CHAPTER

SIX

JAVA

6.1 Java Client

This client provides a Java API for interacting with a kRPC server. A jar containing the `krpc.client` package can be downloaded from GitHub. It requires Java version 1.8.

6.1.1 Using the Library

The kRPC client library depends on the protobuf and javatuples libraries. A prebuilt jar for protobuf is available via Maven. Note that you need protobuf version 3. Version 2 is not compatible with kRPC.

The following example program connects to the server, queries it for its version and prints it out:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KRPC;

import java.io.IOException;

public class Basic {
  public static void main(String[] args) throws IOException, RPCException {
    Connection connection = Connection.newInstance();
    KRPC krpc = KRPC.newInstance(connection);
    System.out.println("Connected to kRPC version " + krpc.getStatus().getVersion());
    connection.close();
  }
}
```

To compile this program using javac on the command line, save the source as `Example.java` and run the following:

```
javac -cp krpc-java-0.4.0.jar:protobuf-java-3.4.0.jar:javatuples-1.2.jar Example.java
```

You may need to change the paths to the JAR files.

6.1.2 Connecting to the Server

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

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KRPC;

import java.io.IOException;

public class Connecting {
  public static void main(String[] args) throws IOException, RPCException {
    Connection connection = Connection.newInstance("Remote example", "my.domain.name",
↪ 1000, 1001);
    System.out.println(KRPC.newInstance(connection).getStatus().getVersion());
    connection.close();
  }
}
```

6.1.3 Calling Remote Procedures

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

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

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

import java.io.IOException;

public class RemoteProcedures {
  public static void main(String[] args) throws IOException, RPCException {
    Connection connection = Connection.newInstance("Vessel Name");
    SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
    Vessel vessel = spaceCenter.getActiveVessel();
    System.out.println(vessel.getName());
    connection.close();
  }
}
```

6.1.4 Streaming Data from the Server

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

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KRPC;
```

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

import java.io.IOException;

public class Streaming1 {
  public static void main(String[] args) throws IOException, RPCException {
    Connection connection = Connection.newInstance();
    SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
    Vessel vessel = spaceCenter.getActiveVessel();
    ReferenceFrame refframe = vessel.getOrbit().getBody().getReferenceFrame();
    while (true) {
      System.out.println(vessel.position(refframe));
    }
  }
}
```

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

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

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

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

import org.javatuples.Triplet;

import java.io.IOException;

public class Streaming2 {
  public static void main(String[] args) throws IOException, RPCException,␣
↪StreamException {
    Connection connection = Connection.newInstance();
    SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
    Vessel vessel = spaceCenter.getActiveVessel();
    ReferenceFrame refframe = vessel.getOrbit().getBody().getReferenceFrame();
    Stream<Triplet<Double,Double,Double>> vesselStream = connection.addStream(vessel,
↪"position", refframe);
    while (true) {
      System.out.println(vesselStream.get());
    }
  }
}
```

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

A stream can be created for any method call by calling *Connection.addStream* and passing it information about

which method to stream. The example above passes a remote object, the name of the method to call, followed by the arguments to pass to the method (if any). *Connection.addStream* returns a stream object of type *Stream*. The most recent value of the stream can be obtained by calling *Stream.get*. A stream can be stopped and removed from the server by calling *Stream.remove* on the stream object. All of a clients streams are automatically stopped when it disconnects.

6.1.5 Synchronizing with Stream Updates

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

Condition Variables

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

The following example waits until the abort button is pressed in game, by waiting for the value of *krpc.client.services.SpaceCenter.Control.getAbort()* to change to true:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.Stream;
import krpc.client.StreamException;
import krpc.client.services.KRPC;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Control;

import java.io.IOException;

public class ConditionVariables {
  public static void main(String[] args) throws IOException, RPCException,␣
↪StreamException {
    Connection connection = Connection.newInstance();
    SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
    Control control = spaceCenter.getActiveVessel().getControl();
    Stream<Boolean> abort = connection.addStream(control, "getAbort");
    synchronized (abort.getCondition()) {
      while (!abort.get()) {
        abort.waitForUpdate();
      }
    }
  }
}
```

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

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

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

Callbacks

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

For example the following program registers two callbacks that are invoked when the value of *krpc.client.services.SpaceCenter.Control.getAbort()* changes:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.Stream;
import krpc.client.StreamException;
import krpc.client.services.KRPC;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Control;

import java.io.IOException;

public class Callbacks {
  public static void main(String[] args) throws IOException, RPCException,␣
↪StreamException {
    Connection connection = Connection.newInstance();
    SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
    Control control = spaceCenter.getActiveVessel().getControl();
    Stream<Boolean> abort = connection.addStream(control, "getAbort");
    abort.addCallback(
      (Boolean x) -> {
        System.out.println("Abort 1 called with a value of " + x);
      });
    abort.addCallback(
      (Boolean x) -> {
        System.out.println("Abort 2 called with a value of " + x);
      });
    abort.start();

    // Keep the program running...
    while (true) {
    }
  }
}
```

Note: When a stream is created it does not start receiving updates until `start` is called. This is implicitly called when accessing the value of a stream, but as this example does not do this an explicit call to `start` is required.

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

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

6.1.6 Custom Events

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

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

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.Event;
import krpc.client.StreamException;
import krpc.client.services.KRPC;
import krpc.client.services.KRPC.Expression;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Flight;
import krpc.schema.KRPC.ProcedureCall;

import java.io.IOException;

public class CustomEvent {
  public static void main(String[] args) throws IOException, RPCException,␣
↪StreamException {
    Connection connection = Connection.newInstance();
    KRPC krpc = KRPC.newInstance(connection);
    SpaceCenter spaceCenter = SpaceCenter.newInstance(connection);
    Flight flight = spaceCenter.getActiveVessel().flight(null);

    // Get the remote procedure call as a message object,
    // so it can be passed to the server
    ProcedureCall meanAltitude = connection.getCall(flight, "getMeanAltitude");

    // Create an expression on the server
    Expression expr = Expression.greaterThan(connection,
      Expression.call(connection, meanAltitude),
      Expression.constantDouble(connection, 1000));

