pole of the celestial body is pointing, as a unit vector, in the specified reference frame.

Parameters **reference_frame** (*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number)

angular_velocity (*reference_frame*)

Returns the angular velocity of the body in the specified reference frame. The magnitude of the vector is the rotational speed of the body, in radians per second, and the direction of the vector indicates the axis of rotation, using the right-hand rule.

Parameters **reference_frame** (*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number)

6.3.4 Flight

class **Flight**

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

Note: To get orbital information, such as the apoapsis or inclination, see `SpaceCenter.Orbit`.

g_force

The current G force acting on the vessel in m/s^2 .

Attribute Read-only, cannot be set

Return type number

mean_altitude

The altitude above sea level, in meters.

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.

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.

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 vector of the vessel. The magnitude of the vector is the speed of the vessel in meters per second. The direction of the vector is the direction of the vessels motion.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

speed

The speed of the vessel in meters per second.

Attribute Read-only, cannot be set

Return type number

horizontal_speed

The horizontal speed of the vessel in meters per second.

Attribute Read-only, cannot be set

Return type number

vertical_speed

The vertical speed of the vessel in meters per second.

Attribute Read-only, cannot be set

Return type number

center_of_mass

The position of the center of mass of the vessel.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

rotation

The rotation of the vessel.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number, number)

direction

The direction vector that the vessel is pointing in.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

pitch

The pitch angle 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 angle of the vessel relative to north, in degrees. A value between 0° and 360°.

Attribute Read-only, cannot be set

Return type number

roll

The roll angle 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 unit direction vector pointing in the prograde direction.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

retrograde

The unit direction vector pointing in the retrograde direction.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

normal

The unit direction vector pointing in the normal direction.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

anti_normal

The unit direction vector pointing in the anti-normal direction.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

radial

The unit direction vector pointing in the radial direction.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

anti_radial

The unit direction vector pointing in the anti-radial direction.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

atmosphere_density

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

Attribute Read-only, cannot be set

Return type number

dynamic_pressure

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

Attribute Read-only, cannot be set

Return type number

Note: Calculated using [KSPs stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

static_pressure

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

Attribute Read-only, cannot be set

Return type number

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

aerodynamic_force

The total aerodynamic forces acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

lift

The [aerodynamic lift](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

drag

The [aerodynamic drag](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Attribute Read-only, cannot be set

Return type Tuple of (number, number, number)

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

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

Note: Not available when [Ferram Aerospace Research](#) is installed.

mach

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

Attribute Read-only, cannot be set

Return type number

Note: Not available when [Ferram Aerospace Research](#) is installed.

equivalent_air_speed

The [equivalent air speed](#) of the vessel, in m/s .

Attribute Read-only, cannot be set

Return type number

Note: Not available when [Ferram Aerospace Research](#) is installed.

terminal_velocity

An estimate of the current terminal velocity of the vessel, in m/s . This is the speed at which the drag forces cancel out the force of gravity.

Attribute Read-only, cannot be set

Return type number

Note: Calculated using [KSP's stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

angle_of_attack

Gets 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

Gets 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 temperature 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

Gets 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

Gets 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

Gets 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

Gets the [ballistic coefficient](#).

Attribute Read-only, cannot be set

Return type number

Note: Requires [Ferram Aerospace Research](#).

thrust_specific_fuel_consumption

Gets 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 product one newton of thrust.

Attribute Read-only, cannot be set

Return type number

Note: Requires [Ferram Aerospace Research](#).

6.3.5 Orbit

class Orbit

Describes an orbit. For example, the orbit of a vessel, obtained by calling *SpaceCenter.Vessel.orbit*, or a celestial body, obtained by calling *SpaceCenter.CelestialBody.orbit*.

body

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

Attribute Read-only, cannot be set

Return type *SpaceCenter.CelestialBody*

apoapsis

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

Attribute Read-only, cannot be set

Return type number

Note: For the apoapsis altitude reported on the in-game map view, use *SpaceCenter.Orbit.apoapsis_altitude*.

periapsis

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

Attribute Read-only, cannot be set

Return type number

Note: For the periapsis altitude reported on the in-game map view, use *SpaceCenter.Orbit.periapsis_altitude*.

apoapsis_altitude

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

Attribute Read-only, cannot be set

Return type number

Note: This is equal to *SpaceCenter.Orbit.apoapsis* minus the equatorial radius of the body.

periapsis_altitude

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

Attribute Read-only, cannot be set

Return type number

Note: This is equal to *SpaceCenter.Orbit.periapsis* minus the equatorial radius of the body.

semi_major_axis

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

Attribute Read-only, cannot be set

Return type number

semi_minor_axis

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

Attribute Read-only, cannot be set

Return type number

radius

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

Attribute Read-only, cannot be set

Return type number

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

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

eccentric_anomaly

The *eccentric anomaly*.

Attribute Read-only, cannot be set

Return type number

static reference_plane_normal (*reference_frame*)

The unit direction vector 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*) –

Return type Tuple of (number, number, number)

static reference_plane_direction (*reference_frame*)

The unit direction vector from which the orbits longitude of ascending node is measured, in the given reference frame.

Parameters **reference_frame** (*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, 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*

6.3.6 Control

class Control

Used to manipulate the controls of a vessel. This includes adjusting the throttle, enabling/disabling systems such as SAS and RCS, or altering the direction in which the vessel is pointing.

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.

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

gear

The state of the landing gear/legs.

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

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

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*

get_action_group(group)

Returns `True` if the given action group is enabled.

Parameters **group** (*number*) – A number between 0 and 9 inclusive.

Return type boolean

set_action_group(group, state)

Sets the state of the given action group (a value between 0 and 9 inclusive).

Parameters

- **group** (*number*) – A number between 0 and 9 inclusive.
- **state** (*boolean*) –

toggle_action_group(group)

Toggles the state of the given action group.

Parameters **group** (*number*) – A number between 0 and 9 inclusive.

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 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

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.

6.3.7 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*
 - *Cargo Bay*
 - *Decoupler*
 - *Docking Port*
 - *Engine*
 - *Fairing*
 - *Intake*
 - *Landing Gear*
 - *Landing Leg*
 - *Launch Clamp*
 - *Light*
 - *Parachute*
 - *Radiator*
 - *Resource Converter*
 - *Resource Harvester*
 - *Reaction Wheel*
 - *Sensor*
 - *Solar Panel*
- *Trees of Parts*
 - *Traversing the Tree*
 - *Attachment Modes*
- *Fuel Lines*
- *Staging*

Parts

class **Parts**

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling `SpaceCenter.Vessel.parts`.

all

A list of all of the vessels parts.

Attribute Read-only, cannot be set

Return type List of `SpaceCenter.Part`

root

The vessels root part.

Attribute Read-only, cannot be set

Return type `SpaceCenter.Part`

Note: See the discussion on *Trees of Parts*.

controlling

The part from which the vessel is controlled.

Attribute Can be read or written

Return type `SpaceCenter.Part`

with_name (*name*)

A list of parts whose *SpaceCenter.Part.name* is *name*.

Parameters **name** (*string*) –

Return type List of *SpaceCenter.Part*

with_title (*title*)

A list of all parts whose *SpaceCenter.Part.title* is *title*.

Parameters **title** (*string*) –

Return type List of *SpaceCenter.Part*

with_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*

cargo_bays

A list of all cargo bays in the vessel.

Attribute Read-only, cannot be set

Return type List of *SpaceCenter.CargoBay*

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*

docking_port_with_name (*name*)

The first docking port in the vessel with the given port name, as returned by *SpaceCenter.DockingPort.name*. Returns nil if there are no such docking ports.

Parameters *name* (*string*) –

Return type *SpaceCenter.DockingPort*

engines

A list of all engines in the vessel.

Attribute Read-only, cannot be set

Return type List of *SpaceCenter.Engine*

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*

landing_gear

A list of all landing gear attached to the vessel.

Attribute Read-only, cannot be set

Return type List of *SpaceCenter.LandingGear*

landing_legs

A list of all landing legs attached to the vessel.

Attribute Read-only, cannot be set

Return type List of *SpaceCenter.LandingLeg*

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*

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*

Part

class Part

Instances of this class represents a part. A vessel is made of multiple parts. Instances can be obtained by various 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

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

impact_tolerance

The impact tolerance of the part, in meters per second.

Attribute Read-only, cannot be set

Return type number

temperature

Temperature of the part, in Kelvin.

Attribute Read-only, cannot be set

Return type number

skin_temperature

Temperature of the skin of the part, in Kelvin.

Attribute Read-only, cannot be set

Return type number

max_temperature

Maximum temperature that the part can survive, in Kelvin.

Attribute Read-only, cannot be set

Return type number

max_skin_temperature

Maximum temperature that the skin of the part can survive, in Kelvin.

Attribute Read-only, cannot be set

Return type number

thermal_mass

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

Attribute Read-only, cannot be set

Return type number

thermal_skin_mass

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

Attribute Read-only, cannot be set

Return type number

thermal_resource_mass

A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

Attribute Read-only, cannot be set

Return type number

thermal_conduction_flux

The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

Attribute Read-only, cannot be set

Return type number

thermal_convection_flux

The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

Attribute Read-only, cannot be set

Return type number

thermal_radiation_flux

The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

Attribute Read-only, cannot be set

Return type number

thermal_internal_flux

The rate at which heat energy is being generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

Attribute Read-only, cannot be set

Return type number

thermal_skin_to_internal_flux

The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

Attribute Read-only, cannot be set

Return type number

resources

A *SpaceCenter.Resources* object for the part.

Attribute Read-only, cannot be set

Return type *SpaceCenter.Resources*

crossfeed

Whether this part is crossfeed capable.

Attribute Read-only, cannot be set

Return type boolean

is_fuel_line

Whether this part is a fuel line.

Attribute Read-only, cannot be set

Return type boolean

fuel_lines_from

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

Attribute Read-only, cannot be set

Return type List of *SpaceCenter.Part*

Note: See the discussion on *Fuel Lines*.

fuel_lines_to

The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

Attribute Read-only, cannot be set

Return type List of *SpaceCenter.Part*

Note: See the discussion on *Fuel Lines*.

modules

The modules for this part.

Attribute Read-only, cannot be set

Return type List of *SpaceCenter.Module*

cargo_bay

A *SpaceCenter.CargoBay* if the part is a cargo bay, otherwise nil.

Attribute Read-only, cannot be set

Return type *SpaceCenter.CargoBay*

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*

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*

landing_gear

A *SpaceCenter.LandingGear* if the part is a landing gear , otherwise nil.

Attribute Read-only, cannot be set

Return type *SpaceCenter.LandingGear*

landing_leg

A *SpaceCenter.LandingLeg* if the part is a landing leg, otherwise nil.

Attribute Read-only, cannot be set

Return type *SpaceCenter.LandingLeg*

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*

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*

position (*reference_frame*)

The position of the part in the given reference frame.

Parameters **reference_frame** (*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number)

direction (*reference_frame*)

The direction of the part in the given reference frame.

Parameters **reference_frame** (*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number)

velocity (*reference_frame*)

The velocity of the part in the given reference frame.

Parameters **reference_frame** (*SpaceCenter.ReferenceFrame*) –

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`) –

Return type Tuple of (number, number, number, number)

reference_frame

The reference frame that is fixed relative to this part.

- The origin is at the position of the part.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

Attribute Read-only, cannot be set

Return type `SpaceCenter.ReferenceFrame`

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by `SpaceCenter.DockingPort.reference_frame`.

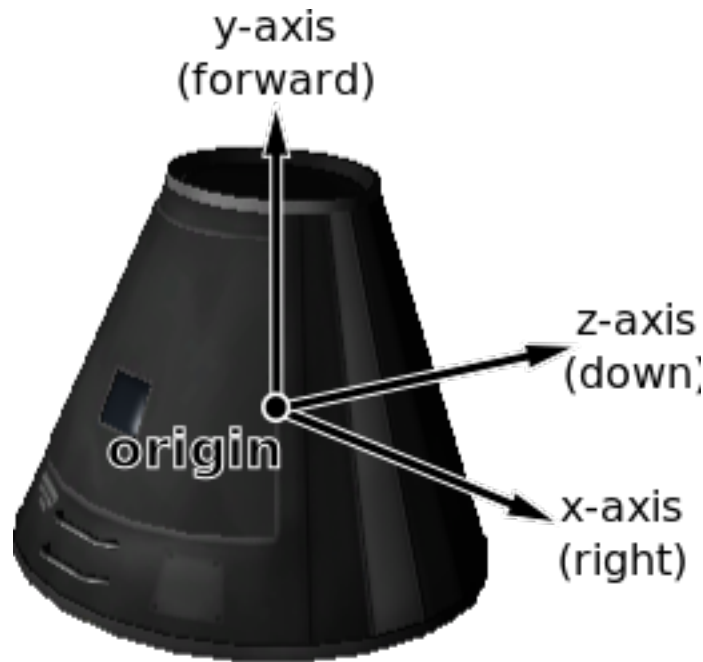


Fig. 6.7: Mk1 Command Pod reference frame origin and axes

Module

class Module

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” `PartModule` that contains all the functionality of an engine. This class allows you to interact with KSPs `PartModules`, and any `PartModules` that have been added by other mods.

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

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.

- *Cargo Bay*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Fairing*
- *Intake*
- *Landing Gear*
- *Landing Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *Sensor*
- *Solar Panel*

Cargo Bay

class **CargoBay**

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

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.

Decoupler**class 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. Has no effect if the decoupler has already fired.

decoupled

Whether the decoupler has fired.

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**

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*

name

The port name of the docking port. This is the name of the port that can be set in the right click menu, when the [Docking Port Alignment Indicator](#) mod is installed. If this mod is not installed, returns the title of the part (*SpaceCenter.Part.title*).

Attribute Can be read or written

Return type string

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 vessel that was undocked from. After undocking, the active vessel may change (*SpaceCenter.active_vessel*). This method can be called for either docking port in a docked pair - both calls will have the same effect. Returns `nil` if the docking port is not docked to anything.

Return type *SpaceCenter.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*) –

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*) –

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*) –

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*.

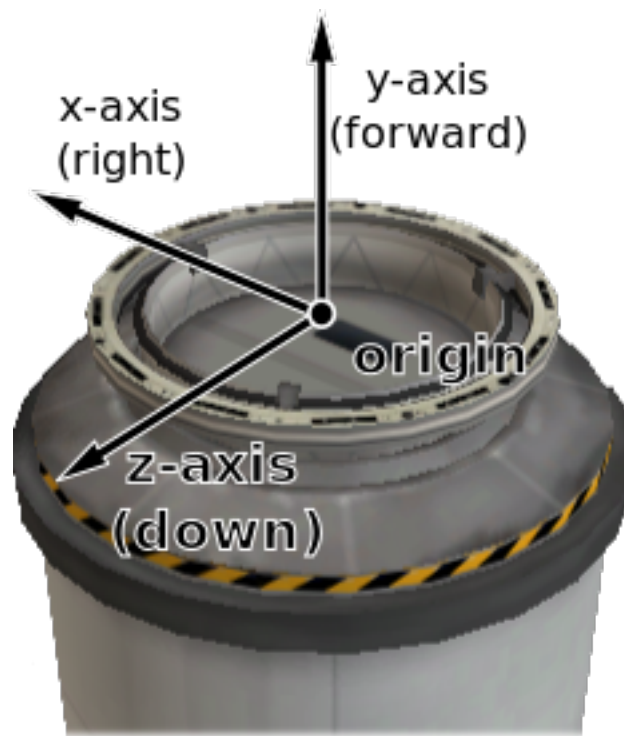


Fig. 6.8: Docking port reference frame origin and axes

class DockingPortState

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).

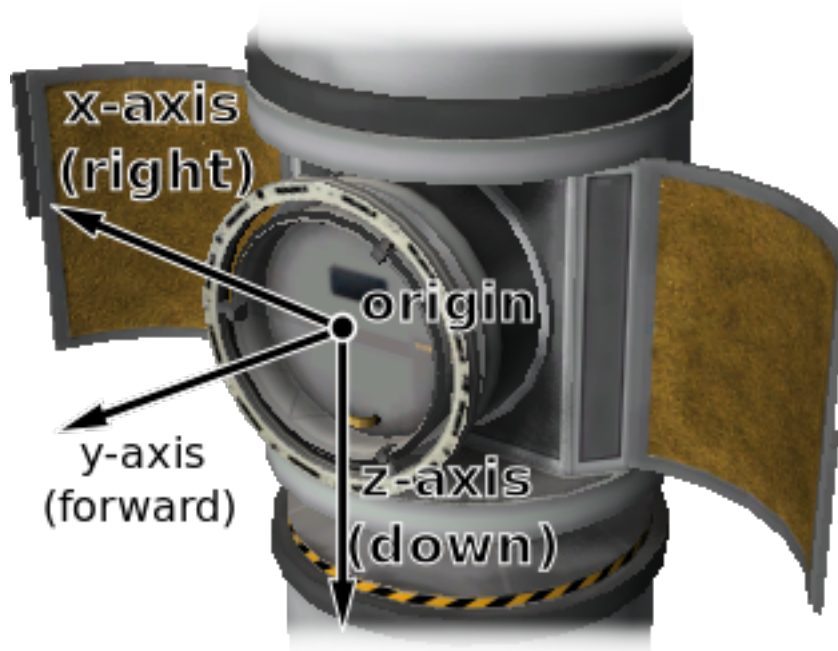


Fig. 6.9: Inline docking port reference frame origin and axes

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 (*SpaceCenter.DockingPort.reengage_distance*).

shielded

The docking port has a shield, and the shield is closed.

moving

The docking ports shield is currently opening/closing.

Engine

class Engine

Obtained by calling *SpaceCenter.Part.engine*.

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. Returns zero if the engine is not active or if it has no fuel.

Attribute Read-only, cannot be set

Return type number

available_thrust

The maximum available amount of thrust that can be produced by the engine, in Newtons. This takes *SpaceCenter.Engine.thrust_limit* into account, and is the amount of thrust produced by the engine when activated and the main throttle is set to 100%. Returns zero if the engine does not have any fuel.

Attribute Read-only, cannot be set

Return type number

max_thrust

Gets the maximum amount of thrust that can be produced by the engine, in Newtons. This is the amount of thrust produced by the engine when activated, *SpaceCenter.Engine.thrust_limit* is set to 100% and the main vessel's throttle is 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

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

propellants

The names of resources that the engine consumes.

Attribute Read-only, cannot be set

Return type List of string

propellant_ratios

The ratios of resources that the engine consumes. A dictionary mapping resource names to the ratios at which they are consumed by the engine.

Attribute Read-only, cannot be set

Return type Map from string to number

has_fuel

Whether the engine has run out of fuel (or flamed out).

Attribute Read-only, cannot be set

Return type boolean

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

Gets 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 nozzle is gimbaled, i.e. can provide a turning force.

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 gimbaled.

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 gimbaled.

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

Fairing**class 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

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

Landing Gear

class LandingGear

Obtained by calling *SpaceCenter.Part.landing_gear*.

part

The part object for this landing gear.

Attribute Read-only, cannot be set

Return type *SpaceCenter.Part*

state

Gets the current state of the landing gear.

Attribute Read-only, cannot be set

Return type *SpaceCenter.LandingGearState*

Note: Fixed landing gear are always deployed.

deployable

Whether the landing gear is deployable.

Attribute Read-only, cannot be set

Return type boolean

deployed

Whether the landing gear is deployed.

Attribute Can be read or written

Return type boolean

Note: Fixed landing gear are always deployed. Returns an error if you try to deploy fixed landing gear.

class LandingGearState

See *SpaceCenter.LandingGear.state*.

deployed

Landing gear is fully deployed.

retracted

Landing gear is fully retracted.

deploying

Landing gear is being deployed.

retracting

Landing gear is being retracted.

Landing Leg**class LandingLeg**

Obtained by calling *SpaceCenter.Part.landing_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.LandingLegState*

deployed

Whether the landing leg is deployed.

Attribute Can be read or written

Return type boolean

class LandingLegState

See *SpaceCenter.LandingLeg.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.

repairing

Landing leg is being repaired.

Launch Clamp**class LaunchClamp**

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**

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

power_usage

The current power usage, in units of charge per second.

Attribute Read-only, cannot be set

Return type number

Parachute

class **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

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.

Attribute Can be read or written

Return type number

deploy_min_pressure

The minimum pressure at which the parachute will semi-deploy, in atmospheres.

Attribute Can be read or written

Return type number

class **ParachuteState**

See *SpaceCenter.Parachute.state*.

stowed

The parachute is safely tucked away inside its housing.

active

The parachute is still stowed, but ready to semi-deploy.

semi_deployed

The parachute has been deployed and is providing some drag, but is not fully deployed yet.

deployed

The parachute is fully deployed.

cut

The parachute has been cut.

Radiator

class 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

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

Obtained by calling *SpaceCenter.Part.resource_converter*.

part

The part object for this converter.

Attribute Read-only, cannot be set

Return type *SpaceCenter.Part*

count

The number of converters in the part.

Attribute Read-only, cannot be set

Return type number

name (*index*)

The name of the specified converter.

Parameters **index** (*number*) – Index of the converter.

Return type string

active (*index*)

True if the specified converter is active.

Parameters **index** (*number*) – Index of the converter.

Return type boolean

start (*index*)

Start the specified converter.

Parameters **index** (*number*) – Index of the converter.

stop (*index*)

Stop the specified converter.

Parameters **index** (*number*) – Index of the converter.

state (*index*)

The state of the specified converter.

Parameters **index** (*number*) – Index of the converter.

Return type *SpaceCenter.ResourceConverterState*

status_info (*index*)

Status information for the specified converter. This is the full status message shown in the in-game UI.

Parameters **index** (*number*) – Index of the converter.

Return type string

inputs (*index*)

List of the names of resources consumed by the specified converter.

Parameters **index** (*number*) – Index of the converter.

Return type List of string

outputs (*index*)

List of the names of resources produced by the specified converter.

Parameters **index** (*number*) – Index of the converter.

Return type List of string

class ResourceConverterState

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

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

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**

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

pitch_torque

The torque in the pitch axis, in Newton meters.

Attribute Read-only, cannot be set

Return type number

yaw_torque

The torque in the yaw axis, in Newton meters.

Attribute Read-only, cannot be set

Return type number

roll_torque

The torque in the roll axis, in Newton meters.

Attribute Read-only, cannot be set

Return type number

Sensor**class Sensor**

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

power_usage

The current power usage of the sensor, in units of charge per second.

Attribute Read-only, cannot be set

Return type number

Solar Panel**class SolarPanel**

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*

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

See *SpaceCenter.SolarPanel.state*.

extended

Solar panel is fully extended.

retracted

Solar panel is fully retracted.

extending

Solar panel is being extended.

retracting

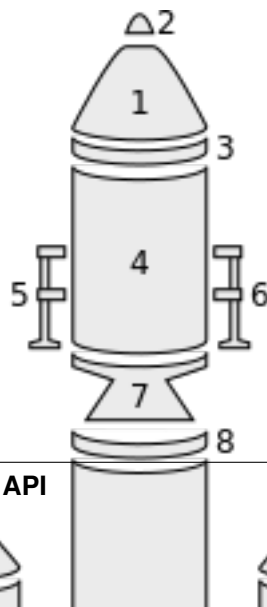
Solar panel is being retracted.

broken

Solar panel is broken.

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 *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 root = vessel.parts.root
local stack = {{root,0}}
while #stack > 0 do
    local part,depth = unpack(table.remove(stack))
    print(string.rep(' ', depth) .. part.title)
    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
TR-18A Stack Decoupler
FL-T400 Fuel Tank
LV-909 Liquid Fuel Engine
TR-18A Stack Decoupler
FL-T800 Fuel Tank
LV-909 Liquid Fuel Engine
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launch Stability Enhancer
FTX-2 External Fuel Duct
LV-909 Liquid Fuel Engine
Aerodynamic Nose Cone
TT-70 Radial Decoupler
FL-T400 Fuel Tank
TT18-A Launch Stability Enhancer
FTX-2 External Fuel Duct
LV-909 Liquid Fuel Engine
Aerodynamic Nose Cone
LT-1 Landing Struts
LT-1 Landing Struts
Mk16 Parachute
```

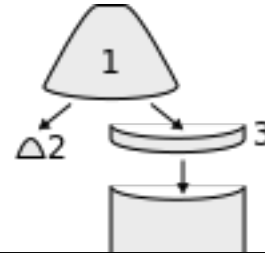
Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

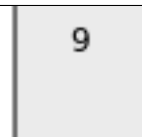
The root part of a vessel (for example the command pod – part 1) does not have a parent part, so does not have an attachment mode. However, the part is considered to be *axially* attached to nothing.

The following Lua example does a depth-first traversal as before, but also prints out the attachment mode used by the part:



```
local root = vessel.parts.root
local stack = {{root, 0}}
while #stack > 0 do
    local part, depth = unpack(table.remove(stack))
    local attach_mode
    if part.axially_attached then
        attach_mode = 'axial'
    else -- radially_attached
        attach_mode = 'radial'
    end
    print(string.rep(' ', depth) .. part.title .. ' - ' .. attach_mode)
    for _, child in ipairs(part.children) do
        table.insert(stack, {child, depth+1})
    end
end
end
```

When this code is executed using the craft file for the example vessel pictured above, the following is printed out:



```
Command Pod Mk1 - axial
TR-18A Stack Decoupler - axial
FL-T400 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TR-18A Stack Decoupler - axial
FL-T800 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
LT-1 Landing Struts - radial
LT-1 Landing Struts - radial
Mk16 Parachute - axial
```

Fuel Lines

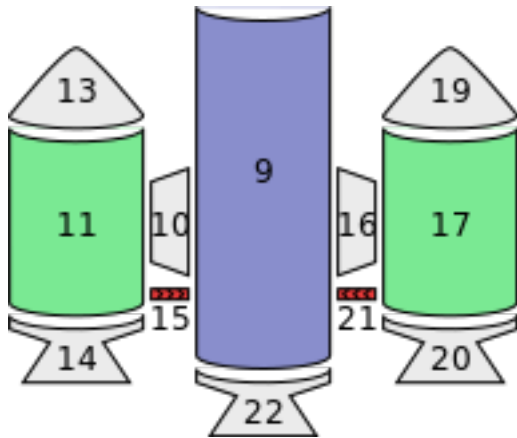


Fig. 6.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.

The attributes `SpaceCenter.Part.fuel_lines_from` and `SpaceCenter.Part.fuel_lines_to` can be used to discover these connections. In the example in Figure 5, when `SpaceCenter.Part.fuel_lines_to` is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When `SpaceCenter.Part.fuel_lines_from` is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).

Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using `SpaceCenter.Part.stage` and `SpaceCenter.Part.decouple_stage` respectively. For parts that are not activated by staging, `SpaceCenter.Part.stage` returns -1. For parts that are never decoupled, `SpaceCenter.Part.decouple_stage` returns a value of -1.

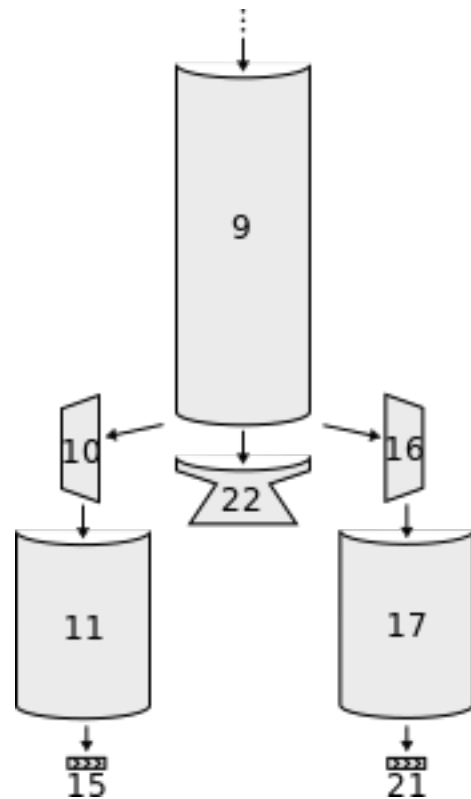


Fig. 6.13: **Figure 4** – A subset of the parts tree from Figure 2 above.

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *activated*. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

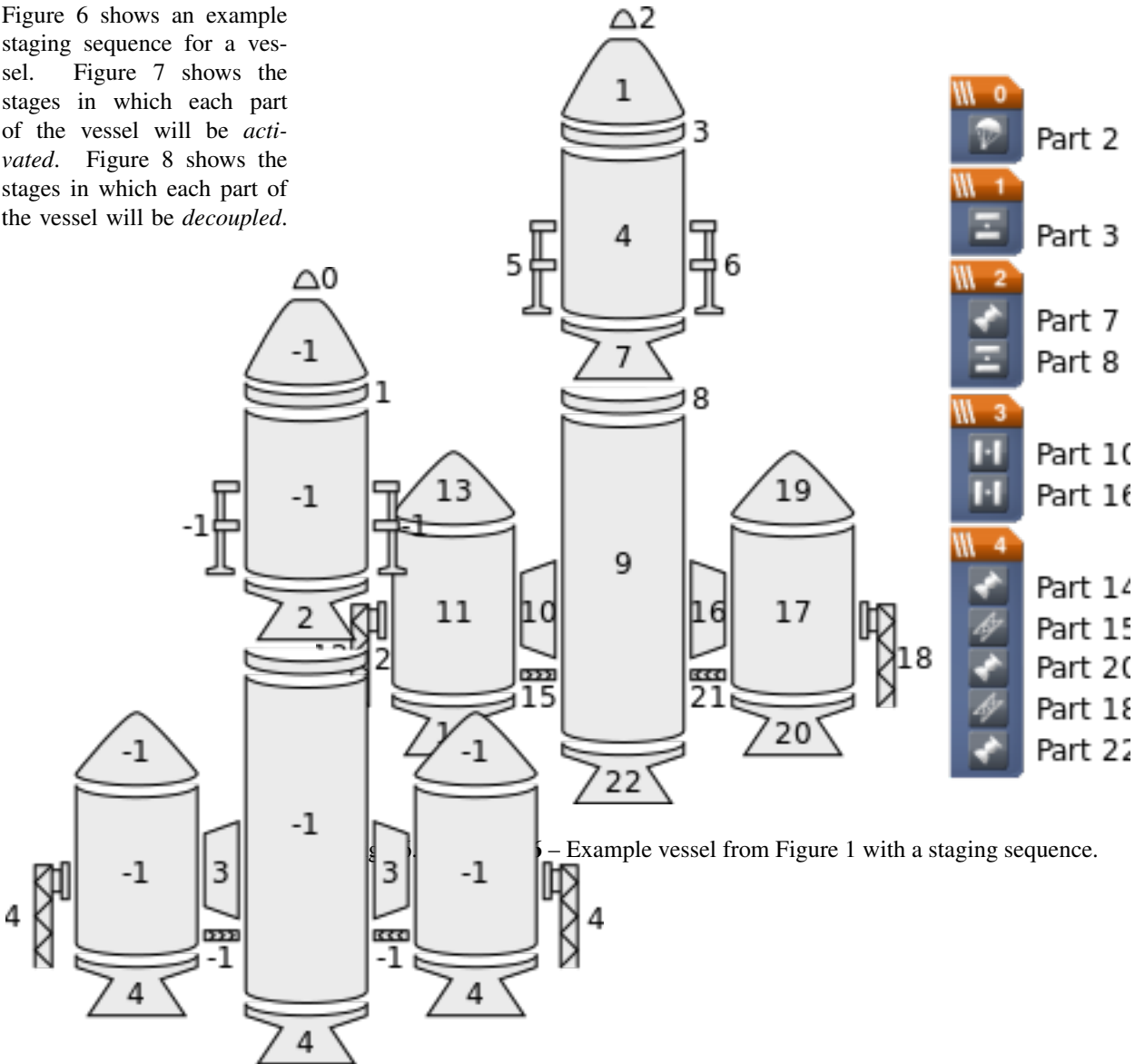
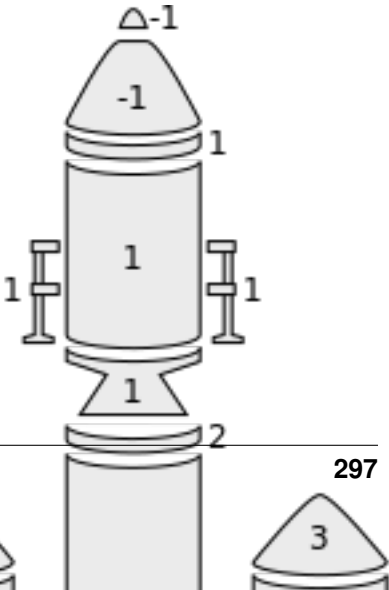


Fig. 6.15: **Figure 7** – The stage in which each part is *activated*.

6.3.8 Resources

class Resources
Created by calling
`SpaceCenter.Vessel.resources,`
`SpaceCenter.Vessel.resources_in_decouple_stage()`
or `SpaceCenter.Part.resources.`
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

max (*name*)

Returns the amount of a resource that can be stored.

Parameters **name** (*string*) – The name of the resource.

Return type number

amount (*name*)

Returns the amount of a resource that is currently 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*

class ResourceFlowMode

See *SpaceCenter.Resources.flow_mode()*.

vessel

The resource flows to any part in the vessel. For example, electric charge.

stage

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

adjacent

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

none

The resource does not flow. For example, solid fuel.

6.3.9 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 a vector whose direction the direction of the maneuver node burn, and whose magnitude is the delta-v of the burn in m/s.

Parameters `reference_frame`

(`SpaceCenter.ReferenceFrame`) –

Return type Tuple of (number, number, number)

Note:	Does	not
change	when	exe-
cuting	the	maneu-
ver	node.	See

SpaceCenter.Node.remaining_burn_vector().

remaining_burn_vector (*[reference_frame = None]*)

Returns a vector whose direction the direction of the maneuver node burn, and whose magnitude is the delta-v of the burn in m/s. The direction and magnitude change as the burn is executed.

Parameters **reference_frame**

(*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number)

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

Gets 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

Gets 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*)

Returns the position vector of the maneuver node in the given reference frame.

Parameters **reference_frame**

(*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number)

direction (*reference_frame*)

Returns the unit direction vector of the maneuver nodes burn in the given reference frame.

Parameters **reference_frame**

(*SpaceCenter.ReferenceFrame*) –

Return type Tuple of (number, number, number)

6.3.10 Comms

class Comms

Used to interact with RemoteTech. Created using a call to *SpaceCenter.Vessel.comms*.

Note: This class requires *RemoteTech* to be installed.

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 RemoteTech flight computer on board.

Attribute Read-only, cannot be set

Return type boolean

has_connection

Whether the vessel can receive commands from the KSC or a command station.

Attribute Read-only, cannot be set

Return type boolean

has_connection_to_ground_station

Whether the vessel can transmit science data to a ground station.

Attribute Read-only, cannot be set

Return type boolean

signal_delay

The signal delay when sending commands 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*)

Returns the signal delay between the current vessel and another vessel, in seconds.

Parameters *other* (`SpaceCenter.Vessel`) –

Return type number

6.3.11 ReferenceFrame

class ReferenceFrame

Represents a reference frame for positions, rotations and velocities. Contains:

- The position of the origin.
- The directions of the x, y and z axes.
- The linear velocity of the frame.
- The angular velocity of the frame.

Note: This class does not contain any properties or methods. It is only used as a parameter to other functions.

6.3.12 AutoPilot

class **AutoPilot**

Provides basic auto-piloting utilities for a vessel. Created by calling `SpaceCenter.Vessel.auto_pilot`.

engage()

Engage the auto-pilot.

disengage()

Disengage the auto-pilot.

wait()

Blocks until the vessel is pointing in the target direction (if set) and has the target roll (if set).

error

The error, in degrees, between the direction the ship has been asked to point in and the direction it is pointing in. Returns zero if the auto-pilot has not been engaged, SAS is not enabled, SAS is in stability assist mode, or no target direction is set.

Attribute Read-only, cannot be set

Return type number

roll_error

The error, in degrees, between the roll the ship has been asked to be in and the actual roll. Returns zero 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`

target_direction

The target direction. `nil` if no target direction is set.

Attribute Can be read or written

Return type Tuple of (number, number, number)

target_pitch_and_heading (*pitch, heading*)
Set (*SpaceCenter.AutoPilot.target_direction*)
from a pitch and heading angle.

Parameters

- **pitch** (*number*) – Target pitch angle, in degrees between -90° and +90°.
- **heading** (*number*) – Target heading angle, in degrees between 0° and 360°.

target_roll
The target roll, in degrees. NaN if no target roll is set.

Attribute Can be read or written

Return type number

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

rotation_speed_multiplier
Target rotation speed multiplier. Defaults to 1.

Attribute Can be read or written

Return type number

max_rotation_speed
Maximum target rotation speed. Defaults to 1.

Attribute Can be read or written

Return type number

roll_speed_multiplier
Target roll speed multiplier. Defaults to 1.

Attribute Can be read or written

Return type number

max_roll_speed

Maximum target roll speed. Defaults to 1.

Attribute Can be read or written

Return type number

set_pid_parameters ($[kp = 1.0][, ki = 0.0][, kd = 0.0]$)

Sets the gains for the rotation rate PID controller.

Parameters

- **kp** (*number*) – Proportional gain.
- **ki** (*number*) – Integral gain.
- **kd** (*number*) – Derivative gain.

6.3.13 Geometry Types

class Vector3

3-dimensional vectors are represented as a 3-tuple.

For example:

```
local krpc = require 'krpc.init'
local conn = krpc.connect()
local v = conn.space_center.active_vessel:flight().prograde
print(v[1], v[2], v[3])
```

class Quaternion

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.init'
local conn = krpc.connect()
local q = conn.space_center.active_vessel:flight().rotation
print(q[1], q[2], q[3], q[4])
```

6.4 InfernalRobotics API

Provides RPCs to interact with the [InfernalRobotics](#) mod. Provides the following classes:

6.4.1 InfernalRobotics

This service provides functionality to interact with the [InfernalRobotics](#) mod.

servo_groups

A list of all the servo groups in the active vessel.