    Event event = krpc.addEvent(expr);
    synchronized (event.getCondition()) {
      event.waitFor();
      System.out.println("Altitude reached 1000m");
    }
  }
}
```

6.1.7 Client API Reference

class **Connection**

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

static *Connection* **newInstance**()

static *Connection* **newInstance**(String *name*)

static *Connection* **newInstance**(String *name*, String *address*)

static *Connection* **newInstance**(String *name*, String *address*, int *rpcPort*, int *streamPort*)

static *Connection* **newInstance**(String *name*, java.net.InetAddress *address*)

static *Connection* **newInstance**(String *name*, java.net.InetAddress *address*, int *rpcPort*, int *streamPort*)

Create a connection to the server, using the given connection details.

Parameters

- **name** (*String*) – A descriptive name for the connection. This is passed to the server and appears in the in-game server window.
- **address** (*String*) – The address of the server to connect to. Can either be a hostname, an IP address as a string or a `java.net.InetAddress` object. Defaults to 127.0.0.1.
- **rpc_port** (*int*) – The port number of the RPC Server. Defaults to 50000. This should match the RPC port number of the server you want to connect to.
- **stream_port** (*int*) – The port number of the Stream Server. Defaults to 50001. This should match the stream port number of the server you want to connect to.

Stream<T> **addStream**(Class<?> *clazz*, String *method*, Object... *args*)

Create a stream for a static method call to the given class.

Stream<T> **addStream**(*RemoteObject instance*, String *method*, Object... *args*)

Create a stream for a method call to the given remote object.

krpc.schema.KRPC.ProcedureCall **getCall**(Class<?> *clazz*, String *method*, Object... *args*)

Returns a procedure call message for the given static method call. This allows descriptions of procedure calls to be passed to the server, for example when constructing custom events. See *Custom Events*.

krpc.schema.KRPC.ProcedureCall **getCall**(*RemoteObject instance*, String *method*, Object... *args*)

Returns a procedure call message for the given method call. This allows descriptions of procedure calls to be passed to the server, for example when constructing custom events. See *Custom Events*.

void **close**()

Close the connection.

class **Stream**<T>

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

Stream objects implement `hashCode` and `equals` such that two stream objects are equal if they are bound to the same stream on the server.

void **start**()

void **startAndWait**()

Starts the stream. When a stream is created it does not start sending updates to the client until this method is called.

The `startAndWait` method will block until at least one update has been received from the server.

The `start` method starts the stream and returns immediately. Subsequent calls to `get()` may throw a `StreamException`.

float **getRate**()

void **setRate**(float *rate*)

The update rate of the stream in Hertz. When set to zero, the rate is unlimited.

T **get**()

Returns the most recent value for the stream. If executing the remote procedure for the stream throws an exception, calling this method will rethrow the exception. Raises a `StreamException` if no update has been received from the server.

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

Object **getCondition**()
A condition variable that is notified (using `notifyAll`) whenever the value of the stream changes.

void **waitForUpdate**()

void **waitForUpdateWithTimeout**(double *timeout*)
These methods block until the value of the stream changes or the operation times out.

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

If *timeout* is specified it is the timeout in seconds for the operation.

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

void **addCallback**(java.util.function.Consumer<T> *callback*)
Adds a callback function that is invoked whenever the value of the stream changes. The callback function should take one argument, which is passed the new value of the stream.

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

---

void **remove**()
Remove the stream from the server.

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

Event objects implement `hashCode` and `equals` such that two event objects are equal if they are bound to the same underlying stream on the server.

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

Object **getCondition**()
The condition variable that is notified (using `notifyAll`) whenever the event occurs.

void **waitFor**()

void **waitForWithTimeout**(double *timeout*)
These methods block until the event occurs or the operation times out.

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

If *timeout* is specified it is the timeout in seconds for the operation.

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

void **addCallback**(java.lang.Callable *callback*)
Adds a callback function that is invoked whenever the event occurs. The callback function should be a function that takes zero arguments.

void **remove**()
Removes the event from the server.

Stream<Boolean> **getStream**()
Returns the underlying stream for the event.

abstract class **RemoteObject**
The abstract base class for all remote objects.

6.2 KRPC API

6.2.1 KRPC

None None None None None

public class **KRPC**
Main kRPC service, used by clients to interact with basic server functionality.

byte[] **getClientID**()
Returns the identifier for the current client.

String **getClientName**()
Returns the name of the current client. This is an empty string if the client has no name.

java.util.List<org.javatuples.Triplet<byte[], String, String>> **getClients**()
A list of RPC clients that are currently connected to the server. Each entry in the list is a clients identifier, name and address.

krpc.schema.KRPC.Status **getStatus**()
Returns some information about the server, such as the version.

krpc.schema.KRPC.Services **getServices**()
Returns information on all services, procedures, classes, properties etc. provided by the server. Can be used by client libraries to automatically create functionality such as stubs.

GameScene **getCurrentGameScene**()
Get the current game scene.

boolean **getPaused**()

void **setPaused**(boolean *value*)
Whether the game is paused.

public enum **GameScene**
The game scene. See *getCurrentGameScene()*.

public *GameScene* **SPACE_CENTER**
The game scene showing the Kerbal Space Center buildings.

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

public *GameScene* **TRACKING_STATION**
The tracking station.

public *GameScene* **EDITOR_VAB**
The Vehicle Assembly Building.

public *GameScene* **EDITOR_SPH**
The Space Plane Hangar.

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

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

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

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

6.2.2 Expressions

public class **Expression**
A server side expression.

static *Expression* **constantDouble**(*Connection connection*, double *value*)
A constant value of double precision floating point type.

Parameters

- **value** (*double*) –

static *Expression* **constantFloat**(*Connection connection*, float *value*)
A constant value of single precision floating point type.

Parameters

- **value** (*float*) –

static *Expression* **constantInt**(*Connection connection*, int *value*)
A constant value of integer type.

Parameters

- **value** (*int*) –

static *Expression* **constantBool**(*Connection connection*, boolean *value*)
A constant value of boolean type.

Parameters

- **value** (*boolean*) –

static *Expression* **constantString**(*Connection connection*, String *value*)
A constant value of string type.

Parameters

- **value** (*String*) –

static *Expression* **call**(*Connection connection*, krpc.schema.KRPC.ProcedureCall *call*)
An RPC call.

Parameters

- **call** (*krpc.schema.KRPC.ProcedureCall*) –

static *Expression* **equal**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Equality comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **notEqual**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Inequality comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **greaterThan**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Greater than numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **greaterThanOrEqual**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Greater than or equal numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **lessThan**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Less than numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **lessThanOrEqual**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Less than or equal numerical comparison.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **and**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Boolean and operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **or**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Boolean or operator.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **exclusiveOr**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Boolean exclusive-or operator.

Parameters

- **arg0** (Expression) –

- **arg1** (Expression) –

static *Expression* **not**(*Connection connection*, *Expression arg*)
Boolean negation operator.

Parameters

- **arg** (Expression) –

static *Expression* **add**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Numerical addition.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **subtract**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Numerical subtraction.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **multiply**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Numerical multiplication.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **divide**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Numerical division.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

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

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

Returns The remainder of arg0 divided by arg1

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

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

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

static *Expression* **leftShift**(*Connection connection*, *Expression arg0*, *Expression arg1*)
Bitwise left shift.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **rightShift** (*Connection connection*, *Expression arg0*, *Expression arg1*)
Bitwise right shift.

Parameters

- **arg0** (Expression) –
- **arg1** (Expression) –

static *Expression* **cast** (*Connection connection*, *Expression arg*, *Type type*)
Perform a cast to the given type.

Parameters

- **arg** (Expression) –
- **type** (Type) – Type to cast the argument to.

static *Expression* **parameter** (*Connection connection*, String *name*, *Type type*)
A named parameter of type double.

Parameters

- **name** (*String*) – The name of the parameter.
- **type** (Type) – The type of the parameter.

Returns A named parameter.

static *Expression* **function** (*Connection connection*, java.util.List<*Expression*> *parameters*, *Expression body*)
A function.

Parameters

- **parameters** (*java.util.List<Expression>*) – The parameters of the function.
- **body** (Expression) – The body of the function.

Returns A function.

static *Expression* **invoke** (*Connection connection*, *Expression function*, java.util.Map<String, *Expression*> *args*)
A function call.

Parameters

- **function** (Expression) – The function to call.
- **args** (*java.util.Map<String,Expression>*) – The arguments to call the function with.

Returns A function call.

static *Expression* **createTuple** (*Connection connection*, java.util.List<*Expression*> *elements*)
Construct a tuple.

Parameters

- **elements** (*java.util.List<Expression>*) – The elements.

Returns The tuple.

static *Expression* **createList**(*Connection connection*, java.util.List<*Expression*> *values*)
Construct a list.

Parameters

- **values** (*java.util.List<Expression>*) – The value. Should all be of the same type.

Returns The list.

static *Expression* **createSet**(*Connection connection*, java.util.Set<*Expression*> *values*)
Construct a set.

Parameters

- **values** (*java.util.Set<Expression>*) – The values. Should all be of the same type.

Returns The set.

static *Expression* **createDictionary**(*Connection connection*, java.util.List<*Expression*> *keys*, java.util.List<*Expression*> *values*)
Construct a dictionary, from a list of corresponding keys and values.

Parameters

- **keys** (*java.util.List<Expression>*) – The keys. Should all be of the same type.
- **values** (*java.util.List<Expression>*) – The values. Should all be of the same type.

Returns The dictionary.

static *Expression* **toList**(*Connection connection*, *Expression arg*)
Convert a collection to a list.

Parameters

- **arg** (Expression) – The collection.

Returns The collection as a list.

static *Expression* **toSet**(*Connection connection*, *Expression arg*)
Convert a collection to a set.

Parameters

- **arg** (Expression) – The collection.

Returns The collection as a set.

static *Expression* **get**(*Connection connection*, *Expression arg*, *Expression index*)
Access an element in a tuple, list or dictionary.

Parameters

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

Returns The element.

static *Expression* **count**(*Connection connection*, *Expression arg*)
Number of elements in a collection.

Parameters

- **arg** (Expression) – The list, set or dictionary.

Returns The number of elements in the collection.

static *Expression* **sum**(*Connection connection*, *Expression arg*)
Sum all elements of a collection.

Parameters

- **arg** (Expression) – The list or set.

Returns The sum of the elements in the collection.

static *Expression* **max**(*Connection connection*, *Expression arg*)
Maximum of all elements in a collection.

Parameters

- **arg** (Expression) – The list or set.

Returns The maximum elements in the collection.

static *Expression* **min**(*Connection connection*, *Expression arg*)
Minimum of all elements in a collection.

Parameters

- **arg** (Expression) – The list or set.

Returns The minimum elements in the collection.

static *Expression* **average**(*Connection connection*, *Expression arg*)
Minimum of all elements in a collection.

Parameters

- **arg** (Expression) – The list or set.

Returns The minimum elements in the collection.

static *Expression* **select**(*Connection connection*, *Expression arg*, *Expression func*)
Run a function on every element in the collection.

Parameters

- **arg** (Expression) – The list or set.
- **func** (Expression) – The function.

Returns The modified collection.

static *Expression* **where**(*Connection connection*, *Expression arg*, *Expression func*)
Run a function on every element in the collection.

Parameters

- **arg** (Expression) – The list or set.
- **func** (Expression) – The function.

Returns The modified collection.

static *Expression* **contains**(*Connection connection*, *Expression arg*, *Expression value*)
Determine if a collection contains a value.

Parameters

- **arg** (Expression) – The collection.

- **value** (Expression) – The value to look for.

Returns Whether the collection contains a value.

static *Expression* **aggregate**(*Connection connection*, *Expression arg*, *Expression func*)
Applies an accumulator function over a sequence.

Parameters

- **arg** (Expression) – The collection.
- **func** (Expression) – The accumulator function.

Returns The accumulated value.

static *Expression* **aggregateWithSeed**(*Connection connection*, *Expression arg*, *Expression seed*, *Expression func*)
Applies an accumulator function over a sequence, with a given seed.

Parameters

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

Returns The accumulated value.

static *Expression* **concat**(*Connection connection*, *Expression arg1*, *Expression arg2*)
Concatenate two sequences.

Parameters

- **arg1** (Expression) – The first sequence.
- **arg2** (Expression) – The second sequence.

Returns The first sequence followed by the second sequence.

static *Expression* **orderBy**(*Connection connection*, *Expression arg*, *Expression key*)
Order a collection using a key function.

Parameters

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

Returns The ordered collection.

static *Expression* **all**(*Connection connection*, *Expression arg*, *Expression predicate*)
Determine whether all items in a collection satisfy a boolean predicate.

Parameters

- **arg** (Expression) – The collection.
- **predicate** (Expression) – The predicate function.

Returns Whether all items satisfy the predicate.

static *Expression* **any**(*Connection connection*, *Expression arg*, *Expression predicate*)
Determine whether any item in a collection satisfies a boolean predicate.

Parameters

- **arg** (Expression) – The collection.

- **predicate** (Expression) – The predicate function.

Returns Whether any item satisfies the predicate.

public class **Type**

A server side expression.

static *Type* **double_**(*Connection connection*)
Double type.

static *Type* **float_**(*Connection connection*)
Float type.

static *Type* **int_**(*Connection connection*)
Int type.

static *Type* **bool**(*Connection connection*)
Bool type.

static *Type* **string**(*Connection connection*)
String type.

6.3 SpaceCenter API

6.3.1 SpaceCenter

public class **SpaceCenter**

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

Vessel **getActiveVessel**()

void **setActiveVessel**(*Vessel value*)
The currently active vessel.

java.util.List<*Vessel*> **getVessels**()
A list of all the vessels in the game.

java.util.Map<String, *CelestialBody*> **getBodies**()
A dictionary of all celestial bodies (planets, moons, etc.) in the game, keyed by the name of the body.

CelestialBody **getTargetBody**()

void **setTargetBody**(*CelestialBody value*)
The currently targeted celestial body.

Vessel **getTargetVessel**()

void **setTargetVessel**(*Vessel value*)
The currently targeted vessel.

DockingPort **getTargetDockingPort**()

void **setTargetDockingPort**(*DockingPort value*)
The currently targeted docking port.

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

java.util.List<String> **launchableVessels**(String *craftDirectory*)
Returns a list of vessels from the given *craftDirectory* that can be launched.

Parameters

- **craftDirectory** (*String*) – Name of the directory in the current saves "Ships" directory. For example `"VAB"` or `"SPH"`.

void **launchVessel** (String *craftDirectory*, String *name*, String *launchSite*)
Launch a vessel.

Parameters

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

void **launchVesselFromVAB** (String *name*)
Launch a new vessel from the VAB onto the launchpad.

Parameters

- **name** (*String*) – Name of the vessel to launch.

Note: This is equivalent to calling *launchVessel(String, String, String)* with the craft directory set to "VAB" and the launch site set to "LaunchPad".

---

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

Parameters

- **name** (*String*) – Name of the vessel to launch.

Note: This is equivalent to calling *launchVessel(String, String, String)* with the craft directory set to "SPH" and the launch site set to "Runway".

---

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

Parameters

- **name** (*String*) –

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

Parameters

- **name** (*String*) –

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

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

---

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

Note: This is the same as calling `load(String)` with the name "quicksave".

boolean **getUIVisible**()

void **setUIVisible**(boolean *value*)
Whether the UI is visible.

boolean **getNavball**()

void **setNavball**(boolean *value*)
Whether the navball is visible.

double **getUT**()
The current universal time in seconds.

double **getG**()
The value of the gravitational constant G in $N(m/kg)^2$.

float **getWarpRate**()
The current warp rate. This is the rate at which time is passing for either on-rails or physical time warp. For example, a value of 10 means time is passing 10x faster than normal. Returns 1 if time warp is not active.

float **getWarpFactor**()
The current warp factor. This is the index of the rate at which time is passing for either regular "on-rails" or physical time warp. Returns 0 if time warp is not active. When in on-rails time warp, this is equal to `getRailsWarpFactor()`, and in physics time warp, this is equal to `getPhysicsWarpFactor()`.

int **getRailsWarpFactor**()

void **setRailsWarpFactor**(int *value*)
The time warp rate, using regular "on-rails" time warp. A value between 0 and 7 inclusive. 0 means no time warp. Returns 0 if physical time warp is active.

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

int **getPhysicsWarpFactor**()

void **setPhysicsWarpFactor**(int *value*)
The physical time warp rate. A value between 0 and 3 inclusive. 0 means no time warp. Returns 0 if regular "on-rails" time warp is active.

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

Parameters

- **factor** (*int*) – The warp factor to check.

int **getMaximumRailsWarpFactor**()
The current maximum regular "on-rails" warp factor that can be set. A value between 0 and 7 inclusive. See the KSP wiki for details.

void **warpTo**(double *ut*, float *maxRailsRate*, float *maxPhysicsRate*)
Uses time acceleration to warp forward to a time in the future, specified by universal time *ut*. This call

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

Parameters

- **ut** (*double*) – The universal time to warp to, in seconds.
- **maxRailsRate** (*float*) – The maximum warp rate in regular "on-rails" time warp.
- **maxPhysicsRate** (*float*) – The maximum warp rate in physical time warp.

Returns When the time warp is complete.

org.javatuples.Triplet<Double, Double, Double> **transformPosition**(org.javatuples.Triplet<Double, Double, Double> *position*, *ReferenceFrame from*, *ReferenceFrame to*)

Converts a position from one reference frame to another.

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position, as a vector, in reference frame *from*.
- **from** (ReferenceFrame) – The reference frame that the position is in.
- **to** (ReferenceFrame) – The reference frame to covert the position to.

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

org.javatuples.Triplet<Double, Double, Double> **transformDirection**(org.javatuples.Triplet<Double, Double, Double> *direction*, *ReferenceFrame from*, *ReferenceFrame to*)

Converts a direction from one reference frame to another.

Parameters

- **direction** (*org.javatuples.Triplet<Double,Double,Double>*) – Direction, as a vector, in reference frame *from*.
- **from** (ReferenceFrame) – The reference frame that the direction is in.
- **to** (ReferenceFrame) – The reference frame to covert the direction to.

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

org.javatuples.Quartet<Double, Double, Double, Double> **transformRotation**(org.javatuples.Quartet<Double, Double, Double, Double> *rotation*, *ReferenceFrame from*, *ReferenceFrame to*)

Converts a rotation from one reference frame to another.

Parameters

- **rotation** (*org.javatuples.Quartet<Double,Double,Double,Double>*) – Rotation, as a quaternion of the form (x, y, z, w), in reference frame *from*.
- **from** (ReferenceFrame) – The reference frame that the rotation is in.

- **to** (ReferenceFrame) – The reference frame to covert the rotation to.

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

org.javatuples.Triplet<Double, Double, Double> **transformVelocity**(org.javatuples.Triplet<Double, Double, Double> *position*, org.javatuples.Triplet<Double, Double, Double> *velocity*, *ReferenceFrame from*, *ReferenceFrame to*)

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

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position, as a vector, in reference frame *from*.
- **velocity** (*org.javatuples.Triplet<Double,Double,Double>*) – Velocity, as a vector that points in the direction of travel and whose magnitude is the speed in meters per second, in reference frame *from*.
- **from** (ReferenceFrame) – The reference frame that the position and velocity are in.
- **to** (ReferenceFrame) – The reference frame to covert the velocity to.

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

double **raycastDistance**(org.javatuples.Triplet<Double, Double, Double> *position*, org.javatuples.Triplet<Double, Double, Double> *direction*, *ReferenceFrame referenceFrame*)

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

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position, as a vector, of the origin of the ray.
- **direction** (*org.javatuples.Triplet<Double,Double,Double>*) – Direction of the ray, as a unit vector.
- **referenceFrame** (ReferenceFrame) – The reference frame that the position and direction are in.

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

Part **raycastPart**(org.javatuples.Triplet<Double, Double, Double> *position*, org.javatuples.Triplet<Double, Double, Double> *direction*, *ReferenceFrame referenceFrame*)

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

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position, as a vector, of the origin of the ray.
- **direction** (*org.javatuples.Triplet<Double,Double,Double>*) – Direction of the ray, as a unit vector.
- **referenceFrame** (ReferenceFrame) – The reference frame that the position and direction are in.

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

boolean **getFARAvailable**()
Whether Ferram Aerospace Research is installed.

WarpMode **getWarpMode**()
The current time warp mode. Returns *WarpMode.NONE* if time warp is not active, *WarpMode.RAILS* if regular "on-rails" time warp is active, or *WarpMode.PHYSICS* if physical time warp is active.

Camera **getCamera**()
An object that can be used to control the camera.

WaypointManager **getWaypointManager**()
The waypoint manager.

ContractManager **getContractManager**()
The contract manager.

public enum **WarpMode**
The time warp mode. Returned by *WarpMode*

public *WarpMode* **RAILS**
Time warp is active, and in regular "on-rails" mode.

public *WarpMode* **PHYSICS**
Time warp is active, and in physical time warp mode.

public *WarpMode* **NONE**
Time warp is not active.

6.3.2 Vessel

public class **Vessel**
These objects are used to interact with vessels in KSP. This includes getting orbital and flight data, manipulating control inputs and managing resources. Created using *getActiveVessel()* or *getVessels()*.

String **getName**()

void **setName**(String *value*)
The name of the vessel.

VesselType **getType**()

void **setType**(*VesselType value*)
The type of the vessel.

VesselSituation **getSituation**()
The situation the vessel is in.

boolean **getRecoverable**()
Whether the vessel is recoverable.

void **recover**()
Recover the vessel.

double **getMET**()
The mission elapsed time in seconds.

String **getBiome**()
The name of the biome the vessel is currently in.

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

Parameters

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

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

Orbit **getOrbit**()
The current orbit of the vessel.

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

Comms **getComms**()
Returns a *Comms* object that can be used to interact with CommNet for this vessel.

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

int **getCrewCapacity**()
The number of crew that can occupy the vessel.

int **getCrewCount**()
The number of crew that are occupying the vessel.

java.util.List<*CrewMember*> **getCrew**()
The crew in the vessel.

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

Resources **resourcesInDecoupleStage**(int *stage*, boolean *cumulative*)
Returns a *Resources* object, that can used to get information about resources stored in a given *stage*.

Parameters

- **stage** (*int*) – Get resources for parts that are decoupled in this stage.
- **cumulative** (*boolean*) – When false, returns the resources for parts decoupled in just the given stage. When true returns the resources decoupled in the given stage and all subsequent stages combined.

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

Parts **getParts**()
A *Parts* object, that can used to interact with the parts that make up this vessel.

float **getMass**()
The total mass of the vessel, including resources, in kg.

float **getDryMass**()
The total mass of the vessel, excluding resources, in kg.

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

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

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

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

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

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

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

org.javatuples.Triplet<Double, Double, Double> **getMomentOfInertia**()
The moment of inertia of the vessel around its center of mass in $kg.m^2$. The inertia values in the returned 3-tuple are around the pitch, roll and yaw directions respectively. This corresponds to the vessels reference frame (*ReferenceFrame*).

java.util.List<Double> **getInertiaTensor**()
The inertia tensor of the vessel around its center of mass, in the vessels reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The maximum torque that the vessel generates. Includes contributions from reaction wheels, RCS, gimballed engines and aerodynamic control surfaces. Returns the torques in $N.m$ around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The maximum torque that the currently active and powered reaction wheels can generate. Returns the torques in $N.m$ around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The maximum torque that the currently active RCS thrusters can generate. Returns the torques in $N.m$ around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The maximum torque that the currently active and gimballed engines can generate. Returns the torques in $N.m$ around each of the coordinate axes of the vessels reference frame (*ReferenceFrame*). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The maximum torque that the aerodynamic control surfaces can generate. Returns the torques in $N.m$ around each of the coordinate axes of the vessels reference frame (`ReferenceFrame`). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The maximum torque that parts (excluding reaction wheels, gimballed engines, RCS and control surfaces) can generate. Returns the torques in $N.m$ around each of the coordinate axes of the vessels reference frame (`ReferenceFrame`). These axes are equivalent to the pitch, roll and yaw axes of the vessel.

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.

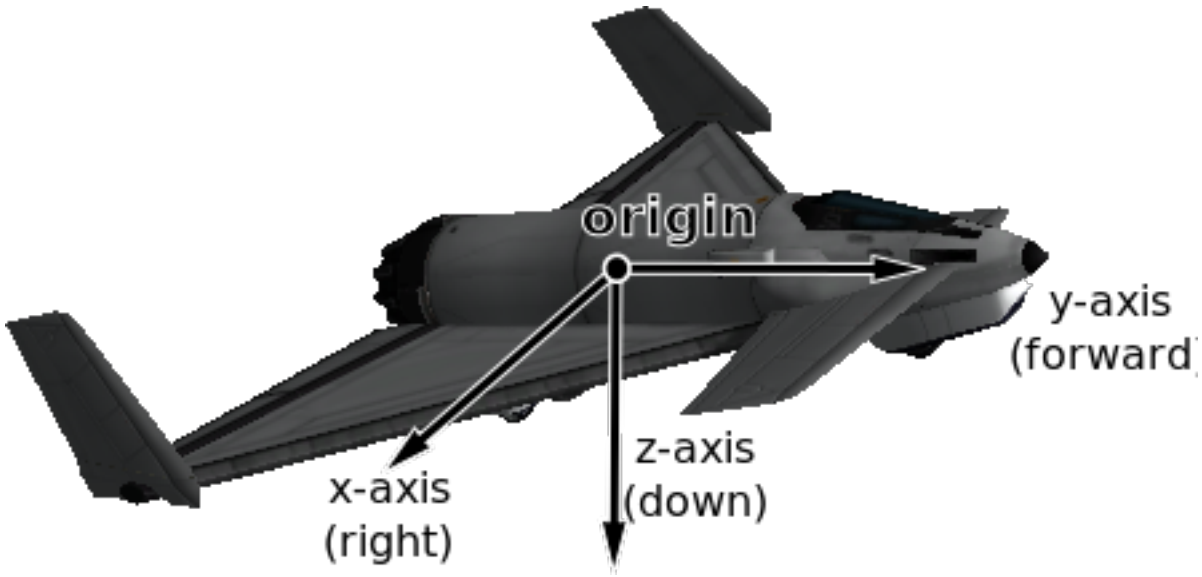


Fig. 6.1: Vessel reference frame origin and axes for the Aeris 3A aircraft

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.

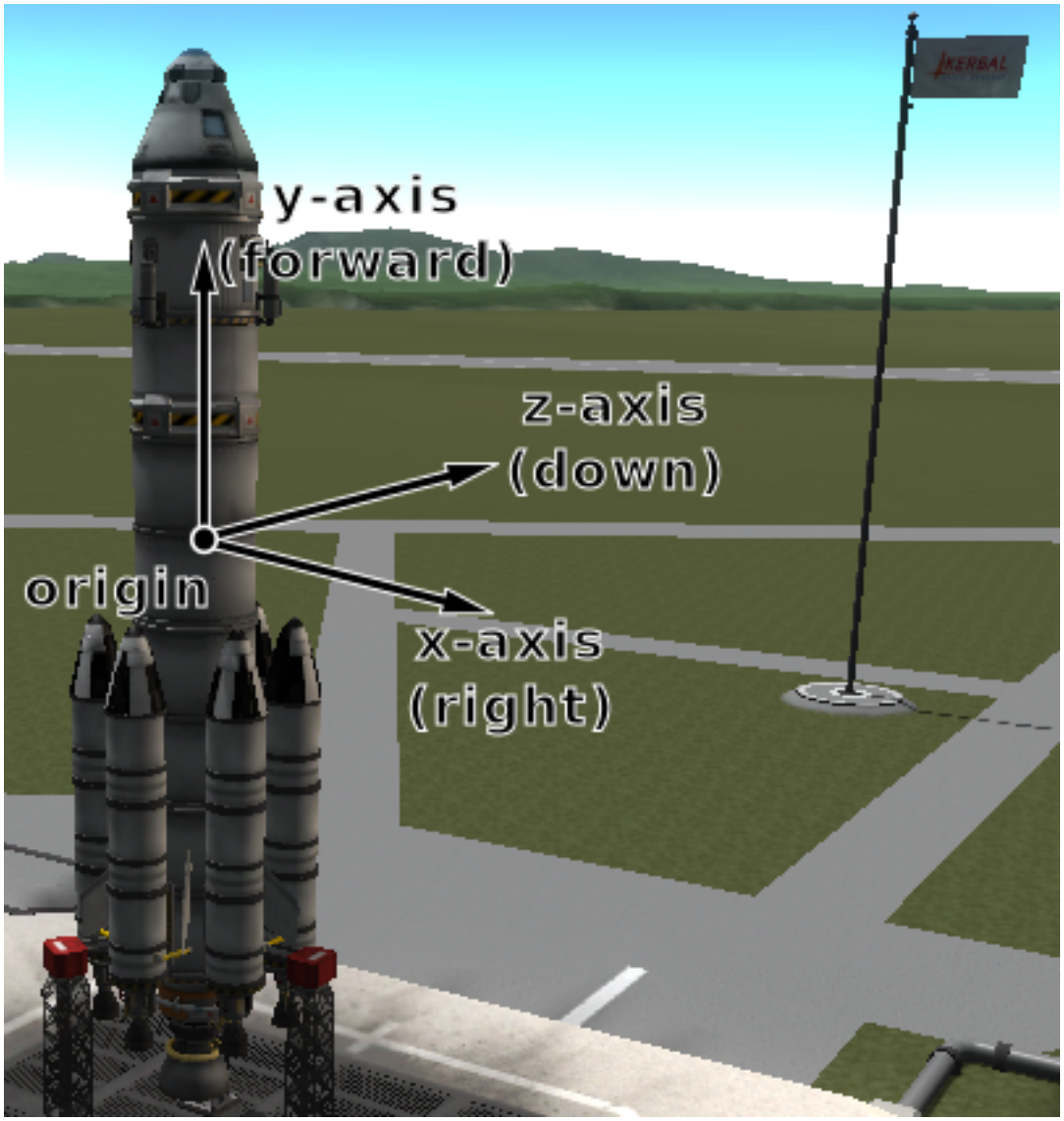


Fig. 6.2: Vessel reference frame origin and axes for the Kerbal-X rocket

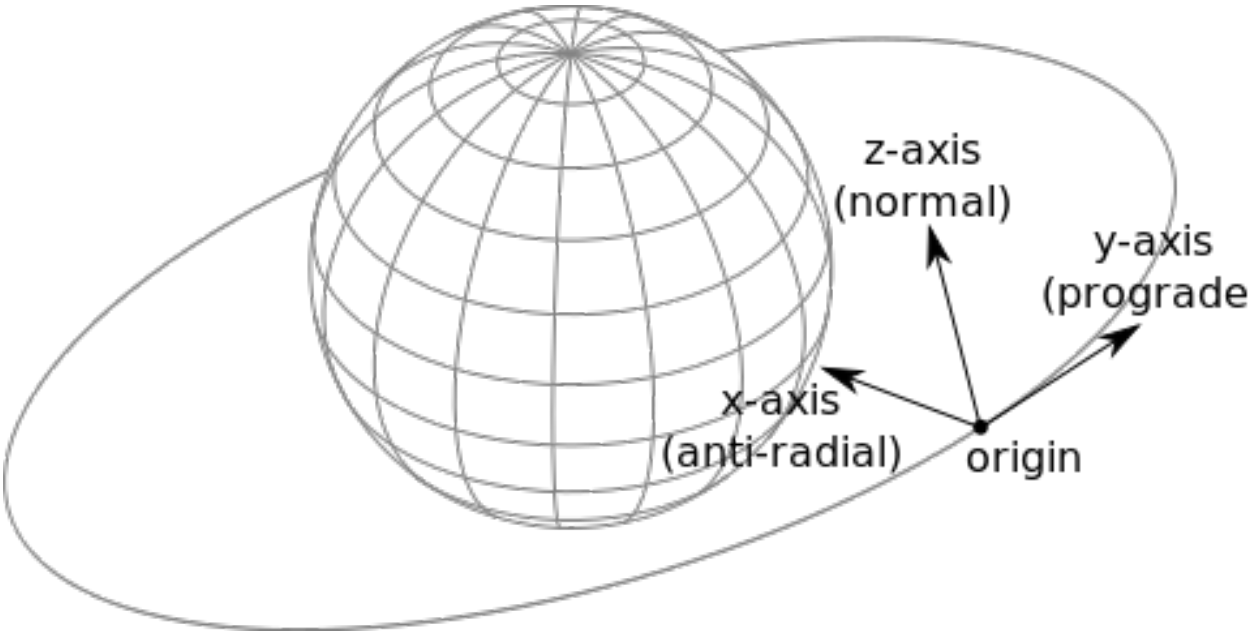


Fig. 6.3: Vessel orbital reference frame origin and axes

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.

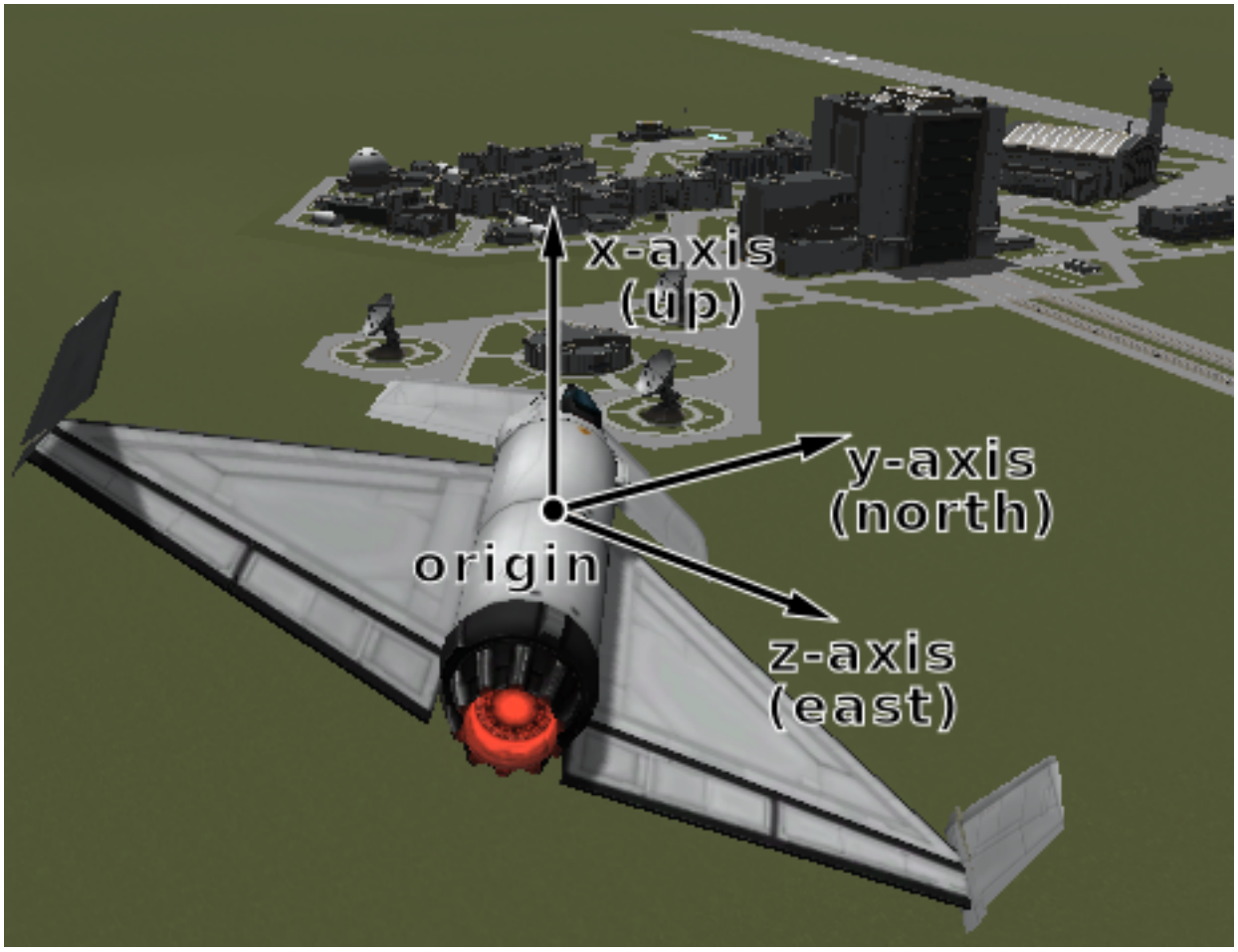


Fig. 6.4: Vessel surface reference frame origin and axes

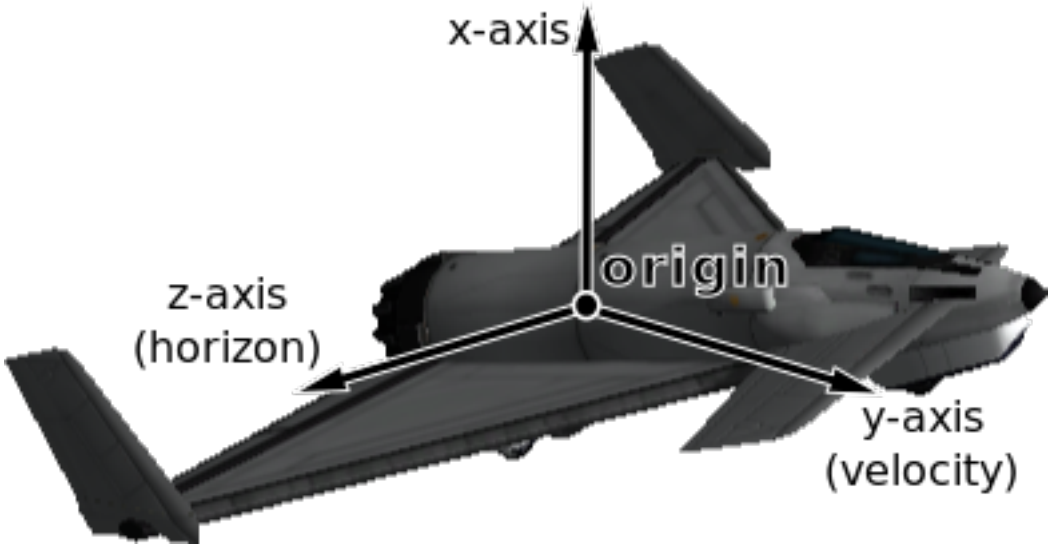


Fig. 6.5: Vessel surface velocity reference frame origin and axes

org.javatuples.Triplet<Double, Double, Double> **position** (*ReferenceFrame referenceFrame*)
The position of the center of mass of the vessel, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **bound**

The axis-aligned bounding box of the vessel in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vectors are in.

Returns The positions of the minimum and maximum vertices of the box, as position vectors.

org.javatuples.Triplet<Double, Double, Double> **velocity** (*ReferenceFrame referenceFrame*)
The velocity of the center of mass of the vessel, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned velocity vector is in.

Returns The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

org.javatuples.Quartet<Double, Double, Double, Double> **rotation** (*ReferenceFrame referenceFrame*)
The rotation of the vessel, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned rotation is in.

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

org.javatuples.Triplet<Double, Double, Double> **direction** (*ReferenceFrame referenceFrame*)
The direction in which the vessel is pointing, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **angularVelocity** (*ReferenceFrame referenceFrame*)
The angular velocity of the vessel, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame the returned angular velocity is in.

Returns The angular velocity as a vector. The magnitude of the vector is the rotational speed of the vessel, in radians per second. The direction of the vector indicates the axis of rotation, using the right-hand rule.

public enum **VesselType**

The type of a vessel. See *Vessel.getType()*.

public *VesselType* **BASE**
Base.

public *VesselType* **DEBRIS**
Debris.

public *VesselType* **LANDER**
Lander.

public *VesselType* **PLANE**
Plane.

public *VesselType* **PROBE**
Probe.

public *VesselType* **RELAY**
Relay.

public *VesselType* **ROVER**
Rover.

public *VesselType* **SHIP**
Ship.

public *VesselType* **STATION**
Station.

public enum **VesselSituation**

The situation a vessel is in. See *Vessel.getSituation()*.

public *VesselSituation* **DOCKED**
Vessel is docked to another.

public *VesselSituation* **ESCAPING**
Escaping.

public *VesselSituation* **FLYING**
Vessel is flying through an atmosphere.

public *VesselSituation* **LANDED**
Vessel is landed on the surface of a body.

public *VesselSituation* **ORBITING**
Vessel is orbiting a body.

public *VesselSituation* **PRE_LAUNCH**
Vessel is awaiting launch.

public *VesselSituation* **SPLASHED**
Vessel has splashed down in an ocean.

public *VesselSituation* **SUB_ORBITAL**
Vessel is on a sub-orbital trajectory.

public class **CrewMember**

Represents crew in a vessel. Can be obtained using *Vessel.getCrew()*.

String **getName**()

void **setName**(String *value*)
The crew members name.

CrewMemberType **getType**()
The type of crew member.

boolean **getOnMission**()
Whether the crew member is on a mission.

float **getCourage**()

void **setCourage**(float *value*)
The crew members courage.

float **getStupidity**()

void **setStupidity**(float *value*)
The crew members stupidity.

float **getExperience**()

void **setExperience**(float *value*)
The crew members experience.

boolean **getBadass**()

void **setBadass**(boolean *value*)
Whether the crew member is a badass.

boolean **getVeteran**()

void **setVeteran**(boolean *value*)
Whether the crew member is a veteran.

public enum **CrewMemberType**
The type of a crew member. See *CrewMember.getType()*.

public *CrewMemberType* **APPLICANT**
An applicant for crew.

public *CrewMemberType* **CREW**
Rocket crew.

public *CrewMemberType* **TOURIST**
A tourist.

public *CrewMemberType* **UNOWNED**
An unowned crew member.

6.3.3 CelestialBody

public class **CelestialBody**
Represents a celestial body (such as a planet or moon). See *getBodies()*.

String **getName**()
The name of the body.

java.util.List<*CelestialBody*> **getSatellites**()
A list of celestial bodies that are in orbit around this celestial body.

Orbit **getOrbit**()
The orbit of the body.

float **getMass**()
The mass of the body, in kilograms.

float **getGravitationalParameter**()
The standard gravitational parameter of the body in m^3s^{-2}.

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

float **getRotationalPeriod**()
The sidereal rotational period of the body, in seconds.

float **getRotationalSpeed**()
The rotational speed of the body, in radians per second.

double **getRotationAngle**()
The current rotation angle of the body, in radians. A value between 0 and 2π

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

float **getEquatorialRadius**()
The equatorial radius of the body, in meters.

double **surfaceHeight**(double *latitude*, double *longitude*)
The height of the surface relative to mean sea level, in meters, at the given position. When over water this is equal to 0.

Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.

double **bedrockHeight**(double *latitude*, double *longitude*)
The height of the surface relative to mean sea level, in meters, at the given position. When over water, this is the height of the sea-bed and is therefore negative value.

Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.

org.javatuples.Triplet<Double, Double, Double> **mSLPosition**(double *latitude*, double *longitude*, *ReferenceFrame referenceFrame*)
The position at mean sea level at the given latitude and longitude, in the given reference frame.

Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **referenceFrame** (ReferenceFrame) – Reference frame for the returned position vector.

Returns Position as a vector.

org.javatuples.Triplet<Double, Double, Double> **surfacePosition**(double *latitude*, double *longitude*, *ReferenceFrame referenceFrame*)
The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position of the surface of the water.

Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **referenceFrame** (ReferenceFrame) – Reference frame for the returned position vector.

Returns Position as a vector.

org.javatuples.Triplet<Double, Double, Double> **bedrockPosition**(double *latitude*, double *longitude*, *ReferenceFrame referenceFrame*)

The position of the surface at the given latitude and longitude, in the given reference frame. When over water, this is the position at the bottom of the sea-bed.

Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **referenceFrame** (ReferenceFrame) – Reference frame for the returned position vector.

Returns Position as a vector.

org.javatuples.Triplet<Double, Double, Double> **positionAtAltitude**(double *latitude*, double *longitude*, double *altitude*, *ReferenceFrame referenceFrame*)

The position at the given latitude, longitude and altitude, in the given reference frame.

Parameters

- **latitude** (*double*) – Latitude in degrees.
- **longitude** (*double*) – Longitude in degrees.
- **altitude** (*double*) – Altitude in meters above sea level.
- **referenceFrame** (ReferenceFrame) – Reference frame for the returned position vector.

Returns Position as a vector.

double **altitudeAtPosition**(org.javatuples.Triplet<Double, Double, Double> *position*, *ReferenceFrame referenceFrame*)

The altitude, in meters, of the given position in the given reference frame.

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position as a vector.
- **referenceFrame** (ReferenceFrame) – Reference frame for the position vector.

double **latitudeAtPosition**(org.javatuples.Triplet<Double, Double, Double> *position*, *ReferenceFrame referenceFrame*)

The latitude of the given position, in the given reference frame.

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position as a vector.
- **referenceFrame** (ReferenceFrame) – Reference frame for the position vector.

double **longitudeAtPosition**(org.javatuples.Triplet<Double, Double, Double> *position*, *ReferenceFrame referenceFrame*)
The longitude of the given position, in the given reference frame.

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position as a vector.
- **referenceFrame** (ReferenceFrame) – Reference frame for the position vector.

float **getSphereOfInfluence**()
The radius of the sphere of influence of the body, in meters.

boolean **getHasAtmosphere**()
true if the body has an atmosphere.

float **getAtmosphereDepth**()
The depth of the atmosphere, in meters.

double **atmosphericDensityAtPosition**(org.javatuples.Triplet<Double, Double, Double> *position*, *ReferenceFrame referenceFrame*)
The atmospheric density at the given position, in kg/m^3, in the given reference frame.

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – The position vector at which to measure the density.
- **referenceFrame** (ReferenceFrame) – Reference frame that the position vector is in.

boolean **getHasAtmosphericOxygen**()
true if there is oxygen in the atmosphere, required for air-breathing engines.

double **temperatureAt**(org.javatuples.Triplet<Double, Double, Double> *position*, *ReferenceFrame referenceFrame*)
The temperature on the body at the given position, in the given reference frame.

Parameters

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position as a vector.
- **referenceFrame** (ReferenceFrame) – The reference frame that the position is in.

Note: This calculation is performed using the bodies current position, which means that the value could be wrong if you want to know the temperature in the far future.

---

double **densityAt**(double *altitude*)
Gets the air density, in kg/m^3, for the specified altitude above sea level, in meters.

Parameters

- **altitude** (*double*) –

Note: This is an approximation, because actual calculations, taking sun exposure into account to compute air temperature, require us to know the exact point on the body where the density is to be computed (knowing the altitude is not enough). However, the difference is small for high altitudes, so it makes very little difference for trajectory prediction.

---

double **pressureAt**(double *altitude*)
Gets the air pressure, in Pascals, for the specified altitude above sea level, in meters.

Parameters

- **altitude** (*double*) –

java.util.Set<String> **getBiomes**()
The biomes present on this body.

String **biomeAt**(double *latitude*, double *longitude*)
The biome at the given latitude and longitude, in degrees.

Parameters

- **latitude** (*double*) –
- **longitude** (*double*) –

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

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

ReferenceFrame **getReferenceFrame**()
The reference frame that is fixed relative to the celestial body.

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

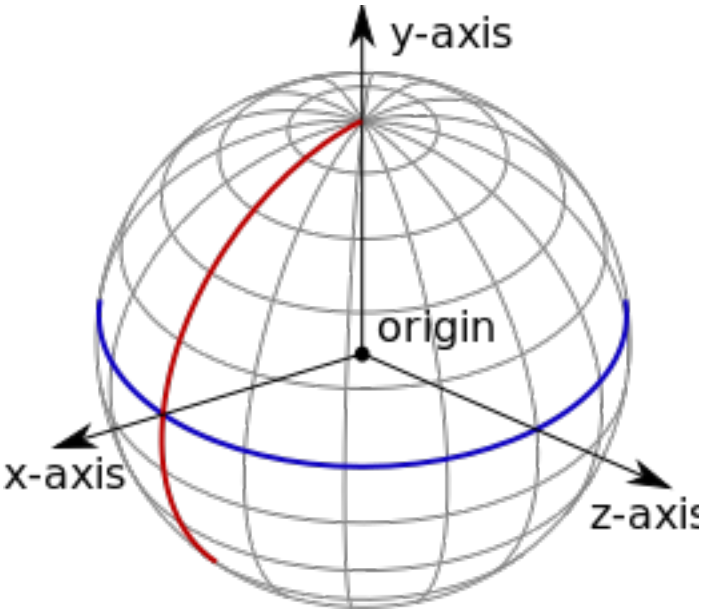


Fig. 6.6: Celestial body reference frame origin and axes. The equator is shown in blue, and the prime meridian in red.

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

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

ReferenceFrame **getOrbitalReferenceFrame**()
The reference frame that is fixed relative to this celestial body, but orientated with the body's orbital prograde/normal/radial directions.

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

org.javatuples.Triplet<Double, Double, Double> **position**(*ReferenceFrame referenceFrame*)
The position of the center of the body, in the specified reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

org.javatuples.Triplet<Double, Double, Double> **velocity**(*ReferenceFrame referenceFrame*)
The linear velocity of the body, in the specified reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned velocity vector is in.

Returns The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

org.javatuples.Quartet<Double, Double, Double, Double> **rotation**(*ReferenceFrame referenceFrame*)
The rotation of the body, in the specified reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned rotation is in.

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

org.javatuples.Triplet<Double, Double, Double> **direction**(*ReferenceFrame referenceFrame*)
The direction in which the north pole of the celestial body is pointing, in the specified reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **angularVelocity**(*ReferenceFrame referenceFrame*)

The angular velocity of the body in the specified reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame the returned angular velocity is in.

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

6.3.4 Flight

public class **Flight**

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

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

---

float **getGForce**()

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

double **getMeanAltitude**()

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

double **getSurfaceAltitude**()

The altitude above the surface of the body or sea level, whichever is closer, in meters. Measured from the center of mass of the vessel.

double **getBedrockAltitude**()

The altitude above the surface of the body, in meters. When over water, this is the altitude above the sea floor. Measured from the center of mass of the vessel.

double **getElevation**()

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

double **getLatitude**()

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

double **getLongitude**()

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

org.javatuples.Triplet<Double, Double, Double> **getVelocity**()

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

Returns The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the vessel in meters per second.

double **getSpeed**()

The speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

double **getHorizontalSpeed**()

The horizontal speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

double **getVerticalSpeed**()
The vertical speed of the vessel in meters per second, in the reference frame *ReferenceFrame*.

org.javatuples.Triplet<Double, Double, Double> **getCenterOfMass**()
The position of the center of mass of the vessel, in the reference frame *ReferenceFrame*

Returns The position as a vector.

org.javatuples.Quartet<Double, Double, Double, Double> **getRotation**()
The rotation of the vessel, in the reference frame *ReferenceFrame*

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

org.javatuples.Triplet<Double, Double, Double> **getDirection**()
The direction that the vessel is pointing in, in the reference frame *ReferenceFrame*.

Returns The direction as a unit vector.

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

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

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

org.javatuples.Triplet<Double, Double, Double> **getPrograde**()
The prograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

Returns The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getRetrograde**()
The retrograde direction of the vessels orbit, in the reference frame *ReferenceFrame*.

Returns The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getNormal**()
The direction normal to the vessels orbit, in the reference frame *ReferenceFrame*.

Returns The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getAntiNormal**()
The direction opposite to the normal of the vessels orbit, in the reference frame *ReferenceFrame*.

Returns The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getRadial**()
The radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

Returns The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **getAntiRadial**()
The direction opposite to the radial direction of the vessels orbit, in the reference frame *ReferenceFrame*.

Returns The direction as a unit vector.

float **getAtmosphereDensity**()
The current density of the atmosphere around the vessel, in kg/m^3.

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

float **getStaticPressure**()
The static atmospheric pressure acting on the vessel, in Pascals.

float **getStaticPressureAtMSL**()
The static atmospheric pressure at mean sea level, in Pascals.

org.javatuples.Triplet<Double, Double, Double> **getAerodynamicForce**()
The total aerodynamic forces acting on the vessel, in reference frame *ReferenceFrame*.

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

org.javatuples.Triplet<Double, Double, Double> **simulateAerodynamicForceAt**(*CelestialBody body*, org.javatuples.Triplet<Double, Double, Double> *position*, org.javatuples.Triplet<Double, Double, Double> *velocity*)
Simulate and return the total aerodynamic forces acting on the vessel, if it where to be traveling with the given velocity at the given position in the atmosphere of the given celestial body.

Parameters

- **body** (CelestialBody) –
- **position** (*org.javatuples.Triplet<Double,Double,Double>*) –
- **velocity** (*org.javatuples.Triplet<Double,Double,Double>*) –

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

org.javatuples.Triplet<Double, Double, Double> **getLift**()
The aerodynamic lift currently acting on the vessel.

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

org.javatuples.Triplet<Double, Double, Double> **getDrag**()
The aerodynamic drag currently acting on the vessel.

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

float **getSpeedOfSound**()
The speed of sound, in the atmosphere around the vessel, in m/s.

float **getMach**()
The speed of the vessel, in multiples of the speed of sound.

float **getReynoldsNumber**()
The vessels Reynolds number.

Note: Requires Ferram Aerospace Research.

---

float **getTrueAirSpeed**()
The true air speed of the vessel, in meters per second.

float **getEquivalentAirSpeed**()
The equivalent air speed of the vessel, in meters per second.

float **getTerminalVelocity**()
An estimate of the current terminal velocity of the vessel, in meters per second. This is the speed at which the drag forces cancel out the force of gravity.

float **getAngleOfAttack**()