Attribute Read-only, cannot be set

Return type List of *InfernalRobotics.ControlGroup*

static servo_group_with_name (*name*)

Returns the servo group with the given *name* or *nil* if none exists. If multiple servo groups have the same name, only one of them is returned.

Parameters **name** (*string*) – Name of servo group to find.

Return type *InfernalRobotics.ControlGroup*

static servo_with_name (*name*)

Returns the servo with the given *name*, from all servo groups, or *nil* if none exists. If multiple servos have the same name, only one of them is returned.

Parameters **name** (*string*) – Name of the servo to find.

Return type *InfernalRobotics.Servo*

6.4.2 ControlGroup

class ControlGroup

A group of servos, obtained by calling *InfernalRobotics.servo_groups* or *InfernalRobotics.servo_group_with_name()*. Represents the “Servo Groups” in the Infernal-Robotics 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 Infernal-Robotics 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*

move_right ()

Moves all of the servos in the group to the right.

move_left ()

Moves all of the servos in the group to the left.

move_center ()

Moves all of the servos in the group to the center.

move_next_preset ()

Moves all of the servos in the group to the next preset.

move_prev_preset ()

Moves all of the servos in the group to the previous preset.

stop ()

Stops the servos in the group.

6.4.3 Servo

class Servo

Represents a servo.
 Obtained using *InfernalRobotics.ControlGroup.servos*,
InfernalRobotics.ControlGroup.servo_with_name()
 or *InfernalRobotics.servo_with_name()*.

name

The name of the servo.

Attribute Can be read or written

Return type string

highlight

Whether the servo should be highlighted in-game.

Attribute Write-only, cannot be read

Return type boolean

position

The position of the servo.

Attribute Read-only, cannot be set

Return type number

min_config_position

The minimum position of the servo, specified by the part configuration.

Attribute Read-only, cannot be set

Return type number

max_config_position

The maximum position of the servo, specified by the part configuration.

Attribute Read-only, cannot be set

Return type number

min_position

The minimum position of the servo, specified by the in-game tweak menu.

Attribute Can be read or written

Return type number

max_position

The maximum position of the servo, specified by the in-game tweak menu.

Attribute Can be read or written

Return type number

config_speed

The speed multiplier of the servo, specified by the part configuration.

Attribute Read-only, cannot be set

Return type number

speed

The speed multiplier of the servo, specified by the in-game tweak menu.

Attribute Can be read or written

Return type number

current_speed

The current speed at which the servo is moving.

Attribute Can be read or written

Return type number

acceleration

The current speed multiplier set in the UI.

Attribute Can be read or written

Return type number

is_moving

Whether the servo is moving.

Attribute Read-only, cannot be set

Return type boolean

is_free_moving

Whether the servo is freely moving.

Attribute Read-only, cannot be set

Return type boolean

is_locked

Whether the servo is locked.

Attribute Can be read or written

Return type boolean

is_axis_inverted

Whether the servos axis is inverted.

Attribute Can be read or written

Return type boolean

move_right ()

Moves the servo to the right.

move_left ()

Moves the servo to the left.

move_center ()

Moves the servo to the center.

move_next_preset ()

Moves the servo to the next preset.

move_prev_preset ()

Moves the servo to the previous preset.

move_to (position, speed)

Moves the servo to *position* and sets the speed multiplier to *speed*.

Parameters

- **position** (*number*) – The position to move the servo to.
- **speed** (*number*) – Speed multiplier for the movement.

stop ()

Stops the servo.

6.4.4 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.init'
local platform = require 'krpc.platform'
local Types = require 'krpc.types'

local conn = krpc.connect(nil, nil, nil, 'InfernalRobotics Example')

local group = conn.infernal_robotics.servo_group_with_name('MyGroup')
if group == Types.none then
    print('Group not found')
    os.exit(1)
end

for _, servo in ipairs(group.servos) do
    print(servo.name, servo.position)
end

group:move_right()
platform.sleep(1)
group:stop()
```

6.5 Kerbal Alarm Clock API

Provides RPCs to interact with the [Kerbal Alarm Clock](#) mod. Provides the following classes:

6.5.1 KerbalAlarmClock

This service provides functionality to interact with the [Kerbal Alarm Clock](#) mod.

alarms

A list of all the alarms.

Attribute Read-only, cannot be set

Return type List of *KerbalAlarmClock.Alarm*

static alarm_with_name (*name*)

Get the alarm with the given *name*, or *nil* if no alarms have that name. If more than one alarm has the name, only returns one of them.

Parameters **name** (*string*) – Name of the alarm to search for.

Return type *KerbalAlarmClock.Alarm*

static alarms_with_type (*type*)

Get a list of alarms of the specified *type*.

Parameters `type` (`KerbalAlarmClock.AlarmType`)
 – Type of alarm to return.

Return type List of `KerbalAlarmClock.Alarm`

static `create_alarm` (`type`, `name`, `ut`)
 Create a new alarm and return it.

Parameters

- **type** (`KerbalAlarmClock.AlarmType`) – Type of the new alarm.
- **name** (`string`) – Name of the new alarm.
- **ut** (`number`) – Time at which the new alarm should trigger.

Return type `KerbalAlarmClock.Alarm`

6.5.2 Alarm

class `Alarm`

Represents an alarm.
 Obtained by calling
`KerbalAlarmClock.alarms`,
`KerbalAlarmClock.alarm_with_name()`
 or `KerbalAlarmClock.alarms_with_type()`.

action

The action that the alarm triggers.

Attribute Can be read or written

Return type `KerbalAlarmClock.AlarmAction`

margin

The number of seconds before the event that the alarm will fire.

Attribute Can be read or written

Return type number

time

The time at which the alarm will fire.

Attribute Can be read or written

Return type number

type

The type of the alarm.

Attribute Read-only, cannot be set

Return type `KerbalAlarmClock.AlarmType`

id

The unique identifier for the alarm.

Attribute Read-only, cannot be set

Return type string

name

The short name of the alarm.

Attribute Can be read or written

Return type string

notes

The long description of the alarm.

Attribute Can be read or written

Return type string

remaining

The number of seconds until the alarm will fire.

Attribute Read-only, cannot be set

Return type number

repeat

Whether the alarm will be repeated after it has fired.

Attribute Can be read or written

Return type boolean

repeat_period

The time delay to automatically create an alarm after it has fired.

Attribute Can be read or written

Return type number

vessel

The vessel that the alarm is attached to.

Attribute Can be read or written

Return type *SpaceCenter.Vessel*

xfer_origin_body

The celestial body the vessel is departing from.

Attribute Can be read or written

Return type *SpaceCenter.CelestialBody*

xfer_target_body

The celestial body the vessel is arriving at.

Attribute Can be read or written

Return type *SpaceCenter.CelestialBody*

remove ()

Removes the alarm.

6.5.3 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`.

6.5.4 AlarmAction

class AlarmAction

The action performed by an alarm when it fires.

do_nothing

Don't do anything at all...

do_nothing_delete_when_passed

Don't do anything, and delete the alarm.

kill_warp

Drop out of time warp.

kill_warp_only

Drop out of time warp.

message_only

Display a message.

pause_game

Pause the game.

6.5.5 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.init'
local conn = krpc.connect(nil, nil, nil, '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
```

7.1 Python Client

This client provides functionality to interact with a kRPC server from programs written in Python. It can be [installed](#) using PyPI or [downloaded from GitHub](#).

7.1.1 Installing the Library

The python client and all of its dependencies can be installed using pip with a single command. It supports Python 2.7+ and 3.x

On linux:

```
pip install krpc
```

On Windows:

```
C:\Python27\Scripts\pip.exe install krpc
```

7.1.2 Using the Library

Once it's installed, simply `import krpc` and you are good to go! You can check what version you have installed by running the following script:

```
import krpc
print(krpc.__version__)
```

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:

```
import krpc
conn = krpc.connect(name='Example')
print(conn.krpc.get_status().version)
```

This function also accepts arguments that specify what address and port numbers to connect to. For example:

```
import krpc
conn = krpc.connect(name='Remote example', address='my.domain.name', rpc_port=1000, stream_port=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.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`. To explore the functionality provided by a service, you can use the `help()` function from an interactive terminal. For example, running `help(conn.space_center)` will list all of the classes, enumerations, procedures and properties provided by the SpaceCenter service. Or for a class, such as the vessel class provided by the SpaceCenter service by calling `help(conn.space_center.Vessel)`.

Calling methods, getting or setting properties, etc. are mapped to remote procedure calls and passed to the server by the python client.

7.1.5 Streaming Data from the Server

A stream repeatedly executes a function on the server, with a fixed set of argument values. It provides a more efficient way of repeatedly getting the result of a function, avoiding the network overhead of having to invoke it directly.

For example, consider the following loop that continuously prints out the position of the active vessel. This loop incurs significant communication overheads, as the `vessel.position` function is called repeatedly.

```
vessel = conn.space_center.active_vessel
refframe = vessel.orbit.body.reference_frame
while True:
    print vessel.position(refframe)
```

The following code achieves the same thing, but is far more efficient. It calls `krpc.client.Client.add_stream()` once at the start of the program to create a stream, and then repeatedly gets the position from the stream.

```
vessel = conn.space_center.active_vessel
refframe = vessel.orbit.body.reference_frame
position = conn.add_stream(vessel.position, refframe)
while True:
    print position()
```

A stream can be created by calling `krpc.client.Client.add_stream()` or using the `with` statement applied to `krpc.client.Client.stream()`. Both of these approaches return an instance of the `krpc.stream.Stream` class.

Both methods and attributes can be streamed. The example given above demonstrates how to stream methods. The following example shows how to stream an attribute (in this case `vessel.control.abort`):

```
abort = conn.add_stream(getattr, vessel.control, 'abort')
while not abort():
    ...
```

7.1.6 Client API Reference

connect (`[address='127.0.0.1']`, `[rpc_port=50000]`, `[stream_port=50001]`, `[name=None]`)

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

- **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.
- **stream_port** (*int*) – The port number of the Stream Server. Defaults to 50001.
- **name** (*str*) – 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.

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()`.

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*.

For example, to stream the result of method call `vessel.position(refframe)`:

```
vessel = conn.space_center.active_vessel
refframe = vessel.orbit.body.reference_frame
with conn.stream(vessel.position, refframe) as pos:
    print('Position =', pos())
```

Or to stream the property `conn.space_center.ut`:

```
with conn.stream(getattr(conn.space_center, 'ut')) as ut:
    print('Universal Time =', ut())
```

close()

Closes the connection to the server.

krpc

The built-in KRPC class, providing basic interactions with the server.

Return type `krpc.client.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.Client.krpc`. Most 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. 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:

```
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:

```
status = conn.krpc.get_status()
print('Data in =', (status.bytes_read_rate/1024.0), 'KB/s')
print('Data out =', (status.bytes_written_rate/1024.0), 'KB/s')
```

class Stream

__call__()
Gets the most recently received value for the stream.

remove()
Remove the stream from the server.

7.2 KRPC API

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

static get_status()
Returns some information about the server, such as the version.

Return type `krpc.schema.KRPC.Status`

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

Return type `krpc.schema.KRPC.Services`

current_game_scene
Get the current game scene.

Attribute Read-only, cannot be set

Return type `GameScene`

static add_stream(request)
Add a streaming request and return its identifier.

Parameters **request** (`krpc.schema.KRPC.Request`) –

Return type `int`

Note: Do not call this method from client code. Use *streams* provided by the Python client library.

static remove_stream(id)
Remove a streaming request.

Parameters **id** (`int`) –

Note: Do not call this method from client code. Use *streams* provided by the Python client library.

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.

7.3 SpaceCenter API

7.3.1 SpaceCenter

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

active_vessel

The currently active vessel.

Attribute Can be read or written

Return type *Vessel*

vessels

A list of all the vessels in the game.

Attribute Read-only, cannot be set

Return type list of *Vessel*

bodies

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

Attribute Read-only, cannot be set

Return type dict from str to *CelestialBody*

target_body

The currently targeted celestial body.

Attribute Can be read or written

Return type *CelestialBody*

target_vessel

The currently targeted vessel.

Attribute Can be read or written

Return type *Vessel*

target_docking_port

The currently targeted docking port.

Attribute Can be read or written

Return type *DockingPort*

static clear_target ()

Clears the current target.

static launch_vessel_from_vab (*name*)

Launch a new vessel from the VAB onto the launchpad.

Parameters **name** (*str*) – Name of the vessel’s craft file.

static launch_vessel_from_sph (*name*)

Launch a new vessel from the SPH onto the runway.

Parameters **name** (*str*) – Name of the vessel’s craft file.

ut

The current universal time in seconds.

Attribute Read-only, cannot be set

Return type float

g

The value of the [gravitational constant](#) G in $N(m/kg)^2$.

Attribute Read-only, cannot be set

Return type float

warp_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](#)

warp_rate

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

Attribute Read-only, cannot be set

Return type float

warp_factor

The current warp factor. This is the index of the rate at which time is passing for either regular “on-rails” or physical time warp. Returns 0 if time warp is not active. When in on-rails time warp, this is equal to [rails_warp_factor](#), and in physics time warp, this is equal to [physics_warp_factor](#).

Attribute Read-only, cannot be set

Return type float

rails_warp_factor

The time warp rate, using regular “on-rails” time warp. A value between 0 and 7 inclusive. 0 means no time warp. Returns 0 if physical time warp is active. If requested time warp factor cannot be set, it will be set to the next lowest possible value. For example, if the vessel is too close to a planet. See [the KSP wiki](#) for details.

Attribute Can be read or written

Return type int

physics_warp_factor

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

Attribute Can be read or written

Return type int

static `can_rails_warp_at` (*[factor = 1]*)

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

Parameters `factor` (*int*) – The warp factor to check.

Return type `bool`

maximum_rails_warp_factor

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

Attribute Read-only, cannot be set

Return type `int`

static `warp_to` (*ut*, *[max_rails_rate = 100000.0]*, *[max_physics_rate = 2.0]*)

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

Parameters

- `ut` (*float*) – The universal time to warp to, in seconds.
- `max_rails_rate` (*float*) – The maximum warp rate in regular “on-rails” time warp.
- `max_physics_rate` (*float*) – The maximum warp rate in physical time warp.

Returns When the time warp is complete.

static `transform_position` (*position*, *from*, *to*)

Converts a position vector from one reference frame to another.

Parameters

- `position` (*tuple*) – Position vector in reference frame *from*.
- `from` (`ReferenceFrame`) – The reference frame that the position vector is in.
- `to` (`ReferenceFrame`) – The reference frame to convert the position vector to.

Returns The corresponding position vector in reference frame *to*.

Return type tuple of (float, float, float)

static `transform_direction` (*direction*, *from*, *to*)

Converts a direction vector from one reference frame to another.

Parameters

- `direction` (*tuple*) – Direction vector in reference frame *from*.
- `from` (`ReferenceFrame`) – The reference frame that the direction vector is in.
- `to` (`ReferenceFrame`) – The reference frame to convert the direction vector to.

Returns The corresponding direction vector in reference frame *to*.

Return type tuple of (float, float, float)

static `transform_rotation` (*rotation*, *from*, *to*)

Converts a rotation from one reference frame to another.

Parameters

- **rotation** (*tuple*) – Rotation in reference frame *from*.
- **from** (*ReferenceFrame*) – The reference frame that the rotation is in.
- **to** (*ReferenceFrame*) – The corresponding rotation in reference frame *to*.

Returns The corresponding rotation in reference frame *to*.

Return type tuple of (float, float, float, float)

static transform_velocity (*position, velocity, from, to*)

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

Parameters

- **position** (*tuple*) – Position vector in reference frame *from*.
- **velocity** (*tuple*) – Velocity vector in reference frame *from*.
- **from** (*ReferenceFrame*) – The reference frame that the position and velocity vectors are in.
- **to** (*ReferenceFrame*) – The reference frame to covert the velocity vector to.

Returns The corresponding velocity in reference frame *to*.

Return type tuple of (float, float, float)

far_available

Whether [Ferram Aerospace Research](#) is installed.

Attribute Read-only, cannot be set

Return type bool

remote_tech_available

Whether [RemoteTech](#) is installed.

Attribute Read-only, cannot be set

Return type bool

static draw_direction (*direction, reference_frame, color*[, *length = 10.0*])

Draw a direction vector on the active vessel.

Parameters

- **direction** (*tuple*) – Direction to draw the line in.
- **reference_frame** (*ReferenceFrame*) – Reference frame that the direction is in.
- **color** (*tuple*) – The color to use for the line, as an RGB color.
- **length** (*float*) – The length of the line. Defaults to 10.

static draw_line (*start, end, reference_frame, color*)

Draw a line.

Parameters

- **start** (*tuple*) – Position of the start of the line.
- **end** (*tuple*) – Position of the end of the line.
- **reference_frame** (*ReferenceFrame*) – Reference frame that the position are in.

- **color** (*tuple*) – The color to use for the line, as an RGB color.

static clear_drawing()

Remove all directions and lines currently being drawn.

class WarpMode

Returned by *WarpMode*

rails

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

physics

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

none

Time warp is not active.

7.3.2 Vessel

class Vessel

These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources.

name

The name of the vessel.

Attribute Can be read or written

Return type str

type

The type of the vessel.

Attribute Can be read or written

Return type *VesselType*

situation

The situation the vessel is in.

Attribute Read-only, cannot be set

Return type *VesselSituation*

met

The mission elapsed time in seconds.

Attribute Read-only, cannot be set

Return type float

flight ([*reference_frame = None*])

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

Parameters **reference_frame** (*ReferenceFrame*) – Reference frame. Defaults to the vessel’s surface reference frame (*Vessel.surface_reference_frame*).

Return type *Flight*

Note: When this is called with no arguments, the vessel's surface reference frame is used. This reference frame moves with the vessel, therefore velocities and speeds returned by the flight object will be zero. See the [reference frames tutorial](#) for examples of getting the *orbital speed* and *surface speed* of a vessel.

target

The target vessel. *None* if there is no target. When setting the target, the target cannot be the current vessel.

Attribute Can be read or written

Return type *Vessel*

orbit

The current orbit of the vessel.

Attribute Read-only, cannot be set

Return type *Orbit*

control

Returns a *Control* object that can be used to manipulate the vessel's control inputs. For example, its pitch/yaw/roll controls, RCS and thrust.

Attribute Read-only, cannot be set

Return type *Control*

auto_pilot

An *AutoPilot* object, that can be used to perform simple auto-piloting of the vessel.

Attribute Read-only, cannot be set

Return type *AutoPilot*

resources

A *Resources* object, that can be used to get information about resources stored in the vessel.

Attribute Read-only, cannot be set

Return type *Resources*

resources_in_decouple_stage (*stage*[, *cumulative* = *True*])

Returns a *Resources* object, that can be used to get information about resources stored in a given *stage*.

Parameters

- **stage** (*int*) – Get resources for parts that are decoupled in this stage.
- **cumulative** (*bool*) – When *False*, returns the resources for parts decoupled in just the given stage. When *True* returns the resources decoupled in the given stage and all subsequent stages combined.

Return type *Resources*

Note: For details on stage numbering, see the discussion on [Staging](#).

parts

A *Parts* object, that can be used to interact with the parts that make up this vessel.

Attribute Read-only, cannot be set

Return type *Parts*

comms

A *Comms* object, that can used to interact with RemoteTech for this vessel.

Attribute Read-only, cannot be set

Return type *Comms*

Note: Requires *RemoteTech* to be installed.

mass

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

Attribute Read-only, cannot be set

Return type float

dry_mass

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

Attribute Read-only, cannot be set

Return type float

thrust

The total thrust currently being produced by the vessel's engines, in Newtons. This is computed by summing *Engine.thrust* for every engine in the vessel.

Attribute Read-only, cannot be set

Return type float

available_thrust

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

Attribute Read-only, cannot be set

Return type float

max_thrust

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

Attribute Read-only, cannot be set

Return type float

max_vacuum_thrust

The total maximum thrust that can be produced by the vessel's active engines when the vessel is in a vacuum, in Newtons. This is computed by summing *Engine.max_vacuum_thrust* for every active engine.

Attribute Read-only, cannot be set

Return type float

specific_impulse

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

Attribute Read-only, cannot be set

Return type float

vacuum_specific_impulse

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

Attribute Read-only, cannot be set

Return type float

kerbin_sea_level_specific_impulse

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

Attribute Read-only, cannot be set

Return type float

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*

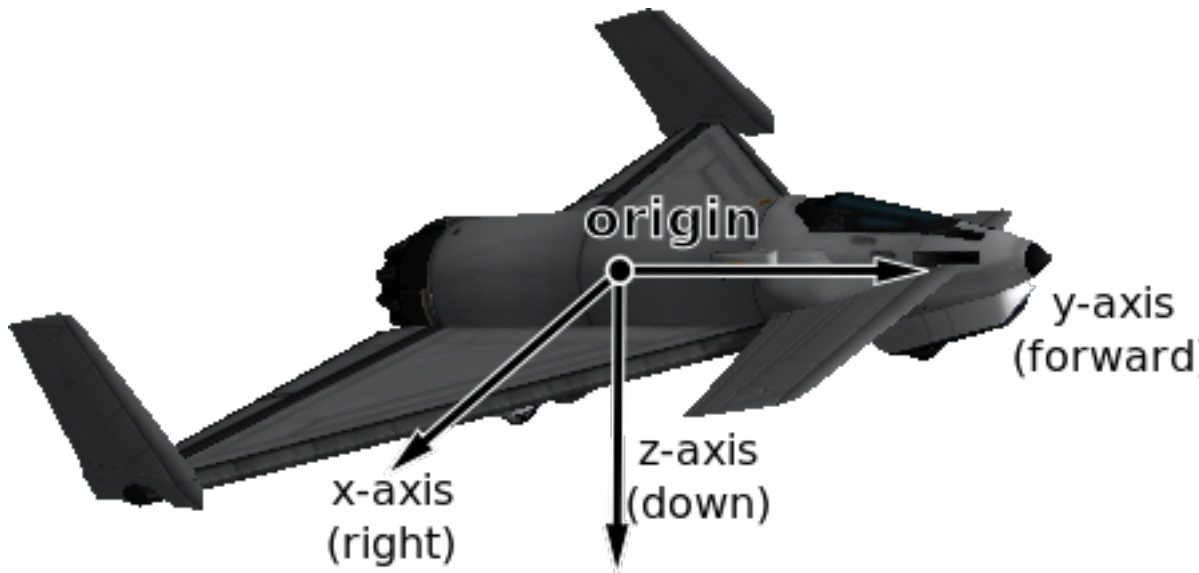


Fig. 7.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

orbital_reference_frame

The reference frame that is fixed relative to the vessel, and orientated with the vessels orbital prograde/normal/radial directions.

- The origin is at the center of mass of the vessel.
- The axes rotate with the orbital prograde/normal/radial directions.

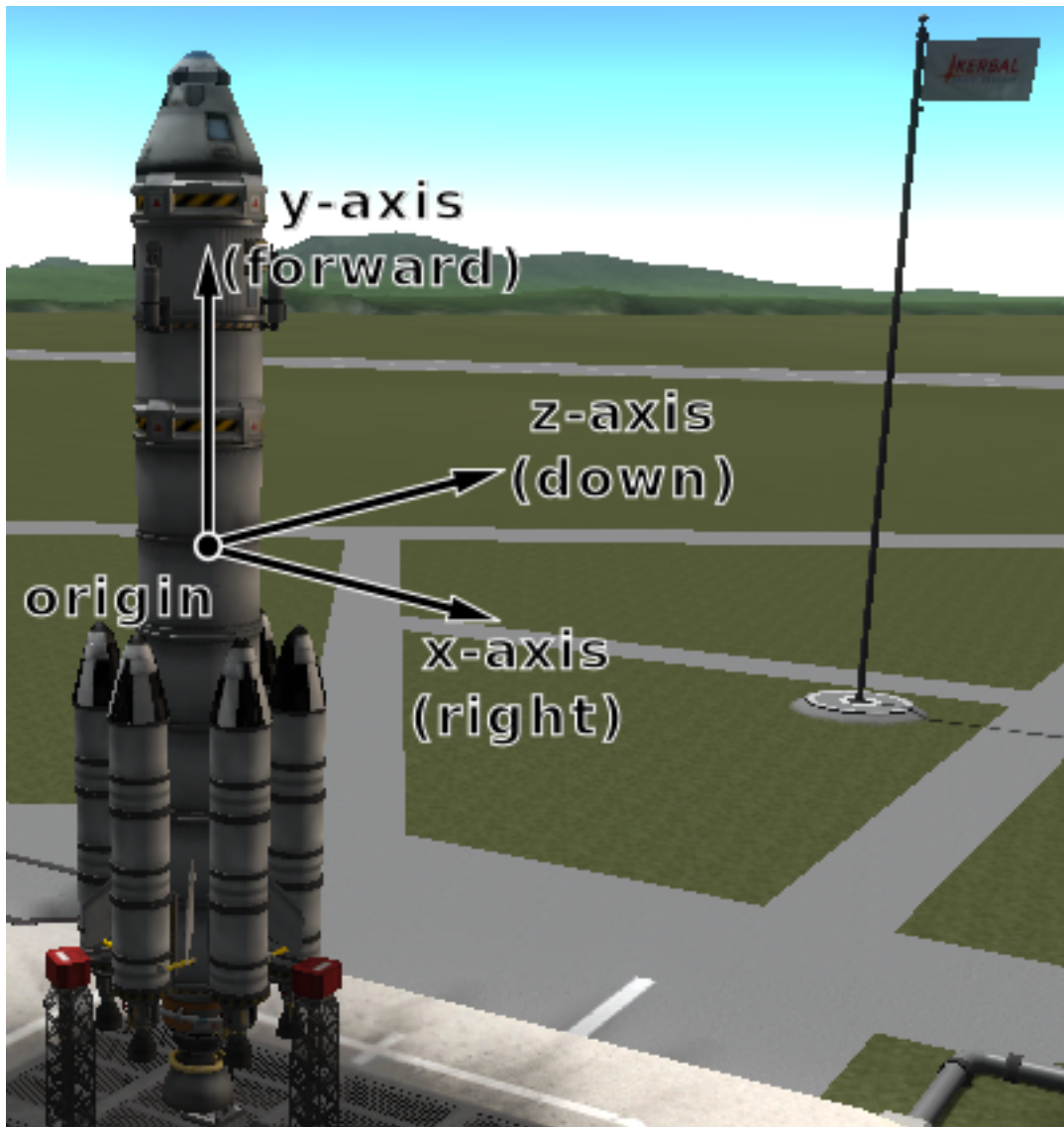


Fig. 7.2: Vessel reference frame origin and axes for the Kerbal-X rocket

- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

Attribute Read-only, cannot be set

Return type *ReferenceFrame*

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

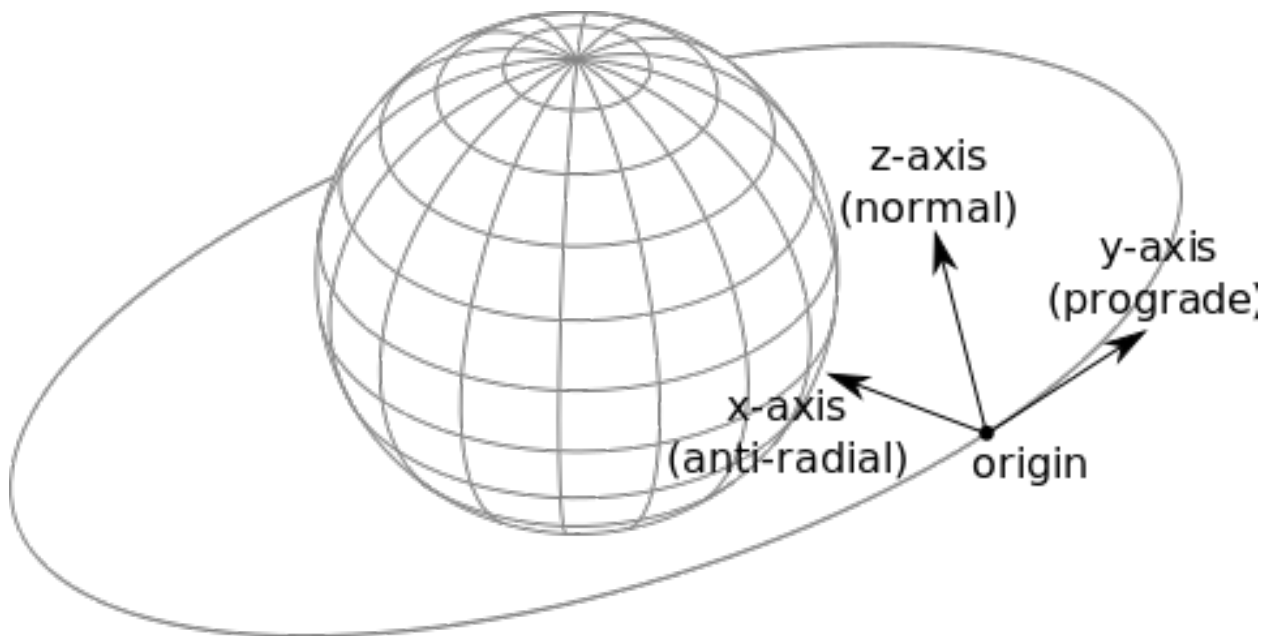


Fig. 7.3: Vessel orbital reference frame origin and axes

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 *ReferenceFrame*

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

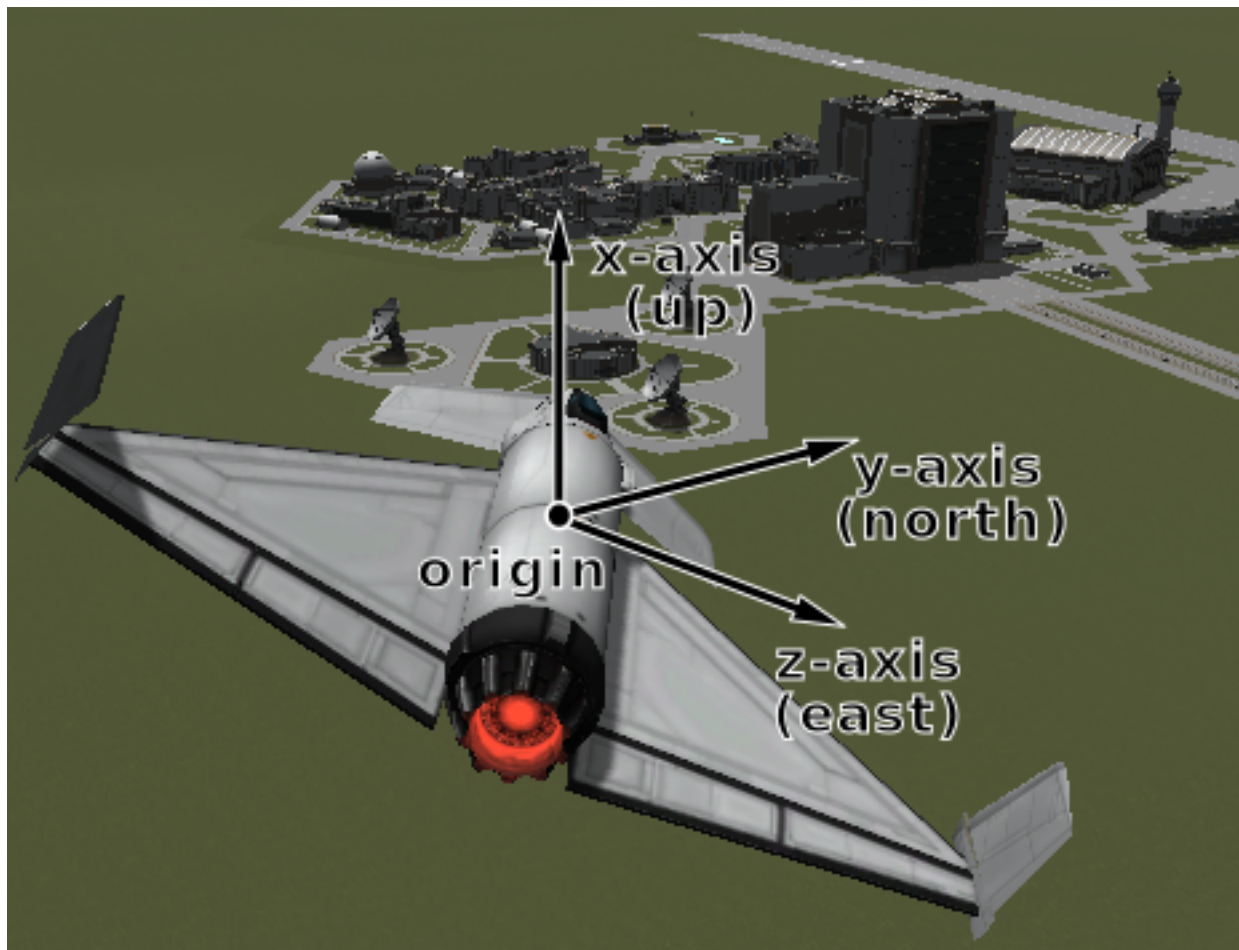


Fig. 7.4: Vessel surface reference frame origin and axes

`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*

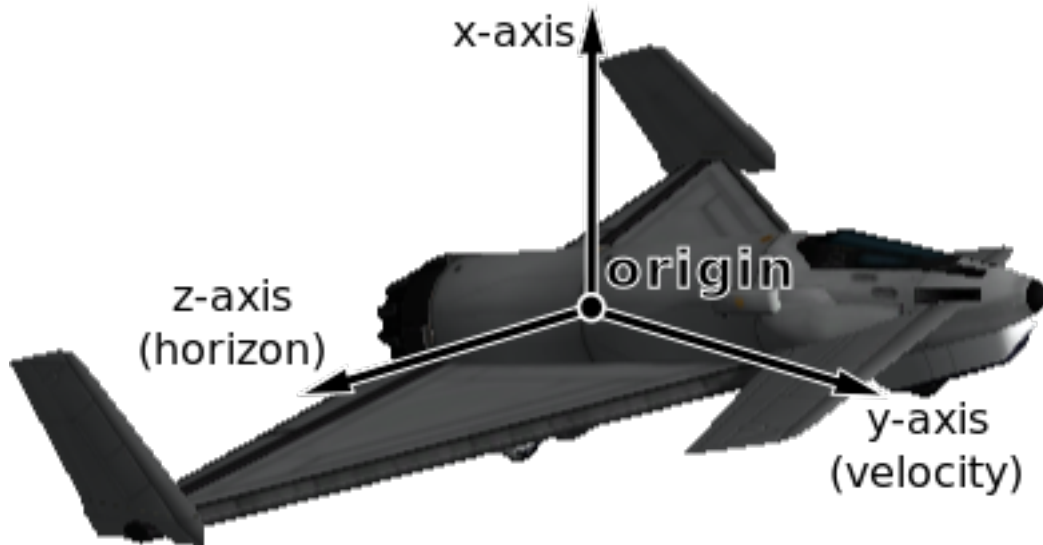


Fig. 7.5: Vessel surface velocity reference frame origin and axes

position (*reference_frame*)

Returns the position vector of the center of mass of the vessel in the given reference frame.

Parameters **reference_frame** ([ReferenceFrame](#)) –

Return type tuple of (float, float, float)

velocity (*reference_frame*)

Returns the velocity vector of the center of mass of the vessel in the given reference frame.

Parameters **reference_frame** ([ReferenceFrame](#)) –

Return type tuple of (float, float, float)

rotation (*reference_frame*)

Returns the rotation of the center of mass of the vessel in the given reference frame.

Parameters **reference_frame** ([ReferenceFrame](#)) –

Return type tuple of (float, float, float, float)

direction (*reference_frame*)

Returns the direction in which the vessel is pointing, as a unit vector, in the given reference frame.

Parameters **reference_frame** ([ReferenceFrame](#)) –

Return type tuple of (float, float, float)

angular_velocity (*reference_frame*)

Returns the angular velocity of the vessel in the given reference frame. The magnitude of the returned vector is the rotational speed in radians per second, and the direction of the vector indicates the axis of rotation (using the right hand rule).

Parameters **reference_frame** ([ReferenceFrame](#)) –

Return type tuple of (float, float, float)

class VesselType

See [Vessel.type](#).

ship
Ship.

station
Station.

lander
Lander.

probe
Probe.

rover
Rover.

base
Base.

debris
Debris.

class VesselSituation

See *Vessel.situation*.

docked
Vessel is docked to another.

escaping
Escaping.

flying
Vessel is flying through an atmosphere.

landed
Vessel is landed on the surface of a body.

orbiting
Vessel is orbiting a body.

pre_launch
Vessel is awaiting launch.

splashed
Vessel has splashed down in an ocean.

sub_orbital
Vessel is on a sub-orbital trajectory.

7.3.3 CelestialBody

class CelestialBody

Represents a celestial body (such as a planet or moon).

name
The name of the body.

Attribute Read-only, cannot be set

Return type str

satellites
A list of celestial bodies that are in orbit around this celestial body.

Attribute Read-only, cannot be set

Return type list of *CelestialBody*

orbit

The orbit of the body.

Attribute Read-only, cannot be set

Return type *Orbit*

mass

The mass of the body, in kilograms.