The pitch angle between the orientation of the vessel and its velocity vector, in degrees.

float **getSideslipAngle**()
The yaw angle between the orientation of the vessel and its velocity vector, in degrees.

float **getTotalAirTemperature**()
The total air temperature of the atmosphere around the vessel, in Kelvin. This includes the *Flight.getStaticAirTemperature()* and the vessel's kinetic energy.

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

float **getStallFraction**()
The current amount of stall, between 0 and 1. A value greater than 0.005 indicates a minor stall and a value greater than 0.5 indicates a large-scale stall.

Note: Requires Ferram Aerospace Research.

---

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

Note: Requires Ferram Aerospace Research.

---

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

Note: Requires Ferram Aerospace Research.

---

float **getBallisticCoefficient**()
The ballistic coefficient.

Note: Requires Ferram Aerospace Research.

---

float **getThrustSpecificFuelConsumption**()
The thrust specific fuel consumption for the jet engines on the vessel. This is a measure of the efficiency of the engines, with a lower value indicating a more efficient vessel. This value is the number of Newtons of fuel that are burned, per hour, to produce one newton of thrust.

Note: Requires Ferram Aerospace Research.

6.3.5 Orbit

public class **Orbit**

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

CelestialBody **getBody**()

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

double **getApoapsis**()

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

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

---

double **getPeriapsis**()

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

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

---

double **getApoapsisAltitude**()

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

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

---

double **getPeriapsisAltitude**()

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

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

---

double **getSemiMajorAxis**()

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

double **getSemiMinorAxis**()

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

double **getRadius**()

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

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

---

double **radiusAt**(double *ut*)

The orbital radius at the given time, in meters.

Parameters

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

org.javatuples.Triplet<Double, Double, Double> **positionAt**(double *ut*, *ReferenceFrame referenceFrame*)

The position at a given time, in the specified reference frame.

Parameters

- **ut** (*double*) – The universal time to measure the position at.
- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

double **getSpeed**()
The current orbital speed of the object in meters per second.

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

---

double **getPeriod**()
The orbital period, in seconds.

double **getTimeToApoapsis**()
The time until the object reaches apoapsis, in seconds.

double **getTimeToPeriapsis**()
The time until the object reaches periapsis, in seconds.

double **getEccentricity**()
The eccentricity of the orbit.

double **getInclination**()
The inclination of the orbit, in radians.

double **getLongitudeOfAscendingNode**()
The longitude of the ascending node, in radians.

double **getArgumentOfPeriapsis**()
The argument of periapsis, in radians.

double **getMeanAnomalyAtEpoch**()
The mean anomaly at epoch.

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

double **getMeanAnomaly**()
The mean anomaly.

double **meanAnomalyAtUT**(double *ut*)
The mean anomaly at the given time.

Parameters

- **ut** (*double*) – The universal time in seconds.

double **getEccentricAnomaly**()
The eccentric anomaly.

double **eccentricAnomalyAtUT**(double *ut*)
The eccentric anomaly at the given universal time.

Parameters

- **ut** (*double*) – The universal time, in seconds.

double **getTrueAnomaly**()
The true anomaly.

double **trueAnomalyAtUT**(double *ut*)
The true anomaly at the given time.

Parameters

- **ut** (*double*) – The universal time in seconds.

double **trueAnomalyAtRadius**(double *radius*)
The true anomaly at the given orbital radius.

Parameters

- **radius** (*double*) – The orbital radius in meters.

double **uTAtTrueAnomaly**(double *trueAnomaly*)
The universal time, in seconds, corresponding to the given true anomaly.

Parameters

- **trueAnomaly** (*double*) – True anomaly.

double **radiusAtTrueAnomaly**(double *trueAnomaly*)
The orbital radius at the point in the orbit given by the true anomaly.

Parameters

- **trueAnomaly** (*double*) – The true anomaly.

double **trueAnomalyAtAN**(*Vessel target*)
The true anomaly of the ascending node with the given target vessel.

Parameters

- **target** (Vessel) – Target vessel.

double **trueAnomalyAtDN**(*Vessel target*)
The true anomaly of the descending node with the given target vessel.

Parameters

- **target** (Vessel) – Target vessel.

double **getOrbitalSpeed**()
The current orbital speed in meters per second.

double **orbitalSpeedAt**(double *time*)
The orbital speed at the given time, in meters per second.

Parameters

- **time** (*double*) – Time from now, in seconds.

static org.javatuples.Triplet<Double, Double, Double> **referencePlaneNormal**(*Connection connection*, *ReferenceFrame referenceFrame*)
The direction that is normal to the orbits reference plane, in the given reference frame. The reference plane is the plane from which the orbits inclination is measured.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

static org.javatuples.Triplet<Double, Double, Double> **referencePlaneDirection**(*Connection connection*, *ReferenceFrame referenceFrame*)

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

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

double **relativeInclination**(*Vessel target*)

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

Parameters

- **target** (Vessel) – Target vessel.

double **getTimeToSOIChange**()

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

Orbit **getNextOrbit**()

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

double **timeOfClosestApproach**(*Vessel target*)

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

Parameters

- **target** (Vessel) – Target vessel.

Returns The universal time at closest approach, in seconds.

double **distanceAtClosestApproach**(*Vessel target*)

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

Parameters

- **target** (Vessel) – Target vessel.

java.util.List<java.util.List<Double>> **listClosestApproaches**(*Vessel target*, int *orbits*)

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

Parameters

- **target** (Vessel) – Target vessel.
- **orbits** (*int*) – The number of future orbits to search.

Returns A list of two lists. The first is a list of times at closest approach, as universal times in seconds. The second is a list of corresponding distances at closest approach, in meters.

6.3.6 Control

public class **Control**

Used to manipulate the controls of a vessel. This includes adjusting the throttle, enabling/disabling systems such as SAS and RCS, or altering the direction in which the vessel is pointing. Obtained by calling *Vessel.getControl()*.

Note: Control inputs (such as pitch, yaw and roll) are zeroed when all clients that have set one or more of these inputs are no longer connected.

ControlSource **getSource**()
The source of the vessels control, for example by a kerbal or a probe core.

ControlState **getState**()
The control state of the vessel.

boolean **getSAS**()

void **setSAS**(boolean *value*)
The state of SAS.

Note: Equivalent to `AutoPilot.getSAS()`

SASMode **getSASMode**()

void **setSASMode**(*SASMode value*)
The current `SASMode`. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

Note: Equivalent to `AutoPilot.getSASMode()`

SpeedMode **getSpeedMode**()

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

boolean **getRCS**()

void **setRCS**(boolean *value*)
The state of RCS.

boolean **getReactionWheels**()

void **setReactionWheels**(boolean *value*)
Returns whether all reactive wheels on the vessel are active, and sets the active state of all reaction wheels. See `ReactionWheel.getActive()`.

boolean **getGear**()

void **setGear**(boolean *value*)
The state of the landing gear/legs.

boolean **getLegs**()

void **setLegs**(boolean *value*)
Returns whether all landing legs on the vessel are deployed, and sets the deployment state of all landing legs. Does not include wheels (for example landing gear). See `Leg.getDeployed()`.

boolean **getWheels**()

void **setWheels**(boolean *value*)
Returns whether all wheels on the vessel are deployed, and sets the deployment state of all wheels. Does not include landing legs. See `Wheel.getDeployed()`.

boolean **getLights**()

void **setLights**(boolean *value*)
The state of the lights.

boolean **getBrakes**()

void **setBrakes**(boolean *value*)
The state of the wheel brakes.

boolean **getAntennas**()

void **setAntennas**(boolean *value*)
Returns whether all antennas on the vessel are deployed, and sets the deployment state of all antennas. See *Antenna.getDeployed()*.

boolean **getCargoBays**()

void **setCargoBays**(boolean *value*)
Returns whether any of the cargo bays on the vessel are open, and sets the open state of all cargo bays. See *CargoBay.getOpen()*.

boolean **getIntakes**()

void **setIntakes**(boolean *value*)
Returns whether all of the air intakes on the vessel are open, and sets the open state of all air intakes. See *Intake.getOpen()*.

boolean **getParachutes**()

void **setParachutes**(boolean *value*)
Returns whether all parachutes on the vessel are deployed, and sets the deployment state of all parachutes. Cannot be set to `false`. See *Parachute.getDeployed()*.

boolean **getRadiators**()

void **setRadiators**(boolean *value*)
Returns whether all radiators on the vessel are deployed, and sets the deployment state of all radiators. See *Radiator.getDeployed()*.

boolean **getResourceHarvesters**()

void **setResourceHarvesters**(boolean *value*)
Returns whether all of the resource harvesters on the vessel are deployed, and sets the deployment state of all resource harvesters. See *ResourceHarvester.getDeployed()*.

boolean **getResourceHarvestersActive**()

void **setResourceHarvestersActive**(boolean *value*)
Returns whether any of the resource harvesters on the vessel are active, and sets the active state of all resource harvesters. See *ResourceHarvester.getActive()*.

boolean **getSolarPanels**()

void **setSolarPanels**(boolean *value*)
Returns whether all solar panels on the vessel are deployed, and sets the deployment state of all solar panels. See *SolarPanel.getDeployed()*.

boolean **getAbort**()

void **setAbort**(boolean *value*)
The state of the abort action group.

float **getThrottle**()

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

ControlInputMode **getInputMode**()

void **setInputMode**(*ControlInputMode value*)
Sets the behavior of the pitch, yaw, roll and translation control inputs. When set to additive, these inputs are added to the vessels current inputs. This mode is the default. When set to override, these inputs (if non-zero) override the vessels inputs. This mode prevents keyboard control, or SAS, from interfering with the controls when they are set.

float **getPitch**()

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

float **getYaw**()

void **setYaw**(float *value*)
The state of the yaw control. A value between -1 and 1. Equivalent to the a and d keys.

float **getRoll**()

void **setRoll**(float *value*)
The state of the roll control. A value between -1 and 1. Equivalent to the q and e keys.

float **getForward**()

void **setForward**(float *value*)
The state of the forward translational control. A value between -1 and 1. Equivalent to the h and n keys.

float **getUp**()

void **setUp**(float *value*)
The state of the up translational control. A value between -1 and 1. Equivalent to the i and k keys.

float **getRight**()

void **setRight**(float *value*)
The state of the right translational control. A value between -1 and 1. Equivalent to the j and l keys.

float **getWheelThrottle**()

void **setWheelThrottle**(float *value*)
The state of the wheel throttle. A value between -1 and 1. A value of 1 rotates the wheels forwards, a value of -1 rotates the wheels backwards.

float **getWheelSteering**()

void **setWheelSteering**(float *value*)
The state of the wheel steering. A value between -1 and 1. A value of 1 steers to the left, and a value of -1 steers to the right.

int **getCurrentStage**()
The current stage of the vessel. Corresponds to the stage number in the in-game UI.

java.util.List<*Vessel*> **activateNextStage**()
Activates the next stage. Equivalent to pressing the space bar in-game.

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

Note: When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to `getActiveVessel()` no longer refer to the active vessel.

---

boolean **getActionGroup**(int *group*)
Returns `true` if the given action group is enabled.

Parameters

- **group** (*int*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the Extended Action Groups mod is installed.

void **setActionGroup**(int *group*, boolean *state*)
Sets the state of the given action group.

Parameters

- **group** (*int*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the Extended Action Groups mod is installed.
- **state** (*boolean*) –

void **toggleActionGroup**(int *group*)
Toggles the state of the given action group.

Parameters

- **group** (*int*) – A number between 0 and 9 inclusive, or between 0 and 250 inclusive when the Extended Action Groups mod is installed.

Node **addNode**(double *ut*, float *prograde*, float *normal*, float *radial*)
Creates a maneuver node at the given universal time, and returns a *Node* object that can be used to modify it. Optionally sets the magnitude of the delta-v for the maneuver node in the prograde, normal and radial directions.

Parameters

- **ut** (*double*) – Universal time of the maneuver node.
- **prograde** (*float*) – Delta-v in the prograde direction.
- **normal** (*float*) – Delta-v in the normal direction.
- **radial** (*float*) – Delta-v in the radial direction.

java.util.List<*Node*> **getNodes**()
Returns a list of all existing maneuver nodes, ordered by time from first to last.

void **removeNodes**()
Remove all maneuver nodes.

public enum **ControlState**
The control state of a vessel. See *Control.getState()*.

public *ControlState* **FULL**
Full controllable.

public *ControlState* **PARTIAL**
Partially controllable.

public *ControlState* **NONE**
Not controllable.

public enum **ControlSource**
The control source of a vessel. See *Control.getSource()*.

public *ControlSource* **KERBAL**
Vessel is controlled by a Kerbal.

public *ControlSource* **PROBE**
Vessel is controlled by a probe core.

public *ControlSource* **NONE**
Vessel is not controlled.

public enum **SASMode**
The behavior of the SAS auto-pilot. See *AutoPilot.getSASMode()*.

public *SASMode* **STABILITY_ASSIST**
Stability assist mode. Dampen out any rotation.

public *SASMode* **MANEUVER**
Point in the burn direction of the next maneuver node.

public *SASMode* **PROGRADE**
Point in the prograde direction.

public *SASMode* **RETROGRADE**
Point in the retrograde direction.

public *SASMode* **NORMAL**
Point in the orbit normal direction.

public *SASMode* **ANTI_NORMAL**
Point in the orbit anti-normal direction.

public *SASMode* **RADIAL**
Point in the orbit radial direction.

public *SASMode* **ANTI_RADIAL**
Point in the orbit anti-radial direction.

public *SASMode* **TARGET**
Point in the direction of the current target.

public *SASMode* **ANTI_TARGET**
Point away from the current target.

public enum **SpeedMode**
The mode of the speed reported in the navball. See *Control.getSpeedMode()*.

public *SpeedMode* **ORBIT**
Speed is relative to the vessel's orbit.

public *SpeedMode* **SURFACE**
Speed is relative to the surface of the body being orbited.

public *SpeedMode* **TARGET**
Speed is relative to the current target.

public enum **ControlInputMode**
See *Control.getInputMode()*.

public *ControlInputMode* **ADDITIVE**
Control inputs are added to the vessels current control inputs.

public *ControlInputMode* **OVERRIDE**
Control inputs (when they are non-zero) override the vessels current control inputs.

6.3.7 Communications

public class **Comms**
Used to interact with CommNet for a given vessel. Obtained by calling *Vessel.getComms()*.

boolean **getCanCommunicate**()
Whether the vessel can communicate with KSC.

boolean **getCanTransmitScience**()
Whether the vessel can transmit science data to KSC.

double **getSignalStrength**()
Signal strength to KSC.

double **getSignalDelay**()
Signal delay to KSC in seconds.

double **getPower**()
The combined power of all active antennae on the vessel.

java.util.List<*CommLink*> **getControlPath**()
The communication path used to control the vessel.

public class **CommLink**
Represents a communication node in the network. For example, a vessel or the KSC.

CommLinkType **getType**()
The type of link.

double **getSignalStrength**()
Signal strength of the link.

CommNode **getStart**()
Start point of the link.

CommNode **getEnd**()
Start point of the link.

public enum **CommLinkType**
The type of a communication link. See *CommLink.getType()*.

public *CommLinkType* **HOME**
Link is to a base station on Kerbin.

public *CommLinkType* **CONTROL**
Link is to a control source, for example a manned spacecraft.

public *CommLinkType* **RELAY**
Link is to a relay satellite.

public class **CommNode**
Represents a communication node in the network. For example, a vessel or the KSC.

String **getName**()
Name of the communication node.

boolean **getIsHome**()
Whether the communication node is on Kerbin.

boolean **getIsControlPoint**()
Whether the communication node is a control point, for example a manned vessel.

boolean **getIsVessel**()
Whether the communication node is a vessel.

Vessel **getVessel**()
The vessel for this communication node.

6.3.8 Parts

The following classes allow interaction with a vessels individual parts.

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

- *Staging*

Parts

public class **Parts**

Instances of this class are used to interact with the parts of a vessel. An instance can be obtained by calling `Vessel.getParts()`.

java.util.List<*Part*> **getAll**()
A list of all of the vessels parts.

Part **getRoot**()
The vessels root part.

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

Part **getControlling**()

void **setControlling**(*Part value*)
The part from which the vessel is controlled.

java.util.List<*Part*> **withName**(String *name*)
A list of parts whose `Part.getName()` is *name*.

Parameters

- **name** (*String*) –

java.util.List<*Part*> **withTitle**(String *title*)
A list of all parts whose `Part.getTitle()` is *title*.

Parameters

- **title** (*String*) –

java.util.List<*Part*> **withTag**(String *tag*)
A list of all parts whose `Part.getTag()` is *tag*.

Parameters

- **tag** (*String*) –

java.util.List<*Part*> **withModule**(String *moduleName*)
A list of all parts that contain a `Module` whose `Module.getName()` is *moduleName*.

Parameters

- **moduleName** (*String*) –

java.util.List<*Part*> **inStage**(int *stage*)
A list of all parts that are activated in the given *stage*.

Parameters

- **stage** (*int*) –

Note: See the discussion on *Staging*.

---

java.util.List<*Part*> **inDecoupleStage**(int *stage*)
A list of all parts that are decoupled in the given *stage*.

Parameters

- **stage** (*int*) –

Note: See the discussion on *Staging*.

java.util.List<*Module*> **modulesWithName**(String *moduleName*)
A list of modules (combined across all parts in the vessel) whose `Module.getName()` is *moduleName*.

Parameters

- **moduleName** (*String*) –

java.util.List<*Antenna*> **getAntennas**()
A list of all antennas in the vessel.

java.util.List<*CargoBay*> **getCargoBays**()
A list of all cargo bays in the vessel.

java.util.List<*ControlSurface*> **getControlSurfaces**()
A list of all control surfaces in the vessel.

java.util.List<*Decoupler*> **getDecouplers**()
A list of all decouplers in the vessel.

java.util.List<*DockingPort*> **getDockingPorts**()
A list of all docking ports in the vessel.

java.util.List<*Engine*> **getEngines**()
A list of all engines in the vessel.

Note: This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters, jet engines and RCS thrusters.

java.util.List<*Experiment*> **getExperiments**()
A list of all science experiments in the vessel.

java.util.List<*Fairing*> **getFairings**()
A list of all fairings in the vessel.

java.util.List<*Intake*> **getIntakes**()
A list of all intakes in the vessel.

java.util.List<*Leg*> **getLegs**()
A list of all landing legs attached to the vessel.

java.util.List<*LaunchClamp*> **getLaunchClamps**()
A list of all launch clamps attached to the vessel.

java.util.List<*Light*> **getLights**()
A list of all lights in the vessel.

java.util.List<*Parachute*> **getParachutes**()
A list of all parachutes in the vessel.

java.util.List<*Radiator*> **getRadiators**()
A list of all radiators in the vessel.

java.util.List<*RCS*> **getRCS**()
A list of all RCS blocks/thrusters in the vessel.

java.util.List<*ReactionWheel*> **getReactionWheels**()
A list of all reaction wheels in the vessel.

java.util.List<*ResourceConverter*> **getResourceConverters**()
A list of all resource converters in the vessel.

java.util.List<*ResourceHarvester*> **getResourceHarvesters**()
A list of all resource harvesters in the vessel.

java.util.List<*Sensor*> **getSensors**()
A list of all sensors in the vessel.

java.util.List<*SolarPanel*> **getSolarPanels**()
A list of all solar panels in the vessel.

java.util.List<*Wheel*> **getWheels**()
A list of all wheels in the vessel.

Part

public class **Part**
Represents an individual part. Vessels are made up of multiple parts. Instances of this class can be obtained by several methods in `Parts`.

String **getName**()
Internal name of the part, as used in part cfg files. For example "Mark1-2Pod".

String **getTitle**()
Title of the part, as shown when the part is right clicked in-game. For example "Mk1-2 Command Pod".

String **getTag**()

void **setTag**(String *value*)
The name tag for the part. Can be set to a custom string using the in-game user interface.

> **Note:** This requires either the NameTag or kOS mod to be installed.

boolean **getHighlighted**()

void **setHighlighted**(boolean *value*)
Whether the part is highlighted.

org.javatuples.Triplet<Double, Double, Double> **getHighlightColor**()

void **setHighlightColor**(org.javatuples.Triplet<Double, Double, Double> *value*)
The color used to highlight the part, as an RGB triple.

double **getCost**()
The cost of the part, in units of funds.

Vessel **getVessel**()
The vessel that contains this part.

Part **getParent**()
The parts parent. Returns `null` if the part does not have a parent. This, in combination with `Part.getChildren()`, can be used to traverse the vessels parts tree.

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

java.util.List<*Part*> **getChildren**()
The parts children. Returns an empty list if the part has no children. This, in combination with `Part.getParent()`, can be used to traverse the vessels parts tree.

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

boolean **getAxiallyAttached**()
Whether the part is axially attached to its parent, i.e. on the top or bottom of its parent. If the part has no parent, returns `false`.

Note: See the discussion on *Attachment Modes*.

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

Note: See the discussion on *Attachment Modes*.

int **getStage**()
The stage in which this part will be activated. Returns -1 if the part is not activated by staging.

Note: See the discussion on *Staging*.

int **getDecoupleStage**()
The stage in which this part will be decoupled. Returns -1 if the part is never decoupled from the vessel.

Note: See the discussion on *Staging*.

boolean **getMassless**()
Whether the part is massless.

double **getMass**()
The current mass of the part, including resources it contains, in kilograms. Returns zero if the part is massless.

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

boolean **getShielded**()
Whether the part is shielded from the exterior of the vessel, for example by a fairing.

float **getDynamicPressure**()
The dynamic pressure acting on the part, in Pascals.

double **getImpactTolerance**()