Attribute Read-only, cannot be set

Return type float

gravitational_parameter

The *standard gravitational parameter* of the body in $m^3 s^{-2}$.

Attribute Read-only, cannot be set

Return type float

surface_gravity

The acceleration due to gravity at sea level (mean altitude) on the body, in m/s^2 .

Attribute Read-only, cannot be set

Return type float

rotational_period

The sidereal rotational period of the body, in seconds.

Attribute Read-only, cannot be set

Return type float

rotational_speed

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

Attribute Read-only, cannot be set

Return type float

equatorial_radius

The equatorial radius of the body, in meters.

Attribute Read-only, cannot be set

Return type float

surface_height (*latitude, longitude*)

The height of the surface relative to mean sea level at the given position, in meters. When over water this is equal to 0.

Parameters

- **latitude** (*float*) – Latitude in degrees
- **longitude** (*float*) – Longitude in degrees

Return type float

bedrock_height (*latitude, longitude*)

The height of the surface relative to mean sea level at the given position, in meters. When over water, this is the height of the sea-bed and is therefore a negative value.

Parameters

- **latitude** (*float*) – Latitude in degrees
- **longitude** (*float*) – Longitude in degrees

Return type float**msl_position** (*latitude, longitude, reference_frame*)

The position at mean sea level at the given latitude and longitude, in the given reference frame.

Parameters

- **latitude** (*float*) – Latitude in degrees
- **longitude** (*float*) – Longitude in degrees
- **reference_frame** ([ReferenceFrame](#)) – Reference frame for the returned position vector

Return type tuple of (float, float, float)**surface_position** (*latitude, longitude, reference_frame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position of the surface of the water.

Parameters

- **latitude** (*float*) – Latitude in degrees
- **longitude** (*float*) – Longitude in degrees
- **reference_frame** ([ReferenceFrame](#)) – Reference frame for the returned position vector

Return type tuple of (float, float, float)**bedrock_position** (*latitude, longitude, reference_frame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position at the bottom of the sea-bed.

Parameters

- **latitude** (*float*) – Latitude in degrees
- **longitude** (*float*) – Longitude in degrees
- **reference_frame** ([ReferenceFrame](#)) – Reference frame for the returned position vector

Return type tuple of (float, float, float)**sphere_of_influence**

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

Attribute Read-only, cannot be set**Return type** float**has_atmosphere**

True if the body has an atmosphere.

Attribute Read-only, cannot be set**Return type** bool**atmosphere_depth**

The depth of the atmosphere, in meters.

Attribute Read-only, cannot be set

Return type float

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

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*

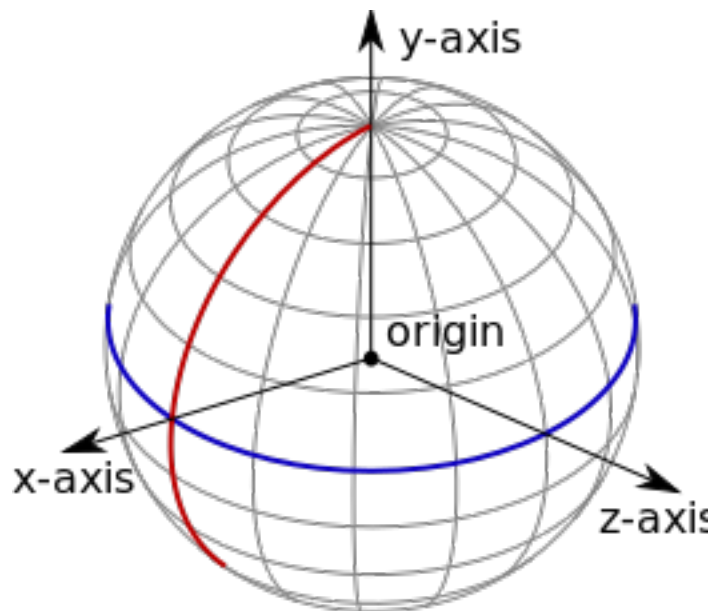


Fig. 7.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

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 *ReferenceFrame*

orbital_reference_frame

Gets the reference frame that is fixed relative to this celestial body, but orientated with the body's orbital prograde/normal/radial directions.

- The origin is at the center of the body.
- The axes rotate with the orbital prograde/normal/radial directions.
- The x-axis points in the orbital anti-radial direction.
- The y-axis points in the orbital prograde direction.
- The z-axis points in the orbital normal direction.

Attribute Read-only, cannot be set

Return type *ReferenceFrame*

position (*reference_frame*)

Returns the position vector of the center of the body in the specified reference frame.

Parameters **reference_frame** (*ReferenceFrame*) –

Return type tuple of (float, float, float)

velocity (*reference_frame*)

Returns the velocity vector of the body in the specified reference frame.

Parameters **reference_frame** (*ReferenceFrame*) –

Return type tuple of (float, float, float)

rotation (*reference_frame*)

Returns the rotation of the body in the specified reference frame.

Parameters **reference_frame** (*ReferenceFrame*) –

Return type tuple of (float, float, float, float)

direction (*reference_frame*)

Returns the direction in which the north pole of the celestial body is pointing, as a unit vector, in the specified reference frame.

Parameters **reference_frame** (*ReferenceFrame*) –

Return type tuple of (float, float, float)

angular_velocity (*reference_frame*)

Returns the angular velocity of the body in the specified reference frame. The magnitude of the vector is the rotational speed of the body, in radians per second, and the direction of the vector indicates the axis of rotation, using the right-hand rule.

Parameters **reference_frame** (*ReferenceFrame*) –

Return type tuple of (float, float, float)

7.3.4 Flight

class Flight

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

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

g_force

The current G force acting on the vessel in m/s^2 .

Attribute Read-only, cannot be set

Return type float

mean_altitude

The altitude above sea level, in meters.

Attribute Read-only, cannot be set

Return type float

surface_altitude

The altitude above the surface of the body or sea level, whichever is closer, in meters.

Attribute Read-only, cannot be set

Return type float

bedrock_altitude

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor.

Attribute Read-only, cannot be set

Return type float

elevation

The elevation of the terrain under the vessel, in meters. This is the height of the terrain above sea level, and is negative when the vessel is over the sea.

Attribute Read-only, cannot be set

Return type float

latitude

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

Attribute Read-only, cannot be set

Return type float

longitude

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

Attribute Read-only, cannot be set

Return type float

velocity

The velocity vector of the vessel. The magnitude of the vector is the speed of the vessel in meters per second. The direction of the vector is the direction of the vessels motion.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

speed

The speed of the vessel in meters per second.

Attribute Read-only, cannot be set

Return type float

horizontal_speed

The horizontal speed of the vessel in meters per second.

Attribute Read-only, cannot be set

Return type float

vertical_speed

The vertical speed of the vessel in meters per second.

Attribute Read-only, cannot be set

Return type float

center_of_mass

The position of the center of mass of the vessel.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

rotation

The rotation of the vessel.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float, float)

direction

The direction vector that the vessel is pointing in.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

pitch

The pitch angle of the vessel relative to the horizon, in degrees. A value between -90° and $+90^\circ$.

Attribute Read-only, cannot be set

Return type float

heading

The heading angle of the vessel relative to north, in degrees. A value between 0° and 360° .

Attribute Read-only, cannot be set

Return type float

roll

The roll angle of the vessel relative to the horizon, in degrees. A value between -180° and $+180^\circ$.

Attribute Read-only, cannot be set

Return type float

prograde

The unit direction vector pointing in the prograde direction.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

retrograde

The unit direction vector pointing in the retrograde direction.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

normal

The unit direction vector pointing in the normal direction.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

anti_normal

The unit direction vector pointing in the anti-normal direction.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

radial

The unit direction vector pointing in the radial direction.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

anti_radial

The unit direction vector pointing in the anti-radial direction.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

atmosphere_density

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

Attribute Read-only, cannot be set

Return type float

dynamic_pressure

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

Attribute Read-only, cannot be set

Return type float

Note: Calculated using [KSPs stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

static_pressure

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

Attribute Read-only, cannot be set

Return type float

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

aerodynamic_force

The total aerodynamic forces acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

lift

The [aerodynamic lift](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

drag

The [aerodynamic drag](#) currently acting on the vessel, as a vector pointing in the direction of the force, with its magnitude equal to the strength of the force in Newtons.

Attribute Read-only, cannot be set

Return type tuple of (float, float, float)

Note: Calculated using [KSPs stock aerodynamic model](#). Not available when [Ferram Aerospace Research](#) is installed.

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

Note: Not available when [Ferram Aerospace Research](#) is installed.

mach

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

Attribute Read-only, cannot be set

Return type float

Note: Not available when [Ferram Aerospace Research](#) is installed.

equivalent_air_speed

The [equivalent air speed](#) of the vessel, in m/s .

Attribute Read-only, cannot be set

Return type float

Note: Not available when [Ferram Aerospace Research](#) is installed.

terminal_velocity

An estimate of the current terminal velocity of the vessel, in m/s . This is the speed at which the drag forces cancel out the force of gravity.

Attribute Read-only, cannot be set

Return type float

Note: Calculated using [KSPs stock aerodynamic model](#), or [Ferram Aerospace Research](#) if it is installed.

angle_of_attack

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

Attribute Read-only, cannot be set

Return type float

sideslip_angle

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

Attribute Read-only, cannot be set

Return type float

total_air_temperature

The [total air temperature](#) of the atmosphere around the vessel, in Kelvin. This temperature includes the [Flight.static_air_temperature](#) and the vessel's kinetic energy.

Attribute Read-only, cannot be set

Return type float

static_air_temperature

The [static \(ambient\) temperature](#) of the atmosphere around the vessel, in Kelvin.

Attribute Read-only, cannot be set

Return type float

stall_fraction

Gets the current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

Attribute Read-only, cannot be set

Return type float

Note: Requires [Ferram Aerospace Research](#).

drag_coefficient

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

Attribute Read-only, cannot be set

Return type float

Note: Requires [Ferram Aerospace Research](#).

lift_coefficient

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

Attribute Read-only, cannot be set

Return type float

Note: Requires [Ferram Aerospace Research](#).

ballistic_coefficient

Gets the [ballistic coefficient](#).

Attribute Read-only, cannot be set

Return type float

Note: Requires [Ferram Aerospace Research](#).

thrust_specific_fuel_consumption

Gets 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 product one newton of thrust.

Attribute Read-only, cannot be set

Return type float

Note: Requires [Ferram Aerospace Research](#).

7.3.5 Orbit

class Orbit

Describes an orbit. For example, the orbit of a vessel, obtained by calling *Vessel.orbit*, or a celestial body, obtained by calling *CelestialBody.orbit*.

body

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

Attribute Read-only, cannot be set

Return type *CelestialBody*

apoapsis

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

Attribute Read-only, cannot be set

Return type float

Note: For the apoapsis altitude reported on the in-game map view, use *Orbit.apoapsis_altitude*.

periapsis

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

Attribute Read-only, cannot be set

Return type float

Note: For the periapsis altitude reported on the in-game map view, use *Orbit.periapsis_altitude*.

apoapsis_altitude

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

Attribute Read-only, cannot be set

Return type float

Note: This is equal to *Orbit.apoapsis* minus the equatorial radius of the body.

periapsis_altitude

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

Attribute Read-only, cannot be set

Return type float

Note: This is equal to *Orbit.periapsis* minus the equatorial radius of the body.

semi_major_axis

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

Attribute Read-only, cannot be set

Return type float

semi_minor_axis

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

Attribute Read-only, cannot be set

Return type float

radius

The current radius of the orbit, in meters. This is the distance between the center of mass of the object in orbit, and the center of mass of the body around which it is orbiting.

Attribute Read-only, cannot be set

Return type float

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

speed

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

Attribute Read-only, cannot be set

Return type float

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

period

The orbital period, in seconds.

Attribute Read-only, cannot be set

Return type float

time_to_apoapsis

The time until the object reaches apoapsis, in seconds.

Attribute Read-only, cannot be set

Return type float

time_to_periapsis

The time until the object reaches periapsis, in seconds.

Attribute Read-only, cannot be set

Return type float

eccentricity

The [eccentricity](#) of the orbit.

Attribute Read-only, cannot be set

Return type float

inclination

The [inclination](#) of the orbit, in radians.

Attribute Read-only, cannot be set

Return type float

longitude_of_ascending_node

The [longitude of the ascending node](#), in radians.

Attribute Read-only, cannot be set

Return type float

argument_of_periapsis

The [argument of periapsis](#), in radians.

Attribute Read-only, cannot be set

Return type float

mean_anomaly_at_epoch

The [mean anomaly at epoch](#).

Attribute Read-only, cannot be set

Return type float

epoch

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

Attribute Read-only, cannot be set

Return type float

mean_anomaly

The [mean anomaly](#).

Attribute Read-only, cannot be set

Return type float

eccentric_anomaly

The [eccentric anomaly](#).

Attribute Read-only, cannot be set

Return type float

static reference_plane_normal (*reference_frame*)

The unit direction vector 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](#)) –

Return type tuple of (float, float, float)

static reference_plane_direction (*reference_frame*)

The unit direction vector from which the orbits longitude of ascending node is measured, in the given reference frame.

Parameters **reference_frame** ([ReferenceFrame](#)) –

Return type tuple of (float, float, float)

time_to_soi_change

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

Attribute Read-only, cannot be set

Return type float

next_orbit

If the object is going to change sphere of influence in the future, returns the new orbit after the change. Otherwise returns None.

Attribute Read-only, cannot be set

Return type [Orbit](#)

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.

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.

sas

The state of SAS.

Attribute Can be read or written

Return type bool

Note: Equivalent to *AutoPilot.sas*

sas_mode

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

Attribute Can be read or written

Return type *SASMode*

Note: Equivalent to *AutoPilot.sas_mode*

speed_mode

The current *SpeedMode* of the navball. This is the mode displayed next to the speed at the top of the navball.

Attribute Can be read or written

Return type *SpeedMode*

rcs

The state of RCS.

Attribute Can be read or written

Return type bool

gear

The state of the landing gear/legs.

Attribute Can be read or written

Return type bool

lights

The state of the lights.

Attribute Can be read or written

Return type bool

brakes

The state of the wheel brakes.

Attribute Can be read or written

Return type bool

abort

The state of the abort action group.

Attribute Can be read or written

Return type bool

throttle

The state of the throttle. A value between 0 and 1.

Attribute Can be read or written

Return type float

pitch

The state of the pitch control. A value between -1 and 1. Equivalent to the w and s keys.

Attribute Can be read or written

Return type float

yaw

The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

Attribute Can be read or written

Return type float

roll

The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

Attribute Can be read or written

Return type float

forward

The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

Attribute Can be read or written

Return type float

up

The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

Attribute Can be read or written

Return type float

right

The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

Attribute Can be read or written

Return type float

wheel_throttle

The state of the wheel throttle. A value between -1 and 1. A value of 1 rotates the wheels forwards, a value of -1 rotates the wheels backwards.

Attribute Can be read or written

Return type float

wheel_steering

The state of the wheel steering. A value between -1 and 1. A value of 1 steers to the left, and a value of -1 steers to the right.

Attribute Can be read or written

Return type float

current_stage

The current stage of the vessel. Corresponds to the stage number in the in-game UI.

Attribute Read-only, cannot be set

Return type `int`

activate_next_stage()

Activates the next stage. Equivalent to pressing the space bar in-game.

Returns A list of vessel objects that are jettisoned from the active vessel.

Return type list of *Vessel*

get_action_group(group)

Returns `True` if the given action group is enabled.

Parameters **group** (*int*) – A number between 0 and 9 inclusive.

Return type `bool`

set_action_group(group, state)

Sets the state of the given action group (a value between 0 and 9 inclusive).

Parameters

- **group** (*int*) – A number between 0 and 9 inclusive.
- **state** (*bool*) –

toggle_action_group(group)

Toggles the state of the given action group.

Parameters **group** (*int*) – A number between 0 and 9 inclusive.

add_node(ut[, prograde = 0.0][[, normal = 0.0][[, radial = 0.0]])

Creates a maneuver node at the given universal time, and returns a *Node* object that can be used to modify it. Optionally sets the magnitude of the delta-v for the maneuver node in the prograde, normal and radial directions.

Parameters

- **ut** (*float*) – Universal time of the maneuver node.
- **prograde** (*float*) – Delta-v in the prograde direction.
- **normal** (*float*) – Delta-v in the normal direction.
- **radial** (*float*) – Delta-v in the radial direction.

Return type *Node*

nodes

Returns a list of all existing maneuver nodes, ordered by time from first to last.

Attribute Read-only, cannot be set

Return type list of *Node*

remove_nodes()

Remove all maneuver nodes.

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

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.

7.3.7 Parts

The following classes allow interaction with a vessels individual parts.

- *Parts*
- *Part*
- *Module*
- *Specific Types of Part*
 - *Cargo Bay*
 - *Decoupler*
 - *Docking Port*
 - *Engine*
 - *Fairing*
 - *Intake*
 - *Landing Gear*
 - *Landing Leg*
 - *Launch Clamp*
 - *Light*
 - *Parachute*
 - *Radiator*
 - *Resource Converter*
 - *Resource Harvester*
 - *Reaction Wheel*
 - *Sensor*
 - *Solar Panel*
- *Trees of Parts*
 - *Traversing the Tree*
 - *Attachment Modes*
- *Fuel Lines*
- *Staging*

Parts

class **Parts**

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling *Vessel.parts*.

all

A list of all of the vessels parts.

Attribute Read-only, cannot be set

Return type list of *Part*

root

The vessels root part.

Attribute Read-only, cannot be set

Return type *Part*

Note: See the discussion on *Trees of Parts*.

controlling

The part from which the vessel is controlled.

Attribute Can be read or written

Return type *Part*

with_name (*name*)

A list of parts whose *Part.name* is *name*.

Parameters **name** (*str*) –

Return type list of *Part*

with_title (*title*)

A list of all parts whose *Part.title* is *title*.

Parameters **title** (*str*) –

Return type list of *Part*

with_module (*module_name*)

A list of all parts that contain a *Module* whose *Module.name* is *module_name*.

Parameters **module_name** (*str*) –

Return type list of *Part*

in_stage (*stage*)

A list of all parts that are activated in the given *stage*.

Parameters **stage** (*int*) –

Return type list of *Part*

Note: See the discussion on *Staging*.

in_decouple_stage (*stage*)

A list of all parts that are decoupled in the given *stage*.

Parameters **stage** (*int*) –

Return type list of *Part*

Note: See the discussion on *Staging*.

modules_with_name (*module_name*)

A list of modules (combined across all parts in the vessel) whose *Module.name* is *module_name*.

Parameters **module_name** (*str*) –

Return type list of *Module*

cargo_bays

A list of all cargo bays in the vessel.

Attribute Read-only, cannot be set

Return type list of *CargoBay*

decouplers

A list of all decouplers in the vessel.

Attribute Read-only, cannot be set

Return type list of *Decoupler*

docking_ports

A list of all docking ports in the vessel.

Attribute Read-only, cannot be set

Return type list of *DockingPort*

docking_port_with_name (*name*)

The first docking port in the vessel with the given port name, as returned by *DockingPort.name*. Returns None if there are no such docking ports.

Parameters *name* (*str*) –

Return type *DockingPort*

engines

A list of all engines in the vessel.

Attribute Read-only, cannot be set

Return type list of *Engine*

fairings

A list of all fairings in the vessel.

Attribute Read-only, cannot be set

Return type list of *Fairing*

intakes

A list of all intakes in the vessel.

Attribute Read-only, cannot be set

Return type list of *Intake*

landing_gear

A list of all landing gear attached to the vessel.

Attribute Read-only, cannot be set

Return type list of *LandingGear*

landing_legs

A list of all landing legs attached to the vessel.

Attribute Read-only, cannot be set

Return type list of *LandingLeg*

launch_clamps

A list of all launch clamps attached to the vessel.

Attribute Read-only, cannot be set

Return type list of *LaunchClamp*

lights

A list of all lights in the vessel.

Attribute Read-only, cannot be set

Return type list of *Light*

parachutes

A list of all parachutes in the vessel.

Attribute Read-only, cannot be set

Return type list of *Parachute*

radiators

A list of all radiators in the vessel.

Attribute Read-only, cannot be set

Return type list of *Radiator*

reaction_wheels

A list of all reaction wheels in the vessel.

Attribute Read-only, cannot be set

Return type list of *ReactionWheel*

resource_converters

A list of all resource converters in the vessel.

Attribute Read-only, cannot be set

Return type list of *ResourceConverter*

resource_harvesters

A list of all resource harvesters in the vessel.

Attribute Read-only, cannot be set

Return type list of *ResourceHarvester*

sensors

A list of all sensors in the vessel.

Attribute Read-only, cannot be set

Return type list of *Sensor*

solar_panels

A list of all solar panels in the vessel.

Attribute Read-only, cannot be set

Return type list of *SolarPanel*

Part

class Part

Instances of this class represents a part. A vessel is made of multiple parts. Instances can be obtained by various methods in *Parts*.

name

Internal name of the part, as used in *part cfg files*. For example “Mark1-2Pod”.

Attribute Read-only, cannot be set

Return type str

title

Title of the part, as shown when the part is right clicked in-game. For example “Mk1-2 Command Pod”.

Attribute Read-only, cannot be set

Return type str

cost

The cost of the part, in units of funds.

Attribute Read-only, cannot be set

Return type float

vessel

The vessel that contains this part.

Attribute Read-only, cannot be set

Return type *Vessel*

parent

The parts parent. Returns *None* if the part does not have a parent. This, in combination with *Part.children*, can be used to traverse the vessels parts tree.

Attribute Read-only, cannot be set

Return type *Part*

Note: See the discussion on *Trees of Parts*.

children

The parts children. Returns an empty list if the part has no children. This, in combination with *Part.parent*, can be used to traverse the vessels parts tree.

Attribute Read-only, cannot be set

Return type list of *Part*

Note: See the discussion on *Trees of Parts*.

axially_attached

Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns *False*.

Attribute Read-only, cannot be set

Return type bool

Note: See the discussion on *Attachment Modes*.

radially_attached

Whether the part is radially attached to its parent, i.e. on the side of its parent. If the part has no parent, returns *False*.

Attribute Read-only, cannot be set

Return type bool

Note: See the discussion on *Attachment Modes*.

stage

The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

Attribute Read-only, cannot be set

Return type int

Note: See the discussion on *Staging*.

decouple_stage

The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

Attribute Read-only, cannot be set

Return type int

Note: See the discussion on *Staging*.

massless

Whether the part is *massless*.

Attribute Read-only, cannot be set

Return type bool

mass

The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

Attribute Read-only, cannot be set

Return type float

dry_mass

The mass of the part, not including any resources it contains, in kilograms. Returns zero if the part is massless.

Attribute Read-only, cannot be set

Return type float

impact_tolerance

The impact tolerance of the part, in meters per second.

Attribute Read-only, cannot be set

Return type float

temperature

Temperature of the part, in Kelvin.

Attribute Read-only, cannot be set

Return type float

skin_temperature

Temperature of the skin of the part, in Kelvin.

Attribute Read-only, cannot be set

Return type float

max_temperature

Maximum temperature that the part can survive, in Kelvin.

Attribute Read-only, cannot be set

Return type float

max_skin_temperature

Maximum temperature that the skin of the part can survive, in Kelvin.

Attribute Read-only, cannot be set

Return type float

thermal_mass

A measure of how much energy it takes to increase the internal temperature of the part, in Joules per Kelvin.

Attribute Read-only, cannot be set

Return type float

thermal_skin_mass

A measure of how much energy it takes to increase the skin temperature of the part, in Joules per Kelvin.

Attribute Read-only, cannot be set

Return type float

thermal_resource_mass

A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

Attribute Read-only, cannot be set

Return type float

thermal_conduction_flux

The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

Attribute Read-only, cannot be set

Return type float

thermal_convection_flux

The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

Attribute Read-only, cannot be set

Return type float

thermal_radiation_flux

The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

Attribute Read-only, cannot be set

Return type float

thermal_internal_flux

The rate at which heat energy is being generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

Attribute Read-only, cannot be set

Return type float

thermal_skin_to_internal_flux

The rate at which heat energy is transferring between the part's skin and its internals. Measured in energy per unit time, or power, in Watts. A positive value means the part's internals are gaining heat energy, and negative means its skin is gaining heat energy.

Attribute Read-only, cannot be set

Return type float

resources

A *Resources* object for the part.

Attribute Read-only, cannot be set

Return type *Resources*

crossfeed

Whether this part is crossfeed capable.

Attribute Read-only, cannot be set

Return type bool

is_fuel_line

Whether this part is a fuel line.

Attribute Read-only, cannot be set

Return type bool

fuel_lines_from

The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

Attribute Read-only, cannot be set

Return type list of *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 *Part*

Note: See the discussion on *Fuel Lines*.

modules

The modules for this part.

Attribute Read-only, cannot be set

Return type list of *Module*

cargo_bay

A *CargoBay* if the part is a cargo bay, otherwise None.

Attribute Read-only, cannot be set

Return type *CargoBay*

decoupler

A *Decoupler* if the part is a decoupler, otherwise None.

Attribute Read-only, cannot be set

Return type *Decoupler*

docking_port

A *DockingPort* if the part is a docking port, otherwise None.

Attribute Read-only, cannot be set

Return type *DockingPort*

engine

An *Engine* if the part is an engine, otherwise None.

Attribute Read-only, cannot be set

Return type *Engine*

fairing

A *Fairing* if the part is a fairing, otherwise None.

Attribute Read-only, cannot be set

Return type *Fairing*

intake

An *Intake* if the part is an intake, otherwise None.

Attribute Read-only, cannot be set

Return type *Intake*

landing_gear

A *LandingGear* if the part is a landing gear , otherwise None.

Attribute Read-only, cannot be set

Return type *LandingGear*

landing_leg

A *LandingLeg* if the part is a landing leg, otherwise None.

Attribute Read-only, cannot be set

Return type *LandingLeg*

launch_clamp

A *LaunchClamp* if the part is a launch clamp, otherwise None.

Attribute Read-only, cannot be set

Return type *LaunchClamp*

light

A *Light* if the part is a light, otherwise None.

Attribute Read-only, cannot be set

Return type *Light*

parachute

A *Parachute* if the part is a parachute, otherwise None.

Attribute Read-only, cannot be set

Return type *Parachute*

radiator

A *Radiator* if the part is a radiator, otherwise None.

Attribute Read-only, cannot be set

Return type *Radiator*

reaction_wheel

A *ReactionWheel* if the part is a reaction wheel, otherwise None.

Attribute Read-only, cannot be set

Return type *ReactionWheel*

resource_converter

A *ResourceConverter* if the part is a resource converter, otherwise None.

Attribute Read-only, cannot be set

Return type *ResourceConverter*

resource_harvester

A *ResourceHarvester* if the part is a resource harvester, otherwise None.

Attribute Read-only, cannot be set

Return type *ResourceHarvester*

sensor

A *Sensor* if the part is a sensor, otherwise None.

Attribute Read-only, cannot be set

Return type *Sensor*

solar_panel

A *SolarPanel* if the part is a solar panel, otherwise None.

Attribute Read-only, cannot be set

Return type *SolarPanel*

position (*reference_frame*)

The position of the part in the given reference frame.

Parameters **reference_frame** (*ReferenceFrame*) –

Return type tuple of (float, float, float)

direction (*reference_frame*)

The direction of the part in the given reference frame.

Parameters **reference_frame** (*ReferenceFrame*) –

Return type tuple of (float, float, float)

velocity (*reference_frame*)

The velocity of the part in the given reference frame.

Parameters **reference_frame** (*ReferenceFrame*) –

Return type tuple of (float, float, float)

rotation (*reference_frame*)

The rotation of the part in the given reference frame.

Parameters `reference_frame` (`ReferenceFrame`) –

Return type tuple of (float, float, float, float)

reference_frame

The reference frame that is fixed relative to this part.

- The origin is at the position of the part.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

Attribute Read-only, cannot be set

Return type `ReferenceFrame`

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by `DockingPort.reference_frame`.

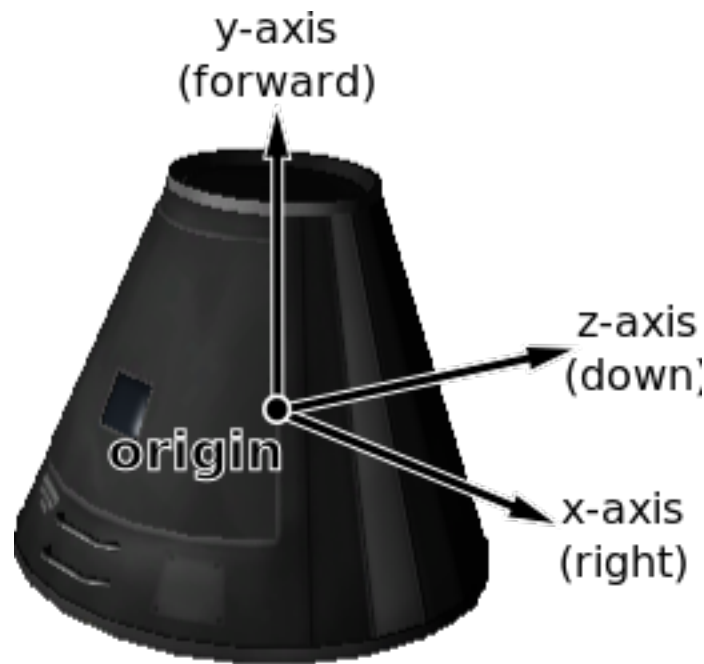


Fig. 7.7: Mk1 Command Pod reference frame origin and axes

Module

class Module

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” `PartModule` that contains all the functionality of an engine. This class allows you to interact with KSPs `PartModules`, and any `PartModules` that have been added by other mods.

name

Name of the `PartModule`. For example, “ModuleEngines”.

Attribute Read-only, cannot be set

Return type str

part

The part that contains this module.

Attribute Read-only, cannot be set

Return type *Part*

fields

The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

Attribute Read-only, cannot be set

Return type dict from str to str

has_field (*name*)

Returns `True` if the module has a field with the given name.

Parameters **name** (*str*) – Name of the field.

Return type bool

get_field (*name*)

Returns the value of a field.

Parameters **name** (*str*) – Name of the field.

Return type str

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 str

has_event (*name*)

`True` if the module has an event with the given name.

Parameters **name** (*str*) –

Return type bool

trigger_event (*name*)

Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters **name** (*str*) –

actions

A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

Attribute Read-only, cannot be set

Return type list of str

has_action (*name*)

`True` if the part has an action with the given name.

Parameters **name** (*str*) –

Return type bool

set_action (*name* [, *value* = *True*])

Set the value of an action with the given name.

Parameters

- **name** (*str*) –
- **value** (*bool*) –

Specific Types of Part

The following classes provide functionality for specific types of part.

- *Cargo Bay*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Fairing*
- *Intake*
- *Landing Gear*
- *Landing Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *Sensor*
- *Solar Panel*

Cargo Bay

class CargoBay

Obtained by calling *Part.cargo_bay*.

part

The part object for this cargo bay.

Attribute Read-only, cannot be set

Return type *Part*

state

The state of the cargo bay.

Attribute Read-only, cannot be set

Return type *CargoBayState*

open

Whether the cargo bay is open.

Attribute Can be read or written

Return type *bool*

class CargoBayState

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.

Decoupler**class Decoupler**

Obtained by calling *Part.decoupler*

part

The part object for this decoupler.

Attribute Read-only, cannot be set

Return type *Part*

decouple ()

Fires the decoupler. Has no effect if the decoupler has already fired.

decoupled

Whether the decoupler has fired.

Attribute Read-only, cannot be set

Return type bool

impulse

The impulse that the decoupler imparts when it is fired, in Newton seconds.

Attribute Read-only, cannot be set

Return type float

Docking Port**class DockingPort**

Obtained by calling *Part.docking_port*

part

The part object for this docking port.

Attribute Read-only, cannot be set

Return type *Part*

name

The port name of the docking port. This is the name of the port that can be set in the right click menu, when the **Docking Port Alignment Indicator** mod is installed. If this mod is not installed, returns the title of the part (*Part.title*).

Attribute Can be read or written

Return type str

state

The current state of the docking port.

Attribute Read-only, cannot be set

Return type *DockingPortState*

docked_part

The part that this docking port is docked to. Returns *None* if this docking port is not docked to anything.

Attribute Read-only, cannot be set

Return type *Part*

undock ()

Undocks the docking port and returns the vessel that was undocked from. After undocking, the active vessel may change (*active_vessel*). This method can be called for either docking port in a docked pair - both calls will have the same effect. Returns *None* if the docking port is not docked to anything.

Return type *Vessel*

reengage_distance

The distance a docking port must move away when it undocks before it becomes ready to dock with another port, in meters.

Attribute Read-only, cannot be set

Return type float

has_shield

Whether the docking port has a shield.

Attribute Read-only, cannot be set

Return type bool

shielded

The state of the docking ports shield, if it has one. Returns *True* if the docking port has a shield, and the shield is closed. Otherwise returns *False*. When set to *True*, the shield is closed, and when set to *False* the shield is opened. If the docking port does not have a shield, setting this attribute has no effect.

Attribute Can be read or written

Return type bool

position (reference_frame)

The position of the docking port in the given reference frame.

Parameters *reference_frame* (*ReferenceFrame*) –

Return type tuple of (float, float, float)

direction (reference_frame)

The direction that docking port points in, in the given reference frame.

Parameters *reference_frame* (*ReferenceFrame*) –

Return type tuple of (float, float, float)

rotation (reference_frame)

The rotation of the docking port, in the given reference frame.

Parameters *reference_frame* (*ReferenceFrame*) –

Return type tuple of (float, float, float, float)

reference_frame

The reference frame that is fixed relative to this docking port, and oriented with the port.

- The origin is at the position of the docking port.
- The axes rotate with the docking port.
- The x-axis points out to the right side of the docking port.
- The y-axis points in the direction the docking port is facing.
- The z-axis points out of the bottom off the docking port.

Attribute Read-only, cannot be set

Return type *ReferenceFrame*

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by *Part.reference_frame*.



Fig. 7.8: Docking port reference frame origin and axes

class DockingPortState

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).

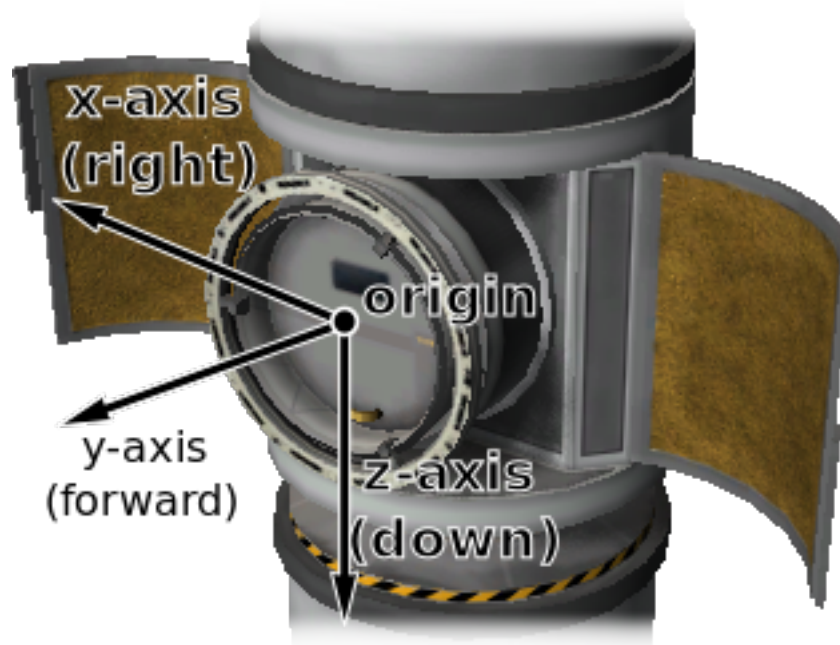


Fig. 7.9: Inline docking port reference frame origin and axes

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

Obtained by calling *Part.engine*.

part

The part object for this engine.

Attribute Read-only, cannot be set

Return type *Part*

active

Whether the engine is active. Setting this attribute may have no effect, depending on *Engine.can_shutdown* and *Engine.can_restart*.

Attribute Can be read or written

Return type bool

thrust

The current amount of thrust being produced by the engine, in Newtons. Returns zero if the engine is not active or if it has no fuel.

Attribute Read-only, cannot be set

Return type float

available_thrust

The maximum available amount of thrust that can be produced by the engine, in Newtons. This takes `Engine.thrust_limit` into account, and is the amount of thrust produced by the engine when activated and the main throttle is set to 100%. Returns zero if the engine does not have any fuel.

Attribute Read-only, cannot be set

Return type float

max_thrust

Gets the maximum amount of thrust that can be produced by the engine, in Newtons. This is the amount of thrust produced by the engine when activated, `Engine.thrust_limit` is set to 100% and the main vessel's throttle is set to 100%.

Attribute Read-only, cannot be set

Return type float

max_vacuum_thrust

The maximum amount of thrust that can be produced by the engine in a vacuum, in Newtons. This is the amount of thrust produced by the engine when activated, `Engine.thrust_limit` is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

Attribute Read-only, cannot be set

Return type float

thrust_limit

The thrust limiter of the engine. A value between 0 and 1. Setting this attribute may have no effect, for example the thrust limit for a solid rocket booster cannot be changed in flight.

Attribute Can be read or written

Return type float

specific_impulse

The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

Attribute Read-only, cannot be set

Return type float

vacuum_specific_impulse

The vacuum specific impulse of the engine, in seconds.

Attribute Read-only, cannot be set

Return type float

kerbin_sea_level_specific_impulse

The specific impulse of the engine at sea level on Kerbin, in seconds.

Attribute Read-only, cannot be set

Return type float

propellants

The names of resources that the engine consumes.

Attribute Read-only, cannot be set

Return type list of str

propellant_ratios

The ratios of resources that the engine consumes. A dictionary mapping resource names to the ratios at which they are consumed by the engine.

Attribute Read-only, cannot be set

Return type dict from str to float

has_fuel

Whether the engine has run out of fuel (or flamed out).

Attribute Read-only, cannot be set

Return type bool

throttle

The current throttle setting for the engine. A value between 0 and 1. This is not necessarily the same as the vessel's main throttle setting, as some engines take time to adjust their throttle (such as jet engines).

Attribute Read-only, cannot be set

Return type float

throttle_locked

Whether the `Control.throttle` affects the engine. For example, this is `True` for liquid fueled rockets, and `False` for solid rocket boosters.

Attribute Read-only, cannot be set

Return type bool

can_restart

Whether the engine can be restarted once shutdown. If the engine cannot be shutdown, returns `False`. For example, this is `True` for liquid fueled rockets and `False` for solid rocket boosters.

Attribute Read-only, cannot be set

Return type bool

can_shutdown

Gets whether the engine can be shutdown once activated. For example, this is `True` for liquid fueled rockets and `False` for solid rocket boosters.

Attribute Read-only, cannot be set

Return type bool

has_modes

Whether the engine has multiple modes of operation.

Attribute Read-only, cannot be set

Return type bool

mode

The name of the current engine mode.

Attribute Can be read or written

Return type str

modes

The available modes for the engine. A dictionary mapping mode names to `Engine` objects.

Attribute Read-only, cannot be set

Return type dict from str to *Engine*

toggle_mode()

Toggle the current engine mode.

auto_mode_switch

Whether the engine will automatically switch modes.

Attribute Can be read or written

Return type bool

gimballed

Whether the engine nozzle is gimballed, i.e. can provide a turning force.

Attribute Read-only, cannot be set

Return type bool

gimbal_range

The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

Attribute Read-only, cannot be set

Return type float

gimbal_locked

Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

Attribute Can be read or written

Return type bool

gimbal_limit

The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

Attribute Can be read or written

Return type float

Fairing

class Fairing

Obtained by calling *Part.fairing*.

part

The part object for this fairing.

Attribute Read-only, cannot be set

Return type *Part*

jettison()

Jettison the fairing. Has no effect if it has already been jettisoned.

jettisoned

Whether the fairing has been jettisoned.

Attribute Read-only, cannot be set

Return type bool

Intake

class Intake

Obtained by calling *Part.intake*.

part

The part object for this intake.

Attribute Read-only, cannot be set

Return type *Part*

open

Whether the intake is open.

Attribute Can be read or written

Return type bool

speed

Speed of the flow into the intake, in *m/s*.

Attribute Read-only, cannot be set

Return type float

flow

The rate of flow into the intake, in units of resource per second.

Attribute Read-only, cannot be set

Return type float

area

The area of the intake's opening, in square meters.

Attribute Read-only, cannot be set

Return type float

Landing Gear

class LandingGear

Obtained by calling *Part.landing_gear*.

part

The part object for this landing gear.

Attribute Read-only, cannot be set

Return type *Part*

state

Gets the current state of the landing gear.

Attribute Read-only, cannot be set

Return type *LandingGearState*

Note: Fixed landing gear are always deployed.

deployable

Whether the landing gear is deployable.

Attribute Read-only, cannot be set

Return type bool

deployed

Whether the landing gear is deployed.

Attribute Can be read or written

Return type bool

Note: Fixed landing gear are always deployed. Returns an error if you try to deploy fixed landing gear.

class LandingGearState

See *LandingGear.state*.

deployed

Landing gear is fully deployed.

retracted

Landing gear is fully retracted.

deploying

Landing gear is being deployed.

retracting

Landing gear is being retracted.

Landing Leg**class LandingLeg**

Obtained by calling *Part.landing_leg*.

part

The part object for this landing leg.

Attribute Read-only, cannot be set

Return type *Part*

state

The current state of the landing leg.

Attribute Read-only, cannot be set

Return type *LandingLegState*

deployed

Whether the landing leg is deployed.

Attribute Can be read or written

Return type bool

class LandingLegState

See *LandingLeg.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.

repairing

Landing leg is being repaired.

Launch Clamp**class LaunchClamp**

Obtained by calling *Part.launch_clamp*.

part

The part object for this launch clamp.

Attribute Read-only, cannot be set

Return type *Part*

release()

Releases the docking clamp. Has no effect if the clamp has already been released.

Light**class Light**

Obtained by calling *Part.light*.

part

The part object for this light.

Attribute Read-only, cannot be set

Return type *Part*

active

Whether the light is switched on.

Attribute Can be read or written

Return type bool

power_usage

The current power usage, in units of charge per second.

Attribute Read-only, cannot be set

Return type float

Parachute**class Parachute**

Obtained by calling *Part.parachute*.

part

The part object for this parachute.

Attribute Read-only, cannot be set

Return type *Part*

deploy()

Deploys the parachute. This has no effect if the parachute has already been deployed.

deployed

Whether the parachute has been deployed.

Attribute Read-only, cannot be set

Return type bool

state

The current state of the parachute.

Attribute Read-only, cannot be set

Return type *ParachuteState*

deploy_altitude

The altitude at which the parachute will full deploy, in meters.

Attribute Can be read or written

Return type float

deploy_min_pressure

The minimum pressure at which the parachute will semi-deploy, in atmospheres.

Attribute Can be read or written

Return type float

class ParachuteState

See *Parachute.state*.

stowed

The parachute is safely tucked away inside its housing.

active

The parachute is still stowed, but ready to semi-deploy.

semi_deployed

The parachute has been deployed and is providing some drag, but is not fully deployed yet.

deployed

The parachute is fully deployed.

cut

The parachute has been cut.

Radiator**class Radiator**

Obtained by calling *Part.radiator*.

part

The part object for this radiator.

Attribute Read-only, cannot be set

Return type *Part*

deployable

Whether the radiator is deployable.

Attribute Read-only, cannot be set

Return type *bool*

deployed

For a deployable radiator, *True* if the radiator is extended. If the radiator is not deployable, this is always *True*.

Attribute Can be read or written

Return type *bool*

state

The current state of the radiator.

Attribute Read-only, cannot be set

Return type *RadiatorState*

Note: A fixed radiator is always *RadiatorState.extended*.

class RadiatorState

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

Obtained by calling *Part.resource_converter*.

part

The part object for this converter.

Attribute Read-only, cannot be set

Return type *Part*

count

The number of converters in the part.

Attribute Read-only, cannot be set

Return type int

name (*index*)

The name of the specified converter.

Parameters **index** (*int*) – Index of the converter.

Return type str

active (*index*)

True if the specified converter is active.

Parameters **index** (*int*) – Index of the converter.

Return type bool

start (*index*)

Start the specified converter.

Parameters **index** (*int*) – Index of the converter.

stop (*index*)

Stop the specified converter.

Parameters **index** (*int*) – Index of the converter.

state (*index*)

The state of the specified converter.

Parameters **index** (*int*) – Index of the converter.

Return type *ResourceConverterState*

status_info (*index*)

Status information for the specified converter. This is the full status message shown in the in-game UI.

Parameters **index** (*int*) – Index of the converter.

Return type str

inputs (*index*)

List of the names of resources consumed by the specified converter.

Parameters **index** (*int*) – Index of the converter.

Return type list of str

outputs (*index*)

List of the names of resources produced by the specified converter.

Parameters **index** (*int*) – Index of the converter.

Return type list of str

class **ResourceConverterState**

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**

Obtained by calling `Part.resource_harvester`.

part

The part object for this harvester.

Attribute Read-only, cannot be set

Return type `Part`

state

The state of the harvester.

Attribute Read-only, cannot be set

Return type `ResourceHarvesterState`

deployed

Whether the harvester is deployed.

Attribute Can be read or written

Return type `bool`

active

Whether the harvester is actively drilling.

Attribute Can be read or written

Return type `bool`

extraction_rate

The rate at which the drill is extracting ore, in units per second.

Attribute Read-only, cannot be set

Return type `float`

thermal_efficiency

The thermal efficiency of the drill, as a percentage of its maximum.

Attribute Read-only, cannot be set

Return type `float`

core_temperature

The core temperature of the drill, in Kelvin.

Attribute Read-only, cannot be set

Return type `float`

optimum_core_temperature

The core temperature at which the drill will operate with peak efficiency, in Kelvin.

Attribute Read-only, cannot be set

Return type float

class ResourceHarvesterState

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**

Obtained by calling *Part.reaction_wheel*.

part

The part object for this reaction wheel.

Attribute Read-only, cannot be set

Return type *Part*

active

Whether the reaction wheel is active.

Attribute Can be read or written

Return type bool

broken

Whether the reaction wheel is broken.

Attribute Read-only, cannot be set

Return type bool

pitch_torque

The torque in the pitch axis, in Newton meters.

Attribute Read-only, cannot be set

Return type float

yaw_torque

The torque in the yaw axis, in Newton meters.

Attribute Read-only, cannot be set

Return type float

roll_torque

The torque in the roll axis, in Newton meters.

Attribute Read-only, cannot be set

Return type float

Sensor**class Sensor**

Obtained by calling *Part.sensor*.

part

The part object for this sensor.

Attribute Read-only, cannot be set

Return type *Part*

active

Whether the sensor is active.

Attribute Can be read or written

Return type bool

value

The current value of the sensor.

Attribute Read-only, cannot be set

Return type str

power_usage

The current power usage of the sensor, in units of charge per second.

Attribute Read-only, cannot be set

Return type float

Solar Panel**class SolarPanel**

Obtained by calling *Part.solar_panel*.

part

The part object for this solar panel.

Attribute Read-only, cannot be set

Return type *Part*

deployed

Whether the solar panel is extended.

Attribute Can be read or written

Return type bool

state

The current state of the solar panel.

Attribute Read-only, cannot be set

Return type *SolarPanelState*

energy_flow

The current amount of energy being generated by the solar panel, in units of charge per second.

Attribute Read-only, cannot be set

Return type float

sun_exposure

The current amount of sunlight that is incident on the solar panel, as a percentage. A value between 0 and 1.

Attribute Read-only, cannot be set

Return type float

class SolarPanelState

See *SolarPanel.state*.

extended

Solar panel is fully extended.

retracted

Solar panel is fully retracted.

extending

Solar panel is being extended.

retracting

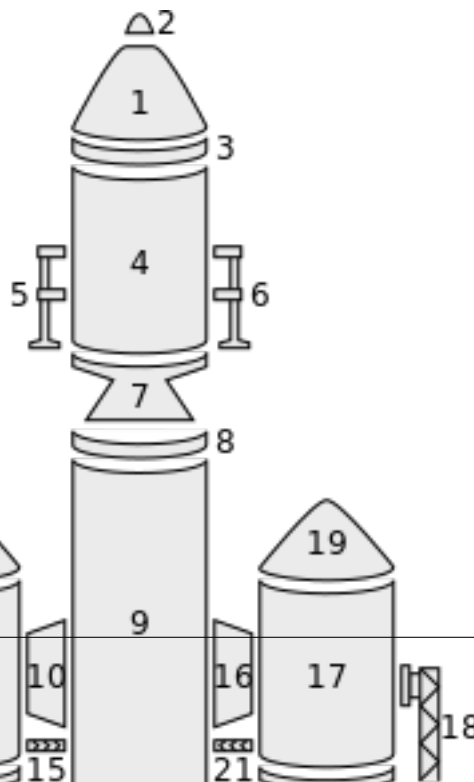
Solar panel is being retracted.

broken

Solar panel is broken.

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 *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 `None`.

The following Python example uses these attributes to perform a depth-first traversal over all of the parts in a vessel:

```
root = vessel.parts.root
stack = [(root, 0)]
while len(stack) > 0:
    part, depth = stack.pop()
    print(' '*depth, part.title)
    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:

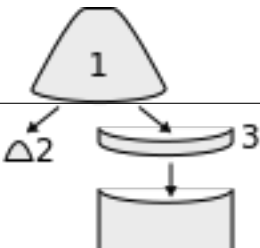
```
Command Pod Mk1
TR-18A Stack Decoupler
FL-T400 Fuel Tank
LV-909 Liquid Fuel Engine
TR-18A Stack Decoupler
FL-T800 Fuel Tank
LV-909 Liquid Fuel Engine
TT-70 Radial Decoupler
FL-T400 Fuel Tank
    TT18-A Launch Stability Enhancer
    FTX-2 External Fuel Duct
    LV-909 Liquid Fuel Engine
    Aerodynamic Nose Cone
TT-70 Radial Decoupler
FL-T400 Fuel Tank
    TT18-A Launch Stability Enhancer
    FTX-2 External Fuel Duct
    LV-909 Liquid Fuel Engine
    Aerodynamic Nose Cone
LT-1 Landing Struts
LT-1 Landing Struts
Mk16 Parachute
```

Attachment Modes

Parts can be attached to other parts either *radially* (on the side of the parent part) or *axially* (on the end of the parent part, to form a stack).

For example, in the vessel pictured above, the parachute (part 2) is *axially* connected to its parent (the command pod – part 1), and the landing leg (part 5) is *radially* connected to its parent (the fuel tank – part 4).

The root part of a vessel (for example the command pod – part 1) does not have a parent part,



so does not have an attachment mode. However, the part is 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:

```
root = vessel.parts.root
stack = [(root, 0)]
while len(stack) > 0:
    part, depth = stack.pop()
    if part.axially_attached:
        attach_mode = 'axial'
    else: # radially_attached
        attach_mode = 'radial'
    print(' '*depth, part.title, '-', attach_mode)
    for child in part.children:
        stack.append((child, depth+1))
```

When this code is execute using the craft file for the example vessel pictured above, the following is printed out:

```
Command Pod Mk1 - axial
TR-18A Stack Decoupler - axial
FL-T400 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TR-18A Stack Decoupler - axial
FL-T800 Fuel Tank - axial
LV-909 Liquid Fuel Engine - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
TT-70 Radial Decoupler - radial
FL-T400 Fuel Tank - radial
TT18-A Launch Stability Enhancer - radial
FTX-2 External Fuel Duct - radial
LV-909 Liquid Fuel Engine - axial
Aerodynamic Nose Cone - axial
LT-1 Landing Struts - radial
LT-1 Landing Struts - radial
Mk16 Parachute - axial
```


Fuel Lines

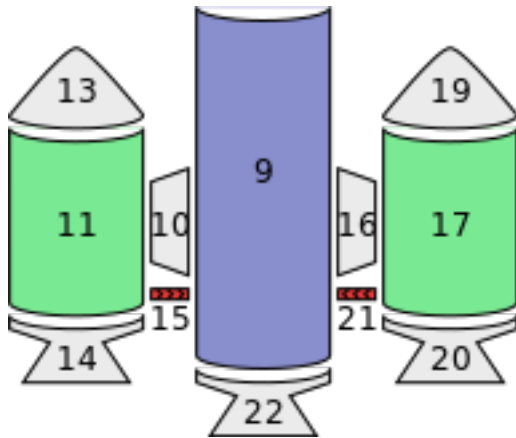


Fig. 7.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

Fuel lines are considered parts, and are included in the parts tree (for example, as pictured in Figure 4). However, the parts tree does not contain information about which parts fuel lines connect to. The parent part of a fuel line is the part from which it will take fuel (as shown in Figure 4) however the part that it will send fuel to is not represented in the parts tree.

Figure 5 shows the fuel lines from the example vessel pictured earlier. Fuel line part 15 (in red) takes fuel from a fuel tank (part 11 – in green) and feeds it into another fuel tank (part 9 – in blue). The fuel line is therefore a child of part 11, but its connection to part 9 is not represented in the tree.

The attributes `SpaceCenter.Part.fuel_lines_from` and `SpaceCenter.Part.fuel_lines_to` can be used to discover these connections. In the example in Figure 5, when `SpaceCenter.Part.fuel_lines_to` is called on fuel tank part 11, it will return a list of parts containing just fuel tank part 9 (the blue part). When `SpaceCenter.Part.fuel_lines_from` is called on fuel tank part 9, it will return a list containing fuel tank parts 11 and 17 (the parts colored green).

Staging

Each part has two staging numbers associated with it: the stage in which the part is *activated* and the stage in which the part is *decoupled*. These values can be obtained using `SpaceCenter.Part.stage` and `SpaceCenter.Part.decouple_stage` respectively. For parts that are not activated by staging, `SpaceCenter.Part.stage` returns -1. For parts that are never decoupled, `SpaceCenter.Part.decouple_stage` returns a value of -1.

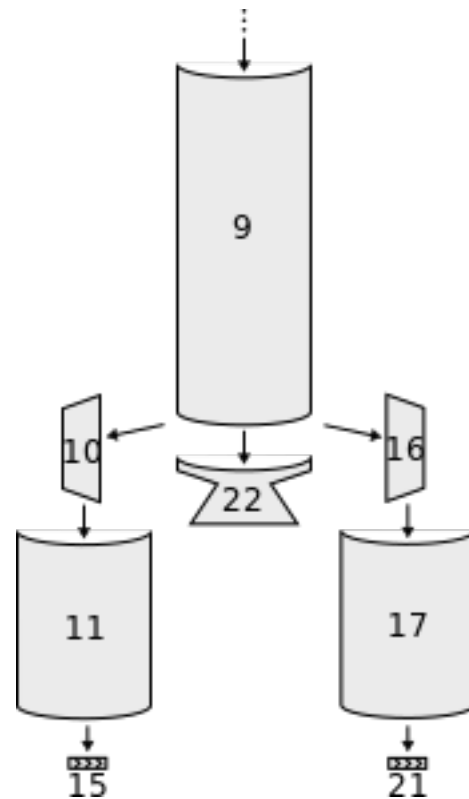


Fig. 7.13: **Figure 4** – A subset of the parts tree from Figure 2 above.

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *activated*. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

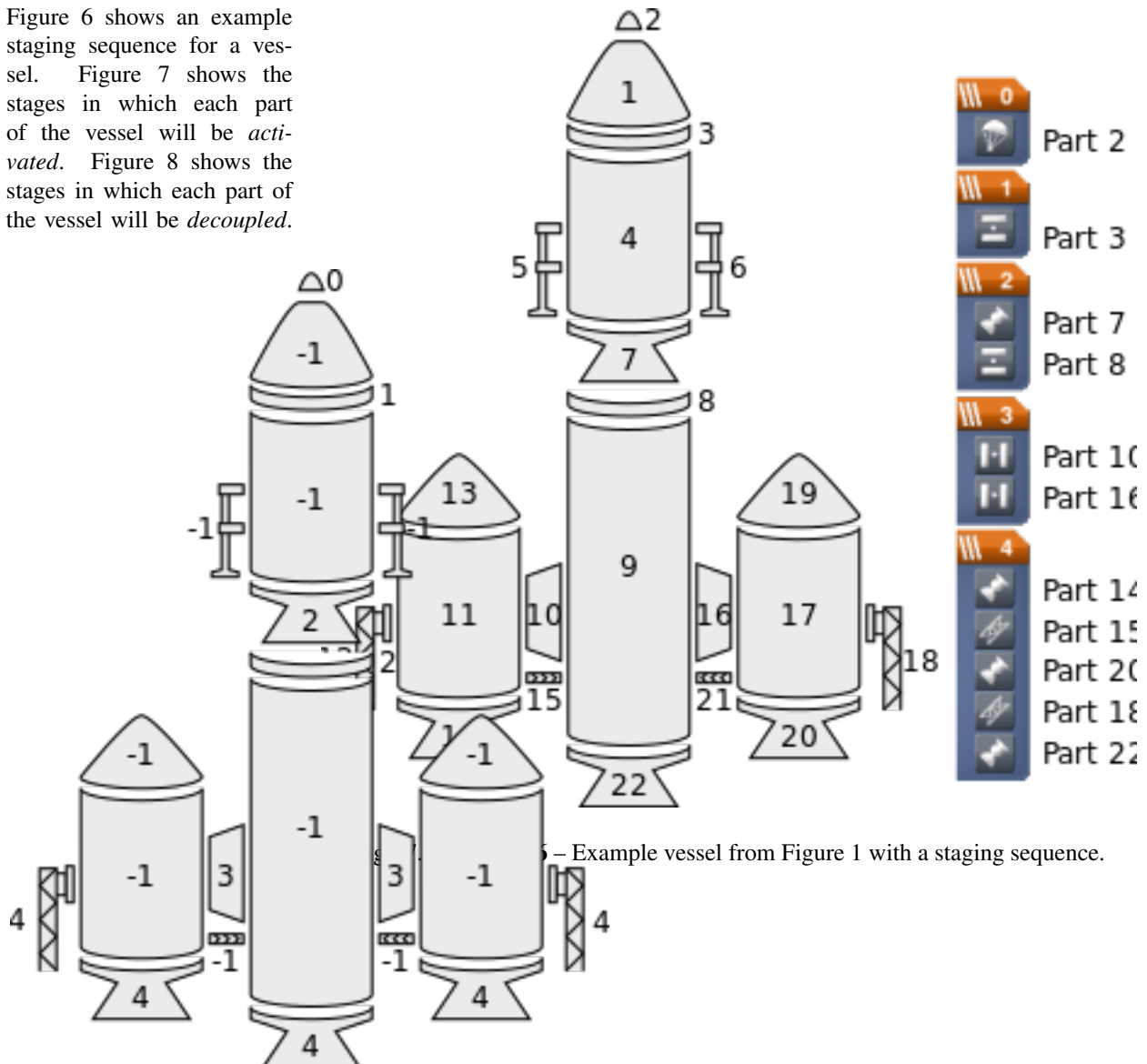


Fig. 7.15: **Figure 7** – The stage in which each part is *activated*.

7.3.8 Resources

class Resources

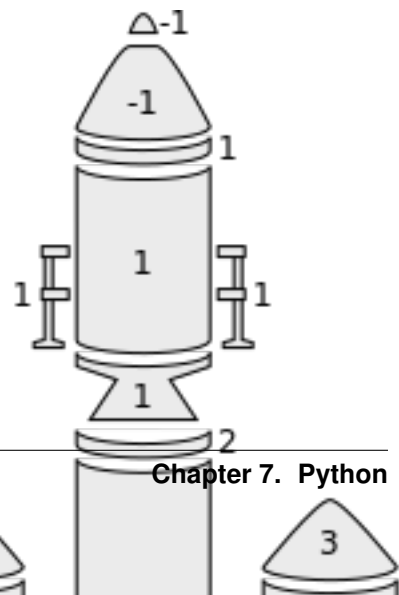
Created by calling
Vessel.resources,
Vessel.resources_in_decouple_stage()
 or *Part.resources*.

names

A list of resource names that can be stored.

Attribute Read-only, cannot be set

Return type list of str



has_resource (*name*)

Check whether the named resource can be stored.

Parameters **name** (*str*) – The name of the resource.

Return type bool

max (*name*)

Returns the amount of a resource that can be stored.

Parameters **name** (*str*) – The name of the resource.

Return type float

amount (*name*)

Returns the amount of a resource that is currently stored.

Parameters **name** (*str*) – The name of the resource.

Return type float

static density (*name*)

Returns the density of a resource, in kg/l.

Parameters **name** (*str*) – The name of the resource.

Return type float

static flow_mode (*name*)

Returns the flow mode of a resource.

Parameters **name** (*str*) – The name of the resource.

Return type *ResourceFlowMode*

class ResourceFlowMode

See *Resources.flow_mode()*.

vessel

The resource flows to any part in the vessel. For example, electric charge.

stage

The resource flows from parts in the first stage, followed by the second, and so on. For example, mono-propellant.

adjacent

The resource flows between adjacent parts within the vessel. For example, liquid fuel or oxidizer.

none

The resource does not flow. For example, solid fuel.

7.3.9 Node

class Node

Represents a maneuver node. Can be created using *Control.add_node()*.

prograde

The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

Attribute Can be read or written

Return type float

normal

The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

Attribute Can be read or written

Return type float

radial

The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

Attribute Can be read or written

Return type float

delta_v

The delta-v of the maneuver node, in meters per second.

Attribute Can be read or written

Return type float

Note: Does not change when executing the maneuver node. See [Node.remaining_delta_v](#).

remaining_delta_v

Gets the remaining delta-v of the maneuver node, in meters per second. Changes as the node is executed. This is equivalent to the delta-v reported in-game.

Attribute Read-only, cannot be set

Return type float

burn_vector ([*reference_frame* = None])

Returns a vector whose direction is the direction of the maneuver node burn, and whose magnitude is the delta-v of the burn in m/s.

Parameters **reference_frame**

([ReferenceFrame](#)) –

Return type tuple of (float, float, float)

Note: Does not change when executing the maneuver node. See [Node.remaining_burn_vector\(\)](#).

remaining_burn_vector ([*reference_frame* = None])

Returns a vector whose direction is the direction of

the maneuver node burn, and whose magnitude is the delta-v of the burn in m/s. The direction and magnitude change as the burn is executed.

Parameters `reference_frame`

(`ReferenceFrame`) –

Return type tuple of (float, float, float)

ut

The universal time at which the maneuver will occur, in seconds.

Attribute Can be read or written

Return type float

time_to

The time until the maneuver node will be encountered, in seconds.

Attribute Read-only, cannot be set

Return type float

orbit

The orbit that results from executing the maneuver node.

Attribute Read-only, cannot be set

Return type `Orbit`

remove()

Removes the maneuver node.

reference_frame

Gets the reference frame that is fixed relative to the maneuver node's burn.

- The origin is at the position of the maneuver node.
- The y-axis points in the direction of the burn.
- The x-axis and z-axis point in arbitrary but fixed directions.

Attribute Read-only, cannot be set

Return type `ReferenceFrame`

orbital_reference_frame

Gets the reference frame that is fixed relative to the maneuver node, and orientated with the orbital prograde/normal/radial directions of the original orbit at the maneuver node's position.

- The origin is at the position of the maneuver node.
- The x-axis points in the orbital anti-radial direction of the original orbit, at the position of the maneuver node.

- The y-axis points in the orbital prograde direction of the original orbit, at the position of the maneuver node.
- The z-axis points in the orbital normal direction of the original orbit, at the position of the maneuver node.

Attribute Read-only, cannot be set

Return type *ReferenceFrame*

position (*reference_frame*)

Returns the position vector of the maneuver node in the given reference frame.

Parameters **reference_frame**

(*ReferenceFrame*) –

Return type tuple of (float, float, float)

direction (*reference_frame*)

Returns the unit direction vector of the maneuver nodes burn in the given reference frame.

Parameters **reference_frame**

(*ReferenceFrame*) –

Return type tuple of (float, float, float)

7.3.10 Comms

class Comms

Used to interact with RemoteTech. Created using a call to *Vessel.comms*.

Note: This class requires *RemoteTech* to be installed.

has_local_control

Whether the vessel can be controlled locally.

Attribute Read-only, cannot be set

Return type bool

has_flight_computer

Whether the vessel has a RemoteTech flight computer on board.

Attribute Read-only, cannot be set

Return type bool

has_connection

Whether the vessel can receive commands from the KSC or a command station.

Attribute Read-only, cannot be set

Return type bool

has_connection_to_ground_station

Whether the vessel can transmit science data to a ground station.

Attribute Read-only, cannot be set

Return type bool

signal_delay

The signal delay when sending commands to the vessel, in seconds.

Attribute Read-only, cannot be set

Return type float

signal_delay_to_ground_station

The signal delay between the vessel and the closest ground station, in seconds.

Attribute Read-only, cannot be set

Return type float

signal_delay_to_vessel (*other*)

Returns the signal delay between the current vessel and another vessel, in seconds.

Parameters **other** (*Vessel*) –

Return type float

7.3.11 ReferenceFrame

class ReferenceFrame

Represents a reference frame for positions, rotations and velocities. Contains:

- The position of the origin.
- The directions of the x, y and z axes.
- The linear velocity of the frame.
- The angular velocity of the frame.

Note: This class does not contain any properties or methods. It is only used as a parameter to other functions.

7.3.12 AutoPilot

class AutoPilot

Provides basic auto-piloting utilities for a vessel. Created by calling *Vessel.auto_pilot*.

engage ()

Engage the auto-pilot.

disengage ()

Disengage the auto-pilot.

wait ()

Blocks until the vessel is pointing in the target direction (if set) and has the target roll (if set).

error

The error, in degrees, between the direction the ship has been asked to point in and the direction it is pointing in. Returns zero if the auto-pilot has not been engaged, SAS is not enabled, SAS is in stability assist mode, or no target direction is set.

Attribute Read-only, cannot be set

Return type float

roll_error

The error, in degrees, between the roll the ship has been asked to be in and the actual roll. Returns zero if the auto-pilot has not been engaged or no target roll is set.

Attribute Read-only, cannot be set

Return type float

reference_frame

The reference frame for the target direction (*AutoPilot.target_direction*).

Attribute Can be read or written

Return type *ReferenceFrame*

target_direction

The target direction. None if no target direction is set.

Attribute Can be read or written

Return type tuple of (float, float, float)

target_pitch_and_heading (pitch, heading)

Set (*AutoPilot.target_direction*) from a pitch and heading angle.

Parameters

- **pitch** (*float*) – Target pitch angle, in degrees between -90° and +90°.
- **heading** (*float*) – Target heading angle, in degrees between 0° and 360°.

target_roll

The target roll, in degrees. NaN if no target roll is set.

Attribute Can be read or written

Return type float

sas

The state of SAS.

Attribute Can be read or written

Return type bool

Note: Equivalent to `Control.sas`

sas_mode

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

Attribute Can be read or written

Return type `SASMode`

Note: Equivalent to `Control.sas_mode`

rotation_speed_multiplier

Target rotation speed multiplier. Defaults to 1.

Attribute Can be read or written

Return type float

max_rotation_speed

Maximum target rotation speed. Defaults to 1.

Attribute Can be read or written

Return type float

roll_speed_multiplier

Target roll speed multiplier. Defaults to 1.

Attribute Can be read or written

Return type float

max_roll_speed

Maximum target roll speed. Defaults to 1.

Attribute Can be read or written

Return type float

set_pid_parameters (`[kp = 1.0][, ki = 0.0][, kd = 0.0]`)

Sets the gains for the rotation rate PID controller.

Parameters

- **kp** (`float`) – Proportional gain.
- **ki** (`float`) – Integral gain.
- **kd** (`float`) – Derivative gain.

7.3.13 Geometry Types

class **Vector3**

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])
```

class **Quaternion**

Quaternions (rotations in 3-dimensional space) are encoded as a 4-tuple containing the x, y, z and w components. For example:

```
import krpc
conn = krpc.connect()
q = conn.space_center.active_vessel.flight().rotation
print(q[0], q[1], q[2], q[3])
```

7.4 InfernalRobotics API

Provides RPCs to interact with the [InfernalRobotics](#) mod. Provides the following classes:

7.4.1 InfernalRobotics

This service provides functionality to interact with the [InfernalRobotics](#) mod.

servo_groups

A list of all the servo groups in the active vessel.

Attribute Read-only, cannot be set

Return type list of [ControlGroup](#)

static servo_group_with_name (*name*)

Returns the servo group with the given *name* or *None* if none exists. If multiple servo groups have the same name, only one of them is returned.

Parameters **name** (*str*) – Name of servo group to find.

Return type [ControlGroup](#)

static servo_with_name (*name*)

Returns the servo with the given *name*, from all servo groups, or *None* if none exists. If multiple servos have the same name, only one of them is returned.

Parameters **name** (*str*) – Name of the servo to find.

Return type [Servo](#)

7.4.2 ControlGroup

class ControlGroup

A group of servos, obtained by calling `servo_groups` or `servo_group_with_name()`. Represents the “Servo Groups” in the InfernalRobotics UI.

name

The name of the group.

Attribute Can be read or written

Return type str

forward_key

The key assigned to be the “forward” key for the group.

Attribute Can be read or written

Return type str

reverse_key

The key assigned to be the “reverse” key for the group.

Attribute Can be read or written

Return type str

speed

The speed multiplier for the group.

Attribute Can be read or written

Return type float

expanded

Whether the group is expanded in the InfernalRobotics UI.

Attribute Can be read or written

Return type bool

servos

The servos that are in the group.

Attribute Read-only, cannot be set

Return type list of *Servo*

servo_with_name(name)

Returns the servo with the given *name* from this group, or *None* if none exists.

Parameters *name* (*str*) – Name of servo to find.

Return type *Servo*

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.

7.4.3 Servo

class Servo
Represents a servo. Obtained using `ControlGroup.servos`, `ControlGroup.servo_with_name()` or `servo_with_name()`.

name
The name of the servo.

Attribute Can be read or written

Return type str

highlight
Whether the servo should be highlighted in-game.

Attribute Write-only, cannot be read

Return type bool

position
The position of the servo.

Attribute Read-only, cannot be set

Return type float

min_config_position
The minimum position of the servo, specified by the part configuration.

Attribute Read-only, cannot be set

Return type float

max_config_position
The maximum position of the servo, specified by the part configuration.

Attribute Read-only, cannot be set

Return type float

min_position

The minimum position of the servo, specified by the in-game tweak menu.

Attribute Can be read or written

Return type float

max_position

The maximum position of the servo, specified by the in-game tweak menu.

Attribute Can be read or written

Return type float

config_speed

The speed multiplier of the servo, specified by the part configuration.

Attribute Read-only, cannot be set

Return type float

speed

The speed multiplier of the servo, specified by the in-game tweak menu.

Attribute Can be read or written

Return type float

current_speed

The current speed at which the servo is moving.

Attribute Can be read or written

Return type float

acceleration

The current speed multiplier set in the UI.

Attribute Can be read or written

Return type float

is_moving

Whether the servo is moving.

Attribute Read-only, cannot be set

Return type bool

is_free_moving

Whether the servo is freely moving.

Attribute Read-only, cannot be set

Return type bool

is_locked

Whether the servo is locked.

Attribute Can be read or written

Return type bool

is_axis_inverted

Whether the servos axis is inverted.

Attribute Can be read or written

Return type bool

move_right()

Moves the servo to the right.

move_left()

Moves the servo to the left.

move_center()

Moves the servo to the center.

move_next_preset()

Moves the servo to the next preset.

move_prev_preset()

Moves the servo to the previous preset.

move_to(position, speed)

Moves the servo to *position* and sets the speed multiplier to *speed*.

Parameters

- **position** (*float*) – The position to move the servo to.
- **speed** (*float*) – Speed multiplier for the movement.

stop()

Stops the servo.

7.4.4 Example

The following example gets the control group named “MyGroup”, prints out the names and positions of all of the servos in the group, then moves all of the servos to the right for 1 second.

```
import krpc, time
conn = krpc.connect(name='InfernalRobotics Example')

group = conn.infernal_robotics.servo_group_with_name('MyGroup')
if group is None:
    print('Group not found')
    exit(1)

for servo in group.servos:
    print(servo.name, servo.position)

group.move_right()
time.sleep(1)
group.stop()
```

7.5 Kerbal Alarm Clock API

Provides RPCs to interact with the `Kerbal Alarm Clock` mod. Provides the following classes:

7.5.1 KerbalAlarmClock

This service provides functionality to interact with the `Kerbal Alarm Clock` mod.

alarms

A list of all the alarms.

Attribute Read-only, cannot be set

Return type list of `Alarm`

static alarm_with_name (*name*)

Get the alarm with the given *name*, or `None` if no alarms have that name. If more than one alarm has the name, only returns one of them.

Parameters **name** (*str*) – Name of the alarm to search for.

Return type `Alarm`

static alarms_with_type (*type*)

Get a list of alarms of the specified *type*.

Parameters **type** (`AlarmType`) – Type of alarm to return.

Return type list of `Alarm`

static create_alarm (*type*, *name*, *ut*)

Create a new alarm and return it.

Parameters

- **type** (`AlarmType`) – Type of the new alarm.
- **name** (*str*) – Name of the new alarm.
- **ut** (*float*) – Time at which the new alarm should trigger.

Return type `Alarm`

7.5.2 Alarm

class Alarm

Represents an alarm. Obtained by calling `alarms`, `alarm_with_name()` or `alarms_with_type()`.

action

The action that the alarm triggers.

Attribute Can be read or written

Return type *AlarmAction*

margin

The number of seconds before the event that the alarm will fire.

Attribute Can be read or written

Return type float

time

The time at which the alarm will fire.

Attribute Can be read or written

Return type float

type

The type of the alarm.

Attribute Read-only, cannot be set

Return type *AlarmType*

id

The unique identifier for the alarm.

Attribute Read-only, cannot be set

Return type str

name

The short name of the alarm.

Attribute Can be read or written

Return type str

notes

The long description of the alarm.

Attribute Can be read or written

Return type str

remaining

The number of seconds until the alarm will fire.

Attribute Read-only, cannot be set

Return type float

repeat

Whether the alarm will be repeated after it has fired.

Attribute Can be read or written

Return type bool

repeat_period

The time delay to automatically create an alarm after it has fired.

Attribute Can be read or written

Return type float

vessel

The vessel that the alarm is attached to.

Attribute Can be read or written

Return type *SpaceCenter.Vessel*

xfer_origin_body

The celestial body the vessel is departing from.

Attribute Can be read or written

Return type *SpaceCenter.CelestialBody*

xfer_target_body

The celestial body the vessel is arriving at.

Attribute Can be read or written

Return type *SpaceCenter.CelestialBody*

remove ()

Removes the alarm.