The impact tolerance of the part, in meters per second.

double **getTemperature**()
Temperature of the part, in Kelvin.

double **getSkinTemperature**()
Temperature of the skin of the part, in Kelvin.

double **getMaxTemperature**()
Maximum temperature that the part can survive, in Kelvin.

double **getMaxSkinTemperature**()
Maximum temperature that the skin of the part can survive, in Kelvin.

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

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

float **getThermalResourceMass**()
A measure of how much energy it takes to increase the temperature of the resources contained in the part, in Joules per Kelvin.

float **getThermalConductionFlux**()
The rate at which heat energy is conducting into or out of the part via contact with other parts. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalConvectionFlux**()
The rate at which heat energy is convecting into or out of the part from the surrounding atmosphere. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalRadiationFlux**()
The rate at which heat energy is radiating into or out of the part from the surrounding environment. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

float **getThermalInternalFlux**()
The rate at which heat energy is begin generated by the part. For example, some engines generate heat by combusting fuel. Measured in energy per unit time, or power, in Watts. A positive value means the part is gaining heat energy, and negative means it is losing heat energy.

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

Resources **getResources**()
A *Resources* object for the part.

boolean **getCrossfeed**()
Whether this part is crossfeed capable.

boolean **getIsFuelLine**()
Whether this part is a fuel line.

java.util.List<*Part*> **getFuelLinesFrom**()
The parts that are connected to this part via fuel lines, where the direction of the fuel line is into this part.

Note: See the discussion on *Fuel Lines*.

java.util.List<*Part*> **getFuelLinesTo**()
The parts that are connected to this part via fuel lines, where the direction of the fuel line is out of this part.

Note: See the discussion on *Fuel Lines*.

java.util.List<*Module*> **getModules**()
The modules for this part.

Antenna **getAntenna**()
A *Antenna* if the part is an antenna, otherwise null.

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

ControlSurface **getControlSurface**()
A *ControlSurface* if the part is an aerodynamic control surface, otherwise null.

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

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

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

Experiment **getExperiment**()
An *Experiment* if the part is a science experiment, otherwise null.

Fairing **getFairing**()
A *Fairing* if the part is a fairing, otherwise null.

Intake **getIntake**()
An *Intake* if the part is an intake, otherwise null.

Note: This includes any part that generates thrust. This covers many different types of engine, including liquid fuel rockets, solid rocket boosters and jet engines. For RCS thrusters see *RCS*.

Leg **getLeg**()
A *Leg* if the part is a landing leg, otherwise null.

LaunchClamp **getLaunchClamp**()
A *LaunchClamp* if the part is a launch clamp, otherwise null.

Light **getLight**()
A *Light* if the part is a light, otherwise null.

Parachute **getParachute**()
A *Parachute* if the part is a parachute, otherwise null.

Radiator **getRadiator**()
A *Radiator* if the part is a radiator, otherwise null.

RCS **getRCS**()
A *RCS* if the part is an RCS block/thruster, otherwise null.

ReactionWheel **getReactionWheel**()
A *ReactionWheel* if the part is a reaction wheel, otherwise null.

ResourceConverter **getResourceConverter**()
A *ResourceConverter* if the part is a resource converter, otherwise null.

ResourceHarvester **getResourceHarvester**()
A *ResourceHarvester* if the part is a resource harvester, otherwise null.

Sensor **getSensor**()
A *Sensor* if the part is a sensor, otherwise null.

SolarPanel **getSolarPanel**()
A *SolarPanel* if the part is a solar panel, otherwise null.

Wheel **getWheel**()
A *Wheel* if the part is a wheel, otherwise null.

org.javatuples.Triplet<Double, Double, Double> **position**(*ReferenceFrame referenceFrame*)
The position of the part in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

Note: This is a fixed position in the part, defined by the parts model. It s not necessarily the same as the parts center of mass. Use *Part.centerOfMass(ReferenceFrame)* to get the parts center of mass.

---

org.javatuples.Triplet<Double, Double, Double> **centerOfMass**(*ReferenceFrame referenceFrame*)
The position of the parts center of mass in the given reference frame. If the part is physicsless, this is equivalent to *Part.position(ReferenceFrame)*.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **bound**

The axis-aligned bounding box of the part in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vectors are in.

Returns The positions of the minimum and maximum vertices of the box, as position vectors.

Note: This is computed from the collision mesh of the part. If the part is not collidable, the box has zero volume and is centered on the *Part.position(ReferenceFrame)* of the part.

---

org.javatuples.Triplet<Double, Double, Double> **direction**(*ReferenceFrame referenceFrame*)
The direction the part points in, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

org.javatuples.Triplet<Double, Double, Double> **velocity**(*ReferenceFrame referenceFrame*)
The linear velocity of the part in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned velocity vector is in.

Returns The velocity as a vector. The vector points in the direction of travel, and its magnitude is the speed of the body in meters per second.

org.javatuples.Quartet<Double, Double, Double, Double> **rotation**(*ReferenceFrame referenceFrame*)
The rotation of the part, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned rotation is in.

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

org.javatuples.Triplet<Double, Double, Double> **getMomentOfInertia**()
The moment of inertia of the part in $kg.m^2$ around its center of mass in the parts reference frame (*ReferenceFrame*).

java.util.List<Double> **getInertiaTensor**()
The inertia tensor of the part in the parts reference frame (*ReferenceFrame*). Returns the 3x3 matrix as a list of elements, in row-major order.

ReferenceFrame **getReferenceFrame**()
The reference frame that is fixed relative to this part, and centered on a fixed position within the part, defined by the parts model.

- The origin is at the position of the part, as returned by *Part.position(ReferenceFrame)*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

Note: For docking port parts, this reference frame is not necessarily equivalent to the reference frame for the docking port, returned by *DockingPort.getReferenceFrame()*.

ReferenceFrame **getCenterOfMassReferenceFrame**()
The reference frame that is fixed relative to this part, and centered on its center of mass.

- The origin is at the center of mass of the part, as returned by *Part.centerOfMass(ReferenceFrame)*.
- The axes rotate with the part.
- The x, y and z axis directions depend on the design of the part.

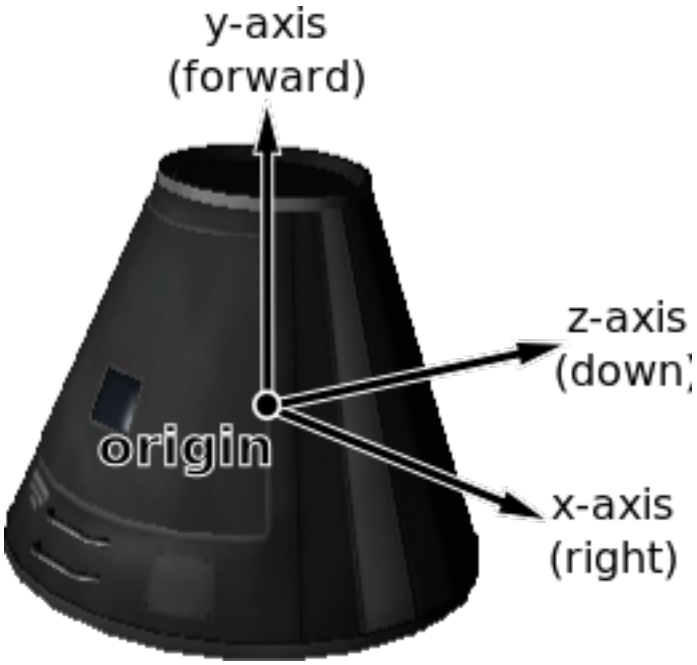


Fig. 6.7: Mk1 Command Pod reference frame origin and axes

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

Force **addForce**(org.javatuples.Triplet<Double, Double, Double> *force*, org.javatuples.Triplet<Double, Double, Double> *position*, *ReferenceFrame referenceFrame*)

Exert a constant force on the part, acting at the given position.

Parameters

- **force** (*org.javatuples.Triplet<Double,Double,Double>*) – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.
- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – The position at which the force acts, as a vector.
- **referenceFrame** (ReferenceFrame) – The reference frame that the force and position are in.

Returns An object that can be used to remove or modify the force.

void **instantaneousForce**(org.javatuples.Triplet<Double, Double, Double> *force*, org.javatuples.Triplet<Double, Double, Double> *position*, *ReferenceFrame referenceFrame*)

Exert an instantaneous force on the part, acting at the given position.

Parameters

- **force** (*org.javatuples.Triplet<Double,Double,Double>*) – A vector pointing in the direction that the force acts, with its magnitude equal to the strength of the force in Newtons.

- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – The position at which the force acts, as a vector.
- **referenceFrame** (ReferenceFrame) – The reference frame that the force and position are in.

Note: The force is applied instantaneously in a single physics update.

public class **Force**

Obtained by calling *Part.addForce(org.javatuples.Triplet<Double,Double,Double>, org.javatuples.Triplet<Double,Double,Double>, ReferenceFrame)*.

Part **getPart**()
The part that this force is applied to.

org.javatuples.Triplet<Double, Double, Double> **getForceVector**()

void **setForceVector**(org.javatuples.Triplet<Double, Double, Double> *value*)
The force vector, in Newtons.

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

org.javatuples.Triplet<Double, Double, Double> **getPosition**()

void **setPosition**(org.javatuples.Triplet<Double, Double, Double> *value*)
The position at which the force acts, in reference frame *ReferenceFrame*.

Returns The position as a vector.

ReferenceFrame **getReferenceFrame**()

void **setReferenceFrame**(*ReferenceFrame value*)
The reference frame of the force vector and position.

void **remove**()
Remove the force.

Module

public class **Module**

This can be used to interact with a specific part module. This includes part modules in stock KSP, and those added by mods.

In KSP, each part has zero or more PartModules associated with it. Each one contains some of the functionality of the part. For example, an engine has a "ModuleEngines" part module that contains all the functionality of an engine.

String **getName**()
Name of the PartModule. For example, "ModuleEngines".

Part **getPart**()
The part that contains this module.

java.util.Map<String, String> **getFields**()
The modules field names and their associated values, as a dictionary. These are the values visible in the right-click menu of the part.

boolean **hasField**(String *name*)
Returns `true` if the module has a field with the given name.

Parameters

- **name** (*String*) – Name of the field.

String **getField**(String *name*)
Returns the value of a field.

Parameters

- **name** (*String*) – Name of the field.

void **setFieldInt**(String *name*, int *value*)
Set the value of a field to the given integer number.

Parameters

- **name** (*String*) – Name of the field.
- **value** (*int*) – Value to set.

void **setFieldFloat**(String *name*, float *value*)
Set the value of a field to the given floating point number.

Parameters

- **name** (*String*) – Name of the field.
- **value** (*float*) – Value to set.

void **setFieldString**(String *name*, String *value*)
Set the value of a field to the given string.

Parameters

- **name** (*String*) – Name of the field.
- **value** (*String*) – Value to set.

void **resetField**(String *name*)
Set the value of a field to its original value.

Parameters

- **name** (*String*) – Name of the field.

java.util.List<String> **getEvents**()
A list of the names of all of the modules events. Events are the clickable buttons visible in the right-click menu of the part.

boolean **hasEvent**(String *name*)
`true` if the module has an event with the given name.

Parameters

- **name** (*String*) –

void **triggerEvent**(String *name*)
Trigger the named event. Equivalent to clicking the button in the right-click menu of the part.

Parameters

- **name** (*String*) –

java.util.List<String> **getActions**()
A list of all the names of the modules actions. These are the parts actions that can be assigned to action groups in the in-game editor.

boolean **hasAction** (String *name*)
`true` if the part has an action with the given name.

Parameters

- **name** (*String*) –

void **setAction** (String *name*, boolean *value*)
Set the value of an action with the given name.

Parameters

- **name** (*String*) –
- **value** (*boolean*) –

Specific Types of Part

The following classes provide functionality for specific types of part.

- *Antenna*
- *Cargo Bay*
- *Control Surface*
- *Decoupler*
- *Docking Port*
- *Engine*
- *Experiment*
- *Fairing*
- *Intake*
- *Leg*
- *Launch Clamp*
- *Light*
- *Parachute*
- *Radiator*
- *Resource Converter*
- *Resource Harvester*
- *Reaction Wheel*
- *RCS*
- *Sensor*
- *Solar Panel*
- *Thruster*
- *Wheel*

Antenna

public class **Antenna**

An antenna. Obtained by calling *Part.getAntenna()*.

Part **getPart**()
The part object for this antenna.

AntennaState **getState**()
The current state of the antenna.

boolean **getDeployable**()
Whether the antenna is deployable.

boolean **getDeployed**()

void **setDeployed**(boolean *value*)
Whether the antenna is deployed.

Note: Fixed antennas are always deployed. Returns an error if you try to deploy a fixed antenna.

---

boolean **getCanTransmit**()
Whether data can be transmitted by this antenna.

void **transmit**()
Transmit data.

void **cancel**()
Cancel current transmission of data.

boolean **getAllowPartial**()

void **setAllowPartial**(boolean *value*)
Whether partial data transmission is permitted.

double **getPower**()
The power of the antenna.

boolean **getCombinable**()
Whether the antenna can be combined with other antennae on the vessel to boost the power.

double **getCombinableExponent**()
Exponent used to calculate the combined power of multiple antennae on a vessel.

float **getPacketInterval**()
Interval between sending packets in seconds.

float **getPacketSize**()
Amount of data sent per packet in Mits.

double **getPacketResourceCost**()
Units of electric charge consumed per packet sent.

public enum **AntennaState**

The state of an antenna. See *Antenna.getState()*.

public *AntennaState* **DEPLOYED**
Antenna is fully deployed.

public *AntennaState* **RETRACTED**
Antenna is fully retracted.

public *AntennaState* **DEPLOYING**
Antenna is being deployed.

public *AntennaState* **RETRACTING**
Antenna is being retracted.

public *AntennaState* **BROKEN**
Antenna is broken.

Cargo Bay

public class **CargoBay**
A cargo bay. Obtained by calling *`Part.getCargoBay()`*.

Part **getPart**()
The part object for this cargo bay.

CargoBayState **getState**()
The state of the cargo bay.

boolean **getOpen**()

void **setOpen**(boolean *value*)
Whether the cargo bay is open.

public enum **CargoBayState**
The state of a cargo bay. See *`CargoBay.getState()`*.

public *CargoBayState* **OPEN**
Cargo bay is fully open.

public *CargoBayState* **CLOSED**
Cargo bay closed and locked.

public *CargoBayState* **OPENING**
Cargo bay is opening.

public *CargoBayState* **CLOSING**
Cargo bay is closing.

Control Surface

public class **ControlSurface**
An aerodynamic control surface. Obtained by calling *`Part.getControlSurface()`*.

Part **getPart**()
The part object for this control surface.

boolean **getPitchEnabled**()

void **setPitchEnabled**(boolean *value*)
Whether the control surface has pitch control enabled.

boolean **getYawEnabled**()

void **setYawEnabled**(boolean *value*)
Whether the control surface has yaw control enabled.

boolean **getRollEnabled**()

void **setRollEnabled**(boolean *value*)
Whether the control surface has roll control enabled.

float **getAuthorityLimiter**()

void **setAuthorityLimiter**(float *value*)
The authority limiter for the control surface, which controls how far the control surface will move.

boolean **getInverted**()

void **setInverted**(boolean *value*)
Whether the control surface movement is inverted.

boolean **getDeployed**()

void **setDeployed**(boolean *value*)
Whether the control surface has been fully deployed.

float **getSurfaceArea**()
Surface area of the control surface in m^2.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The available torque, in Newton meters, that can be produced by this control surface, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.getReferenceFrame()*.

Decoupler

public class **Decoupler**
A decoupler. Obtained by calling *Part.getDecoupler()*

Part **getPart**()
The part object for this decoupler.

Vessel **decouple**()
Fires the decoupler. Returns the new vessel created when the decoupler fires. Throws an exception if the decoupler has already fired.

Note: When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *getActiveVessel()* no longer refer to the active vessel.

---

boolean **getDecoupled**()
Whether the decoupler has fired.

boolean **getStaged**()
Whether the decoupler is enabled in the staging sequence.

float **getImpulse**()
The impulse that the decoupler imparts when it is fired, in Newton seconds.

Docking Port

public class **DockingPort**
A docking port. Obtained by calling *Part.getDockingPort()*

Part **getPart**()
The part object for this docking port.

DockingPortState **getState**()
The current state of the docking port.

Part **getDockedPart**()
The part that this docking port is docked to. Returns `null` if this docking port is not docked to anything.

Vessel **undock**()
Undocks the docking port and returns the new *Vessel* that is created. This method can be called for either docking port in a docked pair. Throws an exception if the docking port is not docked to anything.

Note: When called, the active vessel may change. It is therefore possible that, after calling this function, the object(s) returned by previous call(s) to *getActiveVessel()* no longer refer to the active vessel.

---

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

boolean **getHasShield**()
Whether the docking port has a shield.

boolean **getShielded**()

void **setShielded**(boolean *value*)
The state of the docking ports shield, if it has one.

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

org.javatuples.Triplet<Double, Double, Double> **position**(*ReferenceFrame referenceFrame*)
The position of the docking port, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

org.javatuples.Triplet<Double, Double, Double> **direction**(*ReferenceFrame referenceFrame*)
The direction that docking port points in, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

org.javatuples.Quartet<Double, Double, Double, Double> **rotation**(*ReferenceFrame referenceFrame*)
The rotation of the docking port, in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned rotation is in.

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

ReferenceFrame **getReferenceFrame**()
The reference frame that is fixed relative to this docking port, and oriented with the port.

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

Note: This reference frame is not necessarily equivalent to the reference frame for the part, returned by *`Part.getReferenceFrame()`*.

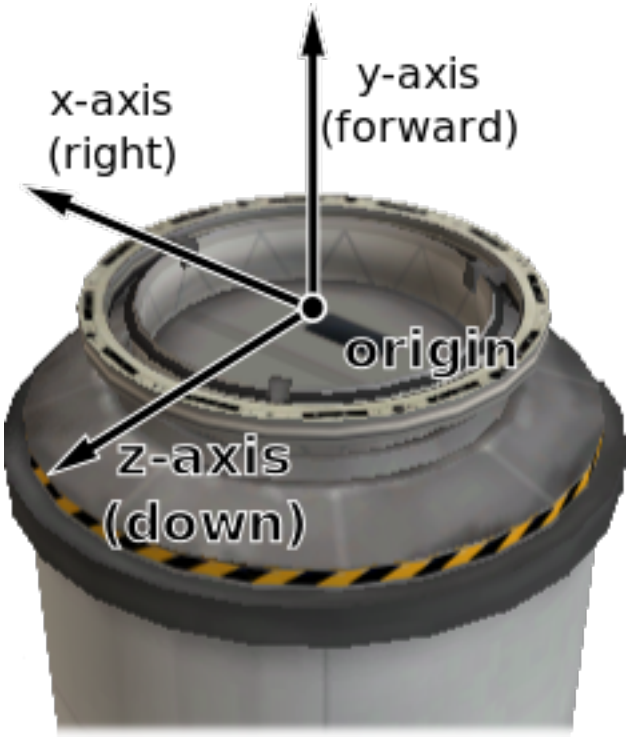


Fig. 6.8: Docking port reference frame origin and axes

public enum **`DockingPortState`**

The state of a docking port. See *`DockingPort.getState()`*.

public *DockingPortState* **`READY`**

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

public *DockingPortState* **`DOCKED`**

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

public *DockingPortState* **`DOCKING`**

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

public *DockingPortState* **`UNDOCKING`**

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

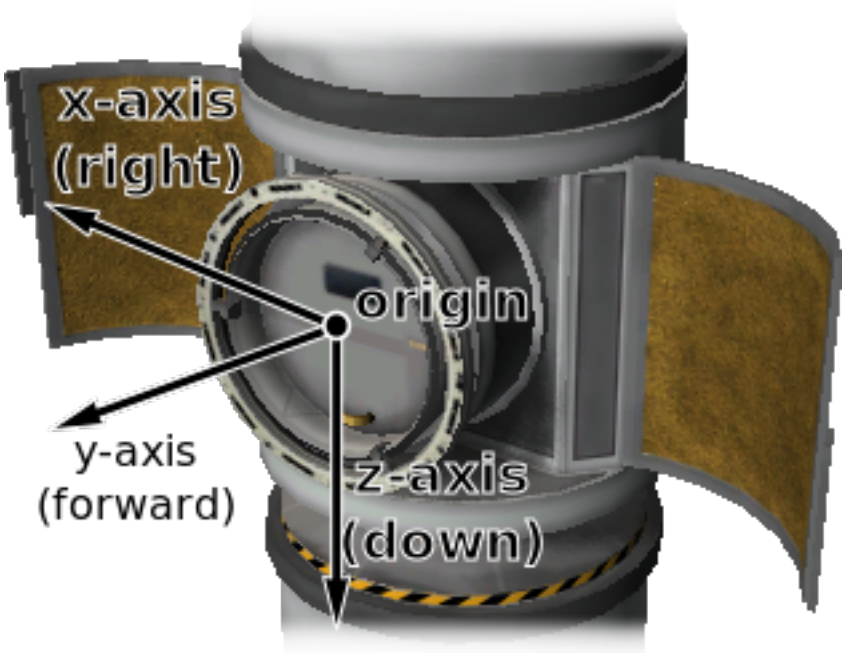


Fig. 6.9: Inline docking port reference frame origin and axes

public *DockingPortState* **SHIELDED**
The docking port has a shield, and the shield is closed.

public *DockingPortState* **MOVING**
The docking ports shield is currently opening/closing.

Engine

public class **Engine**
An engine, including ones of various types. For example liquid fuelled gimballed engines, solid rocket boosters and jet engines. Obtained by calling `Part.getEngine()`.

Note: For RCS thrusters `Part.getRCS()`.

Part **getPart**()
The part object for this engine.

boolean **getActive**()

void **setActive**(boolean *value*)
Whether the engine is active. Setting this attribute may have no effect, depending on `Engine.getCanShutdown()` and `Engine.getCanRestart()`.

float **getThrust**()
The current amount of thrust being produced by the engine, in Newtons.

float **getAvailableThrust**()
The amount of thrust, in Newtons, that would be produced by the engine when activated and with its throttle set to 100%. Returns zero if the engine does not have any fuel. Takes the engine's current `Engine.getThrustLimit()` and atmospheric conditions into account.

float **getMaxThrust**()
The amount of thrust, in Newtons, that would be produced by the engine when activated and fueled, with its throttle and throttle limiter set to 100%.

float **getMaxVacuumThrust**()
The maximum amount of thrust that can be produced by the engine in a vacuum, in Newtons. This is the amount of thrust produced by the engine when activated, *Engine.getThrustLimit()* is set to 100%, the main vessel's throttle is set to 100% and the engine is in a vacuum.

float **getThrustLimit**()

void **setThrustLimit**(float *value*)
The thrust limiter of the engine. A value between 0 and 1. Setting this attribute may have no effect, for example the thrust limit for a solid rocket booster cannot be changed in flight.

java.util.List<*Thruster*> **getThrusters**()
The components of the engine that generate thrust.

Note: For example, this corresponds to the rocket nozzel on a solid rocket booster, or the individual nozzels on a RAPIER engine. The overall thrust produced by the engine, as reported by *Engine.getAvailableThrust()*, *Engine.getMaxThrust()* and others, is the sum of the thrust generated by each thruster.

---

float **getSpecificImpulse**()
The current specific impulse of the engine, in seconds. Returns zero if the engine is not active.

float **getVacuumSpecificImpulse**()
The vacuum specific impulse of the engine, in seconds.

float **getKerbinSeaLevelSpecificImpulse**()
The specific impulse of the engine at sea level on Kerbin, in seconds.

java.util.List<String> **getPropellantNames**()
The names of the propellants that the engine consumes.

java.util.Map<String, Float> **getPropellantRatios**()
The ratio of resources that the engine consumes. A dictionary mapping resource names to the ratio at which they are consumed by the engine.

Note: For example, if the ratios are 0.6 for LiquidFuel and 0.4 for Oxidizer, then for every 0.6 units of LiquidFuel that the engine burns, it will burn 0.4 units of Oxidizer.

---

java.util.List<*Propellant*> **getPropellants**()
The propellants that the engine consumes.

boolean **getHasFuel**()
Whether the engine has any fuel available.

Note: The engine must be activated for this property to update correctly.

---

float **getThrottle**()
The current throttle setting for the engine. A value between 0 and 1. This is not necessarily the same as the vessel's main throttle setting, as some engines take time to adjust their throttle (such as jet engines).

boolean **getThrottleLocked**()
Whether the *Control.getThrottle()* affects the engine. For example, this is `true` for liquid fueled rockets, and `false` for solid rocket boosters.

boolean **getCanRestart**()
Whether the engine can be restarted once shutdown. If the engine cannot be shutdown, returns `false`. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

boolean **getCanShutdown**()
Whether the engine can be shutdown once activated. For example, this is `true` for liquid fueled rockets and `false` for solid rocket boosters.

boolean **getHasModes**()
Whether the engine has multiple modes of operation.

String **getMode**()

void **setMode**(String *value*)
The name of the current engine mode.

java.util.Map<String, *Engine*> **getModes**()
The available modes for the engine. A dictionary mapping mode names to *Engine* objects.

void **toggleMode**()
Toggle the current engine mode.

boolean **getAutoModeSwitch**()

void **setAutoModeSwitch**(boolean *value*)
Whether the engine will automatically switch modes.

boolean **getGimballed**()
Whether the engine is gimballed.

float **getGimbalRange**()
The range over which the gimbal can move, in degrees. Returns 0 if the engine is not gimballed.

boolean **getGimbalLocked**()

void **setGimbalLocked**(boolean *value*)
Whether the engines gimbal is locked in place. Setting this attribute has no effect if the engine is not gimballed.

float **getGimbalLimit**()

void **setGimbalLimit**(float *value*)