7.5.3 AlarmType

class AlarmType

The type of an alarm.

raw

An alarm for a specific date/time or a specific period in the future.

maneuver

An alarm based on the next maneuver node on the current ships flight path. This node will be stored and can be restored when you come back to the ship.

maneuver_auto

See *AlarmType.maneuver*.

apoapsis

An alarm for furthest part of the orbit from the planet.

periapsis

An alarm for nearest part of the orbit from the planet.

ascending_node

Ascending node for the targeted object, or equatorial ascending node.

descending_node

Descending node for the targeted object, or equatorial descending node.

closest

An alarm based on the closest approach of this vessel to the targeted vessel, some number of orbits into the future.

contract

An alarm based on the expiry or deadline of contracts in career modes.

contract_auto

See *AlarmType.contract*.

crew

An alarm that is attached to a crew member.

distance

An alarm that is triggered when a selected target comes within a chosen distance.

earth_time

An alarm based on the time in the “Earth” alternative Universe (aka the Real World).

launch_rendevous

An alarm that fires as your landed craft passes under the orbit of your target.

soi_change

An alarm manually based on when the next SOI point is on the flight path or set to continually monitor the active flight path and add alarms as it detects SOI changes.

soi_change_auto

See *AlarmType.soi_change*.

transfer

An alarm based on Interplanetary Transfer Phase Angles, i.e. when should I launch to planet X? Based on Kosmo Not’s post and used in Olex’s Calculator.

transfer_modelled

See *AlarmType.transfer*.

7.5.4 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.

7.5.5 Example

The following example creates a new alarm for the active vessel. The alarm is set to trigger after 10 seconds have passed, and display a message.

```
import krpc
conn = krpc.connect(name='Kerbal Alarm Clock Example')

alarm = conn.kerbal_alarm_clock.create_alarm(
    conn.kerbal_alarm_clock.AlarmType.raw,
    'My New Alarm',
    conn.space_center.ut+10)

alarm.notes = '10 seconds have now passed since the alarm was created.'
alarm.action = conn.kerbal_alarm_clock.AlarmAction.message_only
```


OTHER CLIENTS, SERVICES AND SCRIPTS

This page links to clients, services, scripts, tools and other useful things for kRPC that have been made by others. If you want your own project added to this page, please feel free to ask [on the forum](#).

8.1 Clients

- [Ruby client](#) by TeWu

8.2 Services

- [krpcmj](#) – remote procedures to interact with MechJeb

8.3 Scripts/Tools/Libraries etc.

- [kautopilly](#) – an autopilot primarily intended for planes.
- [KNav](#) – a flexible platform for implementing computer-assisted navigation and control of vessels.
- [wernher](#) – a toolkit for flight control and orbit analysis.
- [A small logging script](#).

COMPILING KRPC

kRPC uses the [Bazel build system](#).

9.1 Install Dependencies

Bazel automatically downloads most of the required dependencies to build kRPC. However the following will need to be installed on your system:

- [Mono C# compiler and runtime](#)
- Python, including virtualenv and pip
- pdflatex for building the documentation
- RSVG for converting SVGs to PNGs
- libxml, libxslt and Python headers, for building the Java documentation

To install the latest C# compiler and runtime on Ubuntu, follow the instructions on [Mono's website](#). The other dependencies can be installed via apt:

```
$ sudo apt-get install mono-complete \
python-virtualenv python-pip \
texlive-latex-base texlive-latex-recommended \
texlive-fonts-recommended texlive-latex-extra \
librsvg2-bin libxml2-dev libxslt1-dev python-dev
```

9.2 Setup 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 at `/path/to/krpc` you can create a symlink using `ln -s /path/to/ksp /path/to/krpc/lib/ksp`

You may also need to modify the symlink at `lib/mono-4.5` to point to the correct location of your Mono installation.

9.3 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 specific other targets that may be useful. Build them 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/InfernalRobotics`
 - `//service/KerbalAlarmClock`
- 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` – build the documentation and serve it from `http://localhost:8080`
- `tools/install.sh` – build the plugin and the testing tools, and install them into the GameData directory of the copy of KSP found at `lib/ksp`.

9.4 Building the C# projects using an IDE

A C# solution file (`kRPC.sln`) is provided in the root of the project for use with MonoDevelop or a similar C# IDE.

Some of the C# source files it references are generated by the Bazel build scripts. You need to run `bazel build //:csproj` to generate these files before the solution can be built.

Alternatively, if you are unable to run Bazel to build these files, you can [download them from GitHub](#). Simply extract this archive over your copy of the source and you are good to go.

9.4.1 Running the Tests

kRPC contains a suite of tests for the server plugin, services, client libraries and others.

The tests, which do not require KSP to be running, can be executed using: `bazel test //:test`. Bazel will automatically download most of the required dependencies to run the tests, however you will also need to install Lua and LuaRocks on your system. On Ubuntu, these can be installed using: `sudo apt-get install lua5.1 luarocks`

Note: You need to install luarocks version 2.2.0 or higher. On older versions of Ubuntu, the version in the apt repositories is too old and luarocks will need to be installed via other means.

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 connected enabled). Then install the krpc python client, and run the scripts found in `service/SpaceCenter/test`. These tests will automatically load a save game called `test`, launch a vessel and run various tests on it.

EXTENDING KRPC

10.1 The kRPC Architecture

kRPC consists of two components: a server and a client. The server plugin (provided by `KRPC.dll`) runs inside KSP. It provides a collection of *procedures* that clients can run. These procedures are arranged in groups called *services* to keep things organized. It also provides an in-game user interface that can be used to start/stop the server, change settings and monitor active clients.

Clients run outside of KSP. This gives you the freedom to run scripts in whatever environment you want. A client communicates with the server to run procedures using a *communication protocol*. kRPC comes with several client libraries that implement this communication protocol, making it easier to write programs in these languages.

kRPC comes with a collection of standard functionality for interacting with vessels, contained in a service called `SpaceCenter`. This service provides procedures for things like getting flight/orbital data and controlling the active vessel. This service is provided by `KRPC.SpaceCenter.dll`.

10.2 Service API

Third party mods can add functionality to kRPC using the *Service API*. This is done by adding *attributes* to your own classes, methods and properties to make them visible through the server. When the kRPC server starts, it scans all the assemblies loaded by the game, looking for classes, methods and properties with these attributes.

The following example implements a service that can control the throttle and staging of the active vessel. To add this to the server, compile the code and place the DLL in your GameData directory.

```
using UnityEngine;
using KRPC.Service;
using KRPC.Service.Attributes;

namespace LaunchControl {

    /// <summary>
    /// Service for staging vessels and controlling their throttle.
    /// </summary>
    [KRPCService (GameScene = GameScene.Flight)]
    public static class LaunchControl {

        /// <summary>
        /// The current throttle setting for the active vessel, between 0 and 1.
        /// </summary>
        [KRPCProperty]
        public static float Throttle {
            get { return FlightInputHandler.state.mainThrottle; }
        }
    }
}
```

```
        set { FlightInputHandler.state.mainThrottle = value; }
    }

    /// <summary>
    /// Activate the next stage in the vessel.
    /// </summary>
    [KRPCProcedure]
    public static void ActivateStage ()
    {
        Staging.ActivateNextStage ();
    }
}
```

The following example shows how this service can then be used from a python client:

```
import krpc
conn = krpc.connect()
conn.launch_control.throttle = 1
conn.launch_control.activate_stage()
```

Some of the client libraries automatically pick up changes to the functionality provided by the server, including the Python and Lua clients. However, some clients require code to be generated from the service assembly so that they can interact with new or changed functionality. See [clientgen](#) for details on how to generate this code.

10.2.1 Attributes

The following [C# attributes](#) can be used to add functionality to the kRPC server.

KRPCService (*string Name, KRPC.Service.GameScene GameScene*)

Parameters

- **Name** – Optional name for the service. If omitted, the service name is set to the name of the class this attribute is applied to.
- **GameScene** – The game scenes in which the services procedures are available.

This [attribute](#) is applied to a static class, to indicate that all methods, properties and classes declared within it are part of the the same service. The name of the service is set to the name of the class, or – if present – the Name parameter.

Multiple services with the same name can be declared, as long the classes, procedures and methods they contain have unique names. The classes will be merged to appear as a single service on the server.

The type to which this attribute is applied must satisfy the following criteria:

- The type must be a class.
- The class must be `public static`.
- The name of the class, or the Name parameter if specified, must be a valid [kRPC identifier](#).
- The class must not be declared within another class that has the [KRPCService](#) attribute. Nesting of services is not permitted.

Services are configured to be available in specific [game scenes](#) via the GameScene parameter. If the GameScene parameter is not specified, the service is available in any scene. If a procedure is called when the service is not available, it will throw an exception.

Examples

- Declare a service called EVA:

```
[KRPCService]
public static class EVA {
    ...
}
```

- Declare a service called MyEVAService (different to the name of the class):

```
[KRPCService (Name = "MyEVAService")]
public static class EVA {
    ...
}
```

- Declare a service called FlightTools that is only available during the Flight game scene:

```
[KRPCService (GameScene = GameScene.Flight)]
public static class FlightTools {
    ...
}
```

KRPCProcedure

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.

Example

The following defines a service called EVA with a PlantFlag procedure that takes a name and an optional description, and returns a Flag object.

```
[KRPCService]
public static class EVA {
    [KRPCProcedure]
    public static Flag PlantFlag (string name, string description = "")
    {
        ...
    }
}
```

This can be called from a python client as follows:

```
import krpc
conn = krpc.connect()
flag = conn.eva.plant_flag('Landing Site', 'One small step for Kerbal-kind')
```

KRPCClass (string Service)

Parameters

- **Service** – Optional name of the service to add this class to. If omitted, the class is added to the service that contains its definition.

This *attribute* is applied to non-static classes. It adds the class to the server, so that references to instances of the class can be passed between client and server.

A *KRPCClass* must be part of a service, just like a *KRPCProcedure*. However, it would be restrictive if the class had to be declared as a nested class inside a class with the *KRPCService* attribute. Therefore, a *KRPCClass* can be declared outside of any service if it has the *Service* parameter set to the name of the service that it is part of. Also, the service that the *Service* parameter refers to does not have to exist. If it does not exist, a service with the given name is created.

The class to which this attribute is applied must satisfy the following criteria:

- The class must be *public* and *not* *static*.
- The name of the class must be a valid *kRPC identifier*.
- The class must either be declared inside a class that is a *KRPCService*, or have its *Service* parameter set to the name of the service it is part of.

Examples

- Declare a class called *Flag* in the *EVA* service:

```
[KRPCService]
public static class EVA {
    [KRPCClass]
    public class Flag {
        ...
    }
}
```

- Declare a class called *Flag*, without nesting the class definition in a service class:

```
[KRPCClass (Service = "EVA")]
public class Flag {
    ...
}
```

KRPCMethod

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.

Example

Declare a *Remove* method in the *Flag* class:

```
[KRPCClass (Service = "EVA")]
public class Flag {
    [KRPCMethod]
    void Remove()
    {
        ...
    }
}
```

```

    }
}

```

class **KRPCProperty**

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*.

Examples

- Applied to a static property in a service:

```

[KRPCService]
public static class EVA {
    [KRPCProperty]
    public Flag LastFlag
    {
        get { ... }
    }
}

```

This property can be accessed from a python client as follows:

```

import krpc
conn = krpc.connect()
flag = conn.eva.last_flag

```

- Applied to a non-static property in a class:

```

[KRPCClass (Service = "EVA")]
public class Flag {
    [KRPCProperty]
    public void Name { get; set; }

    [KRPCProperty]
    public void Description { get; set; }
}

```

KRPCEnum (*string Service*)

Parameters

- **Service** – Optional name of the service to add this enum to. If omitted, the enum is added to the service that contains its definition.

This **attribute** is applied to enumeration types. It adds the enumeration and its permissible values to the server. This attribute works similarly to *KRPCClass*, but is applied to enumeration types.

A *KRPCEnum* must be part of a service, just like a *KRPCClass*. Similarly, a *KRPCEnum* can be declared outside of a service if it has its `Service` parameter set to the name of the service that it is part of.

The enumeration type to which this attribute is applied must satisfy the following criteria:

- The enumeration must be `public`.
- The name of the enumeration must be a valid *kRPC identifier*.
- The enumeration must either be declared inside a *KRPCService*, or have its `Service` parameter set to the name of the service it is part of.
- The underlying C# type must be an `int`.

Examples

- Declare an enumeration type with two values:

```
[KRPCEnum (Service = "EVA")]
public enum FlagState {
    Raised,
    Lowered
}
```

This can be used from a python client as follows:

```
import krpc
conn = krpc.connect()
state = conn.eva.FlagState.lowered
```

10.2.2 Identifiers

An identifier must only contain alphanumeric characters and underscores. An identifier must not start with an underscore. Identifiers should follow *CamelCase* capitalization conventions.

Note: Although underscores are permitted, they should be avoided as they are used for internal name mangling.

10.2.3 Serializable Types

A type can only be used as a parameter or return type if kRPC knows how to serialize it. The following types are serializable:

- The C# types `double`, `float`, `int`, `long`, `uint`, `ulong`, `bool`, `string` and `byte[]`
- Any type annotated with *KRPCClass*
- Any type annotated with *KRPCEnum*
- Collections of serializable types:
 - `System.Collections.Generic.IList<T>` where `T` is a serializable type
 - `System.Collections.Generic.IDictionary<K,V>` where `K` is one of `int`, `long`, `uint`, `ulong`, `bool` or `string` and `V` is a serializable type
 - `System.Collections.HashSet<V>` where `V` is a serializable type
- Return types can be `void`
- Protocol buffer message types from namespace `KRPC.Schema.KRPC`

10.2.4 Game Scenes

Each service is configured to be available from a particular game scene, or scenes.

enum `KRPC.Service.GameScene`

SpaceCenter

The game scene showing the Kerbal Space Center buildings.

Flight

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

TrackingStation

The tracking station.

EditorVAB

The Vehicle Assembly Building.

EditorSPH

The Space Plane Hangar.

Editor

Either the VAB or the SPH.

All

All game scenes.

Examples

- Declare a service that is available in the `KRPC.Service.GameScene.Flight` game scene:

```
[KRPCService (GameScene = GameScene.Flight)]
public static class MyService {
    ...
}
```

- Declare a service that is available in the `KRPC.Service.GameScene.Flight` and `KRPC.Service.GameScene.Editor` game scenes:

```
[KRPCService (GameScene = (GameScene.Flight | GameScene.Editor))]
public static class MyService {
    ...
}
```

10.3 Documentation

Documentation can be added using [C# XML documentation](#). The documentation will be automatically exported to clients when they connect.

10.4 Further Examples

See the [SpaceCenter](#) service implementation for more extensive examples.

10.5 Generating Service Code for Static Clients

Some of the client libraries dynamically construct the code necessary to interact with the server when they connect. This means that these libraries will automatically pick up changes to service code. Such client libraries include those for Python and Lua.

Other client libraries required code to be generated and compiled into them statically. They do not automatically pick up changes to service code. Such client libraries include those for C++ and C#.

Code for these ‘static’ libraries is generated using the `krpc-clientgen` tool. This is provided as part of the [krpctools python package](#). It can be installed using `pip`:

```
pip install krpctools
```

You can then run the script from the command line:

```
$ krpc-clientgen --help

usage: krpc-clientgen [-h] [-v] [-o OUTPUT] [--ksp KSP]
                    [--output-defs OUTPUT_DEFS]
                    {cpp,csharp,java} service input [input ...]

Generate client source code for kRPC services.

positional arguments:
  {cpp,csharp,java}      Language to generate
  service                Name of service to generate
  input                  Path to service definition JSON file or assembly
                        DLL(s)

optional arguments:
  -h, --help            show this help message and exit
  -v, --version          show program's version number and exit
  -o OUTPUT, --output OUTPUT
                        Path to write source code to. If not specified, writes
                        source code to standard output.
  --ksp KSP             Path to Kerbal Space Program directory. Required when
                        reading from an assembly DLL(s)
  --output-defs OUTPUT_DEFS
                        When generating client code from a DLL, output the
                        service definitions to the given JSON file
```

Client code can be generated either directly from an assembly DLL containing the service, or from a JSON file that has previously been generated from an assembly DLL (using the `--output-defs` flag).

Generating client code from an assembly DLL requires a copy of Kerbal Space Program and a C# runtime to be available on the machine. In contrast, generating client code from a JSON file does not have these requirements and so is more portable.

10.5.1 Example

The following demonstrates how to generate code for the C++ and C# clients to interact with the LaunchControl service, given in an example previously.

`krpc-clientgen` expects to be passed the location of your copy of Kerbal Space Program, the name of the language to generate, the name of the service (from the `KRPCService` attribute), a path to the assembly containing the service and the path to write the generated code to.

For C++, run the following:

```
krpc-clientgen --ksp=/path/to/ksp cpp LaunchControl LaunchControl.dll  
launch_control.hpp
```

To then use the LaunchControl service from C++, you need to link your code against the C++ client library, and include *launch_control.hpp*.

For C#, run the following:

```
krpc-clientgen --ksp=/path/to/ksp csharp LaunchControl LaunchControl.dll  
LaunchControl.cs
```

To then use the LaunchControl service from a C# client, you need to reference the *KRPC.Client.dll* and include *LaunchControl.cs* in your project.

COMMUNICATION PROTOCOL

Clients invoke Remote Procedure Calls (RPCs) by communicating with the server using [Protocol Buffer v3 messages](#) sent over a TCP/IP connection. 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](#).

The following sections describe how to communicate with kRPC using snippets of Python code. A complete example script made from these snippets can be [downloaded here](#).

11.1 Establishing a Connection

kRPC consists of two servers: an *RPC server* (over which clients send and receive RPCs) and a *stream server* (over which clients receive *Streams*). A client first connects to the *RPC Server*, then (optionally) to the *Stream Server*.

11.1.1 Connecting to the RPC Server

To establish a connection to the RPC server, a client must do the following:

1. Open a TCP socket to the server on its RPC port (which defaults to 50000).
2. Send this 12 byte hello message: 0x48 0x45 0x4C 0x4C 0x4F 0x2D 0x52 0x50 0x43 0x00 0x00 0x00
3. Send a 32 byte message containing a name for the connection, that will be displayed on the in-game server window. This should be a UTF-8 encoded string, up to a maximum of 32 bytes in length. If the string is shorter than 32 bytes, it should be padded with zeros.
4. Receive a 16 byte unique client identifier. This is sent to the client when the connection is granted, for example after the user has clicked accept on the in-game UI.

For example, this python code will connect to the RPC server at address 127.0.0.1:50000 using the identifier Jeb:

```
import socket
rpc_conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
rpc_conn.connect(('127.0.0.1', 50000))
# Send the 12 byte hello message
rpc_conn.sendall(b'\x48\x45\x4C\x4C\x4F\x2D\x52\x50\x43\x00\x00\x00')
# Send the 32 byte client name 'Jeb' padded with zeroes
name = 'Jeb'.encode('utf-8')
name += (b'\x00' * (32-len(name)))
rpc_conn.sendall(name)
# Receive the 16 byte client identifier
```

```
identifier = b''
while len(identifier) < 16:
    identifier += rpc_conn.recv(16 - len(identifier))
# Connection successful. Print out a message along with the client identifier.
import binascii
printable_identifier = binascii.hexlify(bytearray(identifier))
print('Connected to RPC server, client identifier = %s' % printable_identifier)
```

11.1.2 Connecting to the Stream Server

To establish a connection to the stream server, a client must first *connect to the RPC Server* then do the following:

1. Open a TCP socket to the server on its stream port (which defaults to 50001).
2. Send this 12 byte hello message: 0x48 0x45 0x4C 0x4C 0x4F 0x2D 0x53 0x54 0x52 0x45 0x41 0x4D
3. Send a 16 byte message containing the client's unique identifier. This identifier is given to the client after it successfully connects to the RPC server.
4. Receive a 2 byte OK message: 0x4F 0x4B This indicates a successful connection.

Note: Connecting to the Stream Server is optional. If the client doesn't require stream functionality, there is no need to connect.

For example, this python code will connect to the stream server at address 127.0.0.1:50001. Note that *identifier* is the unique client identifier received when *connecting to the RPC server*.

```
stream_conn = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
stream_conn.connect(('127.0.0.1', 50001))
# Send the 12 byte hello message
stream_conn.sendall(b'\x48\x45\x4C\x4C\x4F\x2D\x53\x54\x52\x45\x41\x4D')
# Send the 16 byte client identifier
stream_conn.sendall(identifier)
# Receive the 2 byte OK message
ok_message = b''
while len(ok_message) < 2:
    ok_message += stream_conn.recv(2 - len(ok_message))
# Connection successful
print('Connected to stream server')
```

11.2 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.

11.2.1 Invoking Remote Procedures

Remote procedures are invoked by sending a request message to the RPC server, and waiting for a response message. These messages are encoded as Protocol Buffer messages.

The request message contains the name of the procedure to invoke, and the values of any arguments to pass it. The response message contains the value returned by the procedure (if any) and any errors that were encountered.

Requests are processed in order of receipt. The next request from a client will not be processed until the previous one completes execution and its response has been received by the client. When there are multiple client connections, requests are processed in round-robin order.

11.2.2 Anatomy of a Request

A request is sent to the server using a Request Protocol Buffer message with the following format:

```
message Request {
  string service = 1;
  string procedure = 2;
  repeated Argument arguments = 3;
}

message Argument {
  uint32 position = 1;
  bytes value = 2;
}
```

The fields are:

- `service` - The name of the service in which the remote procedure is defined.
- `procedure` - The name of the remote procedure to invoke.
- `arguments` - A sequence of `Argument` messages containing the values of the procedure's arguments. The fields are:
 - `position` - The zero-indexed position of the of the argument in the procedure's signature.
 - `value` - The value of the argument, encoded in Protocol Buffer format.

The `Argument` messages have a `position` field to allow values for default arguments to be omitted. See [Protocol Buffer Encoding](#) for details on how to serialize the argument values.

11.2.3 Anatomy of a Response

A response is sent to the client using a Response Protocol Buffer message with the following format:

```
message Response {
  double time = 1;
  bool has_error = 2;
  string error = 3;
  bool has_return_value = 4;
  bytes return_value = 5;
}
```

The fields are:

- `time` - The universal time (in seconds) when the request completed processing.
- `has_error` - True if there was an error executing the remote procedure.
- `error` - If `has_error` is true, contains a description of the error.
- `has_return_value` - True if the remote procedure returned a value.
- `return_value` - If `has_return_value` is true and `has_error` is false, contains the value returned by the remote procedure, encoded in protocol buffer format.

See [Protocol Buffer Encoding](#) for details on how to unserialize the return value.

11.2.4 Encoding and Sending Requests and Responses

To send a request:

1. Encode a Request message using the *Protocol Buffer Encoding*.
2. Send the size in bytes of the encoded Request message, encoded as a Protocol Buffer varint.
3. Send the message data.

To receive a response:

1. Read a Protocol Buffer varint, which contains the length of the Response message data in bytes.
2. Receive and decode the Response message.

11.2.5 Example RPC invocation

The following Python script 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 Protocol Buffer message formats `Request`, `Response` and `Status` compiled to python code using the Protocol Buffer compiler. The `EncodeVarint` and `DecodeVarint` functions are used to encode/decode integers to/from the Protocol Buffer varint format.

```
# import the krpc.proto schema
import krpc.schema

# Utility functions to encode and decode integers to protobuf format
import google.protobuf

def EncodeVarint(value):
    data = []
    def write(x):
        data.append(x)
    google.protobuf.internal.encoder._SignedVarintEncoder()(write, value)
    return b''.join(data)

def DecodeVarint(data):
    return google.protobuf.internal.decoder._DecodeSignedVarint(data, 0)[0]

# Create Request message
request = krpc.schema.KRPC.Request()
request.service = 'KRPC'
request.procedure = 'GetStatus'

# Encode and send the request
data = request.SerializeToString()
header = EncodeVarint(len(data))
rpc_conn.sendall(header + data)

# Receive the size of the response data
data = b''
while True:
    data += rpc_conn.recv(1)
    try:
        size = DecodeVarint(data)
        break
    except IndexError:
        pass
```



```

# Receive the response data
data = b''
while len(data) < size:
    data += rpc_conn.recv(size - len(data))

# Decode the response message
response = krpc.schema.KRPC.Response()
response.ParseFromString(data)

# Check for an error response
if response.has_error:
    print('ERROR:', response.error)

# Decode the return value as a Status message
else:
    status = krpc.schema.KRPC.Status()
    assert response.has_return_value
    status.ParseFromString(response.return_value)

    # Print out the version string from the Status message
    print(status.version)

```

11.3 Protocol Buffer Encoding

Values passed as arguments or received as return values are encoded using the Protocol Buffer version 3 serialization format:

- Documentation for this encoding can be found here: <https://developers.google.com/protocol-buffers/docs/encoding>
- Protocol Buffer libraries in many languages are available here: <https://github.com/google/protobuf/releases>

11.4 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*. 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 message*. This is then sent to the client over the stream server's TCP/IP connection. If the value returned by a stream's 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.

11.4.1 Anatomy of a Stream Message

A stream message is sent to the client using a *StreamMessage* Protocol Buffer message with the following format:

```

message StreamMessage {
    repeated StreamResponse responses = 1;
}

```

This contains a list of `StreamResponse` 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 StreamResponse {  
    uint32 id = 1;  
    Response response = 2;  
}
```

The fields are:

- `id` - The identifier of the stream. This is the value returned by *AddStream* when the stream is created.
- `response` - A `Response` message containing the result of the stream's RPC. This is identical to the `Response` message returned when calling the RPC directly. See *Anatomy of a Response* for details on the format and contents of this message.

11.5 KRPC Service

The server provides a service called `KRPC` containing procedures that are used to retrieve information about the server and to manage streams.

11.5.1 GetStatus

The `GetStatus` procedure returns status information about the server. It returns a Protocol Buffer message with the format:

```
message Status {  
    string version = 1;  
    uint64 bytes_read = 2;  
    uint64 bytes_written = 3;  
    float bytes_read_rate = 4;  
    float bytes_written_rate = 5;  
    uint64 rpcs_executed = 6;  
    float rpc_rate = 7;  
    bool one_rpc_per_update = 8;  
    uint32 max_time_per_update = 9;  
    bool adaptive_rate_control = 10;  
    bool blocking_recv = 11;  
    uint32 recv_timeout = 12;  
    float time_per_rpc_update = 13;  
    float poll_time_per_rpc_update = 14;  
    float exec_time_per_rpc_update = 15;  
    uint32 stream_rpcs = 16;  
    uint64 stream_rpcs_executed = 17;  
    float stream_rpc_rate = 18;  
    float time_per_stream_update = 19;  
}
```

The `version` field contains the version string of the server. The remaining fields contain performance information about the server.

11.5.2 GetServices

The `GetServices` procedure returns a Protocol Buffer message containing information about all of the services and procedures provided by the server. It also provides type information about each procedure, in the form of *attributes*.

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*.

11.5.3 AddStream

The `AddStream` procedure adds a new stream to the server. It takes a single argument containing the RPC to invoke, encoded as a `Request` object. See *Anatomy of a Request* for the format and contents of this object. See *Streams* for more information on working with streams.

11.5.4 RemoveStream

The `RemoveStream` procedure removes a stream from the server. It takes a single argument – the identifier of the stream to be removed. This is the identifier returned when the stream was added by calling *AddStream*. See *Streams* for more information on working with streams.

11.6 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;
  string documentation = 5;
}
```

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.
- `documentation` - Documentation for the service, as *C# XML documentation*.

Note: See the *Extending kRPC* documentation for more details about *KRPCClass* and *KRPCEnum*.

11.6.1 Procedures

Details about a procedure are given by a `Procedure` message, with the format:

```
message Procedure {
    string name = 1;
    repeated Parameter parameters = 2;
    bool has_return_type = 3;
    string return_type = 4;
    repeated string attributes = 5;
    string documentation = 6;
}

message Parameter {
    string name = 1;
    string type = 2;
    bool has_default_argument = 3;
    bytes default_argument = 4;
}
```

The fields are:

- name - The name of the procedure.
- 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.
 - has_default_argument - True if the parameter has a default value.
 - default_argument - If has_default_argument is true, contains the value of the default value of the parameter, *encoded using Protocol Buffer format*.
- has_return_type - True if the procedure returns a value.
- return_type - If has_return_type is true, contains the *return type* of the procedure.
- attributes - The procedure's *attributes*.
- documentation - Documentation for the procedure, as *C# XML documentation*.

11.6.2 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*.

11.6.3 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](#).

11.6.4 Attributes

Additional type information about a procedure is encoded as a list of attributes, and included in the `Procedure` message. For example, if the procedure implements a method for a class (see [proxy objects](#)) this fact will be specified in the attributes.

The following attributes specify what the procedure implements:

- `Property.Get(property-name)`
Indicates that the procedure is a property getter (for the service) with the given `property-name`.
- `Property.Set(property-name)`
Indicates that the procedure is a property setter (for the service) with the given `property-name`.
- `Class.Method(class-name,method-name)`
Indicates that the procedure is a method for a class with the given `class-name` and `method-name`.
- `Class.StaticMethod(class-name,method-name)`
Indicates that the procedure is a static method for a class with the given `class-name` and `method-name`.
- `Class.Property.Get(class-name,property-name)`
Indicates that the procedure is a property getter for a class with the given `class-name` and `property-name`.
- `Class.Property.Set(class-name,property-name)`
Indicates that the procedure is a property setter for a class with the given `class-name` and `property-name`.

The following attributes specify more details about the return and parameter types of the procedure.

- `ReturnType.type-name`
Specifies the actual [return type](#) of the procedure, if it differs to the type specified in the `Procedure` message. For example, this is used with [proxy objects](#).

- `ParameterType(parameter-position).type-name`

Specifies the actual *parameter type* of the procedure, if it differs to the type of the corresponding parameter specified in the `Parameter` message. For example, this is used with *proxy objects*.

11.6.5 Type Names

The `GetServices` procedure returns type information about parameters and return values as strings. Type names can be any of the following:

- A Protocol Buffer value type. One of `float`, `double`, `int32`, `int64`, `uint32`, `uint64`, `bool`, `string` or `bytes`.
- A *KRPCClass* in the format `Class(ClassName)`
- A *KRPCEnum* in the format `Enum(ClassName)`
- A Protocol Buffer message type, in the format `KRPC.MessageType`. Only message types defined in `krpc.proto` are permitted.

11.6.6 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, the procedure will have the attribute `ReturnType.Class(ClassName)` where `ClassName` is the name of the class.

When a procedure takes a proxy object as a parameter, the procedure will have the attribute `ParameterType(n).Class(ClassName)` where `n` is the position of the parameter and `ClassName` is the name of the class.

INTERNALS OF KRPC

12.1 Server Performance Settings

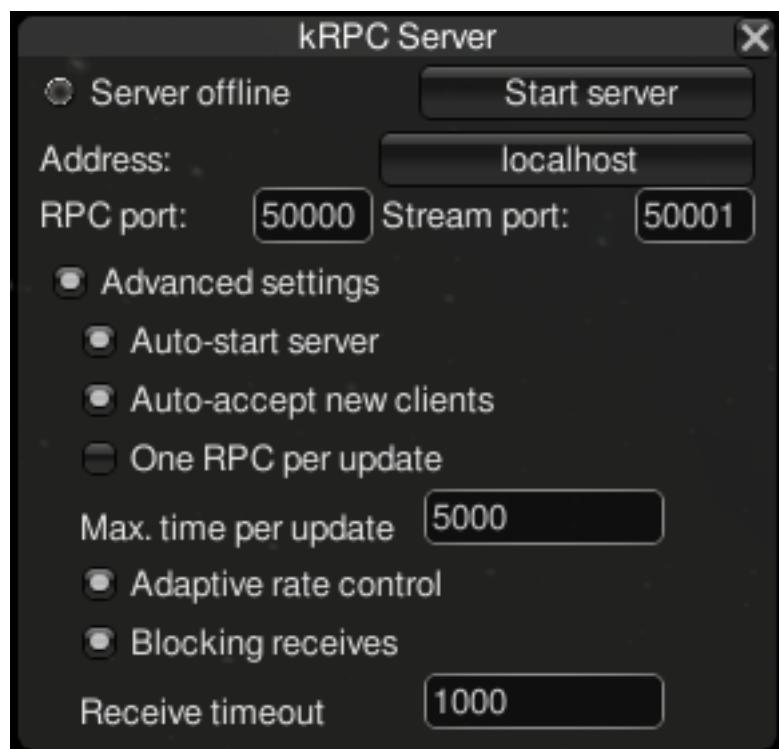


Fig. 12.1: Server window showing the advanced settings.

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 allow 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.

i

InfernalRobotics, [390](#)

k

KerbalAlarmClock, [395](#)

krpc, [316](#)

KRPC, [318](#)

krpc.client, [317](#)

krpc.stream, [318](#)

s

SpaceCenter, [319](#)

i

InfernalRobotics, [305](#)

k

KerbalAlarmClock, [310](#)

KRPC, [233](#)

krpc, [232](#)

s

SpaceCenter, [234](#)

Symbols

`__call__()` (Stream method), 318

A

abort (Control attribute), 260, 345

acceleration (Servo attribute), 308, 393

action (Alarm attribute), 311, 395

actions (Module attribute), 275, 360

activate_next_stage() (Control method), 262, 347

activateNextStage() (Java method), 186

active (Engine attribute), 280, 365

ACTIVE (Java field), 204, 207

active (Light attribute), 286, 371

active (ReactionWheel attribute), 291, 376

active (ResourceHarvester attribute), 290, 375

active (Sensor attribute), 292, 377

active() (ResourceConverter method), 289, 374

active(int) (Java method), 206

active_vessel (in module SpaceCenter), 234, 319

add_node() (Control method), 262, 347

add_stream() (Client method), 317

add_stream() (in module KRPC), 233, 318

addNode(double, float, float, float) (Java method), 187

addStream(Class, String, Object) (Java method), 164

addStream(krpc.schema.KRPC.Request) (Java method), 164

addStream(RemoteObject, String, Object) (Java method), 164

ADJACENT (Java field), 215

aerodynamic_force (Flight attribute), 254, 339

Alarm (class in KerbalAlarmClock), 311, 395

Alarm (Java class), 226

alarm_with_name() (in module KerbalAlarmClock), 310, 395

AlarmAction (class in KerbalAlarmClock), 314, 398

AlarmAction (Java enum), 229

AlarmAction.do_nothing (in module KerbalAlarmClock), 314, 398

AlarmAction.do_nothing_delete_when_passed (in module KerbalAlarmClock), 314, 398

AlarmAction.kill_warp (in module KerbalAlarmClock), 314, 398

AlarmAction.kill_warp_only (in module KerbalAlarmClock), 314, 398

AlarmAction.message_only (in module KerbalAlarmClock), 314, 398

AlarmAction.pause_game (in module KerbalAlarmClock), 314, 398

alarms (in module KerbalAlarmClock), 310, 395

alarms_with_type() (in module KerbalAlarmClock), 310, 395

alarmsWithType(AlarmType) (Java method), 226

AlarmType (class in KerbalAlarmClock), 312, 397

AlarmType (Java enum), 227

AlarmType.apoapsis (in module KerbalAlarmClock), 313, 397

AlarmType.ascending_node (in module KerbalAlarmClock), 313, 397

AlarmType.closest (in module KerbalAlarmClock), 313, 397

AlarmType.contract (in module KerbalAlarmClock), 313, 397

AlarmType.contract_auto (in module KerbalAlarmClock), 313, 398

AlarmType.crew (in module KerbalAlarmClock), 313, 398

AlarmType.descending_node (in module KerbalAlarmClock), 313, 397

AlarmType.distance (in module KerbalAlarmClock), 313, 398

AlarmType.earth_time (in module KerbalAlarmClock), 313, 398

AlarmType.launch_rendevous (in module KerbalAlarmClock), 313, 398

AlarmType.maneuver (in module KerbalAlarmClock), 313, 397

AlarmType.maneuver_auto (in module KerbalAlarmClock), 313, 397

AlarmType.periapsis (in module KerbalAlarmClock), 313, 397

AlarmType.raw (in module KerbalAlarmClock), 312, 397

AlarmType soi_change (in module KerbalAlarmClock), 313, 398

AlarmType soi_change_auto (in module KerbalAlarm-

Clock), 313, 398
 AlarmType.transfer (in module KerbalAlarmClock), 314, 398
 AlarmType.transfer_modelled (in module KerbalAlarmClock), 314, 398
 alarmWithName(String) (Java method), 226
 all (Parts attribute), 264, 349
 amount() (Resources method), 298, 383
 amount(String) (Java method), 214
 angle_of_attack (Flight attribute), 255, 340
 angular_velocity() (CelestialBody method), 250, 335
 angular_velocity() (Vessel method), 245, 330
 angularVelocity(ReferenceFrame) (Java method), 175, 179
 anti_normal (Flight attribute), 253, 338
 ANTI_NORMAL (Java field), 187
 anti_radial (Flight attribute), 253, 338
 ANTI_RADIAL (Java field), 187
 ANTI_TARGET (Java field), 187
 APOAPSIS (Java field), 228
 apoapsis (Orbit attribute), 257, 342
 apoapsis_altitude (Orbit attribute), 257, 342
 area (Intake attribute), 284, 369
 argument_of_periapsis (Orbit attribute), 258, 343
 ASCENDING_NODE (Java field), 228
 atmosphere_density (Flight attribute), 253, 338
 atmosphere_depth (CelestialBody attribute), 248, 333
 auto_mode_switch (Engine attribute), 283, 368
 auto_pilot (Vessel attribute), 239, 324
 AutoPilot (class in SpaceCenter), 303, 387
 AutoPilot (Java class), 219
 available_thrust (Engine attribute), 281, 366
 available_thrust (Vessel attribute), 240, 325
 axially_attached (Part attribute), 268, 353

B

ballistic_coefficient (Flight attribute), 256, 341
 BASE (Java field), 175
 bedrock_altitude (Flight attribute), 251, 336
 bedrock_height() (CelestialBody method), 247, 332
 bedrock_position() (CelestialBody method), 248, 333
 bedrockHeight(double, double) (Java method), 177
 bedrockPosition(double, double, ReferenceFrame) (Java method), 177
 bodies (in module SpaceCenter), 234, 319
 body (Orbit attribute), 256, 341
 brakes (Control attribute), 260, 345
 BROKEN (Java field), 203, 205, 209
 broken (ReactionWheel attribute), 291, 376
 burn_vector() (Node method), 299, 384
 burnVector(ReferenceFrame) (Java method), 216

C

can_rails_warp_at() (in module SpaceCenter), 235, 320

can_restart (Engine attribute), 282, 367
 can_shutdown (Engine attribute), 282, 367
 canRailsWarpAt(int) (Java method), 166
 CAPACITY (Java field), 207
 cargo_bay (Part attribute), 271, 356
 cargo_bays (Parts attribute), 265, 350
 CargoBay (class in SpaceCenter), 276, 361
 CargoBay (Java class), 196
 CargoBayState (class in SpaceCenter), 276, 361
 CargoBayState (Java enum), 197
 CargoBayState.closed (in module SpaceCenter), 277, 362
 CargoBayState.closing (in module SpaceCenter), 277, 362
 CargoBayState.open (in module SpaceCenter), 277, 362
 CargoBayState.opening (in module SpaceCenter), 277, 362
 CelestialBody (class in SpaceCenter), 246, 331
 CelestialBody (Java class), 176
 center_of_mass (Flight attribute), 252, 337
 children (Part attribute), 268, 353
 clear_drawing() (in module SpaceCenter), 237, 323
 clear_target() (in module SpaceCenter), 234, 319
 clearDrawing() (Java method), 168
 clearTarget() (Java method), 165
 Client (class in krpc), 232
 Client (class in krpc.client), 317
 close() (Client method), 232, 317
 close() (Java method), 164
 CLOSED (Java field), 197
 CLOSEST (Java field), 228
 CLOSING (Java field), 197
 Comms (class in SpaceCenter), 301, 386
 Comms (Java class), 217
 comms (Vessel attribute), 240, 324
 config_speed (Servo attribute), 308, 393
 connect() (in module krpc), 232, 316
 Connection (Java class), 163
 CONTRACT (Java field), 228
 CONTRACT_AUTO (Java field), 228
 Control (class in SpaceCenter), 259, 344
 Control (Java class), 185
 control (Vessel attribute), 239, 324
 ControlGroup (class in InfernalRobotics), 306, 391
 ControlGroup (Java class), 222
 controlling (Parts attribute), 264, 349
 core_temperature (ResourceHarvester attribute), 290, 375
 cost (Part attribute), 267, 352
 count (ResourceConverter attribute), 289, 373
 create_alarm() (in module KerbalAlarmClock), 311, 395
 createAlarm(AlarmType, String, double) (Java method), 226
 CREW (Java field), 228
 crossfeed (Part attribute), 271, 356
 current_game_scene (in module KRPC), 233, 318

current_speed (Servo attribute), 308, 393
current_stage (Control attribute), 261, 346
CUT (Java field), 205

D

DEBRIS (Java field), 175
decouple() (Decoupler method), 277, 362
decouple() (Java method), 197
decouple_stage (Part attribute), 269, 354
decoupled (Decoupler attribute), 277, 362
Decoupler (class in SpaceCenter), 277, 362
Decoupler (Java class), 197
decoupler (Part attribute), 271, 356
decouplers (Parts attribute), 265, 350
delta_v (Node attribute), 299, 384
density() (Resources static method), 298, 383
density(String) (Java method), 215
deploy() (Java method), 204
deploy() (Parachute method), 287, 372
deploy_altitude (Parachute attribute), 287, 372
deploy_min_pressure (Parachute attribute), 287, 372
deployable (LandingGear attribute), 285, 369
deployable (Radiator attribute), 288, 373
DEPLOYED (Java field), 203, 204, 207
deployed (LandingGear attribute), 285, 370
deployed (LandingLeg attribute), 285, 370
deployed (Parachute attribute), 287, 372
deployed (Radiator attribute), 288, 373
deployed (ResourceHarvester attribute), 290, 375
deployed (SolarPanel attribute), 292, 377
DEPLOYING (Java field), 203, 207
DESCENDING_NODE (Java field), 228
direction (Flight attribute), 252, 337
direction() (CelestialBody method), 250, 335
direction() (DockingPort method), 278, 363
direction() (Node method), 301, 386
direction() (Part method), 273, 358
direction() (Vessel method), 245, 330
direction(ReferenceFrame) (Java method), 175, 179, 194, 198, 217
disengage() (AutoPilot method), 303, 388
disengage() (Java method), 219
DISTANCE (Java field), 228
DO_NOTHING (Java field), 229
DO_NOTHING_DELETE_WHEN_PASSED (Java field), 229
DOCKED (Java field), 175, 198
docked_part (DockingPort attribute), 278, 363
DOCKING (Java field), 198
docking_port (Part attribute), 272, 357
docking_port_with_name() (Parts method), 266, 351
docking_ports (Parts attribute), 265, 350
DockingPort (class in SpaceCenter), 277, 362
DockingPort (Java class), 197

DockingPortState (class in SpaceCenter), 279, 364
DockingPortState (Java enum), 198
DockingPortState.docked (in module SpaceCenter), 279, 364
DockingPortState.docking (in module SpaceCenter), 279, 364
DockingPortState.moving (in module SpaceCenter), 280, 365
DockingPortState.ready (in module SpaceCenter), 279, 364
DockingPortState.shielded (in module SpaceCenter), 280, 365
DockingPortState.undocking (in module SpaceCenter), 280, 365
dockingPortWithName(String) (Java method), 190
drag (Flight attribute), 254, 339
drag_coefficient (Flight attribute), 256, 341
draw_direction() (in module SpaceCenter), 237, 322
draw_line() (in module SpaceCenter), 237, 322
drawDirection(org.javatuples.Triplet, ReferenceFrame, org.javatuples.Triplet, float) (Java method), 168
drawLine(org.javatuples.Triplet, org.javatuples.Triplet, ReferenceFrame, org.javatuples.Triplet) (Java method), 168
dry_mass (Part attribute), 269, 354
dry_mass (Vessel attribute), 240, 325
dynamic_pressure (Flight attribute), 253, 338

E

EARTH_TIME (Java field), 228
eccentric_anomaly (Orbit attribute), 259, 344
eccentricity (Orbit attribute), 258, 343
EDITOR_SPH (Java field), 165
EDITOR_VAB (Java field), 165
elevation (Flight attribute), 251, 336
energy_flow (SolarPanel attribute), 293, 378
engage() (AutoPilot method), 303, 387
engage() (Java method), 219
Engine (class in SpaceCenter), 280, 365
Engine (Java class), 200
engine (Part attribute), 272, 357
engines (Parts attribute), 266, 351
epoch (Orbit attribute), 259, 344
equatorial_radius (CelestialBody attribute), 247, 332
equivalent_air_speed (Flight attribute), 255, 340
error (AutoPilot attribute), 303, 388
ESCAPING (Java field), 176
events (Module attribute), 275, 360
expanded (ControlGroup attribute), 306, 391
EXTENDED (Java field), 205, 209
EXTENDING (Java field), 205, 209
extraction_rate (ResourceHarvester attribute), 290, 375

F

Fairing (class in SpaceCenter), 283, 368
 Fairing (Java class), 202
 fairing (Part attribute), 272, 357
 fairings (Parts attribute), 266, 351
 far_available (in module SpaceCenter), 237, 322
 fields (Module attribute), 275, 360
 Flight (class in SpaceCenter), 251, 336
 Flight (Java class), 179
 FLIGHT (Java field), 165
 flight() (Vessel method), 238, 323
 flight(ReferenceFrame) (Java method), 169
 flow (Intake attribute), 284, 369
 flow_mode() (Resources static method), 298, 383
 flowMode(String) (Java method), 215
 FLYING (Java field), 176
 forward (Control attribute), 261, 346
 forward_key (ControlGroup attribute), 306, 391
 fuel_lines_from (Part attribute), 271, 356
 fuel_lines_to (Part attribute), 271, 356

G

g (in module SpaceCenter), 235, 320
 g_force (Flight attribute), 251, 336
 GameScene (class in KRPC), 233, 318
 GameScene (Java enum), 165
 GameScene.editor_sph (in module KRPC), 233, 319
 GameScene.editor_vab (in module KRPC), 233, 319
 GameScene.flight (in module KRPC), 233, 318
 GameScene.space_center (in module KRPC), 233, 318
 GameScene.tracking_station (in module KRPC), 233, 319
 gear (Control attribute), 260, 345
 get() (Java method), 164
 get_action_group() (Control method), 262, 347
 get_field() (Module method), 275, 360
 get_services() (in module KRPC), 233, 318
 get_status() (in module KRPC), 233, 318
 get_status() (KRPC method), 232, 317
 getAbort() (Java method), 185
 getAcceleration() (Java method), 224
 getAction() (Java method), 226
 getActionGroup(int) (Java method), 186
 getActions() (Java method), 195
 getActive() (Java method), 200, 204, 207, 208
 getActiveVessel() (Java method), 165
 getAerodynamicForce() (Java method), 181
 getAlarms() (Java method), 226
 getAll() (Java method), 188
 getAngleOfAttack() (Java method), 182
 getAntiNormal() (Java method), 180
 getAntiRadial() (Java method), 180
 getApoapsis() (Java method), 183
 getApoapsisAltitude() (Java method), 183

getArea() (Java method), 202
 getArgumentOfPeriapsis() (Java method), 184
 getAtmosphereDensity() (Java method), 180
 getAtmosphereDepth() (Java method), 177
 getAutoModeSwitch() (Java method), 201
 getAutoPilot() (Java method), 170
 getAvailableThrust() (Java method), 170, 200
 getAxiallyAttached() (Java method), 191
 getBallisticCoefficient() (Java method), 182
 getBedrockAltitude() (Java method), 179
 getBodies() (Java method), 165
 getBody() (Java method), 183
 getBrakes() (Java method), 185
 getBroken() (Java method), 208
 getCanRestart() (Java method), 201
 getCanShutdown() (Java method), 201
 getCargoBay() (Java method), 193
 getCargoBays() (Java method), 189
 getCenterOfMass() (Java method), 180
 getChildren() (Java method), 191
 getComms() (Java method), 170
 getConfigSpeed() (Java method), 224
 getControl() (Java method), 169
 getControlling() (Java method), 189
 getCoreTemperature() (Java method), 207
 getCost() (Java method), 191
 getCount() (Java method), 205
 getCrossfeed() (Java method), 192
 getCurrentGameScene() (Java method), 164
 getCurrentSpeed() (Java method), 224
 getCurrentStage() (Java method), 186
 getDecoupled() (Java method), 197
 getDecoupler() (Java method), 193
 getDecouplers() (Java method), 189
 getDecoupleStage() (Java method), 191
 getDeltaV() (Java method), 216
 getDeployable() (Java method), 202, 205
 getDeployAltitude() (Java method), 204
 getDeployed() (Java method), 202–205, 207, 208
 getDeployMinPressure() (Java method), 204
 getDirection() (Java method), 180
 getDockedPart() (Java method), 197
 getDockingPort() (Java method), 193
 getDockingPorts() (Java method), 189
 getDrag() (Java method), 181
 getDragCoefficient() (Java method), 182
 getDryMass() (Java method), 170, 191
 getDynamicPressure() (Java method), 180
 getEccentricAnomaly() (Java method), 184
 getEccentricity() (Java method), 184
 getElevation() (Java method), 179
 getEnergyFlow() (Java method), 208
 getEngine() (Java method), 193
 getEngines() (Java method), 190

getEpoch() (Java method), 184
 getEquatorialRadius() (Java method), 176
 getEquivalentAirSpeed() (Java method), 181
 getError() (Java method), 219
 getEvents() (Java method), 195
 getExpanded() (Java method), 222
 getExtractionRate() (Java method), 207
 getFairing() (Java method), 193
 getFairings() (Java method), 190
 getFARAvailable() (Java method), 168
 getField(String) (Java method), 195
 getFields() (Java method), 194
 getFlow() (Java method), 202
 getForward() (Java method), 186
 getForwardKey() (Java method), 222
 getFuelLinesFrom() (Java method), 193
 getFuelLinesTo() (Java method), 193
 getG() (Java method), 166
 getGear() (Java method), 185
 getGForce() (Java method), 179
 getGimballed() (Java method), 201
 getGimbalLimit() (Java method), 201
 getGimbalLocked() (Java method), 201
 getGimbalRange() (Java method), 201
 getGravitationalParameter() (Java method), 176
 getHasAtmosphere() (Java method), 177
 getHasAtmosphericOxygen() (Java method), 177
 getHasConnection() (Java method), 218
 getHasConnectionToGroundStation() (Java method), 218
 getHasFlightComputer() (Java method), 218
 getHasFuel() (Java method), 201
 getHasLocalControl() (Java method), 218
 getHasModes() (Java method), 201
 getHasShield() (Java method), 198
 getHeading() (Java method), 180
 getHorizontalSpeed() (Java method), 180
 getID() (Java method), 227
 getImpactTolerance() (Java method), 192
 getImpulse() (Java method), 197
 getInclination() (Java method), 184
 getIntake() (Java method), 193
 getIntakes() (Java method), 190
 getIsAxisInverted() (Java method), 224
 getIsFreeMoving() (Java method), 224
 getIsFuelLine() (Java method), 192
 getIsLocked() (Java method), 224
 getIsMoving() (Java method), 224
 getJettisoned() (Java method), 202
 getKerbinSeaLevelSpecificImpulse() (Java method), 171, 200
 getLandingGear() (Java method), 190, 193
 getLandingLeg() (Java method), 193
 getLandingLegs() (Java method), 190
 getLatitude() (Java method), 180
 getLaunchClamp() (Java method), 193
 getLaunchClamps() (Java method), 190
 getLift() (Java method), 181
 getLiftCoefficient() (Java method), 182
 getLight() (Java method), 193
 getLights() (Java method), 185, 190
 getLongitude() (Java method), 180
 getLongitudeOfAscendingNode() (Java method), 184
 getMach() (Java method), 181
 getMargin() (Java method), 226
 getMass() (Java method), 170, 176, 191
 getMassless() (Java method), 191
 getMaxConfigPosition() (Java method), 223
 getMaximumRailsWarpFactor() (Java method), 166
 getMaxPosition() (Java method), 224
 getMaxRollSpeed() (Java method), 220
 getMaxRotationSpeed() (Java method), 220
 getMaxSkinTemperature() (Java method), 192
 getMaxTemperature() (Java method), 192
 getMaxThrust() (Java method), 170, 200
 getMaxVacuumThrust() (Java method), 170, 200
 getMeanAltitude() (Java method), 179
 getMeanAnomaly() (Java method), 184
 getMeanAnomalyAtEpoch() (Java method), 184
 getMET() (Java method), 169
 getMinConfigPosition() (Java method), 223
 getMinPosition() (Java method), 223
 getMode() (Java method), 201
 getModes() (Java method), 201
 getModules() (Java method), 193
 getName() (Java method), 169, 176, 190, 194, 197, 222, 223, 227
 getNames() (Java method), 214
 getNextOrbit() (Java method), 184
 getNodes() (Java method), 187
 getNonRotatingReferenceFrame() (Java method), 178
 getNormal() (Java method), 180, 215
 getNotes() (Java method), 227
 getOpen() (Java method), 196, 202
 getOptimumCoreTemperature() (Java method), 207
 getOrbit() (Java method), 169, 176, 216
 getOrbitalReferenceFrame() (Java method), 171, 178, 217
 getParachute() (Java method), 193
 getParachutes() (Java method), 190
 getParent() (Java method), 191
 getPart() (Java method), 194, 196, 197, 200, 202–205, 207, 208
 getParts() (Java method), 170
 getPeriapsis() (Java method), 183
 getPeriapsisAltitude() (Java method), 183
 getPeriod() (Java method), 184
 getPhysicsWarpFactor() (Java method), 166
 getPitch() (Java method), 180, 186
 getPitchTorque() (Java method), 208

getPosition() (Java method), 223
 getPowerUsage() (Java method), 204, 208
 getPrograde() (Java method), 180, 215
 getPropellantRatios() (Java method), 201
 getPropellants() (Java method), 201
 getRadial() (Java method), 180, 215
 getRadiallyAttached() (Java method), 191
 getRadiator() (Java method), 193
 getRadiators() (Java method), 190
 getRadius() (Java method), 183
 getRailsWarpFactor() (Java method), 166
 getRCS() (Java method), 185
 getReactionWheel() (Java method), 193
 getReactionWheels() (Java method), 190
 getReengageDistance() (Java method), 198
 getReferenceFrame() (Java method), 171, 177, 194, 198, 217, 219
 getRemaining() (Java method), 227
 getRemainingDeltaV() (Java method), 216
 getRemoteTechAvailable() (Java method), 168
 getRepeat() (Java method), 227
 getRepeatPeriod() (Java method), 227
 getResourceConverter() (Java method), 193
 getResourceConverters() (Java method), 190
 getResourceHarvester() (Java method), 193
 getResourceHarvesters() (Java method), 190
 getResources() (Java method), 170, 192
 getRetrograde() (Java method), 180
 getReverseKey() (Java method), 222
 getRight() (Java method), 186
 getRoll() (Java method), 180, 186
 getRollError() (Java method), 219
 getRollSpeedMultiplier() (Java method), 220
 getRollTorque() (Java method), 208
 getRoot() (Java method), 188
 getRotation() (Java method), 180
 getRotationalPeriod() (Java method), 176
 getRotationalSpeed() (Java method), 176
 getRotationSpeedMultiplier() (Java method), 220
 getSAS() (Java method), 185, 219
 getSASMode() (Java method), 185, 220
 getSatellites() (Java method), 176
 getSemiMajorAxis() (Java method), 183
 getSemiMinorAxis() (Java method), 183
 getSensor() (Java method), 193
 getSensors() (Java method), 190
 getServices() (Java method), 164
 getServoGroups() (Java method), 221
 getServos() (Java method), 222
 getShielded() (Java method), 198
 getSideslipAngle() (Java method), 182
 getSignalDelay() (Java method), 218
 getSignalDelayToGroundStation() (Java method), 218
 getSituation() (Java method), 169
 getSkinTemperature() (Java method), 192
 getSolarPanel() (Java method), 194
 getSolarPanels() (Java method), 190
 getSpecificImpulse() (Java method), 170, 200
 getSpeed() (Java method), 180, 183, 202, 222, 224
 getSpeedMode() (Java method), 185
 getSpeedOfSound() (Java method), 181
 getSphereOfInfluence() (Java method), 177
 getStage() (Java method), 191
 getStallFraction() (Java method), 182
 getState() (Java method), 196, 197, 202–205, 207, 208
 getStaticAirTemperature() (Java method), 182
 getStaticPressure() (Java method), 181
 getStatus() (Java method), 164
 getSunExposure() (Java method), 209
 getSurfaceAltitude() (Java method), 179
 getSurfaceGravity() (Java method), 176
 getSurfaceReferenceFrame() (Java method), 171
 getSurfaceVelocityReferenceFrame() (Java method), 173
 getTarget() (Java method), 169
 getTargetBody() (Java method), 165
 getTargetDirection() (Java method), 219
 getTargetDockingPort() (Java method), 165
 getTargetRoll() (Java method), 219
 getTargetVessel() (Java method), 165
 getTemperature() (Java method), 192
 getTerminalVelocity() (Java method), 182
 getThermalConductionFlux() (Java method), 192
 getThermalConvectionFlux() (Java method), 192
 getThermalEfficiency() (Java method), 207
 getThermalInternalFlux() (Java method), 192
 getThermalMass() (Java method), 192
 getThermalRadiationFlux() (Java method), 192
 getThermalResourceMass() (Java method), 192
 getThermalSkinMass() (Java method), 192
 getThermalSkinToInternalFlux() (Java method), 192
 getThrottle() (Java method), 185, 201
 getThrottleLocked() (Java method), 201
 getThrust() (Java method), 170, 200
 getThrustLimit() (Java method), 200
 getThrustSpecificFuelConsumption() (Java method), 182
 getTime() (Java method), 226
 getTimeTo() (Java method), 216
 getTimeToApoapsis() (Java method), 184
 getTimeToPeriapsis() (Java method), 184
 getTimeToSOIChange() (Java method), 184
 getTitle() (Java method), 190
 getTotalAirTemperature() (Java method), 182
 getType() (Java method), 169, 226
 getUp() (Java method), 186
 getUT() (Java method), 166, 216
 getVacuumSpecificImpulse() (Java method), 170, 200
 getValue() (Java method), 208
 getVelocity() (Java method), 180

getVerticalSpeed() (Java method), 180
 getVessel() (Java method), 191, 227
 getVessels() (Java method), 165
 getWarpFactor() (Java method), 166
 getWarpMode() (Java method), 166
 getWarpRate() (Java method), 166
 getWheelSteering() (Java method), 186
 getWheelThrottle() (Java method), 186
 getXferOriginBody() (Java method), 227
 getXferTargetBody() (Java method), 227
 getYaw() (Java method), 186
 getYawTorque() (Java method), 208
 gimbal_limit (Engine attribute), 283, 368
 gimbal_locked (Engine attribute), 283, 368
 gimbal_range (Engine attribute), 283, 368
 gimballed (Engine attribute), 283, 368
 gravitational_parameter (CelestialBody attribute), 247, 332

H

has_action() (Module method), 275, 360
 has_atmosphere (CelestialBody attribute), 248, 333
 has_atmospheric_oxygen (CelestialBody attribute), 249, 334
 has_connection (Comms attribute), 302, 386
 has_connection_to_ground_station (Comms attribute), 302, 387
 has_event() (Module method), 275, 360
 has_field() (Module method), 275, 360
 has_flight_computer (Comms attribute), 301, 386
 has_fuel (Engine attribute), 282, 367
 has_local_control (Comms attribute), 301, 386
 has_modes (Engine attribute), 282, 367
 has_resource() (Resources method), 297, 382
 has_shield (DockingPort attribute), 278, 363
 hasAction(String) (Java method), 196
 hasEvent(String) (Java method), 195
 hasField(String) (Java method), 195
 hasResource(String) (Java method), 214
 heading (Flight attribute), 252, 337
 highlight (Servo attribute), 307, 392
 horizontal_speed (Flight attribute), 252, 337

I

id (Alarm attribute), 311, 396
 IDLE (Java field), 206
 impact_tolerance (Part attribute), 269, 354
 impulse (Decoupler attribute), 277, 362
 in_decouple_stage() (Parts method), 265, 350
 in_stage() (Parts method), 265, 350
 inclination (Orbit attribute), 258, 343
 inDecoupleStage(int) (Java method), 189
 InfernalRobotics (Java class), 221
 InfernalRobotics (module), 305, 390

inputs() (ResourceConverter method), 289, 374
 inputs(int) (Java method), 206
 inStage(int) (Java method), 189
 Intake (class in SpaceCenter), 284, 369
 Intake (Java class), 202
 intake (Part attribute), 272, 357
 intakes (Parts attribute), 266, 351
 is_axis_inverted (Servo attribute), 309, 393
 is_free_moving (Servo attribute), 309, 393
 is_fuel_line (Part attribute), 271, 356
 is_locked (Servo attribute), 309, 393
 is_moving (Servo attribute), 309, 393

J

jettison() (Fairing method), 283, 368
 jettison() (Java method), 202
 jettisoned (Fairing attribute), 283, 368

K

KerbalAlarmClock (Java class), 226
 KerbalAlarmClock (module), 310, 395
 kerbin_sea_level_specific_impulse (Engine attribute), 281, 366
 kerbin_sea_level_specific_impulse (Vessel attribute), 241, 326
 KILL_WARP (Java field), 229
 KILL_WARP_ONLY (Java field), 229
 KRPC (class in krpc), 232
 KRPC (class in krpc.client), 317
 krpc (Client attribute), 232, 317
 KRPC (Java class), 164
 KRPC (module), 233, 318
 krpc (module), 232, 316
 krpc.client (module), 317
 krpc.client (package), 163
 krpc.stream (module), 318
 krpc::Client (C++ class), 94
 krpc::connect (C++ function), 94
 krpc::services::InfernalRobotics (C++ class), 152
 krpc::services::InfernalRobotics::ControlGroup (C++ class), 152
 krpc::services::InfernalRobotics::ControlGroup::expanded (C++ function), 153
 krpc::services::InfernalRobotics::ControlGroup::forward_key (C++ function), 152
 krpc::services::InfernalRobotics::ControlGroup::move_center (C++ function), 153
 krpc::services::InfernalRobotics::ControlGroup::move_left (C++ function), 153
 krpc::services::InfernalRobotics::ControlGroup::move_next_preset (C++ function), 153
 krpc::services::InfernalRobotics::ControlGroup::move_prev_preset (C++ function), 153

<code>krpc::services::InfernalRobotics::ControlGroup::move_right</code>	(C++ function), 155
<code>krpc::services::InfernalRobotics::ControlGroup::name</code>	(C++ function), 152
<code>krpc::services::InfernalRobotics::ControlGroup::reverse_key</code>	(C++ function), 152
<code>krpc::services::InfernalRobotics::ControlGroup::servo_with_name</code>	(C++ function), 153
<code>krpc::services::InfernalRobotics::ControlGroup::servos</code>	(C++ function), 153
<code>krpc::services::InfernalRobotics::ControlGroup::set_expanded</code>	(C++ function), 153
<code>krpc::services::InfernalRobotics::ControlGroup::set_forward_key</code>	(C++ function), 152
<code>krpc::services::InfernalRobotics::ControlGroup::set_name</code>	(C++ function), 152
<code>krpc::services::InfernalRobotics::ControlGroup::set_reverse_key</code>	(C++ function), 152
<code>krpc::services::InfernalRobotics::ControlGroup::set_speed</code>	(C++ function), 153
<code>krpc::services::InfernalRobotics::ControlGroup::speed</code>	(C++ function), 152
<code>krpc::services::InfernalRobotics::ControlGroup::stop</code>	(C++ function), 153
<code>krpc::services::InfernalRobotics::InfernalRobotics</code>	(C++ function), 152
<code>krpc::services::InfernalRobotics::Servo</code>	(C++ class), 153
<code>krpc::services::InfernalRobotics::Servo::acceleration</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::config_speed</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::current_speed</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::is_axis_inverted</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::is_free_moving</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::is_locked</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::is_moving</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::max_config_position</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::max_position</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::min_config_position</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::min_position</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::move_center</code>	(C++ function), 155
<code>krpc::services::InfernalRobotics::Servo::move_left</code>	(C++ function), 155
<code>krpc::services::InfernalRobotics::Servo::move_next_preset</code>	(C++ function), 155
<code>krpc::services::InfernalRobotics::Servo::move_prev_preset</code>	(C++ function), 155
<code>krpc::services::InfernalRobotics::Servo::move_right</code>	(C++ function), 155
<code>krpc::services::InfernalRobotics::Servo::move_to</code>	(C++ function), 155
<code>krpc::services::InfernalRobotics::Servo::name</code>	(C++ function), 153
<code>krpc::services::InfernalRobotics::Servo::position</code>	(C++ function), 153
<code>krpc::services::InfernalRobotics::Servo::set_acceleration</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::set_current_speed</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::set_highlight</code>	(C++ function), 153
<code>krpc::services::InfernalRobotics::Servo::set_is_axis_inverted</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::set_is_locked</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::set_max_position</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::set_min_position</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::set_name</code>	(C++ function), 153
<code>krpc::services::InfernalRobotics::Servo::set_speed</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::speed</code>	(C++ function), 154
<code>krpc::services::InfernalRobotics::Servo::stop</code>	(C++ function), 155
<code>krpc::services::InfernalRobotics::servo_group_with_name</code>	(C++ function), 152
<code>krpc::services::InfernalRobotics::servo_groups</code>	(C++ function), 152
<code>krpc::services::InfernalRobotics::servo_with_name</code>	(C++ function), 152
<code>krpc::services::KerbalAlarmClock</code>	(C++ class), 156
<code>krpc::services::KerbalAlarmClock::Alarm</code>	(C++ class), 156
<code>krpc::services::KerbalAlarmClock::Alarm::action</code>	(C++ function), 157
<code>krpc::services::KerbalAlarmClock::Alarm::id</code>	(C++ function), 157
<code>krpc::services::KerbalAlarmClock::Alarm::margin</code>	(C++ function), 157
<code>krpc::services::KerbalAlarmClock::Alarm::name</code>	(C++ function), 157
<code>krpc::services::KerbalAlarmClock::Alarm::notes</code>	(C++ function), 157
<code>krpc::services::KerbalAlarmClock::Alarm::remaining</code>	(C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::remove (C++ function), 158

krpc::services::KerbalAlarmClock::Alarm::repeat (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::repeat_period (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_action (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_margin (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_name (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_notes (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_repeat (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_repeat_period (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_time (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_vessel (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_xfer_origin_body (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::set_xfer_target_body (C++ function), 158

krpc::services::KerbalAlarmClock::Alarm::time (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::type (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::vessel (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::xfer_origin_body (C++ function), 157

krpc::services::KerbalAlarmClock::Alarm::xfer_target_body (C++ function), 157

krpc::services::KerbalAlarmClock::alarm_with_name (C++ function), 156

krpc::services::KerbalAlarmClock::AlarmAction (C++ enum), 159

krpc::services::KerbalAlarmClock::AlarmAction::do_nothing (C++ enumerator), 159

krpc::services::KerbalAlarmClock::AlarmAction::do_nothing_if_not_active (C++ enumerator), 159

krpc::services::KerbalAlarmClock::AlarmAction::kill_warp (C++ enumerator), 159

krpc::services::KerbalAlarmClock::AlarmAction::kill_warp_if_not_active (C++ enumerator), 159

krpc::services::KerbalAlarmClock::AlarmAction::message_only (C++ enumerator), 159

krpc::services::KerbalAlarmClock::AlarmAction::pause_game (C++ enumerator), 159

krpc::services::KerbalAlarmClock::alarms (C++ function), 156

krpc::services::KerbalAlarmClock::alarms_with_type (C++ function), 156

krpc::services::KerbalAlarmClock::AlarmType (C++ enum), 158

krpc::services::KerbalAlarmClock::AlarmType::apoapsis (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::ascending_node (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::closest (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::contract (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::contract_auto (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::crew (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::descending_node (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::distance (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::earth_time (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::launch_rendevous (C++ enumerator), 159

krpc::services::KerbalAlarmClock::AlarmType::maneuver (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::maneuver_auto (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::periapsis (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::raw (C++ enumerator), 158

krpc::services::KerbalAlarmClock::AlarmType::soi_change (C++ enumerator), 159

krpc::services::KerbalAlarmClock::AlarmType::soi_change_auto (C++ enumerator), 159

krpc::services::KerbalAlarmClock::AlarmType::transfer (C++ enumerator), 159

krpc::services::KerbalAlarmClock::AlarmType::transfer_modelled (C++ enumerator), 159

krpc::services::KerbalAlarmClock::create_alarm (C++ function), 156

krpc::services::KerbalAlarmClock::KerbalAlarmClock (C++ function), 156

krpc::services::KRPC (C++ class), 94, 95

krpc::services::KRPC::add_stream (C++ function), 95

krpc::services::KRPC::current_game_scene (C++ function), 95

krpc::services::KRPC::GameScene (C++ enum), 96

krpc::services::KRPC::GameScene::editor_sph (C++ enumerator), 96

krpc::services::KRPC::GameScene::editor_vab (C++ enumerator), 96

krpc::services::KRPC::GameScene::flight (C++ enumer-

ator), 96
 krpc::services::KRPC::GameScene::space_center (C++
 enumerator), 96
 krpc::services::KRPC::GameScene::tracking_station
 (C++ enumerator), 96
 krpc::services::KRPC::get_services (C++ function), 95
 krpc::services::KRPC::get_status (C++ function), 94, 95
 krpc::services::KRPC::KRPC (C++ function), 94, 95
 krpc::services::KRPC::remove_stream (C++ function),
 96
 krpc::services::SpaceCenter (C++ class), 96
 krpc::services::SpaceCenter::active_vessel (C++ func-
 tion), 96
 krpc::services::SpaceCenter::AutoPilot (C++ class), 149
 krpc::services::SpaceCenter::AutoPilot::disengage (C++
 function), 149
 krpc::services::SpaceCenter::AutoPilot::engage (C++
 function), 149
 krpc::services::SpaceCenter::AutoPilot::error (C++ func-
 tion), 149
 krpc::services::SpaceCenter::AutoPilot::max_roll_speed
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::max_rotation_speed
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::reference_frame
 (C++ function), 149
 krpc::services::SpaceCenter::AutoPilot::roll_error (C++
 function), 149
 krpc::services::SpaceCenter::AutoPilot::roll_speed_multiplier
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::rotation_speed_multiplier
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::sas (C++ func-
 tion), 150
 krpc::services::SpaceCenter::AutoPilot::sas_mode (C++
 function), 150
 krpc::services::SpaceCenter::AutoPilot::set_max_roll_speed
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::set_max_rotation_speed
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::set_pid_parameters
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::set_reference_frame
 (C++ function), 149
 krpc::services::SpaceCenter::AutoPilot::set_roll_speed_multiplier
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::set_rotation_speed_multiplier
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::set_sas (C++
 function), 150
 krpc::services::SpaceCenter::AutoPilot::set_sas_mode
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::set_target_direction
 (C++ function), 149
 krpc::services::SpaceCenter::AutoPilot::set_target_roll
 (C++ function), 150
 krpc::services::SpaceCenter::AutoPilot::target_direction
 (C++ function), 149
 krpc::services::SpaceCenter::AutoPilot::target_pitch_and_heading
 (C++ function), 149
 krpc::services::SpaceCenter::AutoPilot::target_roll (C++
 function), 150
 krpc::services::SpaceCenter::AutoPilot::wait (C++ func-
 tion), 149
 krpc::services::SpaceCenter::bodies (C++ function), 96
 krpc::services::SpaceCenter::can_rails_warp_at (C++
 function), 97
 krpc::services::SpaceCenter::CargoBay (C++ class), 127
 krpc::services::SpaceCenter::CargoBay::open (C++ func-
 tion), 127
 krpc::services::SpaceCenter::CargoBay::part (C++ func-
 tion), 127
 krpc::services::SpaceCenter::CargoBay::set_open (C++
 function), 127
 krpc::services::SpaceCenter::CargoBay::state (C++ func-
 tion), 127
 krpc::services::SpaceCenter::CargoBayState (C++
 enum), 127
 krpc::services::SpaceCenter::CargoBayState::closed
 (C++ enumerator), 127
 krpc::services::SpaceCenter::CargoBayState::closing
 (C++ enumerator), 127
 krpc::services::SpaceCenter::CargoBayState::open (C++
 enumerator), 127
 krpc::services::SpaceCenter::CargoBayState::opening
 (C++ enumerator), 127
 krpc::services::SpaceCenter::CelestialBody (C++ class),
 107
 krpc::services::SpaceCenter::CelestialBody::angular_velocity
 (C++ function), 110
 krpc::services::SpaceCenter::CelestialBody::atmosphere_depth
 (C++ function), 108
 krpc::services::SpaceCenter::CelestialBody::bedrock_height
 (C++ function), 107
 krpc::services::SpaceCenter::CelestialBody::bedrock_position
 (C++ function), 108
 krpc::services::SpaceCenter::CelestialBody::direction
 (C++ function), 110
 krpc::services::SpaceCenter::CelestialBody::equatorial_radius
 (C++ function), 107
 krpc::services::SpaceCenter::CelestialBody::gravitational_parameter
 (C++ function), 107
 krpc::services::SpaceCenter::CelestialBody::has_atmosphere
 (C++ function), 108
 krpc::services::SpaceCenter::CelestialBody::has_atmospheric_oxygen
 (C++ function), 108
 krpc::services::SpaceCenter::CelestialBody::mass (C++
 function), 107