The gimbal limiter of the engine. A value between 0 and 1. Returns 0 if the gimbal is locked.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The available torque, in Newton meters, that can be produced by this engine, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.getReferenceFrame()*. Returns zero if the engine is inactive, or not gimballed.

public class **Propellant**
A propellant for an engine. Obtains by calling *Engine.getPropellants()*.

String **getName**()
The name of the propellant.

double **getCurrentAmount**()
The current amount of propellant.

double **getCurrentRequirement**()
The required amount of propellant.

double **getTotalResourceAvailable**()
The total amount of the underlying resource currently reachable given resource flow rules.

double **getTotalResourceCapacity**()
The total vehicle capacity for the underlying propellant resource, restricted by resource flow rules.

boolean **getIgnoreForIsp**()
If this propellant should be ignored when calculating required mass flow given specific impulse.

boolean **getIgnoreForThrustCurve**()
If this propellant should be ignored for thrust curve calculations.

boolean **getDrawStackGauge**()
If this propellant has a stack gauge or not.

boolean **getIsDeprived**()
If this propellant is deprived.

float **getRatio**()
The propellant ratio.

Experiment

public class **Experiment**
Obtained by calling *`Part.getExperiment()`*.

Part **getPart**()
The part object for this experiment.

void **run**()
Run the experiment.

void **transmit**()
Transmit all experimental data contained by this part.

void **dump**()
Dump the experimental data contained by the experiment.

void **reset**()
Reset the experiment.

boolean **getDeployed**()
Whether the experiment has been deployed.

boolean **getRerunnable**()
Whether the experiment can be re-run.

boolean **getInoperable**()
Whether the experiment is inoperable.

boolean **getHasData**()
Whether the experiment contains data.

java.util.List<*ScienceData*> **getData**()
The data contained in this experiment.

String **getBiome**()
The name of the biome the experiment is currently in.

boolean **getAvailable**()
Determines if the experiment is available given the current conditions.

ScienceSubject **getScienceSubject**()
Containing information on the corresponding specific science result for the current conditions. Returns `null` if the experiment is unavailable.

public class **ScienceData**
Obtained by calling *Experiment.getData()*.

float **getDataAmount**()
Data amount.

float **getScienceValue**()
Science value.

float **getTransmitValue**()
Transmit value.

public class **ScienceSubject**
Obtained by calling *Experiment.getScienceSubject()*.

String **getTitle**()
Title of science subject, displayed in science archives

boolean **getIsComplete**()
Whether the experiment has been completed.

float **getScience**()
Amount of science already earned from this subject, not updated until after transmission/recovery.

float **getScienceCap**()
Total science allowable for this subject.

float **getDataScale**()
Multiply science value by this to determine data amount in mits.

float **getSubjectValue**()
Multiplier for specific Celestial Body/Experiment Situation combination.

float **getScientificValue**()
Diminishing value multiplier for decreasing the science value returned from repeated experiments.

Fairing

public class **Fairing**
A fairing. Obtained by calling *Part.getFairing()*.

Part **getPart**()
The part object for this fairing.

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

boolean **getJettisoned**()
Whether the fairing has been jettisoned.

Intake

public class **Intake**
An air intake. Obtained by calling *Part.getIntake()*.

Part **getPart**()
The part object for this intake.

boolean **getOpen**()

void **setOpen**(boolean *value*)
Whether the intake is open.

float **getSpeed**()
Speed of the flow into the intake, in m/s.

float **getFlow**()
The rate of flow into the intake, in units of resource per second.

float **getArea**()
The area of the intake's opening, in square meters.

Leg

public class **Leg**
A landing leg. Obtained by calling *Part.getLeg()*.

Part **getPart**()
The part object for this landing leg.

LegState **getState**()
The current state of the landing leg.

boolean **getDeployable**()
Whether the leg is deployable.

boolean **getDeployed**()

void **setDeployed**(boolean *value*)
Whether the landing leg is deployed.

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

boolean **getIsGrounded**()
Returns whether the leg is touching the ground.

public enum **LegState**
The state of a landing leg. See *Leg.getState()*.

public *LegState* **DEPLOYED**
Landing leg is fully deployed.

public *LegState* **RETRACTED**
Landing leg is fully retracted.

public *LegState* **DEPLOYING**
Landing leg is being deployed.

public *LegState* **RETRACTING**
Landing leg is being retracted.

public *LegState* **BROKEN**
Landing leg is broken.

Launch Clamp

public class **LaunchClamp**
A launch clamp. Obtained by calling *Part.getLaunchClamp()*.

Part **getPart**()
The part object for this launch clamp.

void **release**()
Releases the docking clamp. Has no effect if the clamp has already been released.

Light

public class **Light**
A light. Obtained by calling *Part.getLight()*.

Part **getPart**()
The part object for this light.

boolean **getActive**()

void **setActive**(boolean *value*)
Whether the light is switched on.

org.javatuples.Triplet<Float, Float, Float> **getColor**()

void **setColor**(org.javatuples.Triplet<Float, Float, Float> *value*)
The color of the light, as an RGB triple.

float **getPowerUsage**()
The current power usage, in units of charge per second.

Parachute

public class **Parachute**
A parachute. Obtained by calling *Part.getParachute()*.

Part **getPart**()
The part object for this parachute.

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

boolean **getDeployed**()
Whether the parachute has been deployed.

void **arm**()
Deploys the parachute. This has no effect if the parachute has already been armed or deployed. Only applicable to RealChutes parachutes.

boolean **getArmed**()
Whether the parachute has been armed or deployed. Only applicable to RealChutes parachutes.

ParachuteState **getState**()
The current state of the parachute.

float **getDeployAltitude**()

void **setDeployAltitude**(float *value*)
The altitude at which the parachute will full deploy, in meters. Only applicable to stock parachutes.

float **getDeployMinPressure**()

void **setDeployMinPressure**(float *value*)
The minimum pressure at which the parachute will semi-deploy, in atmospheres. Only applicable to stock parachutes.

public enum **ParachuteState**
The state of a parachute. See *Parachute.getState()*.

public *ParachuteState* **STOWED**
The parachute is safely tucked away inside its housing.

public *ParachuteState* **ARMED**
The parachute is armed for deployment. (RealChutes only)

public *ParachuteState* **ACTIVE**
The parachute is still stowed, but ready to semi-deploy. (Stock parachutes only)

public *ParachuteState* **SEMI_DEPLOYED**
The parachute has been deployed and is providing some drag, but is not fully deployed yet. (Stock parachutes only)

public *ParachuteState* **DEPLOYED**
The parachute is fully deployed.

public *ParachuteState* **CUT**
The parachute has been cut.

Radiator

public class **Radiator**
A radiator. Obtained by calling *Part.getRadiator()*.

Part **getPart**()
The part object for this radiator.

boolean **getDeployable**()
Whether the radiator is deployable.

boolean **getDeployed**()

void **setDeployed**(boolean *value*)
For a deployable radiator, `true` if the radiator is extended. If the radiator is not deployable, this is always `true`.

RadiatorState **getState**()
The current state of the radiator.

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

---

public enum **RadiatorState**
The state of a radiator. *RadiatorState*

public *RadiatorState* **EXTENDED**
Radiator is fully extended.

public *RadiatorState* **RETRACTED**
Radiator is fully retracted.

public *RadiatorState* **EXTENDING**
Radiator is being extended.

public *RadiatorState* **RETRACTING**
Radiator is being retracted.

public *RadiatorState* **BROKEN**
Radiator is being broken.

Resource Converter

public class **ResourceConverter**
A resource converter. Obtained by calling *Part.getResourceConverter()*.

Part **getPart**()
The part object for this converter.

int **getCount**()
The number of converters in the part.

String **name**(int *index*)
The name of the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

boolean **active**(int *index*)
True if the specified converter is active.

Parameters

- **index** (*int*) – Index of the converter.

void **start**(int *index*)
Start the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

void **stop**(int *index*)
Stop the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

ResourceConverterState **state**(int *index*)
The state of the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

String **statusInfo**(int *index*)
Status information for the specified converter. This is the full status message shown in the in-game UI.

Parameters

- **index** (*int*) – Index of the converter.

java.util.List<String> **inputs**(int *index*)
List of the names of resources consumed by the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

java.util.List<String> **outputs** (int *index*)
List of the names of resources produced by the specified converter.

Parameters

- **index** (*int*) – Index of the converter.

float **getOptimumCoreTemperature** ()
The core temperature at which the converter will operate with peak efficiency, in Kelvin.

float **getCoreTemperature** ()
The core temperature of the converter, in Kelvin.

float **getThermalEfficiency** ()
The thermal efficiency of the converter, as a percentage of its maximum.

public enum **ResourceConverterState**
The state of a resource converter. See *ResourceConverter.state(int)*.

public *ResourceConverterState* **RUNNING**
Converter is running.

public *ResourceConverterState* **IDLE**
Converter is idle.

public *ResourceConverterState* **MISSING_RESOURCE**
Converter is missing a required resource.

public *ResourceConverterState* **STORAGE_FULL**
No available storage for output resource.

public *ResourceConverterState* **CAPACITY**
At preset resource capacity.

public *ResourceConverterState* **UNKNOWN**
Unknown state. Possible with modified resource converters. In this case, check *ResourceConverter.statusInfo(int)* for more information.

Resource Harvester

public class **ResourceHarvester**
A resource harvester (drill). Obtained by calling *Part.getResourceHarvester()*.

Part **getPart** ()
The part object for this harvester.

ResourceHarvesterState **getState** ()
The state of the harvester.

boolean **getDeployed** ()

void **setDeployed** (boolean *value*)
Whether the harvester is deployed.

boolean **getActive** ()

void **setActive** (boolean *value*)
Whether the harvester is actively drilling.

float **getExtractionRate**()
The rate at which the drill is extracting ore, in units per second.

float **getThermalEfficiency**()
The thermal efficiency of the drill, as a percentage of its maximum.

float **getCoreTemperature**()
The core temperature of the drill, in Kelvin.

float **getOptimumCoreTemperature**()
The core temperature at which the drill will operate with peak efficiency, in Kelvin.

public enum **ResourceHarvesterState**
The state of a resource harvester. See *ResourceHarvester.getState()*.

public *ResourceHarvesterState* **DEPLOYING**
The drill is deploying.

public *ResourceHarvesterState* **DEPLOYED**
The drill is deployed and ready.

public *ResourceHarvesterState* **RETRACTING**
The drill is retracting.

public *ResourceHarvesterState* **RETRACTED**
The drill is retracted.

public *ResourceHarvesterState* **ACTIVE**
The drill is running.

Reaction Wheel

public class **ReactionWheel**
A reaction wheel. Obtained by calling *Part.getReactionWheel()*.

Part **getPart**()
The part object for this reaction wheel.

boolean **getActive**()

void **setActive**(boolean *value*)
Whether the reaction wheel is active.

boolean **getBroken**()
Whether the reaction wheel is broken.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The available torque, in Newton meters, that can be produced by this reaction wheel, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.getReferenceFrame()*. Returns zero if the reaction wheel is inactive or broken.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getMax**
The maximum torque, in Newton meters, that can be produced by this reaction wheel, when it is active, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the *Vessel.getReferenceFrame()*.

RCS

public class **RCS**
An RCS block or thruster. Obtained by calling *Part.getRCS()*.

Part **getPart**()
The part object for this RCS.

boolean **getActive**()
Whether the RCS thrusters are active. An RCS thruster is inactive if the RCS action group is disabled (`Control.getRCS()`), the RCS thruster itself is not enabled (`RCS.getEnabled()`) or it is covered by a fairing (`Part.getShielded()`).

boolean **getEnabled**()

void **setEnabled**(boolean *value*)
Whether the RCS thrusters are enabled.

boolean **getPitchEnabled**()

void **setPitchEnabled**(boolean *value*)
Whether the RCS thruster will fire when pitch control input is given.

boolean **getYawEnabled**()

void **setYawEnabled**(boolean *value*)
Whether the RCS thruster will fire when yaw control input is given.

boolean **getRollEnabled**()

void **setRollEnabled**(boolean *value*)
Whether the RCS thruster will fire when roll control input is given.

boolean **getForwardEnabled**()

void **setForwardEnabled**(boolean *value*)
Whether the RCS thruster will fire when pitch control input is given.

boolean **getUpEnabled**()

void **setUpEnabled**(boolean *value*)
Whether the RCS thruster will fire when yaw control input is given.

boolean **getRightEnabled**()

void **setRightEnabled**(boolean *value*)
Whether the RCS thruster will fire when roll control input is given.

org.javatuples.Pair<org.javatuples.Triplet<Double, Double, Double>, org.javatuples.Triplet<Double, Double, Double>> **getAva**
The available torque, in Newton meters, that can be produced by this RCS, in the positive and negative pitch, roll and yaw axes of the vessel. These axes correspond to the coordinate axes of the `Vessel.getReferenceFrame()`. Returns zero if RCS is disable.

float **getMaxThrust**()
The maximum amount of thrust that can be produced by the RCS thrusters when active, in Newtons.

float **getMaxVacuumThrust**()
The maximum amount of thrust that can be produced by the RCS thrusters when active in a vacuum, in Newtons.

java.util.List<*Thruster*> **getThrusters**()
A list of thrusters, one of each nozzel in the RCS part.

float **getSpecificImpulse**()
The current specific impulse of the RCS, in seconds. Returns zero if the RCS is not active.

float **getVacuumSpecificImpulse**()
The vacuum specific impulse of the RCS, in seconds.

float **getKerbinSeaLevelSpecificImpulse**()
The specific impulse of the RCS at sea level on Kerbin, in seconds.

java.util.List<String> **getPropellants**()
The names of resources that the RCS consumes.

java.util.Map<String, Float> **getPropellantRatios**()
The ratios of resources that the RCS consumes. A dictionary mapping resource names to the ratios at which they are consumed by the RCS.

boolean **getHasFuel**()
Whether the RCS has fuel available.

Note: The RCS thruster must be activated for this property to update correctly.

Sensor

public class **Sensor**
A sensor, such as a thermometer. Obtained by calling *`Part.getSensor()`*.

Part **getPart**()
The part object for this sensor.

boolean **getActive**()

void **setActive**(boolean *value*)
Whether the sensor is active.

String **getValue**()
The current value of the sensor.

Solar Panel

public class **SolarPanel**
A solar panel. Obtained by calling *`Part.getSolarPanel()`*.

Part **getPart**()
The part object for this solar panel.

boolean **getDeployable**()
Whether the solar panel is deployable.

boolean **getDeployed**()

void **setDeployed**(boolean *value*)
Whether the solar panel is extended.

SolarPanelState **getState**()
The current state of the solar panel.

float **getEnergyFlow**()
The current amount of energy being generated by the solar panel, in units of charge per second.

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

public enum **SolarPanelState**
The state of a solar panel. See *SolarPanel.getState()*.

public *SolarPanelState* **EXTENDED**
Solar panel is fully extended.

public *SolarPanelState* **RETRACTED**
Solar panel is fully retracted.

public *SolarPanelState* **EXTENDING**
Solar panel is being extended.

public *SolarPanelState* **RETRACTING**
Solar panel is being retracted.

public *SolarPanelState* **BROKEN**
Solar panel is broken.

Thruster

public class **Thruster**
The component of an *Engine* or *RCS* part that generates thrust. Can obtained by calling *Engine.getThrusters()* or *RCS.getThrusters()*.

Note: Engines can consist of multiple thrusters. For example, the S3 KS-25x4 "Mammoth" has four rocket nozzels, and so consists of four thrusters.

Part **getPart**()
The *Part* that contains this thruster.

org.javatuples.Triplet<Double, Double, Double> **thrustPosition**(*ReferenceFrame referenceFrame*)
The position at which the thruster generates thrust, in the given reference frame. For gimballed engines, this takes into account the current rotation of the gimbal.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

org.javatuples.Triplet<Double, Double, Double> **thrustDirection**(*ReferenceFrame referenceFrame*)
The direction of the force generated by the thruster, in the given reference frame. This is opposite to the direction in which the thruster expels propellant. For gimballed engines, this takes into account the current rotation of the gimbal.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

ReferenceFrame **getThrustReferenceFrame**()
A reference frame that is fixed relative to the thruster and orientated with its thrust direction (*Thruster.thrustDirection(ReferenceFrame)*). For gimballed engines, this takes into account the current rotation of the gimbal.

- The origin is at the position of thrust for this thruster (*Thruster.thrustPosition(ReferenceFrame)*).
- The axes rotate with the thrust direction. This is the direction in which the thruster expels propellant, including any gimballing.
- The y-axis points along the thrust direction.
- The x-axis and z-axis are perpendicular to the thrust direction.

boolean **getGimballed**()
Whether the thruster is gimballed.

org.javatuples.Triplet<Double, Double, Double> **gimbalPosition**(*ReferenceFrame referenceFrame*)
Position around which the gimbal pivots.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

org.javatuples.Triplet<Double, Double, Double> **getGimbalAngle**()
The current gimbal angle in the pitch, roll and yaw axes, in degrees.

org.javatuples.Triplet<Double, Double, Double> **initialThrustPosition**(*ReferenceFrame referenceFrame*)
The position at which the thruster generates thrust, when the engine is in its initial position (no gimballing), in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

Note: This position can move when the gimbal rotates. This is because the thrust position and gimbal position are not necessarily the same.

---

org.javatuples.Triplet<Double, Double, Double> **initialThrustDirection**(*ReferenceFrame referenceFrame*)
The direction of the force generated by the thruster, when the engine is in its initial position (no gimballing), in the given reference frame. This is opposite to the direction in which the thruster expels propellant.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

Wheel

public class **Wheel**
A wheel. Includes landing gear and rover wheels. Obtained by calling *Part.getWheel()*. Can be used to control the motors, steering and deployment of wheels, among other things.

Part **getPart**()
The part object for this wheel.

WheelState **getState**()
The current state of the wheel.

float **getRadius**()
Radius of the wheel, in meters.

boolean **getGrounded**()
Whether the wheel is touching the ground.

boolean **getHasBrakes**()
Whether the wheel has brakes.

float **getBrakes**()

void **setBrakes**(float *value*)
The braking force, as a percentage of maximum, when the brakes are applied.

boolean **getAutoFrictionControl**()

void **setAutoFrictionControl**(boolean *value*)
Whether automatic friction control is enabled.

float **getManualFrictionControl**()

void **setManualFrictionControl**(float *value*)
Manual friction control value. Only has an effect if automatic friction control is disabled. A value between 0 and 5 inclusive.

boolean **getDeployable**()
Whether the wheel is deployable.

boolean **getDeployed**()

void **setDeployed**(boolean *value*)
Whether the wheel is deployed.

boolean **getPowered**()
Whether the wheel is powered by a motor.

boolean **getMotorEnabled**()

void **setMotorEnabled**(boolean *value*)
Whether the motor is enabled.

boolean **getMotorInverted**()

void **setMotorInverted**(boolean *value*)
Whether the direction of the motor is inverted.

MotorState **getMotorState**()
Whether the direction of the motor is inverted.

float **getMotorOutput**()
The output of the motor. This is the torque currently being generated, in Newton meters.

boolean **getTractionControlEnabled**()

void **setTractionControlEnabled**(boolean *value*)
Whether automatic traction control is enabled. A wheel only has traction control if it is powered.

float **getTractionControl**()

void **setTractionControl**(float *value*)
Setting for the traction control. Only takes effect if the wheel has automatic traction control enabled. A value between 0 and 5 inclusive.

float **getDriveLimiter**()

void **setDriveLimiter**(float *value*)
Manual setting for the motor limiter. Only takes effect if the wheel has automatic traction control disabled. A value between 0 and 100 inclusive.

boolean **getSteerable**()
Whether the wheel has steering.

boolean **getSteeringEnabled**()

void **setSteeringEnabled**(boolean *value*)
Whether the wheel steering is enabled.

boolean **getSteeringInverted**()

void **setSteeringInverted**(boolean *value*)
Whether the wheel steering is inverted.

boolean **getHasSuspension**()
Whether the wheel has suspension.

float **getSuspensionSpringStrength**()
Suspension spring strength, as set in the editor.

float **getSuspensionDamperStrength**()
Suspension damper strength, as set in the editor.

boolean **getBroken**()
Whether the wheel is broken.

boolean **getRepairable**()
Whether the wheel is repairable.

float **getStress**()
Current stress on the wheel.

float **getStressTolerance**()
Stress tolerance of the wheel.

float **getStressPercentage**()
Current stress on the wheel as a percentage of its stress tolerance.

float **getDeflection**()
Current deflection of the wheel.

float **getSlip**()
Current slip of the wheel.

public enum **WheelState**
The state of a wheel. See *Wheel.getState()*.

public *WheelState* **DEPLOYED**
Wheel is fully deployed.

public *WheelState* **RETRACTED**
Wheel is fully retracted.

public *WheelState* **DEPLOYING**
Wheel is being deployed.

public *WheelState* **RETRACTING**
Wheel is being retracted.

public *WheelState* **BROKEN**
Wheel is broken.

public enum **MotorState**
The state of the motor on a powered wheel. See *Wheel.getMotorState()*.

public *MotorState* **IDLE**
The motor is idle.

public *MotorState* **RUNNING**
The motor is running.

public *MotorState* **DISABLED**
The motor is disabled.

public *MotorState* **INOPERABLE**
The motor is inoperable.

public *MotorState* **NOT_ENOUGH_RESOURCES**
The motor does not have enough resources to run.

Trees of Parts

Vessels in KSP are comprised of a number of parts, connected to one another in a *tree* structure. An example vessel is shown in Figure 1, and the corresponding tree of parts in Figure 2. The craft file for this example can also be downloaded here.

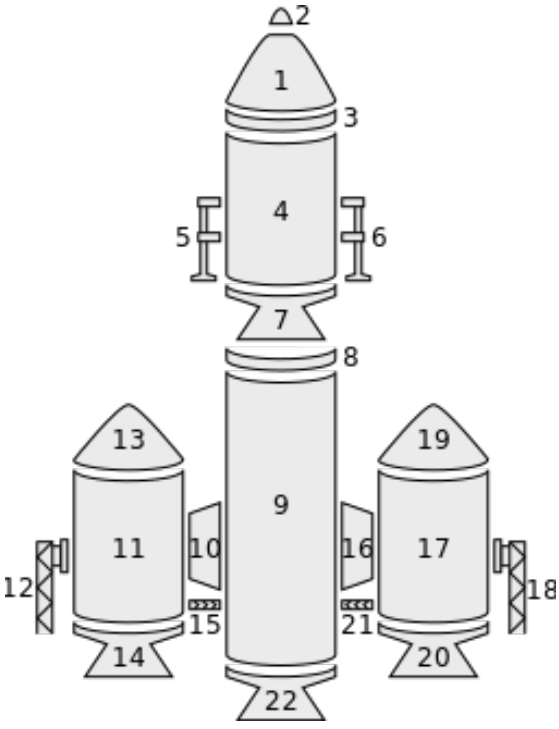


Fig. 6.10: **Figure 1** – Example parts making up a vessel.

Traversing the Tree

The tree of parts can be traversed using the attributes *Parts.getRoot()*, *Part.getParent()* and *Part.getChildren()*.

The root of the tree is the same as the vessels *root part* (part number 1 in the example above) and can be obtained by calling *Parts.getRoot()*. A parts children can be obtained by calling *Part.getChildren()*. If the part does not have any children, *Part.getChildren()* returns an empty list. A parts parent can be obtained by calling *Part.getParent()*. If the part does not have a parent (as is the case for the root part), *Part.getParent()* returns `null`.

The following Java example uses these attributes to perform a depth-first traversal over all of the parts in a vessel:

```
import krpc.client.C
import krpc.client.R
import krpc.client.s
import krpc.client.s
import
↪krpc.client.servic

import org.javatuple
```

```
import java.io.IOExc
import java.util.Arr
import java.util.Deq

public class TreeTra
    public static vo
↪args) throws IOExc
        Connection␣
↪connection = Conne
        Vessel vesse
↪newInstance(connec
    ␣
↪    Part root = ve
        Deque<Pair<P
↪= new ArrayDeque<P
        stack.
↪push(new Pair<Part
        while (stack
    ␣
↪    Pair<Part, In
            Part par
            int dept
            String p
            for (int
                pref
            }
            System.
↪out.println(prefix
    ␣
↪    for (Part chi
                stac
↪Pair<Part, Integer
            }
        }
        connection.c
    }
}
```

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

```
Command Pod Mk1
 TR-18A Stack Decoup
  FL-T400 Fuel Tank
   LV-909 Liquid Fue
    TR-18A Stack Dec
     FL-T800 Fuel Ta
      LV-909 Liquid
      TT-70 Radial D
       FL-T400 Fuel
        TT18-A Launc
        FTX-2 Extern
        LV-909 Liqui
        Aerodynamic
      TT-70 Radial D
       FL-T400 Fuel
        TT18-A Launc
        FTX-2 Extern
```

```
                                                    LV-909 Liqui
                                                    Aerodynamic
                                                LT-1 Landing Stru
                                                LT-1 Landing Stru
                                               Mk16 Parachute
```

Attachment Modes

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

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

The root part of a vessel (for example the command pod – part 1) does not have a parent part, so does not have an attachment mode. However, the part is consider to be *axially* attached to nothing.

The following Java example does a depth-first traversal as before, but also prints out the attachment mode used by the part:

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Part;
import␣
↪krpc.client.services.SpaceCenter.Vessel;

import org.javatuples.Pair;

import java.io.IOException;
import java.util.ArrayDeque;
import java.util.Deque;

public class AttachmentModes {
    public static void main(String[]␣
↪args) throws IOException, RPCException {
        Connection␣
↪connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.
↪newInstance(connection).getActiveVessel();
    ␣
↪     Part root = vessel.getParts().getRoot();
        Deque<Pair<Part, Integer>> stack␣
↪= new ArrayDeque<Pair<Part, Integer>>();
        stack.
↪push(new Pair<Part, Integer>(root, 0));
        while (stack.size() > 0) {
    ␣
↪      Pair<Part, Integer> item = stack.pop();
            Part part = item.getValue0();
            int depth = item.getValue1();
```

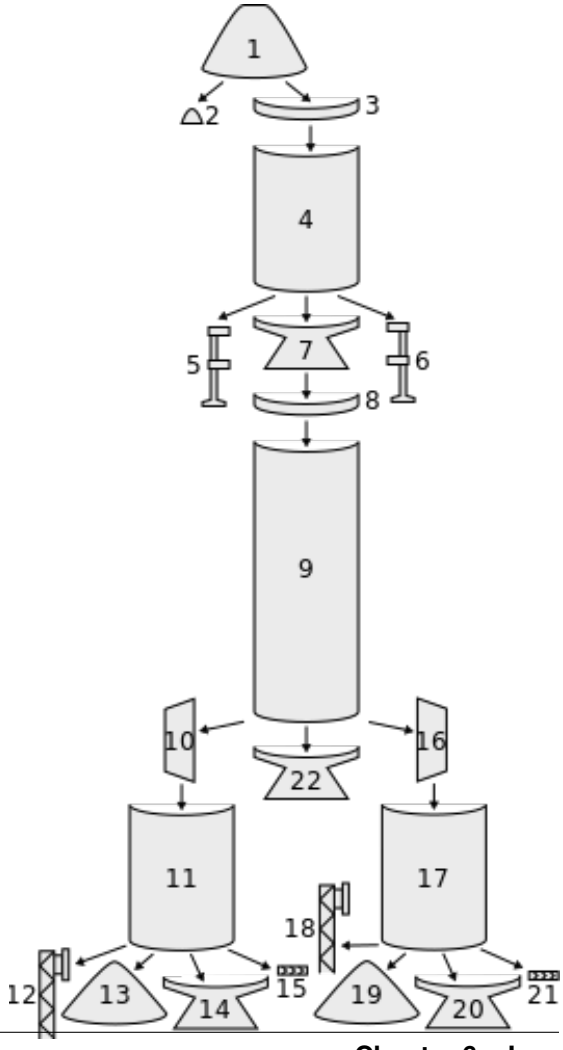


Fig. 6.11: **Figure 2** – Tree of parts for the vessel in Figure 1. Arrows point from the parent part to the child part.

```
            String prefix = "";
            for (int i = 0; i < depth; i++) {
                prefix += " ";
            }
            String attachMode = part.
↪getAxiallyAttached() ? "axial" : "radial";
            System.out.println(prefix␣
↪+ part.getTitle() + " - " + attachMode);
␣
↪      for (Part child : part.getChildren()) {
                stack.push(new␣
↪Pair<Part, Integer>(child, depth + 1));
            }
        }
        connection.close();
    }
}
```

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

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

Fuel Lines

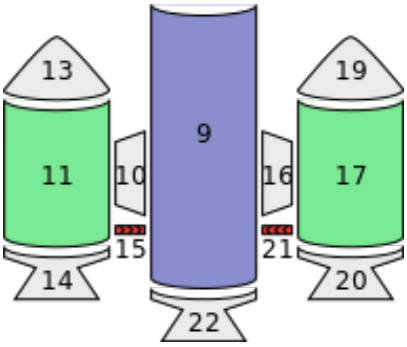


Fig. 6.12: **Figure 5** – Fuel lines from the example in Figure 1. Fuel flows from the parts highlighted in green, into the part highlighted in blue.

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

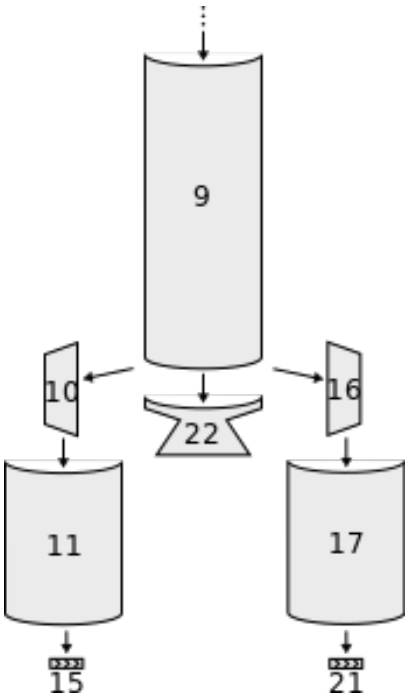


Fig. 6.13: **Figure 4** – A subset of the parts tree from Figure 2 above.

Staging

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

Figure 6 shows an example staging sequence for a vessel. Figure 7 shows the stages in which each part of the vessel will be *activated*. Figure 8 shows the stages in which each part of the vessel will be *decoupled*.

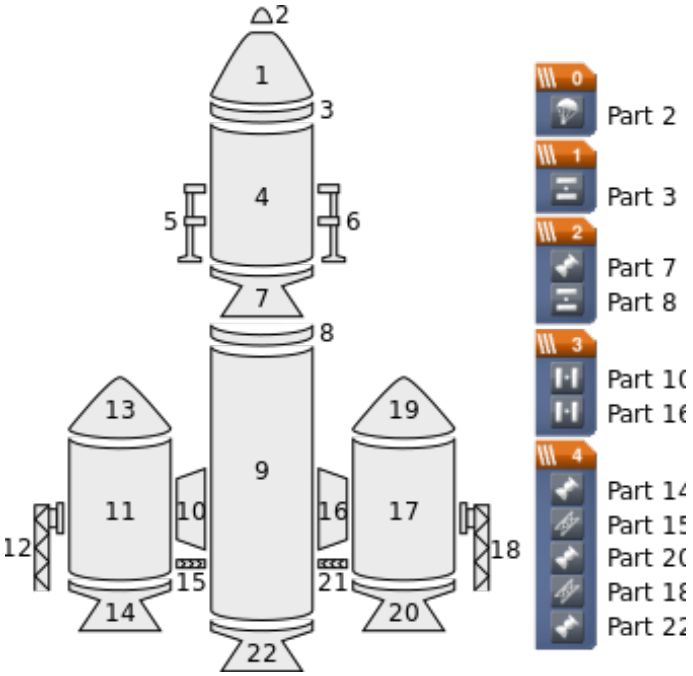


Fig. 6.14: **Figure 6** – Example vessel from Figure 1 with a staging sequence.

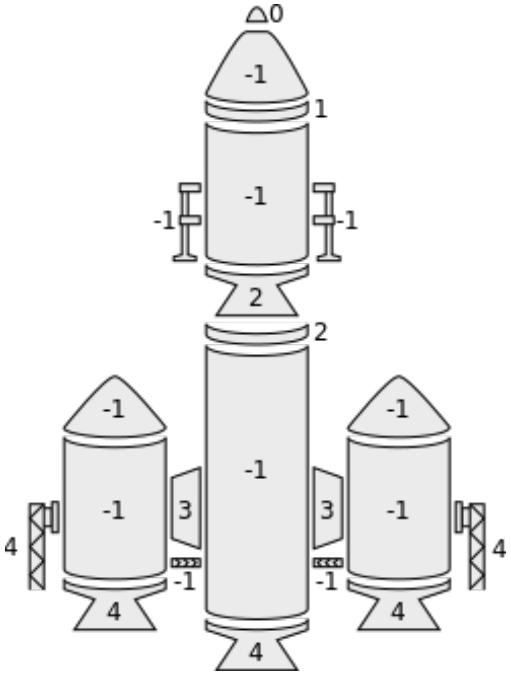
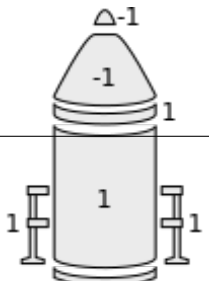


Fig. 6.15: **Figure 7** – The stage in which each part is *activated*.

6.3.9 Resources

public class **`Resources`**

Represents the collection of resources stored in a vessel, stage



or part. Created by calling *Vessel.getResources()*, *Vessel.resourcesInDecoupleStage(int, boolean)* or *Part.getResources()*.

java.util.List<*Resource*> **getAll**()
All the individual resources that can be stored.

java.util.List<*Resource*> **withResource**(String *name*)
All the individual resources with the given name that can be stored.

Parameters

- **name** (*String*) –

java.util.List<String> **getNames**()
A list of resource names that can be stored.

boolean **hasResource**(String *name*)
Check whether the named resource can be stored.

Parameters

- **name** (*String*) – The name of the resource.

float **amount**(String *name*)
Returns the amount of a resource that is currently stored.

Parameters

- **name** (*String*) – The name of the resource.

float **max**(String *name*)
Returns the amount of a resource that can be stored.

Parameters

- **name** (*String*) – The name of the resource.

static float **density**(*Connection connection*, String *name*)
Returns the density of a resource, in kg/l.

Parameters

- **name** (*String*) – The name of the resource.

static *ResourceFlowMode* **flowMode**(*Connection connection*, String *name*)
Returns the flow mode of a resource.

Parameters

- **name** (*String*) – The name of the resource.

boolean **getEnabled**()

void **setEnabled**(boolean *value*)
Whether use of all the resources are enabled.

Note: This is `true` if all of the resources are enabled. If any of the resources are not enabled, this is `false`.

---

public class **Resource**
An individual resource stored within a part. Created using methods in the *Resources* class.

String **getName**()
The name of the resource.

Part **getPart**()
The part containing the resource.

float **getAmount**()
The amount of the resource that is currently stored in the part.

float **getMax**()
The total amount of the resource that can be stored in the part.

float **getDensity**()
The density of the resource, in kg/l.

ResourceFlowMode **getFlowMode**()
The flow mode of the resource.

boolean **getEnabled**()

void **setEnabled**(boolean *value*)
Whether use of this resource is enabled.

public class **ResourceTransfer**
Transfer resources between parts.

static *ResourceTransfer* **start**(*Connection connection*, *Part fromPart*, *Part toPart*, String *resource*, float *maxAmount*)
Start transferring a resource transfer between a pair of parts. The transfer will move at most *maxAmount* units of the resource, depending on how much of the resource is available in the source part and how much storage is available in the destination part. Use *ResourceTransfer.getComplete()* to check if the transfer is complete. Use *ResourceTransfer.getAmount()* to see how much of the resource has been transferred.

Parameters

- **fromPart** (*Part*) – The part to transfer to.
- **toPart** (*Part*) – The part to transfer from.
- **resource** (*String*) – The name of the resource to transfer.
- **maxAmount** (*float*) – The maximum amount of resource to transfer.

float **getAmount**()
The amount of the resource that has been transferred.

boolean **getComplete**()
Whether the transfer has completed.

public enum **ResourceFlowMode**
The way in which a resource flows between parts. See *Resources.flowMode(String)*.

public *ResourceFlowMode* **VESSEL**
The resource flows to any part in the vessel. For example, electric charge.

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

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

public *ResourceFlowMode* **NONE**
The resource does not flow. For example, solid fuel.

6.3.10 Node

public class **Node**
Represents a maneuver node. Can be created using *Control.addNode(double, float, float, float)*.

double **getPrograde**()

void **setPrograde**(double *value*)
The magnitude of the maneuver nodes delta-v in the prograde direction, in meters per second.

double **getNormal**()

void **setNormal**(double *value*)
The magnitude of the maneuver nodes delta-v in the normal direction, in meters per second.

double **getRadial**()

void **setRadial**(double *value*)
The magnitude of the maneuver nodes delta-v in the radial direction, in meters per second.

double **getDeltaV**()

void **setDeltaV**(double *value*)
The delta-v of the maneuver node, in meters per second.

Note: Does not change when executing the maneuver node. See *Node.getRemainingDeltaV()*.

---

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

org.javatuples.Triplet<Double, Double, Double> **burnVector**(*ReferenceFrame referenceFrame*)
Returns the burn vector for the maneuver node.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned vector is in. Defaults to *Vessel.getOrbitalReferenceFrame()*.

Returns A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

Note: Does not change when executing the maneuver node. See *Node.remainingBurnVector(ReferenceFrame)*.

org.javatuples.Triplet<Double, Double, Double> **remainingBurnVector**(*ReferenceFrame referenceFrame*)

Returns the remaining burn vector for the maneuver node.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned vector is in. Defaults to *Vessel.getOrbitalReferenceFrame()*.

Returns A vector whose direction is the direction of the maneuver node burn, and magnitude is the delta-v of the burn in meters per second.

Note: Changes as the maneuver node is executed. See *Node.burnVector(ReferenceFrame)*.

double **getUT**()

void **setUT**(double *value*)
The universal time at which the maneuver will occur, in seconds.

double **getTimeTo**()
The time until the maneuver node will be encountered, in seconds.

Orbit **getOrbit**()
The orbit that results from executing the maneuver node.

void **remove**()
Removes the maneuver node.

ReferenceFrame **getReferenceFrame**()
The reference frame that is fixed relative to the maneuver node's burn.

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

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

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

org.javatuples.Triplet<Double, Double, Double> **position**(*ReferenceFrame referenceFrame*)
The position vector of the maneuver node in the given reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned position vector is in.

Returns The position as a vector.

org.javatuples.Triplet<Double, Double, Double> **direction**(*ReferenceFrame referenceFrame*)
The direction of the maneuver nodes burn.

Parameters

- **referenceFrame** (ReferenceFrame) – The reference frame that the returned direction is in.

Returns The direction as a unit vector.

6.3.11 ReferenceFrame

public class **ReferenceFrame**
Represents a reference frame for positions, rotations and velocities.
Contains:

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

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

---

static *ReferenceFrame* **createRelative**(*Connection connection*, *ReferenceFrame referenceFrame*, org.javatuples.Triplet<Double, Double, Double> *position*, org.javatuples.Quartet<Double, Double, Double, Double> *rotation*, org.javatuples.Triplet<Double, Double, Double> *velocity*, org.javatuples.Triplet<Double, Double, Double> *angularVelocity*)
Create a relative reference frame. This is a custom reference frame whose components offset the components of a parent reference frame.

Parameters

- **referenceFrame** (ReferenceFrame) – The parent reference frame on which to base this reference frame.
- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – The offset of the position of the origin, as a position vector. Defaults to $(0, 0, 0)$
- **rotation** (*org.javatuples.Quartet<Double,Double,Double,Double>*) – The rotation to apply to the parent frames rotation, as a quaternion of the form (x, y, z, w). Defaults to $(0, 0, 0, 1)$ (i.e. no rotation)

- **velocity** (*org.javatuples.Triplet<Double,Double,Double>*) – The linear velocity to offset the parent frame by, as a vector pointing in the direction of travel, whose magnitude is the speed in meters per second. Defaults to $(0, 0, 0)$.
- **angularVelocity** (*org.javatuples.Triplet<Double,Double,Double>*) – The angular velocity to offset the parent frame by, as a vector. This vector points in the direction of the axis of rotation, and its magnitude is the speed of the rotation in radians per second. Defaults to $(0, 0, 0)$.

static *ReferenceFrame* **createHybrid**(*Connection connection*, *ReferenceFrame position*, *ReferenceFrame rotation*, *ReferenceFrame velocity*, *ReferenceFrame angularVelocity*)

Create a hybrid reference frame. This is a custom reference frame whose components inherited from other reference frames.

Parameters

- **position** (ReferenceFrame) – The reference frame providing the position of the origin.
- **rotation** (ReferenceFrame) – The reference frame providing the rotation of the frame.
- **velocity** (ReferenceFrame) – The reference frame providing the linear velocity of the frame.
- **angularVelocity** (ReferenceFrame) – The reference frame providing the angular velocity of the frame.

Note: The *position* reference frame is required but all other reference frames are optional. If omitted, they are set to the *position* reference frame.

6.3.12 AutoPilot

public class **AutoPilot**

Provides basic auto-piloting utilities for a vessel. Created by calling *Vessel.getAutoPilot()*.

Note: If a client engages the auto-pilot and then closes its connection to the server, the auto-pilot will be disengaged and its target reference frame, direction and roll reset to default.