krpc::services::SpaceCenter::CelestialBody::msl_position (C++ function), 108	krpc::services::SpaceCenter::Control::activate_next_stage (C++ function), 117
krpc::services::SpaceCenter::CelestialBody::name (C++ function), 107	krpc::services::SpaceCenter::Control::add_node (C++ function), 117
krpc::services::SpaceCenter::CelestialBody::non_rotating_reference_frame (C++ function), 108	krpc::services::SpaceCenter::Control::brakes (C++ function), 116
krpc::services::SpaceCenter::CelestialBody::orbit (C++ function), 107	krpc::services::SpaceCenter::Control::current_stage (C++ function), 117
krpc::services::SpaceCenter::CelestialBody::orbital_reference_frame (C++ function), 109	krpc::services::SpaceCenter::Control::forward (C++ function), 116
krpc::services::SpaceCenter::CelestialBody::position (C++ function), 109	krpc::services::SpaceCenter::Control::gear (C++ function), 116
krpc::services::SpaceCenter::CelestialBody::reference_frame (C++ function), 108	krpc::services::SpaceCenter::Control::get_action_group (C++ function), 117
krpc::services::SpaceCenter::CelestialBody::rotation (C++ function), 109	krpc::services::SpaceCenter::Control::lights (C++ function), 116
krpc::services::SpaceCenter::CelestialBody::rotational_period (C++ function), 107	krpc::services::SpaceCenter::Control::nodes (C++ function), 118
krpc::services::SpaceCenter::CelestialBody::rotational_speed (C++ function), 107	krpc::services::SpaceCenter::Control::pitch (C++ function), 116
krpc::services::SpaceCenter::CelestialBody::satellites (C++ function), 107	krpc::services::SpaceCenter::Control::rcs (C++ function), 116
krpc::services::SpaceCenter::CelestialBody::sphere_of_influence (C++ function), 108	krpc::services::SpaceCenter::Control::remove_nodes (C++ function), 118
krpc::services::SpaceCenter::CelestialBody::surface_gravity (C++ function), 107	krpc::services::SpaceCenter::Control::right (C++ function), 117
krpc::services::SpaceCenter::CelestialBody::surface_height (C++ function), 107	krpc::services::SpaceCenter::Control::roll (C++ function), 116
krpc::services::SpaceCenter::CelestialBody::surface_position (C++ function), 108	krpc::services::SpaceCenter::Control::sas (C++ function), 115
krpc::services::SpaceCenter::CelestialBody::velocity (C++ function), 109	krpc::services::SpaceCenter::Control::sas_mode (C++ function), 116
krpc::services::SpaceCenter::clear_drawing (C++ function), 99	krpc::services::SpaceCenter::Control::set_abort (C++ function), 116
krpc::services::SpaceCenter::clear_target (C++ function), 97	krpc::services::SpaceCenter::Control::set_action_group (C++ function), 117
krpc::services::SpaceCenter::Comms (C++ class), 148	krpc::services::SpaceCenter::Control::set_brakes (C++ function), 116
krpc::services::SpaceCenter::Comms::has_connection (C++ function), 148	krpc::services::SpaceCenter::Control::set_forward (C++ function), 116
krpc::services::SpaceCenter::Comms::has_connection_to_ground_station (C++ function), 148	krpc::services::SpaceCenter::Control::set_gear (C++ function), 116
krpc::services::SpaceCenter::Comms::has_flight_computer (C++ function), 148	krpc::services::SpaceCenter::Control::set_lights (C++ function), 116
krpc::services::SpaceCenter::Comms::has_local_control (C++ function), 148	krpc::services::SpaceCenter::Control::set_pitch (C++ function), 116
krpc::services::SpaceCenter::Comms::signal_delay (C++ function), 148	krpc::services::SpaceCenter::Control::set_rcs (C++ function), 116
krpc::services::SpaceCenter::Comms::signal_delay_to_ground_station (C++ function), 148	krpc::services::SpaceCenter::Control::set_right (C++ function), 117
krpc::services::SpaceCenter::Comms::signal_delay_to_vessel (C++ function), 148	krpc::services::SpaceCenter::Control::set_roll (C++ function), 116
krpc::services::SpaceCenter::Control (C++ class), 115	krpc::services::SpaceCenter::Control::set_sas (C++ function), 115
krpc::services::SpaceCenter::Control::abort (C++ function), 116	

krpc::services::SpaceCenter::Control::set_sas_mode
 (C++ function), 116
 krpc::services::SpaceCenter::Control::set_speed_mode
 (C++ function), 116
 krpc::services::SpaceCenter::Control::set_throttle (C++
 function), 116
 krpc::services::SpaceCenter::Control::set_up (C++ func-
 tion), 117
 krpc::services::SpaceCenter::Control::set_wheel_steering
 (C++ function), 117
 krpc::services::SpaceCenter::Control::set_wheel_throttle
 (C++ function), 117
 krpc::services::SpaceCenter::Control::set_yaw (C++
 function), 116
 krpc::services::SpaceCenter::Control::speed_mode (C++
 function), 116
 krpc::services::SpaceCenter::Control::throttle (C++ func-
 tion), 116
 krpc::services::SpaceCenter::Control::toggle_action_group
 (C++ function), 117
 krpc::services::SpaceCenter::Control::up (C++ function),
 117
 krpc::services::SpaceCenter::Control::wheel_steering
 (C++ function), 117
 krpc::services::SpaceCenter::Control::wheel_throttle
 (C++ function), 117
 krpc::services::SpaceCenter::Control::yaw (C++ func-
 tion), 116
 krpc::services::SpaceCenter::Decoupler (C++ class), 127
 krpc::services::SpaceCenter::Decoupler::decouple (C++
 function), 127
 krpc::services::SpaceCenter::Decoupler::decoupled (C++
 function), 127
 krpc::services::SpaceCenter::Decoupler::impulse (C++
 function), 127
 krpc::services::SpaceCenter::Decoupler::part (C++ func-
 tion), 127
 krpc::services::SpaceCenter::DockingPort (C++ class),
 127
 krpc::services::SpaceCenter::DockingPort::direction
 (C++ function), 128
 krpc::services::SpaceCenter::DockingPort::docked_part
 (C++ function), 128
 krpc::services::SpaceCenter::DockingPort::has_shield
 (C++ function), 128
 krpc::services::SpaceCenter::DockingPort::name (C++
 function), 127
 krpc::services::SpaceCenter::DockingPort::part (C++
 function), 127
 krpc::services::SpaceCenter::DockingPort::position (C++
 function), 128
 krpc::services::SpaceCenter::DockingPort::reengage_distance
 (C++ function), 128
 krpc::services::SpaceCenter::DockingPort::reference_frame
 (C++ function), 128
 krpc::services::SpaceCenter::DockingPort::rotation (C++
 function), 128
 krpc::services::SpaceCenter::DockingPort::set_name
 (C++ function), 127
 krpc::services::SpaceCenter::DockingPort::set_shielded
 (C++ function), 128
 krpc::services::SpaceCenter::DockingPort::shielded
 (C++ function), 128
 krpc::services::SpaceCenter::DockingPort::state (C++
 function), 128
 krpc::services::SpaceCenter::DockingPort::undock (C++
 function), 128
 krpc::services::SpaceCenter::DockingPortState (C++
 enum), 128
 krpc::services::SpaceCenter::DockingPortState::docked
 (C++ enumerator), 130
 krpc::services::SpaceCenter::DockingPortState::docking
 (C++ enumerator), 130
 krpc::services::SpaceCenter::DockingPortState::moving
 (C++ enumerator), 130
 krpc::services::SpaceCenter::DockingPortState::ready
 (C++ enumerator), 130
 krpc::services::SpaceCenter::DockingPortState::shielded
 (C++ enumerator), 130
 krpc::services::SpaceCenter::DockingPortState::undocking
 (C++ enumerator), 130
 krpc::services::SpaceCenter::draw_direction (C++ func-
 tion), 99
 krpc::services::SpaceCenter::draw_line (C++ function),
 99
 krpc::services::SpaceCenter::Engine (C++ class), 130
 krpc::services::SpaceCenter::Engine::active (C++ func-
 tion), 130
 krpc::services::SpaceCenter::Engine::auto_mode_switch
 (C++ function), 131
 krpc::services::SpaceCenter::Engine::available_thrust
 (C++ function), 130
 krpc::services::SpaceCenter::Engine::can_restart (C++
 function), 131
 krpc::services::SpaceCenter::Engine::can_shutdown
 (C++ function), 131
 krpc::services::SpaceCenter::Engine::gimbal_limit (C++
 function), 132
 krpc::services::SpaceCenter::Engine::gimbal_locked
 (C++ function), 132
 krpc::services::SpaceCenter::Engine::gimbal_range (C++
 function), 131
 krpc::services::SpaceCenter::Engine::gimballed (C++
 function), 131
 krpc::services::SpaceCenter::Engine::has_fuel (C++
 function), 131
 krpc::services::SpaceCenter::Engine::has_modes (C++
 function), 131

krpc::services::SpaceCenter::Engine::kerbin_sea_level_specific_impulse (C++ function), 131	krpc::services::SpaceCenter::Flight::angle_of_attack (C++ function), 112
krpc::services::SpaceCenter::Engine::max_thrust (C++ function), 130	krpc::services::SpaceCenter::Flight::anti_normal (C++ function), 111
krpc::services::SpaceCenter::Engine::max_vacuum_thrust (C++ function), 130	krpc::services::SpaceCenter::Flight::anti_radial (C++ function), 111
krpc::services::SpaceCenter::Engine::mode (C++ function), 131	krpc::services::SpaceCenter::Flight::atmosphere_density (C++ function), 111
krpc::services::SpaceCenter::Engine::modes (C++ function), 131	krpc::services::SpaceCenter::Flight::ballistic_coefficient (C++ function), 113
krpc::services::SpaceCenter::Engine::part (C++ function), 130	krpc::services::SpaceCenter::Flight::bedrock_altitude (C++ function), 110
krpc::services::SpaceCenter::Engine::propellant_ratios (C++ function), 131	krpc::services::SpaceCenter::Flight::center_of_mass (C++ function), 111
krpc::services::SpaceCenter::Engine::propellants (C++ function), 131	krpc::services::SpaceCenter::Flight::direction (C++ function), 111
krpc::services::SpaceCenter::Engine::set_active (C++ function), 130	krpc::services::SpaceCenter::Flight::drag (C++ function), 112
krpc::services::SpaceCenter::Engine::set_auto_mode_switch (C++ function), 131	krpc::services::SpaceCenter::Flight::drag_coefficient (C++ function), 113
krpc::services::SpaceCenter::Engine::set_gimbal_limit (C++ function), 132	krpc::services::SpaceCenter::Flight::dynamic_pressure (C++ function), 111
krpc::services::SpaceCenter::Engine::set_gimbal_locked (C++ function), 132	krpc::services::SpaceCenter::Flight::elevation (C++ function), 110
krpc::services::SpaceCenter::Engine::set_mode (C++ function), 131	krpc::services::SpaceCenter::Flight::equivalent_air_speed (C++ function), 112
krpc::services::SpaceCenter::Engine::set_thrust_limit (C++ function), 130	krpc::services::SpaceCenter::Flight::g_force (C++ function), 110
krpc::services::SpaceCenter::Engine::specific_impulse (C++ function), 131	krpc::services::SpaceCenter::Flight::heading (C++ function), 111
krpc::services::SpaceCenter::Engine::throttle (C++ function), 131	krpc::services::SpaceCenter::Flight::horizontal_speed (C++ function), 110
krpc::services::SpaceCenter::Engine::throttle_locked (C++ function), 131	krpc::services::SpaceCenter::Flight::latitude (C++ function), 110
krpc::services::SpaceCenter::Engine::thrust (C++ function), 130	krpc::services::SpaceCenter::Flight::lift (C++ function), 112
krpc::services::SpaceCenter::Engine::thrust_limit (C++ function), 130	krpc::services::SpaceCenter::Flight::lift_coefficient (C++ function), 113
krpc::services::SpaceCenter::Engine::toggle_mode (C++ function), 131	krpc::services::SpaceCenter::Flight::longitude (C++ function), 110
krpc::services::SpaceCenter::Engine::vacuum_specific_impulse (C++ function), 131	krpc::services::SpaceCenter::Flight::mach (C++ function), 112
krpc::services::SpaceCenter::Fairing (C++ class), 132	krpc::services::SpaceCenter::Flight::mean_altitude (C++ function), 110
krpc::services::SpaceCenter::Fairing::jettison (C++ function), 132	krpc::services::SpaceCenter::Flight::normal (C++ function), 111
krpc::services::SpaceCenter::Fairing::jettisoned (C++ function), 132	krpc::services::SpaceCenter::Flight::pitch (C++ function), 111
krpc::services::SpaceCenter::Fairing::part (C++ function), 132	krpc::services::SpaceCenter::Flight::prograde (C++ function), 111
krpc::services::SpaceCenter::far_available (C++ function), 99	krpc::services::SpaceCenter::Flight::radial (C++ function), 111
krpc::services::SpaceCenter::Flight (C++ class), 110	krpc::services::SpaceCenter::Flight::retrograde (C++ function), 111
krpc::services::SpaceCenter::Flight::aerodynamic_force (C++ function), 111	

krpc::services::SpaceCenter::Flight::roll (C++ function), 111	krpc::services::SpaceCenter::LandingGearState (C++ enum), 133
krpc::services::SpaceCenter::Flight::rotation (C++ function), 111	krpc::services::SpaceCenter::LandingGearState::deployed (C++ enumerator), 133
krpc::services::SpaceCenter::Flight::sideslip_angle (C++ function), 112	krpc::services::SpaceCenter::LandingGearState::deploying (C++ enumerator), 133
krpc::services::SpaceCenter::Flight::speed (C++ function), 110	krpc::services::SpaceCenter::LandingGearState::retracted (C++ enumerator), 133
krpc::services::SpaceCenter::Flight::speed_of_sound (C++ function), 112	krpc::services::SpaceCenter::LandingGearState::retracting (C++ enumerator), 133
krpc::services::SpaceCenter::Flight::stall_fraction (C++ function), 113	krpc::services::SpaceCenter::LandingLeg (C++ class), 133
krpc::services::SpaceCenter::Flight::static_air_temperature (C++ function), 113	krpc::services::SpaceCenter::LandingLeg::deployed (C++ function), 133
krpc::services::SpaceCenter::Flight::static_pressure (C++ function), 111	krpc::services::SpaceCenter::LandingLeg::part (C++ function), 133
krpc::services::SpaceCenter::Flight::surface_altitude (C++ function), 110	krpc::services::SpaceCenter::LandingLeg::set_deployed (C++ function), 133
krpc::services::SpaceCenter::Flight::terminal_velocity (C++ function), 112	krpc::services::SpaceCenter::LandingLeg::state (C++ function), 133
krpc::services::SpaceCenter::Flight::thrust_specific_fuel_consumption (C++ function), 113	krpc::services::SpaceCenter::LandingLegState (C++ enum), 133
krpc::services::SpaceCenter::Flight::total_air_temperature (C++ function), 112	krpc::services::SpaceCenter::LandingLegState::broken (C++ enumerator), 134
krpc::services::SpaceCenter::Flight::velocity (C++ function), 110	krpc::services::SpaceCenter::LandingLegState::deployed (C++ enumerator), 133
krpc::services::SpaceCenter::Flight::vertical_speed (C++ function), 111	krpc::services::SpaceCenter::LandingLegState::deploying (C++ enumerator), 133
krpc::services::SpaceCenter::g (C++ function), 97	krpc::services::SpaceCenter::LandingLegState::repairing (C++ enumerator), 134
krpc::services::SpaceCenter::Intake (C++ class), 132	krpc::services::SpaceCenter::LandingLegState::retracted (C++ enumerator), 133
krpc::services::SpaceCenter::Intake::area (C++ function), 132	krpc::services::SpaceCenter::LandingLegState::retracting (C++ enumerator), 133
krpc::services::SpaceCenter::Intake::flow (C++ function), 132	krpc::services::SpaceCenter::launch_vessel_from_sph (C++ function), 97
krpc::services::SpaceCenter::Intake::open (C++ function), 132	krpc::services::SpaceCenter::launch_vessel_from_vab (C++ function), 97
krpc::services::SpaceCenter::Intake::part (C++ function), 132	krpc::services::SpaceCenter::LaunchClamp (C++ class), 134
krpc::services::SpaceCenter::Intake::set_open (C++ function), 132	krpc::services::SpaceCenter::LaunchClamp::part (C++ function), 134
krpc::services::SpaceCenter::Intake::speed (C++ function), 132	krpc::services::SpaceCenter::LaunchClamp::release (C++ function), 134
krpc::services::SpaceCenter::LandingGear (C++ class), 132	krpc::services::SpaceCenter::Light (C++ class), 134
krpc::services::SpaceCenter::LandingGear::deployable (C++ function), 133	krpc::services::SpaceCenter::Light::active (C++ function), 134
krpc::services::SpaceCenter::LandingGear::deployed (C++ function), 133	krpc::services::SpaceCenter::Light::part (C++ function), 134
krpc::services::SpaceCenter::LandingGear::part (C++ function), 132	krpc::services::SpaceCenter::Light::power_usage (C++ function), 134
krpc::services::SpaceCenter::LandingGear::set_deployed (C++ function), 133	krpc::services::SpaceCenter::Light::set_active (C++ function), 134
krpc::services::SpaceCenter::LandingGear::state (C++ function), 132	krpc::services::SpaceCenter::maximum_rails_warp_factor

(C++ function), 98
 krpc::services::SpaceCenter::Module (C++ class), 125
 krpc::services::SpaceCenter::Module::actions (C++ function), 126
 krpc::services::SpaceCenter::Module::events (C++ function), 126
 krpc::services::SpaceCenter::Module::fields (C++ function), 125
 krpc::services::SpaceCenter::Module::get_field (C++ function), 125
 krpc::services::SpaceCenter::Module::has_action (C++ function), 126
 krpc::services::SpaceCenter::Module::has_event (C++ function), 126
 krpc::services::SpaceCenter::Module::has_field (C++ function), 125
 krpc::services::SpaceCenter::Module::name (C++ function), 125
 krpc::services::SpaceCenter::Module::part (C++ function), 125
 krpc::services::SpaceCenter::Module::set_action (C++ function), 126
 krpc::services::SpaceCenter::Module::trigger_event (C++ function), 126
 krpc::services::SpaceCenter::Node (C++ class), 145
 krpc::services::SpaceCenter::Node::burn_vector (C++ function), 146
 krpc::services::SpaceCenter::Node::delta_v (C++ function), 146
 krpc::services::SpaceCenter::Node::direction (C++ function), 147
 krpc::services::SpaceCenter::Node::normal (C++ function), 146
 krpc::services::SpaceCenter::Node::orbit (C++ function), 147
 krpc::services::SpaceCenter::Node::orbital_reference_frame (C++ function), 147
 krpc::services::SpaceCenter::Node::position (C++ function), 147
 krpc::services::SpaceCenter::Node::prograde (C++ function), 145
 krpc::services::SpaceCenter::Node::radial (C++ function), 146
 krpc::services::SpaceCenter::Node::reference_frame (C++ function), 147
 krpc::services::SpaceCenter::Node::remaining_burn_vector (C++ function), 146
 krpc::services::SpaceCenter::Node::remaining_delta_v (C++ function), 146
 krpc::services::SpaceCenter::Node::remove (C++ function), 147
 krpc::services::SpaceCenter::Node::set_delta_v (C++ function), 146
 krpc::services::SpaceCenter::Node::set_normal (C++ function), 146
 krpc::services::SpaceCenter::Node::set_prograde (C++ function), 146
 krpc::services::SpaceCenter::Node::set_radial (C++ function), 146
 krpc::services::SpaceCenter::Node::set_ut (C++ function), 147
 krpc::services::SpaceCenter::Node::time_to (C++ function), 147
 krpc::services::SpaceCenter::Node::ut (C++ function), 147
 krpc::services::SpaceCenter::Orbit (C++ class), 113
 krpc::services::SpaceCenter::Orbit::apoapsis (C++ function), 113
 krpc::services::SpaceCenter::Orbit::apoapsis_altitude (C++ function), 114
 krpc::services::SpaceCenter::Orbit::argument_of_periapsis (C++ function), 115
 krpc::services::SpaceCenter::Orbit::body (C++ function), 113
 krpc::services::SpaceCenter::Orbit::eccentric_anomaly (C++ function), 115
 krpc::services::SpaceCenter::Orbit::eccentricity (C++ function), 114
 krpc::services::SpaceCenter::Orbit::epoch (C++ function), 115
 krpc::services::SpaceCenter::Orbit::inclination (C++ function), 114
 krpc::services::SpaceCenter::Orbit::longitude_of_ascending_node (C++ function), 115
 krpc::services::SpaceCenter::Orbit::mean_anomaly (C++ function), 115
 krpc::services::SpaceCenter::Orbit::mean_anomaly_at_epoch (C++ function), 115
 krpc::services::SpaceCenter::Orbit::next_orbit (C++ function), 115
 krpc::services::SpaceCenter::Orbit::periapsis (C++ function), 114
 krpc::services::SpaceCenter::Orbit::periapsis_altitude (C++ function), 114
 krpc::services::SpaceCenter::Orbit::period (C++ function), 114
 krpc::services::SpaceCenter::Orbit::radius (C++ function), 114
 krpc::services::SpaceCenter::Orbit::reference_plane_direction (C++ function), 115
 krpc::services::SpaceCenter::Orbit::reference_plane_normal (C++ function), 115
 krpc::services::SpaceCenter::Orbit::semi_major_axis (C++ function), 114
 krpc::services::SpaceCenter::Orbit::semi_minor_axis (C++ function), 114
 krpc::services::SpaceCenter::Orbit::speed (C++ function), 114

<code>krpc::services::SpaceCenter::Orbit::time_to_apoapsis</code> (C++ function), 114	<code>krpc::services::SpaceCenter::Part::dry_mass</code> (C++ function), 122
<code>krpc::services::SpaceCenter::Orbit::time_to_periapsis</code> (C++ function), 114	<code>krpc::services::SpaceCenter::Part::engine</code> (C++ function), 123
<code>krpc::services::SpaceCenter::Orbit::time_to_soi_change</code> (C++ function), 115	<code>krpc::services::SpaceCenter::Part::fairing</code> (C++ function), 123
<code>krpc::services::SpaceCenter::Parachute</code> (C++ class), 134	<code>krpc::services::SpaceCenter::Part::fuel_lines_from</code> (C++ function), 123
<code>krpc::services::SpaceCenter::Parachute::deploy</code> (C++ function), 134	<code>krpc::services::SpaceCenter::Part::fuel_lines_to</code> (C++ function), 123
<code>krpc::services::SpaceCenter::Parachute::deploy_altitude</code> (C++ function), 134	<code>krpc::services::SpaceCenter::Part::impact_tolerance</code> (C++ function), 122
<code>krpc::services::SpaceCenter::Parachute::deploy_min_pressure</code> (C++ function), 134	<code>krpc::services::SpaceCenter::Part::intake</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Parachute::deployed</code> (C++ function), 134	<code>krpc::services::SpaceCenter::Part::is_fuel_line</code> (C++ function), 123
<code>krpc::services::SpaceCenter::Parachute::part</code> (C++ function), 134	<code>krpc::services::SpaceCenter::Part::landing_gear</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Parachute::set_deploy_altitude</code> (C++ function), 134	<code>krpc::services::SpaceCenter::Part::landing_leg</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Parachute::set_deploy_min_pressure</code> (C++ function), 134	<code>krpc::services::SpaceCenter::Part::launch_clamp</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Parachute::state</code> (C++ function), 134	<code>krpc::services::SpaceCenter::Part::light</code> (C++ function), 124
<code>krpc::services::SpaceCenter::ParachuteState</code> (C++ enum), 135	<code>krpc::services::SpaceCenter::Part::mass</code> (C++ function), 122
<code>krpc::services::SpaceCenter::ParachuteState::active</code> (C++ enumerator), 135	<code>krpc::services::SpaceCenter::Part::massless</code> (C++ function), 122
<code>krpc::services::SpaceCenter::ParachuteState::cut</code> (C++ enumerator), 135	<code>krpc::services::SpaceCenter::Part::max_skin_temperature</code> (C++ function), 122
<code>krpc::services::SpaceCenter::ParachuteState::deployed</code> (C++ enumerator), 135	<code>krpc::services::SpaceCenter::Part::max_temperature</code> (C++ function), 122
<code>krpc::services::SpaceCenter::ParachuteState::semi_deployed</code> (C++ enumerator), 135	<code>krpc::services::SpaceCenter::Part::modules</code> (C++ function), 123
<code>krpc::services::SpaceCenter::ParachuteState::stowed</code> (C++ enumerator), 135	<code>krpc::services::SpaceCenter::Part::name</code> (C++ function), 121
<code>krpc::services::SpaceCenter::Part</code> (C++ class), 121	<code>krpc::services::SpaceCenter::Part::parachute</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Part::axially_attached</code> (C++ function), 121	<code>krpc::services::SpaceCenter::Part::parent</code> (C++ function), 121
<code>krpc::services::SpaceCenter::Part::cargo_bay</code> (C++ function), 123	<code>krpc::services::SpaceCenter::Part::position</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Part::children</code> (C++ function), 121	<code>krpc::services::SpaceCenter::Part::radially_attached</code> (C++ function), 122
<code>krpc::services::SpaceCenter::Part::cost</code> (C++ function), 121	<code>krpc::services::SpaceCenter::Part::radiator</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Part::crossfeed</code> (C++ function), 123	<code>krpc::services::SpaceCenter::Part::reaction_wheel</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Part::decouple_stage</code> (C++ function), 122	<code>krpc::services::SpaceCenter::Part::reference_frame</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Part::decoupler</code> (C++ function), 123	<code>krpc::services::SpaceCenter::Part::resource_converter</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Part::direction</code> (C++ function), 124	<code>krpc::services::SpaceCenter::Part::resource_harvester</code> (C++ function), 124
<code>krpc::services::SpaceCenter::Part::docking_port</code> (C++ function), 123	

krpc::services::SpaceCenter::Part::resources (C++ function), 123
 krpc::services::SpaceCenter::Part::rotation (C++ function), 124
 krpc::services::SpaceCenter::Part::sensor (C++ function), 124
 krpc::services::SpaceCenter::Part::skin_temperature (C++ function), 122
 krpc::services::SpaceCenter::Part::solar_panel (C++ function), 124
 krpc::services::SpaceCenter::Part::stage (C++ function), 122
 krpc::services::SpaceCenter::Part::temperature (C++ function), 122
 krpc::services::SpaceCenter::Part::thermal_conduction_flux (C++ function), 122
 krpc::services::SpaceCenter::Part::thermal_convection_flux (C++ function), 123
 krpc::services::SpaceCenter::Part::thermal_internal_flux (C++ function), 123
 krpc::services::SpaceCenter::Part::thermal_mass (C++ function), 122
 krpc::services::SpaceCenter::Part::thermal_radiation_flux (C++ function), 123
 krpc::services::SpaceCenter::Part::thermal_resource_mass (C++ function), 122
 krpc::services::SpaceCenter::Part::thermal_skin_mass (C++ function), 122
 krpc::services::SpaceCenter::Part::thermal_skin_to_internal_flux (C++ function), 123
 krpc::services::SpaceCenter::Part::title (C++ function), 121
 krpc::services::SpaceCenter::Part::velocity (C++ function), 124
 krpc::services::SpaceCenter::Part::vessel (C++ function), 121
 krpc::services::SpaceCenter::Parts (C++ class), 119
 krpc::services::SpaceCenter::Parts::all (C++ function), 119
 krpc::services::SpaceCenter::Parts::cargo_bays (C++ function), 120
 krpc::services::SpaceCenter::Parts::controlling (C++ function), 119
 krpc::services::SpaceCenter::Parts::decouplers (C++ function), 120
 krpc::services::SpaceCenter::Parts::docking_port_with_name (C++ function), 120
 krpc::services::SpaceCenter::Parts::docking_ports (C++ function), 120
 krpc::services::SpaceCenter::Parts::engines (C++ function), 120
 krpc::services::SpaceCenter::Parts::fairings (C++ function), 120
 krpc::services::SpaceCenter::Parts::in_decouple_stage (C++ function), 120
 krpc::services::SpaceCenter::Parts::in_stage (C++ function), 120
 krpc::services::SpaceCenter::Parts::intakes (C++ function), 120
 krpc::services::SpaceCenter::Parts::landing_gear (C++ function), 120
 krpc::services::SpaceCenter::Parts::landing_legs (C++ function), 120
 krpc::services::SpaceCenter::Parts::launch_clamps (C++ function), 120
 krpc::services::SpaceCenter::Parts::lights (C++ function), 120
 krpc::services::SpaceCenter::Parts::modules_with_name (C++ function), 120
 krpc::services::SpaceCenter::Parts::parachutes (C++ function), 121
 krpc::services::SpaceCenter::Parts::radiators (C++ function), 121
 krpc::services::SpaceCenter::Parts::reaction_wheels (C++ function), 121
 krpc::services::SpaceCenter::Parts::resource_converters (C++ function), 121
 krpc::services::SpaceCenter::Parts::resource_harvesters (C++ function), 121
 krpc::services::SpaceCenter::Parts::root (C++ function), 119
 krpc::services::SpaceCenter::Parts::sensors (C++ function), 121
 krpc::services::SpaceCenter::Parts::set_controlling (C++ function), 119
 krpc::services::SpaceCenter::Parts::solar_panels (C++ function), 121
 krpc::services::SpaceCenter::Parts::with_module (C++ function), 120
 krpc::services::SpaceCenter::Parts::with_name (C++ function), 119
 krpc::services::SpaceCenter::Parts::with_title (C++ function), 119
 krpc::services::SpaceCenter::physics_warp_factor (C++ function), 97
 krpc::services::SpaceCenter::Radiator (C++ class), 135
 krpc::services::SpaceCenter::Radiator::deployable (C++ function), 135
 krpc::services::SpaceCenter::Radiator::deployed (C++ function), 135
 krpc::services::SpaceCenter::Radiator::part (C++ function), 135
 krpc::services::SpaceCenter::Radiator::set_deployed (C++ function), 135
 krpc::services::SpaceCenter::Radiator::state (C++ function), 135
 krpc::services::SpaceCenter::RadiatorState (C++ enum), 135

krpc::services::SpaceCenter::RadiatorState::broken (C++ enumerator), 135	krpc::services::SpaceCenter::ResourceConverterState (C++ enum), 136
krpc::services::SpaceCenter::RadiatorState::extended (C++ enumerator), 135	krpc::services::SpaceCenter::ResourceConverterState::capacity (C++ enumerator), 137
krpc::services::SpaceCenter::RadiatorState::extending (C++ enumerator), 135	krpc::services::SpaceCenter::ResourceConverterState::idle (C++ enumerator), 137
krpc::services::SpaceCenter::RadiatorState::retracted (C++ enumerator), 135	krpc::services::SpaceCenter::ResourceConverterState::missing_resource (C++ enumerator), 137
krpc::services::SpaceCenter::RadiatorState::retracting (C++ enumerator), 135	krpc::services::SpaceCenter::ResourceConverterState::running (C++ enumerator), 137
krpc::services::SpaceCenter::rails_warp_factor (C++ function), 97	krpc::services::SpaceCenter::ResourceConverterState::storage_full (C++ enumerator), 137
krpc::services::SpaceCenter::ReactionWheel (C++ class), 138	krpc::services::SpaceCenter::ResourceConverterState::unknown (C++ enumerator), 137
krpc::services::SpaceCenter::ReactionWheel::active (C++ function), 138	krpc::services::SpaceCenter::ResourceFlowMode (C++ enum), 145
krpc::services::SpaceCenter::ReactionWheel::broken (C++ function), 138	krpc::services::SpaceCenter::ResourceFlowMode::adjacent (C++ enumerator), 145
krpc::services::SpaceCenter::ReactionWheel::part (C++ function), 138	krpc::services::SpaceCenter::ResourceFlowMode::none (C++ enumerator), 145
krpc::services::SpaceCenter::ReactionWheel::pitch_torque (C++ function), 138	krpc::services::SpaceCenter::ResourceFlowMode::stage (C++ enumerator), 145
krpc::services::SpaceCenter::ReactionWheel::roll_torque (C++ function), 138	krpc::services::SpaceCenter::ResourceFlowMode::vessel (C++ enumerator), 145
krpc::services::SpaceCenter::ReactionWheel::set_active (C++ function), 138	krpc::services::SpaceCenter::ResourceHarvester (C++ class), 137
krpc::services::SpaceCenter::ReactionWheel::yaw_torque (C++ function), 138	krpc::services::SpaceCenter::ResourceHarvester::active (C++ function), 137
krpc::services::SpaceCenter::ReferenceFrame (C++ class), 148	krpc::services::SpaceCenter::ResourceHarvester::core_temperature (C++ function), 137
krpc::services::SpaceCenter::remote_tech_available (C++ function), 99	krpc::services::SpaceCenter::ResourceHarvester::deployed (C++ function), 137
krpc::services::SpaceCenter::ResourceConverter (C++ class), 136	krpc::services::SpaceCenter::ResourceHarvester::extraction_rate (C++ function), 137
krpc::services::SpaceCenter::ResourceConverter::active (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvester::optimum_core_temperature (C++ function), 137
krpc::services::SpaceCenter::ResourceConverter::count (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvester::part (C++ function), 137
krpc::services::SpaceCenter::ResourceConverter::inputs (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvester::set_active (C++ function), 137
krpc::services::SpaceCenter::ResourceConverter::name (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvester::set_deployed (C++ function), 137
krpc::services::SpaceCenter::ResourceConverter::outputs (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvester::state (C++ function), 137
krpc::services::SpaceCenter::ResourceConverter::part (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvester::thermal_efficiency (C++ function), 137
krpc::services::SpaceCenter::ResourceConverter::start (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvesterState (C++ enum), 137
krpc::services::SpaceCenter::ResourceConverter::state (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvesterState::active (C++ enumerator), 138
krpc::services::SpaceCenter::ResourceConverter::status_info (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvesterState::deployed (C++ enumerator), 137
krpc::services::SpaceCenter::ResourceConverter::stop (C++ function), 136	krpc::services::SpaceCenter::ResourceHarvesterState::deploying (C++ enumerator), 137

krpc::services::SpaceCenter::ResourceHarvesterState::retracted (C++ enumerator), 138
 krpc::services::SpaceCenter::ResourceHarvesterState::retracting (C++ enumerator), 138
 krpc::services::SpaceCenter::Resources (C++ class), 144
 krpc::services::SpaceCenter::Resources::amount (C++ function), 145
 krpc::services::SpaceCenter::Resources::density (C++ function), 145
 krpc::services::SpaceCenter::Resources::flow_mode (C++ function), 145
 krpc::services::SpaceCenter::Resources::has_resource (C++ function), 144
 krpc::services::SpaceCenter::Resources::max (C++ function), 145
 krpc::services::SpaceCenter::Resources::names (C++ function), 144
 krpc::services::SpaceCenter::SASMode (C++ enum), 118
 krpc::services::SpaceCenter::SASMode::anti_normal (C++ enumerator), 118
 krpc::services::SpaceCenter::SASMode::anti_radial (C++ enumerator), 118
 krpc::services::SpaceCenter::SASMode::anti_target (C++ enumerator), 118
 krpc::services::SpaceCenter::SASMode::maneuver (C++ enumerator), 118
 krpc::services::SpaceCenter::SASMode::normal (C++ enumerator), 118
 krpc::services::SpaceCenter::SASMode::prograde (C++ enumerator), 118
 krpc::services::SpaceCenter::SASMode::radial (C++ enumerator), 118
 krpc::services::SpaceCenter::SASMode::retrograde (C++ enumerator), 118
 krpc::services::SpaceCenter::SASMode::stability_assist (C++ enumerator), 118
 krpc::services::SpaceCenter::SASMode::target (C++ enumerator), 118
 krpc::services::SpaceCenter::Sensor (C++ class), 138
 krpc::services::SpaceCenter::Sensor::active (C++ function), 138
 krpc::services::SpaceCenter::Sensor::part (C++ function), 138
 krpc::services::SpaceCenter::Sensor::power_usage (C++ function), 138
 krpc::services::SpaceCenter::Sensor::set_active (C++ function), 138
 krpc::services::SpaceCenter::Sensor::value (C++ function), 138
 krpc::services::SpaceCenter::set_active_vessel (C++ function), 96
 krpc::services::SpaceCenter::set_physics_warp_factor (C++ function), 97
 krpc::services::SpaceCenter::set_rails_warp_factor (C++ function), 97
 krpc::services::SpaceCenter::set_target_body (C++ function), 96
 krpc::services::SpaceCenter::set_target_docking_port (C++ function), 97
 krpc::services::SpaceCenter::set_target_vessel (C++ function), 96
 krpc::services::SpaceCenter::SolarPanel (C++ class), 139
 krpc::services::SpaceCenter::SolarPanel::deployed (C++ function), 139
 krpc::services::SpaceCenter::SolarPanel::energy_flow (C++ function), 139
 krpc::services::SpaceCenter::SolarPanel::part (C++ function), 139
 krpc::services::SpaceCenter::SolarPanel::set_deployed (C++ function), 139
 krpc::services::SpaceCenter::SolarPanel::state (C++ function), 139
 krpc::services::SpaceCenter::SolarPanel::sun_exposure (C++ function), 139
 krpc::services::SpaceCenter::SolarPanelState (C++ enum), 139
 krpc::services::SpaceCenter::SolarPanelState::broken (C++ enumerator), 139
 krpc::services::SpaceCenter::SolarPanelState::extended (C++ enumerator), 139
 krpc::services::SpaceCenter::SolarPanelState::extending (C++ enumerator), 139
 krpc::services::SpaceCenter::SolarPanelState::retracted (C++ enumerator), 139
 krpc::services::SpaceCenter::SolarPanelState::retracting (C++ enumerator), 139
 krpc::services::SpaceCenter::SpaceCenter (C++ function), 96
 krpc::services::SpaceCenter::SpeedMode (C++ enum), 118
 krpc::services::SpaceCenter::SpeedMode::orbit (C++ enumerator), 118
 krpc::services::SpaceCenter::SpeedMode::surface (C++ enumerator), 118
 krpc::services::SpaceCenter::SpeedMode::target (C++ enumerator), 118
 krpc::services::SpaceCenter::target_body (C++ function), 96
 krpc::services::SpaceCenter::target_docking_port (C++ function), 97
 krpc::services::SpaceCenter::target_vessel (C++ function), 96
 krpc::services::SpaceCenter::transform_direction (C++ function), 98
 krpc::services::SpaceCenter::transform_position (C++ function), 98
 krpc::services::SpaceCenter::transform_rotation (C++ function), 98

krpc::services::SpaceCenter::transform_velocity (C++ function), 99
 krpc::services::SpaceCenter::ut (C++ function), 97
 krpc::services::SpaceCenter::Vessel (C++ class), 100
 krpc::services::SpaceCenter::Vessel::angular_velocity (C++ function), 106
 krpc::services::SpaceCenter::Vessel::auto_pilot (C++ function), 100
 krpc::services::SpaceCenter::Vessel::available_thrust (C++ function), 101
 krpc::services::SpaceCenter::Vessel::comms (C++ function), 101
 krpc::services::SpaceCenter::Vessel::control (C++ function), 100
 krpc::services::SpaceCenter::Vessel::direction (C++ function), 106
 krpc::services::SpaceCenter::Vessel::dry_mass (C++ function), 101
 krpc::services::SpaceCenter::Vessel::flight (C++ function), 100
 krpc::services::SpaceCenter::Vessel::kerbin_sea_level_specific_impulse (C++ function), 102
 krpc::services::SpaceCenter::Vessel::mass (C++ function), 101
 krpc::services::SpaceCenter::Vessel::max_thrust (C++ function), 101
 krpc::services::SpaceCenter::Vessel::max_vacuum_thrust (C++ function), 101
 krpc::services::SpaceCenter::Vessel::met (C++ function), 100
 krpc::services::SpaceCenter::Vessel::name (C++ function), 100
 krpc::services::SpaceCenter::Vessel::orbit (C++ function), 100
 krpc::services::SpaceCenter::Vessel::orbital_reference_frame (C++ function), 102
 krpc::services::SpaceCenter::Vessel::parts (C++ function), 101
 krpc::services::SpaceCenter::Vessel::position (C++ function), 104
 krpc::services::SpaceCenter::Vessel::reference_frame (C++ function), 102
 krpc::services::SpaceCenter::Vessel::resources (C++ function), 101
 krpc::services::SpaceCenter::Vessel::resources_in_decouple_stage (C++ function), 101
 krpc::services::SpaceCenter::Vessel::rotation (C++ function), 106
 krpc::services::SpaceCenter::Vessel::set_name (C++ function), 100
 krpc::services::SpaceCenter::Vessel::set_target (C++ function), 100
 krpc::services::SpaceCenter::Vessel::set_type (C++ function), 100
 krpc::services::SpaceCenter::Vessel::situation (C++ function), 100
 krpc::services::SpaceCenter::Vessel::specific_impulse (C++ function), 101
 krpc::services::SpaceCenter::Vessel::surface_reference_frame (C++ function), 102
 krpc::services::SpaceCenter::Vessel::surface_velocity_reference_frame (C++ function), 104
 krpc::services::SpaceCenter::Vessel::target (C++ function), 100
 krpc::services::SpaceCenter::Vessel::thrust (C++ function), 101
 krpc::services::SpaceCenter::Vessel::type (C++ function), 100
 krpc::services::SpaceCenter::Vessel::vacuum_specific_impulse (C++ function), 101
 krpc::services::SpaceCenter::Vessel::velocity (C++ function), 106
 krpc::services::SpaceCenter::vessels (C++ function), 96
 krpc::services::SpaceCenter::VesselSituation (C++ enum), 106
 krpc::services::SpaceCenter::VesselSituation::docked (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselSituation::escaping (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselSituation::flying (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselSituation::landed (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselSituation::orbiting (C++ enumerator), 107
 krpc::services::SpaceCenter::VesselSituation::pre_launch (C++ enumerator), 107
 krpc::services::SpaceCenter::VesselSituation::splashed (C++ enumerator), 107
 krpc::services::SpaceCenter::VesselSituation::sub_orbital (C++ enumerator), 107
 krpc::services::SpaceCenter::VesselType (C++ enum), 106
 krpc::services::SpaceCenter::VesselType::base (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselType::debris (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselType::lander (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselType::probe (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselType::rover (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselType::ship (C++ enumerator), 106
 krpc::services::SpaceCenter::VesselType::station (C++ enumerator), 106
 krpc::services::SpaceCenter::warp_factor (C++ function),

97
 krpc::services::SpaceCenter::warp_mode (C++ function), 97
 krpc::services::SpaceCenter::warp_rate (C++ function), 97
 krpc::services::SpaceCenter::warp_to (C++ function), 98
 krpc::services::SpaceCenter::WarpMode (C++ enum), 99
 krpc::services::SpaceCenter::WarpMode::none (C++ enumerator), 100
 krpc::services::SpaceCenter::WarpMode::physics (C++ enumerator), 100
 krpc::services::SpaceCenter::WarpMode::rails (C++ enumerator), 100
 krpc::Stream<T> (C++ class), 95
 krpc::Stream<T>::operator() (C++ function), 95
 krpc::Stream<T>::remove (C++ function), 95

L

LANDED (Java field), 176
 LANDER (Java field), 175
 landing_gear (Part attribute), 272, 357
 landing_gear (Parts attribute), 266, 351
 landing_leg (Part attribute), 272, 357
 landing_legs (Parts attribute), 266, 351
 LandingGear (class in SpaceCenter), 284, 369
 LandingGear (Java class), 202
 LandingGearState (class in SpaceCenter), 285, 370
 LandingGearState (Java enum), 203
 LandingGearState.deployed (in module SpaceCenter), 285, 370
 LandingGearState.deploying (in module SpaceCenter), 285, 370
 LandingGearState.retracted (in module SpaceCenter), 285, 370
 LandingGearState.retracting (in module SpaceCenter), 285, 370
 LandingLeg (class in SpaceCenter), 285, 370
 LandingLeg (Java class), 203
 LandingLegState (class in SpaceCenter), 285, 370
 LandingLegState (Java enum), 203
 LandingLegState.broken (in module SpaceCenter), 286, 371
 LandingLegState.deployed (in module SpaceCenter), 286, 370
 LandingLegState.deploying (in module SpaceCenter), 286, 371
 LandingLegState.repairing (in module SpaceCenter), 286, 371
 LandingLegState.retracted (in module SpaceCenter), 286, 370
 LandingLegState.retracting (in module SpaceCenter), 286, 371
 latitude (Flight attribute), 251, 336
 launch_clamp (Part attribute), 272, 357

launch_clamps (Parts attribute), 266, 351
 LAUNCH_RENDEVOUS (Java field), 228
 launch_vessel_from_sph() (in module SpaceCenter), 234, 320
 launch_vessel_from_vab() (in module SpaceCenter), 234, 319
 LaunchClamp (class in SpaceCenter), 286, 371
 LaunchClamp (Java class), 203
 launchVesselFromSPH(String) (Java method), 166
 launchVesselFromVAB(String) (Java method), 165
 lift (Flight attribute), 254, 339
 lift_coefficient (Flight attribute), 256, 341
 Light (class in SpaceCenter), 286, 371
 Light (Java class), 204
 light (Part attribute), 272, 357
 lights (Control attribute), 260, 345
 lights (Parts attribute), 266, 351
 longitude (Flight attribute), 251, 336
 longitude_of_ascending_node (Orbit attribute), 258, 343

M

mach (Flight attribute), 254, 339
 MANEUVER (Java field), 187, 228
 MANEUVER_AUTO (Java field), 228
 margin (Alarm attribute), 311, 396
 mass (CelestialBody attribute), 247, 332
 mass (Part attribute), 269, 354
 mass (Vessel attribute), 240, 325
 massless (Part attribute), 269, 354
 max() (Resources method), 298, 383
 max(String) (Java method), 214
 max_config_position (Servo attribute), 308, 392
 max_position (Servo attribute), 308, 393
 max_roll_speed (AutoPilot attribute), 305, 389
 max_rotation_speed (AutoPilot attribute), 304, 389
 max_skin_temperature (Part attribute), 269, 354
 max_temperature (Part attribute), 269, 354
 max_thrust (Engine attribute), 281, 366
 max_thrust (Vessel attribute), 240, 325
 max_vacuum_thrust (Engine attribute), 281, 366
 max_vacuum_thrust (Vessel attribute), 240, 325
 maximum_rails_warp_factor (in module SpaceCenter), 235, 321
 mean_altitude (Flight attribute), 251, 336
 mean_anomaly (Orbit attribute), 259, 344
 mean_anomaly_at_epoch (Orbit attribute), 258, 343
 MESSAGE_ONLY (Java field), 229
 met (Vessel attribute), 238, 323
 min_config_position (Servo attribute), 308, 392
 min_position (Servo attribute), 308, 392
 MISSING_RESOURCE (Java field), 206
 mode (Engine attribute), 282, 367
 modes (Engine attribute), 282, 367
 Module (class in SpaceCenter), 274, 359

Module (Java class), 194
modules (Part attribute), 271, 356
modules_with_name() (Parts method), 265, 350
modulesWithName(String) (Java method), 189
move_center() (ControlGroup method), 307, 392
move_center() (Servo method), 309, 394
move_left() (ControlGroup method), 307, 391
move_left() (Servo method), 309, 394
move_next_preset() (ControlGroup method), 307, 392
move_next_preset() (Servo method), 309, 394
move_prev_preset() (ControlGroup method), 307, 392
move_prev_preset() (Servo method), 309, 394
move_right() (ControlGroup method), 307, 391
move_right() (Servo method), 309, 394
move_to() (Servo method), 309, 394
moveCenter() (Java method), 223, 224
moveLeft() (Java method), 223, 224
moveNextPreset() (Java method), 223, 224
movePrevPreset() (Java method), 223, 224
moveRight() (Java method), 223, 224
moveTo(float, float) (Java method), 225
MOVING (Java field), 200
msl_position() (CelestialBody method), 248, 333
mSLPosition(double, double, ReferenceFrame) (Java method), 177

N

name (Alarm attribute), 312, 396
name (CelestialBody attribute), 246, 331
name (ControlGroup attribute), 306, 391
name (DockingPort attribute), 277, 362
name (Module attribute), 274, 359
name (Part attribute), 267, 352
name (Servo attribute), 307, 392
name (Vessel attribute), 238, 323
name() (ResourceConverter method), 289, 374
name(int) (Java method), 205
names (Resources attribute), 297, 382
newInstance() (Java method), 163
newInstance(String) (Java method), 163
newInstance(String, java.net.InetAddress) (Java method), 163
newInstance(String, java.net.InetAddress, int, int) (Java method), 163
newInstance(String, String) (Java method), 163
newInstance(String, String, int, int) (Java method), 163
next_orbit (Orbit attribute), 259, 344
Node (class in SpaceCenter), 299, 383
Node (Java class), 215
nodes (Control attribute), 262, 347
non_rotating_reference_frame (CelestialBody attribute), 249, 334
NONE (Java field), 169, 215
normal (Flight attribute), 253, 338

NORMAL (Java field), 187
normal (Node attribute), 299, 384
notes (Alarm attribute), 312, 396

O

open (CargoBay attribute), 276, 361
open (Intake attribute), 284, 369
OPEN (Java field), 197
OPENING (Java field), 197
optimum_core_temperature (ResourceHarvester attribute), 291, 375
orbit (CelestialBody attribute), 247, 332
Orbit (class in SpaceCenter), 256, 341
Orbit (Java class), 183
ORBIT (Java field), 187
orbit (Node attribute), 300, 385
orbit (Vessel attribute), 239, 324
orbital_reference_frame (CelestialBody attribute), 250, 335
orbital_reference_frame (Node attribute), 300, 385
orbital_reference_frame (Vessel attribute), 241, 326
ORBITING (Java field), 176
outputs() (ResourceConverter method), 289, 374
outputs(int) (Java method), 206

P

Parachute (class in SpaceCenter), 287, 371
Parachute (Java class), 204
parachute (Part attribute), 272, 357
parachutes (Parts attribute), 266, 351
ParachuteState (class in SpaceCenter), 287, 372
ParachuteState (Java enum), 204
ParachuteState.active (in module SpaceCenter), 287, 372
ParachuteState.cut (in module SpaceCenter), 287, 372
ParachuteState.deployed (in module SpaceCenter), 287, 372
ParachuteState.semi_deployed (in module SpaceCenter), 287, 372
ParachuteState.stowed (in module SpaceCenter), 287, 372
parent (Part attribute), 268, 353
part (CargoBay attribute), 276, 361
Part (class in SpaceCenter), 267, 352
part (Decoupler attribute), 277, 362
part (DockingPort attribute), 277, 362
part (Engine attribute), 280, 365
part (Fairing attribute), 283, 368
part (Intake attribute), 284, 369
Part (Java class), 190
part (LandingGear attribute), 284, 369
part (LandingLeg attribute), 285, 370
part (LaunchClamp attribute), 286, 371
part (Light attribute), 286, 371
part (Module attribute), 275, 360
part (Parachute attribute), 287, 371

part (Radiator attribute), 288, 372
 part (ReactionWheel attribute), 291, 376
 part (ResourceConverter attribute), 288, 373
 part (ResourceHarvester attribute), 290, 375
 part (Sensor attribute), 292, 377
 part (SolarPanel attribute), 292, 377
 Parts (class in SpaceCenter), 264, 349
 Parts (Java class), 188
 parts (Vessel attribute), 239, 324
 PAUSE_GAME (Java field), 229
 PERIAPSIS (Java field), 228
 periapsis (Orbit attribute), 257, 342
 periapsis_altitude (Orbit attribute), 257, 342
 period (Orbit attribute), 258, 343
 PHYSICS (Java field), 169
 physics_warp_factor (in module SpaceCenter), 235, 320
 pitch (Control attribute), 261, 346
 pitch (Flight attribute), 252, 337
 pitch_torque (ReactionWheel attribute), 291, 376
 position (Servo attribute), 308, 392
 position() (CelestialBody method), 250, 335
 position() (DockingPort method), 278, 363
 position() (Node method), 301, 386
 position() (Part method), 273, 358
 position() (Vessel method), 244, 329
 position(ReferenceFrame) (Java method), 173, 179, 194, 198, 217
 power_usage (Light attribute), 286, 371
 power_usage (Sensor attribute), 292, 377
 PRE_LAUNCH (Java field), 176
 PROBE (Java field), 175
 prograde (Flight attribute), 252, 337
 PROGRADE (Java field), 187
 prograde (Node attribute), 299, 383
 propellant_ratios (Engine attribute), 282, 367
 propellants (Engine attribute), 281, 366

Q

Quaternion (C++ class), 151
 Quaternion (class in SpaceCenter), 305, 390

R

radial (Flight attribute), 253, 338
 RADIAL (Java field), 187
 radial (Node attribute), 299, 384
 radially_attached (Part attribute), 268, 353
 Radiator (class in SpaceCenter), 288, 372
 Radiator (Java class), 205
 radiator (Part attribute), 273, 358
 radiators (Parts attribute), 267, 351
 RadiatorState (class in SpaceCenter), 288, 373
 RadiatorState (Java enum), 205
 RadiatorState.broken (in module SpaceCenter), 288, 373

RadiatorState.extended (in module SpaceCenter), 288, 373
 RadiatorState.extending (in module SpaceCenter), 288, 373
 RadiatorState.retracted (in module SpaceCenter), 288, 373
 RadiatorState.retracting (in module SpaceCenter), 288, 373
 radius (Orbit attribute), 257, 342
 RAILS (Java field), 169
 rails_warp_factor (in module SpaceCenter), 235, 320
 RAW (Java field), 227
 rcs (Control attribute), 260, 345
 reaction_wheel (Part attribute), 273, 358
 reaction_wheels (Parts attribute), 267, 352
 ReactionWheel (class in SpaceCenter), 291, 376
 ReactionWheel (Java class), 208
 READY (Java field), 198
 reengage_distance (DockingPort attribute), 278, 363
 reference_frame (AutoPilot attribute), 303, 388
 reference_frame (CelestialBody attribute), 249, 334
 reference_frame (DockingPort attribute), 279, 363
 reference_frame (Node attribute), 300, 385
 reference_frame (Part attribute), 274, 359
 reference_frame (Vessel attribute), 241, 326
 reference_plane_direction() (Orbit static method), 259, 344
 reference_plane_normal() (Orbit static method), 259, 344
 ReferenceFrame (class in SpaceCenter), 302, 387
 ReferenceFrame (Java class), 218
 referencePlaneDirection(ReferenceFrame) (Java method), 184
 referencePlaneNormal(ReferenceFrame) (Java method), 184
 release() (Java method), 204
 release() (LaunchClamp method), 286, 371
 remaining (Alarm attribute), 312, 396
 remaining_burn_vector() (Node method), 300, 384
 remaining_delta_v (Node attribute), 299, 384
 remainingBurnVector(ReferenceFrame) (Java method), 216
 remote_tech_available (in module SpaceCenter), 237, 322
 RemoteObject (Java class), 164
 remove() (Alarm method), 312, 397
 remove() (Java method), 164, 217, 227
 remove() (Node method), 300, 385
 remove() (Stream method), 318
 remove_nodes() (Control method), 262, 347
 remove_stream() (in module KRPC), 233, 318
 removeNodes() (Java method), 187
 removeStream(int) (Java method), 164
 REPAIRING (Java field), 203
 repeat (Alarm attribute), 312, 396
 repeat_period (Alarm attribute), 312, 396

- resource_converter (Part attribute), 273, 358
 - resource_converters (Parts attribute), 267, 352
 - resource_harvester (Part attribute), 273, 358
 - resource_harvesters (Parts attribute), 267, 352
 - ResourceConverter (class in SpaceCenter), 288, 373
 - ResourceConverter (Java class), 205
 - ResourceConverterState (class in SpaceCenter), 289, 374
 - ResourceConverterState (Java enum), 206
 - ResourceConverterState.capacity (in module SpaceCenter), 290, 375
 - ResourceConverterState.idle (in module SpaceCenter), 289, 374
 - ResourceConverterState.missing_resource (in module SpaceCenter), 290, 374
 - ResourceConverterState.running (in module SpaceCenter), 289, 374
 - ResourceConverterState.storage_full (in module SpaceCenter), 290, 374
 - ResourceConverterState.unknown (in module SpaceCenter), 290, 375
 - ResourceFlowMode (class in SpaceCenter), 298, 383
 - ResourceFlowMode (Java enum), 215
 - ResourceFlowMode.adjacent (in module SpaceCenter), 298, 383
 - ResourceFlowMode.none (in module SpaceCenter), 298, 383
 - ResourceFlowMode.stage (in module SpaceCenter), 298, 383
 - ResourceFlowMode.vessel (in module SpaceCenter), 298, 383
 - ResourceHarvester (class in SpaceCenter), 290, 375
 - ResourceHarvester (Java class), 207
 - ResourceHarvesterState (class in SpaceCenter), 291, 376
 - ResourceHarvesterState (Java enum), 207
 - ResourceHarvesterState.active (in module SpaceCenter), 291, 376
 - ResourceHarvesterState.deployed (in module SpaceCenter), 291, 376
 - ResourceHarvesterState.deploying (in module SpaceCenter), 291, 376
 - ResourceHarvesterState.retracted (in module SpaceCenter), 291, 376
 - ResourceHarvesterState.retracting (in module SpaceCenter), 291, 376
 - Resources (class in SpaceCenter), 297, 382
 - Resources (Java class), 214
 - resources (Part attribute), 271, 356
 - resources (Vessel attribute), 239, 324
 - resources_in_decouple_stage() (Vessel method), 239, 324
 - resourcesInDecoupleStage(int, boolean) (Java method), 170
 - RETRACTED (Java field), 203, 205, 207, 209
 - RETRACTING (Java field), 203, 205, 207, 209
 - retrograde (Flight attribute), 253, 338
 - RETROGRADE (Java field), 187
 - reverse_key (ControlGroup attribute), 306, 391
 - right (Control attribute), 261, 346
 - roll (Control attribute), 261, 346
 - roll (Flight attribute), 252, 337
 - roll_error (AutoPilot attribute), 303, 388
 - roll_speed_multiplier (AutoPilot attribute), 304, 389
 - roll_torque (ReactionWheel attribute), 292, 376
 - root (Parts attribute), 264, 349
 - rotation (Flight attribute), 252, 337
 - rotation() (CelestialBody method), 250, 335
 - rotation() (DockingPort method), 278, 363
 - rotation() (Part method), 273, 358
 - rotation() (Vessel method), 245, 330
 - rotation(ReferenceFrame) (Java method), 175, 179, 194, 198
 - rotation_speed_multiplier (AutoPilot attribute), 304, 389
 - rotational_period (CelestialBody attribute), 247, 332
 - rotational_speed (CelestialBody attribute), 247, 332
 - ROVER (Java field), 175
 - RUNNING (Java field), 206
- ## S
- sas (AutoPilot attribute), 304, 389
 - sas (Control attribute), 260, 345
 - sas_mode (AutoPilot attribute), 304, 389
 - sas_mode (Control attribute), 260, 345
 - SASMode (class in SpaceCenter), 262, 347
 - SASMode (Java enum), 187
 - SASMode.anti_normal (in module SpaceCenter), 263, 348
 - SASMode.anti_radial (in module SpaceCenter), 263, 348
 - SASMode.anti_target (in module SpaceCenter), 263, 348
 - SASMode.maneuver (in module SpaceCenter), 262, 347
 - SASMode.normal (in module SpaceCenter), 263, 348
 - SASMode.prograde (in module SpaceCenter), 263, 348
 - SASMode.radial (in module SpaceCenter), 263, 348
 - SASMode.retrograde (in module SpaceCenter), 263, 348
 - SASMode.stability_assist (in module SpaceCenter), 262, 347
 - SASMode.target (in module SpaceCenter), 263, 348
 - satellites (CelestialBody attribute), 246, 331
 - SEMI_DEPLOYED (Java field), 204
 - semi_major_axis (Orbit attribute), 257, 342
 - semi_minor_axis (Orbit attribute), 257, 342
 - Sensor (class in SpaceCenter), 292, 377
 - Sensor (Java class), 208
 - sensor (Part attribute), 273, 358
 - sensors (Parts attribute), 267, 352
 - Servo (class in InfernalRobotics), 307, 392
 - Servo (Java class), 223
 - servo_group_with_name() (in module InfernalRobotics), 305, 390
 - servo_groups (in module InfernalRobotics), 305, 390

- [servo_with_name\(\)](#) (ControlGroup method), 307, 391
[servo_with_name\(\)](#) (in module `InfernalRobotics`), 306, 390
[servoGroupWithName\(String\)](#) (Java method), 221
[servos](#) (ControlGroup attribute), 307, 391
[servoWithName\(String\)](#) (Java method), 222
[set_action\(\)](#) (Module method), 275, 360
[set_action_group\(\)](#) (Control method), 262, 347
[set_pid_parameters\(\)](#) (AutoPilot method), 305, 389
[setAbort\(boolean\)](#) (Java method), 185
[setAcceleration\(float\)](#) (Java method), 224
[setAction\(AlarmAction\)](#) (Java method), 226
[setAction\(String, boolean\)](#) (Java method), 196
[setActionGroup\(int, boolean\)](#) (Java method), 186
[setActive\(boolean\)](#) (Java method), 200, 204, 207, 208
[setActiveVessel\(Vessel\)](#) (Java method), 165
[setAutoModeSwitch\(boolean\)](#) (Java method), 201
[setBrakes\(boolean\)](#) (Java method), 185
[setControlling\(Part\)](#) (Java method), 189
[setCurrentSpeed\(float\)](#) (Java method), 224
[setDeltaV\(float\)](#) (Java method), 216
[setDeployAltitude\(float\)](#) (Java method), 204
[setDeployed\(boolean\)](#) (Java method), 202, 203, 205, 207, 208
[setDeployMinPressure\(float\)](#) (Java method), 204
[setExpanded\(boolean\)](#) (Java method), 222
[setForward\(float\)](#) (Java method), 186
[setForwardKey\(String\)](#) (Java method), 222
[setGear\(boolean\)](#) (Java method), 185
[setGimbalLimit\(float\)](#) (Java method), 201
[setGimbalLocked\(boolean\)](#) (Java method), 201
[setHighlight\(boolean\)](#) (Java method), 223
[setIsAxisInverted\(boolean\)](#) (Java method), 224
[setIsLocked\(boolean\)](#) (Java method), 224
[setLights\(boolean\)](#) (Java method), 185
[setMargin\(double\)](#) (Java method), 226
[setMaxPosition\(float\)](#) (Java method), 224
[setMaxRollSpeed\(float\)](#) (Java method), 220
[setMaxRotationSpeed\(float\)](#) (Java method), 220
[setMinPosition\(float\)](#) (Java method), 223
[setMode\(String\)](#) (Java method), 201
[setName\(String\)](#) (Java method), 169, 197, 222, 223, 227
[setNormal\(float\)](#) (Java method), 215
[setNotes\(String\)](#) (Java method), 227
[setOpen\(boolean\)](#) (Java method), 196, 202
[setPhysicsWarpFactor\(int\)](#) (Java method), 166
[setPIDParameters\(float, float, float\)](#) (Java method), 220
[setPitch\(float\)](#) (Java method), 186
[setPrograde\(float\)](#) (Java method), 215
[setRadial\(float\)](#) (Java method), 216
[setRailsWarpFactor\(int\)](#) (Java method), 166
[setRCS\(boolean\)](#) (Java method), 185
[setReferenceFrame\(ReferenceFrame\)](#) (Java method), 219
[setRepeat\(boolean\)](#) (Java method), 227
[setRepeatPeriod\(double\)](#) (Java method), 227
[setReverseKey\(String\)](#) (Java method), 222
[setRight\(float\)](#) (Java method), 186
[setRoll\(float\)](#) (Java method), 186
[setRollSpeedMultiplier\(float\)](#) (Java method), 220
[setRotationSpeedMultiplier\(float\)](#) (Java method), 220
[setSAS\(boolean\)](#) (Java method), 185, 220
[setSASMode\(SASMode\)](#) (Java method), 185, 220
[setShielded\(boolean\)](#) (Java method), 198
[setSpeed\(float\)](#) (Java method), 222, 224
[setSpeedMode\(SpeedMode\)](#) (Java method), 185
[setTarget\(Vessel\)](#) (Java method), 169
[setTargetBody\(CelestialBody\)](#) (Java method), 165
[setTargetDirection\(org.javatuples.Triplet\)](#) (Java method), 219
[setTargetDockingPort\(DockingPort\)](#) (Java method), 165
[setTargetRoll\(float\)](#) (Java method), 219
[setTargetVessel\(Vessel\)](#) (Java method), 165
[setThrottle\(float\)](#) (Java method), 185
[setThrustLimit\(float\)](#) (Java method), 200
[setTime\(double\)](#) (Java method), 226
[setType\(VesselType\)](#) (Java method), 169
[setUp\(float\)](#) (Java method), 186
[setUT\(double\)](#) (Java method), 216
[setVessel\(Vessel\)](#) (Java method), 227
[setWheelSteering\(float\)](#) (Java method), 186
[setWheelThrottle\(float\)](#) (Java method), 186
[setXferOriginBody\(CelestialBody\)](#) (Java method), 227
[setXferTargetBody\(CelestialBody\)](#) (Java method), 227
[setYaw\(float\)](#) (Java method), 186
[shielded](#) (DockingPort attribute), 278, 363
[SHIELDED](#) (Java field), 200
[SHIP](#) (Java field), 175
[sideslip_angle](#) (Flight attribute), 255, 340
[signal_delay](#) (Comms attribute), 302, 387
[signal_delay_to_ground_station](#) (Comms attribute), 302, 387
[signal_delay_to_vessel\(\)](#) (Comms method), 302, 387
[signalDelayToVessel\(Vessel\)](#) (Java method), 218
[situation](#) (Vessel attribute), 238, 323
[skin_temperature](#) (Part attribute), 269, 354
[SOI_CHANGE](#) (Java field), 228
[SOI_CHANGE_AUTO](#) (Java field), 228
[solar_panel](#) (Part attribute), 273, 358
[solar_panels](#) (Parts attribute), 267, 352
[SolarPanel](#) (class in `SpaceCenter`), 292, 377
[SolarPanel](#) (Java class), 208
[SolarPanelState](#) (class in `SpaceCenter`), 293, 378
[SolarPanelState](#) (Java enum), 209
[SolarPanelState.broken](#) (in module `SpaceCenter`), 293, 378
[SolarPanelState.extended](#) (in module `SpaceCenter`), 293, 378

- SolarPanelState.extending (in module SpaceCenter), 293, 378
 - SolarPanelState.retracted (in module SpaceCenter), 293, 378
 - SolarPanelState.retracting (in module SpaceCenter), 293, 378
 - SPACE_CENTER (Java field), 165
 - SpaceCenter (Java class), 165
 - SpaceCenter (module), 234, 319
 - specific_impulse (Engine attribute), 281, 366
 - specific_impulse (Vessel attribute), 240, 325
 - speed (ControlGroup attribute), 306, 391
 - speed (Flight attribute), 252, 337
 - speed (Intake attribute), 284, 369
 - speed (Orbit attribute), 258, 343
 - speed (Servo attribute), 308, 393
 - speed_mode (Control attribute), 260, 345
 - speed_of_sound (Flight attribute), 254, 339
 - SpeedMode (class in SpaceCenter), 263, 348
 - SpeedMode (Java enum), 187
 - SpeedMode.orbit (in module SpaceCenter), 263, 348
 - SpeedMode.surface (in module SpaceCenter), 263, 348
 - SpeedMode.target (in module SpaceCenter), 263, 348
 - sphere_of_influence (CelestialBody attribute), 248, 333
 - SPLASHED (Java field), 176
 - STABILITY_ASSIST (Java field), 187
 - STAGE (Java field), 215
 - stage (Part attribute), 268, 353
 - stall_fraction (Flight attribute), 255, 340
 - start() (ResourceConverter method), 289, 374
 - start(int) (Java method), 206
 - state (CargoBay attribute), 276, 361
 - state (DockingPort attribute), 278, 363
 - state (LandingGear attribute), 284, 369
 - state (LandingLeg attribute), 285, 370
 - state (Parachute attribute), 287, 372
 - state (Radiator attribute), 288, 373
 - state (ResourceHarvester attribute), 290, 375
 - state (SolarPanel attribute), 293, 377
 - state() (ResourceConverter method), 289, 374
 - state(int) (Java method), 206
 - static_air_temperature (Flight attribute), 255, 340
 - static_pressure (Flight attribute), 253, 338
 - STATION (Java field), 175
 - status_info() (ResourceConverter method), 289, 374
 - statusInfo(int) (Java method), 206
 - stop() (ControlGroup method), 307, 392
 - stop() (Java method), 223, 225
 - stop() (ResourceConverter method), 289, 374
 - stop() (Servo method), 309, 394
 - stop(int) (Java method), 206
 - STORAGE_FULL (Java field), 206
 - STOWED (Java field), 204
 - Stream (class in krpc.stream), 318
 - Stream (Java class), 164
 - stream() (Client method), 317
 - SUB_ORBITAL (Java field), 176
 - sun_exposure (SolarPanel attribute), 293, 378
 - SURFACE (Java field), 188
 - surface_altitude (Flight attribute), 251, 336
 - surface_gravity (CelestialBody attribute), 247, 332
 - surface_height() (CelestialBody method), 247, 332
 - surface_position() (CelestialBody method), 248, 333
 - surface_reference_frame (Vessel attribute), 243, 328
 - surface_velocity_reference_frame (Vessel attribute), 244, 329
 - surfaceHeight(double, double) (Java method), 176
 - surfacePosition(double, double, ReferenceFrame) (Java method), 177
- ## T
- TARGET (Java field), 187, 188
 - target (Vessel attribute), 239, 324
 - target_body (in module SpaceCenter), 234, 319
 - target_direction (AutoPilot attribute), 303, 388
 - target_docking_port (in module SpaceCenter), 234, 319
 - target_pitch_and_heading() (AutoPilot method), 303, 388
 - target_roll (AutoPilot attribute), 304, 388
 - target_vessel (in module SpaceCenter), 234, 319
 - targetPitchAndHeading(float, float) (Java method), 219
 - temperature (Part attribute), 269, 354
 - terminal_velocity (Flight attribute), 255, 340
 - thermal_conduction_flux (Part attribute), 270, 355
 - thermal_convection_flux (Part attribute), 270, 355
 - thermal_efficiency (ResourceHarvester attribute), 290, 375
 - thermal_internal_flux (Part attribute), 270, 355
 - thermal_mass (Part attribute), 270, 355
 - thermal_radiation_flux (Part attribute), 270, 355
 - thermal_resource_mass (Part attribute), 270, 355
 - thermal_skin_mass (Part attribute), 270, 355
 - thermal_skin_to_internal_flux (Part attribute), 270, 355
 - throttle (Control attribute), 261, 346
 - throttle (Engine attribute), 282, 367
 - throttle_locked (Engine attribute), 282, 367
 - thrust (Engine attribute), 280, 365
 - thrust (Vessel attribute), 240, 325
 - thrust_limit (Engine attribute), 281, 366
 - thrust_specific_fuel_consumption (Flight attribute), 256, 341
 - time (Alarm attribute), 311, 396
 - time_to (Node attribute), 300, 385
 - time_to_apoapsis (Orbit attribute), 258, 343
 - time_to_periapsis (Orbit attribute), 258, 343
 - time_to_soi_change (Orbit attribute), 259, 344
 - title (Part attribute), 267, 352
 - toggle_action_group() (Control method), 262, 347
 - toggle_mode() (Engine method), 283, 368

toggleActionGroup(int) (Java method), 187
 toggleMode() (Java method), 201
 total_air_temperature (Flight attribute), 255, 340
 TRACKING_STATION (Java field), 165
 TRANSFER (Java field), 229
 TRANSFER_MODELLED (Java field), 229
 transform_direction() (in module SpaceCenter), 236, 321
 transform_position() (in module SpaceCenter), 236, 321
 transform_rotation() (in module SpaceCenter), 236, 321
 transform_velocity() (in module SpaceCenter), 237, 322
 transformDirection(org.javatuples.Triplet, ReferenceFrame, ReferenceFrame) (Java method), 167
 transformPosition(org.javatuples.Triplet, ReferenceFrame, ReferenceFrame) (Java method), 167
 transformRotation(org.javatuples.Quartet, ReferenceFrame, ReferenceFrame) (Java method), 167
 transformVelocity(org.javatuples.Triplet, org.javatuples.Triplet, ReferenceFrame, ReferenceFrame) (Java method), 167
 trigger_event() (Module method), 275, 360
 triggerEvent(String) (Java method), 195
 type (Alarm attribute), 311, 396
 type (Vessel attribute), 238, 323

U

undock() (DockingPort method), 278, 363
 undock() (Java method), 197
 UNDOCKING (Java field), 200
 UNKNOWN (Java field), 207
 up (Control attribute), 261, 346
 ut (in module SpaceCenter), 234, 320
 ut (Node attribute), 300, 385

V

vacuum_specific_impulse (Engine attribute), 281, 366
 vacuum_specific_impulse (Vessel attribute), 241, 325
 value (Sensor attribute), 292, 377
 Vector3 (C++ class), 151
 Vector3 (class in SpaceCenter), 305, 390
 velocity (Flight attribute), 251, 336
 velocity() (CelestialBody method), 250, 335
 velocity() (Part method), 273, 358
 velocity() (Vessel method), 245, 330
 velocity(ReferenceFrame) (Java method), 175, 179, 194
 vertical_speed (Flight attribute), 252, 337
 vessel (Alarm attribute), 312, 396
 Vessel (class in SpaceCenter), 238, 323
 Vessel (Java class), 169
 VESSEL (Java field), 215
 vessel (Part attribute), 268, 353
 vessels (in module SpaceCenter), 234, 319

VesselSituation (class in SpaceCenter), 246, 331
 VesselSituation (Java enum), 175
 VesselSituation.docked (in module SpaceCenter), 246, 331
 VesselSituation.escaping (in module SpaceCenter), 246, 331
 VesselSituation.flying (in module SpaceCenter), 246, 331
 VesselSituation.landed (in module SpaceCenter), 246, 331
 VesselSituation.orbiting (in module SpaceCenter), 246, 331
 VesselSituation.pre_launch (in module SpaceCenter), 246, 331
 VesselSituation.splashed (in module SpaceCenter), 246, 331
 VesselSituation.sub_orbital (in module SpaceCenter), 246, 331
 VesselType (class in SpaceCenter), 245, 330
 VesselType (Java enum), 175
 VesselType.base (in module SpaceCenter), 246, 331
 VesselType.debris (in module SpaceCenter), 246, 331
 VesselType.lander (in module SpaceCenter), 246, 331
 VesselType.probe (in module SpaceCenter), 246, 331
 VesselType.rover (in module SpaceCenter), 246, 331
 VesselType.ship (in module SpaceCenter), 245, 330
 VesselType.station (in module SpaceCenter), 246, 331

W

wait() (AutoPilot method), 303, 388
 wait() (Java method), 219
 warp_factor (in module SpaceCenter), 235, 320
 warp_mode (in module SpaceCenter), 235, 320
 warp_rate (in module SpaceCenter), 235, 320
 warp_to() (in module SpaceCenter), 236, 321
 WarpMode (class in SpaceCenter), 238, 323
 WarpMode (Java enum), 169
 WarpMode.none (in module SpaceCenter), 238, 323
 WarpMode.physics (in module SpaceCenter), 238, 323
 WarpMode.rails (in module SpaceCenter), 238, 323
 warpTo(double, float, float) (Java method), 166
 wheel_steering (Control attribute), 261, 346
 wheel_throttle (Control attribute), 261, 346
 with_module() (Parts method), 265, 350
 with_name() (Parts method), 264, 349
 with_title() (Parts method), 265, 350
 withModule(String) (Java method), 189
 withName(String) (Java method), 189
 withTitle(String) (Java method), 189

X

xfer_origin_body (Alarm attribute), 312, 397
 xfer_target_body (Alarm attribute), 312, 397

Y

yaw (Control attribute), [261](#), [346](#)

yaw_torque (ReactionWheel attribute), [291](#), [376](#)