---

void **engage**()

Engage the auto-pilot.

void **disengage**()

Disengage the auto-pilot.

void **wait_**()

Blocks until the vessel is pointing in the target direction and has the target roll (if set). Throws an exception if the auto-pilot has not been engaged.

float **getError**()

The error, in degrees, between the direction the ship has been asked

to point in and the direction it is pointing in. Throws an exception if the auto-pilot has not been engaged and SAS is not enabled or is in stability assist mode.

float **getPitchError**()
The error, in degrees, between the vessels current and target pitch. Throws an exception if the auto-pilot has not been engaged.

float **getHeadingError**()
The error, in degrees, between the vessels current and target heading. Throws an exception if the auto-pilot has not been engaged.

float **getRollError**()
The error, in degrees, between the vessels current and target roll. Throws an exception if the auto-pilot has not been engaged or no target roll is set.

ReferenceFrame **getReferenceFrame**()

void **setReferenceFrame**(*ReferenceFrame value*)
The reference frame for the target direction (`AutoPilot.getTargetDirection()`).

Note: An error will be thrown if this property is set to a reference frame that rotates with the vessel being controlled, as it is impossible to rotate the vessel in such a reference frame.

---

float **getTargetPitch**()

void **setTargetPitch**(float *value*)
The target pitch, in degrees, between -90° and +90°.

float **getTargetHeading**()

void **setTargetHeading**(float *value*)
The target heading, in degrees, between 0° and 360°.

float **getTargetRoll**()

void **setTargetRoll**(float *value*)
The target roll, in degrees. `NaN` if no target roll is set.

org.javatuples.Triplet<Double, Double, Double> **getTargetDirection**()

void **setTargetDirection**(org.javatuples.Triplet<Double, Double, Double> *value*)
Direction vector corresponding to the target pitch and heading. This is in the reference frame specified by `ReferenceFrame`.

void **targetPitchAndHeading**(float *pitch*, float *heading*)
Set target pitch and heading angles.

Parameters

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

boolean **getSAS**()

void **setSAS**(boolean *value*)
The state of SAS.

Note: Equivalent to `Control.getSAS()`

SASMode **getSASMode**()

void **setSASMode**(*SASMode value*)
The current `SASMode`. These modes are equivalent to the mode buttons to the left of the navball that appear when SAS is enabled.

Note: Equivalent to `Control.getSASMode()`

double **getRollThreshold**()

void **setRollThreshold**(double *value*)
The threshold at which the autopilot will try to match the target roll angle, if any. Defaults to 5 degrees.

org.javatuples.Triplet<Double, Double, Double> **getStoppingTime**()

void **setStoppingTime**(org.javatuples.Triplet<Double, Double, Double> *value*)
The maximum amount of time that the vessel should need to come to a complete stop. This determines the maximum angular velocity of the vessel. A vector of three stopping times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 0.5 seconds for each axis.

org.javatuples.Triplet<Double, Double, Double> **getDecelerationTime**()

void **setDecelerationTime**(org.javatuples.Triplet<Double, Double, Double> *value*)
The time the vessel should take to come to a stop pointing in the target direction. This determines the angular acceleration used to decelerate the vessel. A vector of three times, in seconds, one for each of the pitch, roll and yaw axes. Defaults to 5 seconds for each axis.

org.javatuples.Triplet<Double, Double, Double> **getAttenuationAngle**()

void **setAttenuationAngle**(org.javatuples.Triplet<Double, Double, Double> *value*)
The angle at which the autopilot considers the vessel to be pointing close to the target. This determines the midpoint of the target velocity attenuation function. A vector of three angles, in degrees, one for each of the pitch, roll and yaw axes. Defaults to 1° for each axis.

boolean **getAutoTune**()

void **setAutoTune**(boolean *value*)
Whether the rotation rate controllers PID parameters should be automatically tuned using the vessels moment of inertia and available torque. Defaults to `true`. See *`AutoPilot.getTimeToPeak()`* and *`AutoPilot.getOvershoot()`*.

org.javatuples.Triplet<Double, Double, Double> **getTimeToPeak**()

void **setTimeToPeak**(org.javatuples.Triplet<Double, Double, Double> *value*)
The target time to peak used to autotune the PID controllers. A vector of three times, in seconds, for each of the pitch, roll and yaw axes. Defaults to 3 seconds for each axis.

org.javatuples.Triplet<Double, Double, Double> **getOvershoot**()

void **setOvershoot**(org.javatuples.Triplet<Double, Double, Double> *value*)
The target overshoot percentage used to autotune the PID controllers. A vector of three values, between 0 and 1, for each of the pitch, roll and yaw axes. Defaults to 0.01 for each axis.

org.javatuples.Triplet<Double, Double, Double> **getPitchPIDGains**()

void **setPitchPIDGains**(org.javatuples.Triplet<Double, Double, Double> *value*)
Gains for the pitch PID controller.

Note: When *`AutoPilot.getAutoTune()`* is true, these values are updated automatically, which will overwrite any manual changes.

---

org.javatuples.Triplet<Double, Double, Double> **getRollPIDGains**()

void **setRollPIDGains**(org.javatuples.Triplet<Double, Double, Double> *value*)
Gains for the roll PID controller.

Note: When *`AutoPilot.getAutoTune()`* is true, these values are updated automatically, which will overwrite any manual changes.

---

org.javatuples.Triplet<Double, Double, Double> **getYawPIDGains**()

void **setYawPIDGains**(org.javatuples.Triplet<Double, Double, Double> *value*)
Gains for the yaw PID controller.

Note: When *`AutoPilot.getAutoTune()`* is true, these values are updated automatically, which will overwrite any manual changes.

6.3.13 Camera

public class **Camera**
Controls the game's camera. Obtained by calling `getCamera()`.

CameraMode **getMode**()

void **setMode**(*CameraMode value*)
The current mode of the camera.

float **getPitch**()

void **setPitch**(float *value*)
The pitch of the camera, in degrees. A value between `Camera.getMinPitch()` and `Camera.getMaxPitch()`

float **getHeading**()

void **setHeading**(float *value*)
The heading of the camera, in degrees.

float **getDistance**()

void **setDistance**(float *value*)
The distance from the camera to the subject, in meters. A value between `Camera.getMinDistance()` and `Camera.getMaxDistance()`.

float **getMinPitch**()
The minimum pitch of the camera.

float **getMaxPitch**()
The maximum pitch of the camera.

float **getMinDistance**()
Minimum distance from the camera to the subject, in meters.

float **getMaxDistance**()
Maximum distance from the camera to the subject, in meters.

float **getDefaultDistance**()
Default distance from the camera to the subject, in meters.

CelestialBody **getFocussedBody**()

void **setFocussedBody**(*CelestialBody value*)
In map mode, the celestial body that the camera is focussed on. Returns `null` if the camera is not focussed on a celestial body. Returns an error is the camera is not in map mode.

Vessel **getFocussedVessel**()

void **setFocussedVessel**(*Vessel value*)
In map mode, the vessel that the camera is focussed on. Returns `null` if the camera is not focussed on a vessel. Returns an error is the camera is not in map mode.

Node **getFocussedNode**()

void **setFocussedNode**(*Node value*)
In map mode, the maneuver node that the camera is focussed on. Returns `null` if the camera is not focussed on a maneuver node. Returns an error is the camera is not in map mode.

public enum **CameraMode**
See *`Camera.getMode()`*.

public *CameraMode* **AUTOMATIC**
The camera is showing the active vessel, in "auto" mode.

public *CameraMode* **FREE**
The camera is showing the active vessel, in "free" mode.

public *CameraMode* **CHASE**
The camera is showing the active vessel, in "chase" mode.

public *CameraMode* **LOCKED**
The camera is showing the active vessel, in "locked" mode.

public *CameraMode* **ORBITAL**
The camera is showing the active vessel, in "orbital" mode.

public *CameraMode* **IVA**
The Intra-Vehicular Activity view is being shown.

public *CameraMode* **MAP**
The map view is being shown.

6.3.14 Waypoints

public class **WaypointManager**
Waypoints are the location markers you can see on the map view showing you where contracts are targeted for. With this structure, you can obtain coordinate data for the locations of these waypoints. Obtained by calling *`getWaypointManager()`*.

java.util.List<*Waypoint*> **getWaypoints**()
A list of all existing waypoints.

Waypoint **addWaypoint**(double *latitude*, double *longitude*, *CelestialBody body*, String *name*)
Creates a waypoint at the given position at ground level, and returns a *`Waypoint`* object that can be used to modify it.

Parameters

- **latitude** (*`double`*) – Latitude of the waypoint.
- **longitude** (*`double`*) – Longitude of the waypoint.
- **body** (`CelestialBody`) – Celestial body the waypoint is attached to.
- **name** (*`String`*) – Name of the waypoint.

Waypoint **addWaypointAtAltitude**(double *latitude*, double *longitude*, double *altitude*, *CelestialBody body*, String *name*)
Creates a waypoint at the given position and altitude, and returns a *`Waypoint`* object that can be used to modify it.

Parameters

- **latitude** (*double*) – Latitude of the waypoint.
- **longitude** (*double*) – Longitude of the waypoint.
- **altitude** (*double*) – Altitude (above sea level) of the waypoint.
- **body** (CelestialBody) – Celestial body the waypoint is attached to.
- **name** (*String*) – Name of the waypoint.

java.util.Map<String, Integer> **getColors**()
An example map of known color - seed pairs. Any other integers may be used as seed.

java.util.List<String> **getIcons**()
Returns all available icons (from "GameData/Squad/Contracts/Icons/").

public class **Waypoint**
Represents a waypoint. Can be created using *WaypointManager.addWaypoint(double, double, CelestialBody, String)*.

CelestialBody **getBody**()

void **setBody**(*CelestialBody value*)
The celestial body the waypoint is attached to.

String **getName**()

void **setName**(String *value*)
The name of the waypoint as it appears on the map and the contract.

int **getColor**()

void **setColor**(int *value*)
The seed of the icon color. See *WaypointManager.getColors()* for example colors.

String **getIcon**()

void **setIcon**(String *value*)
The icon of the waypoint.

double **getLatitude**()

void **setLatitude**(double *value*)
The latitude of the waypoint.

double **getLongitude**()

void **setLongitude**(double *value*)
The longitude of the waypoint.

double **getMeanAltitude**()

void **setMeanAltitude**(double *value*)
The altitude of the waypoint above sea level, in meters.

double **getSurfaceAltitude**()

void **setSurfaceAltitude**(double *value*)
The altitude of the waypoint above the surface of the body or sea level, whichever is closer, in meters.

double **getBedrockAltitude**()

void **setBedrockAltitude**(double *value*)
The altitude of the waypoint above the surface of the body, in meters. When over water, this is the altitude above the sea floor.

boolean **getNearSurface**()
`true` if the waypoint is near to the surface of a body.

boolean **getGrounded**()
`true` if the waypoint is attached to the ground.

int **getIndex**()
The integer index of this waypoint within its cluster of sibling waypoints. In other words, when you have a cluster of waypoints called "Somewhere Alpha", "Somewhere Beta" and "Somewhere Gamma", the alpha site has index 0, the beta site has index 1 and the gamma site has index 2. When *`Waypoint.getClustered()`* is `false`, this is zero.

boolean **getClustered**()
`true` if this waypoint is part of a set of clustered waypoints with greek letter names appended (Alpha, Beta, Gamma, etc). If `true`, there is a one-to-one correspondence with the greek letter name and the *`Waypoint.getIndex()`*.

boolean **getHasContract**()
Whether the waypoint belongs to a contract.

Contract **getContract**()
The associated contract.

void **remove**()
Removes the waypoint.

6.3.15 Contracts

public class **ContractManager**
Contracts manager. Obtained by calling *`getWaypointManager()`*.

java.util.Set<String> **getTypes**()
A list of all contract types.

java.util.List<*Contract*> **getAllContracts**()
A list of all contracts.

java.util.List<*Contract*> **getActiveContracts**()
A list of all active contracts.

java.util.List<*Contract*> **getOfferedContracts**()
A list of all offered, but unaccepted, contracts.

java.util.List<*Contract*> **getCompletedContracts**()
A list of all completed contracts.

java.util.List<*Contract*> **getFailedContracts**()
A list of all failed contracts.

public class **Contract**
A contract. Can be accessed using *getContractManager()*.

String **getType**()
Type of the contract.

String **getTitle**()
Title of the contract.

String **getDescription**()
Description of the contract.

String **getNotes**()
Notes for the contract.

String **getSynopsis**()
Synopsis for the contract.

java.util.List<String> **getKeywords**()
Keywords for the contract.

ContractState **getState**()
State of the contract.

boolean **getSeen**()
Whether the contract has been seen.

boolean **getRead**()
Whether the contract has been read.

boolean **getActive**()
Whether the contract is active.

boolean **getFailed**()
Whether the contract has been failed.

boolean **getCanBeCanceled**()
Whether the contract can be canceled.

boolean **getCanBeDeclined**()
Whether the contract can be declined.

boolean **getCanBeFailed**()
Whether the contract can be failed.

void **accept**()
Accept an offered contract.

void **cancel**()
Cancel an active contract.

void **decline**()
Decline an offered contract.

double **getFundsAdvance**()
Funds received when accepting the contract.

double **getFundsCompletion**()
Funds received on completion of the contract.

double **getFundsFailure**()
Funds lost if the contract is failed.

double **getReputationCompletion**()
Reputation gained on completion of the contract.

double **getReputationFailure**()
Reputation lost if the contract is failed.

double **getScienceCompletion**()
Science gained on completion of the contract.

java.util.List<*ContractParameter*> **getParameters**()
Parameters for the contract.

public enum **ContractState**
The state of a contract. See *Contract.getState()*.

public *ContractState* **ACTIVE**
The contract is active.

public *ContractState* **CANCELED**
The contract has been canceled.

public *ContractState* **COMPLETED**
The contract has been completed.

public *ContractState* **DEADLINE_EXPIRED**
The deadline for the contract has expired.

public *ContractState* **DECLINED**
The contract has been declined.

public *ContractState* **FAILED**
The contract has been failed.

public *ContractState* **GENERATED**
The contract has been generated.

public *ContractState* **OFFERED**
The contract has been offered to the player.

public *ContractState* **OFFER_EXPIRED**
The contract was offered to the player, but the offer expired.

public *ContractState* **WITHDRAWN**
The contract has been withdrawn.

public class **ContractParameter**
A contract parameter. See *Contract.getParameters()*.

String **getTitle**()
Title of the parameter.

String **getNotes**()
Notes for the parameter.

java.util.List<*ContractParameter*> **getChildren**()
Child contract parameters.

boolean **getCompleted**()
Whether the parameter has been completed.

boolean **getFailed**()
Whether the parameter has been failed.

boolean **getOptional**()
Whether the contract parameter is optional.

double **getFundsCompletion**()
Funds received on completion of the contract parameter.

double **getFundsFailure**()
Funds lost if the contract parameter is failed.

double **getReputationCompletion**()
Reputation gained on completion of the contract parameter.

double **getReputationFailure**()
Reputation lost if the contract parameter is failed.

double **getScienceCompletion**()
Science gained on completion of the contract parameter.

6.3.16 Geometry Types

Vectors

3-dimensional vectors are represented as a 3-tuple. For example:

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

import org.javatuples.Triplet;

import java.io.IOException;

public class Vector3 {
    public static void main(String[]␣
↪args) throws IOException, RPCException {
        Connection␣
↪connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.
↪newInstance(connection).getActiveVessel();
        Triplet<Double, Double,
↪ Double> v = vessel.flight(null).getPrograde();
        System.out.println(v.getValue0()␣
↪+ ", " + v.getValue1() + ", " + v.getValue2());
        connection.close();
    }
}
```

Quaternions

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

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

import org.javatuples.Quartet;

import java.io.IOException;

public class Quaternion {
    public static void main(String[]␣
↪args) throws IOException, RPCException {
        Connection␣
↪connection = Connection.newInstance();
        Vessel vessel = SpaceCenter.
↪newInstance(connection).getActiveVessel();
        Quartet<Double, Double, Double,
↪ Double> q = vessel.flight(null).getRotation();
        System.out.
↪println(q.getValue0() + ", " + q.getValue1()␣
↪+ ", " + q.getValue2() + ", " + q.getValue3());
        connection.close();
    }
}
```

6.4 Drawing API

6.4.1 Drawing

public class **Drawing**

Provides functionality for drawing objects in the flight scene.

Line **addLine**(org.javatuples.Triplet<Double, Double, Double> *start*, org.javatuples.Triplet<Double, Double, Double> *end*, *SpaceCenter.ReferenceFrame referenceFrame*, boolean *visible*)

Draw a line in the scene.

Parameters

- **start** (*org.javatuples.Triplet<Double,Double,Double>*) – Position of the start of the line.
- **end** (*org.javatuples.Triplet<Double,Double,Double>*) – Position of the end of the line.
- **referenceFrame** (SpaceCenter.ReferenceFrame) – Reference frame that the positions are in.
- **visible** (*boolean*) – Whether the line is visible.

Line **addDirection**(org.javatuples.Triplet<Double, Double, Double> *direction*, *SpaceCenter.ReferenceFrame referenceFrame*, float *length*, boolean *visible*)

Draw a direction vector in the scene, from the center of mass of the active vessel.

Parameters

- **direction** (*org.javatuples.Triplet<Double,Double,Double>*) – Direction to draw the line in.
- **referenceFrame** (SpaceCenter.ReferenceFrame) – Reference frame that the direction is in.
- **length** (*float*) – The length of the line.
- **visible** (*boolean*) – Whether the line is visible.

Polygon **addPolygon** (java.util.List<org.javatuples.Triplet<Double, Double, Double>> *vertices*, *SpaceCenter.ReferenceFrame referenceFrame*, boolean *visible*)

Draw a polygon in the scene, defined by a list of vertices.

Parameters

- **vertices** (*java.util.List<org.javatuples.Triplet<Double, Double,Double>>*) – Vertices of the polygon.
- **referenceFrame** (SpaceCenter.ReferenceFrame) – Reference frame that the vertices are in.
- **visible** (*boolean*) – Whether the polygon is visible.

Text **addText** (String *text*, *SpaceCenter.ReferenceFrame referenceFrame*, org.javatuples.Triplet<Double, Double, Double> *position*, org.javatuples.Quartet<Double, Double, Double, Double> *rotation*, boolean *visible*)

Draw text in the scene.

Parameters

- **text** (*String*) – The string to draw.
- **referenceFrame** (SpaceCenter.ReferenceFrame) – Reference frame that the text position is in.
- **position** (*org.javatuples.Triplet<Double,Double,Double>*) – Position of the text.
- **rotation** (*org.javatuples.Quartet<Double,Double,Double, Double>*) – Rotation of the text, as a quaternion.
- **visible** (*boolean*) – Whether the text is visible.

void **clear** (boolean *clientOnly*)

Remove all objects being drawn.

Parameters

- **clientOnly** (*boolean*) – If true, only remove objects created by the calling client.

6.4.2 Line

public class **Line**

A line. Created using *addLine(org.javatuples.Triplet<Double,Double,Double>, org.javatuples.Triplet<Double,Double,Double>, SpaceCenter.ReferenceFrame, boolean)*.

org.javatuples.Triplet<Double, Double, Double> **getStart** ()

void **setStart**(org.javatuples.Triplet<Double, Double, Double> *value*)
Start position of the line.

org.javatuples.Triplet<Double, Double, Double> **getEnd**()

void **setEnd**(org.javatuples.Triplet<Double, Double, Double> *value*)
End position of the line.

SpaceCenter.ReferenceFrame **getReferenceFrame**()

void **setReferenceFrame**(*SpaceCenter.ReferenceFrame value*)
Reference frame for the positions of the object.

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the object is visible.

org.javatuples.Triplet<Double, Double, Double> **getColor**()

void **setColor**(org.javatuples.Triplet<Double, Double, Double> *value*)
Set the color

String **getMaterial**()

void **setMaterial**(String *value*)
Material used to render the object. Creates the material from a shader with the given name.

float **getThickness**()

void **setThickness**(float *value*)
Set the thickness

void **remove**()
Remove the object.

6.4.3 Polygon

public class **Polygon**
A polygon. Created using *addPolygon(java.util.List<org.javatuples.Triplet<Double,Double,Double>>, SpaceCenter.ReferenceFrame, boolean)*.

java.util.List<org.javatuples.Triplet<Double, Double, Double>> **getVertices**()

void **setVertices**(java.util.List<org.javatuples.Triplet<Double, Double, Double>> *value*)
Vertices for the polygon.

SpaceCenter.ReferenceFrame **getReferenceFrame**()

void **setReferenceFrame**(*SpaceCenter.ReferenceFrame value*)
Reference frame for the positions of the object.

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the object is visible.

void **remove**()
Remove the object.

org.javatuples.Triplet<Double, Double, Double> **getColor**()

void **setColor**(org.javatuples.Triplet<Double, Double, Double> *value*)
Set the color

String **getMaterial**()

void **setMaterial**(String *value*)
Material used to render the object. Creates the material from a shader with the given name.

float **getThickness**()

void **setThickness**(float *value*)
Set the thickness

6.4.4 Text

public class **Text**
Text. Created using *addText(String, SpaceCenter.ReferenceFrame, org.javatuples.Triplet<Double,Double,Double>, org.javatuples.Quartet<Double,Double,Double,Double>, boolean)*.

org.javatuples.Triplet<Double, Double, Double> **getPosition**()

void **setPosition**(org.javatuples.Triplet<Double, Double, Double> *value*)
Position of the text.

org.javatuples.Quartet<Double, Double, Double, Double> **getRotation**()

void **setRotation**(org.javatuples.Quartet<Double, Double, Double, Double> *value*)
Rotation of the text as a quaternion.

SpaceCenter.ReferenceFrame **getReferenceFrame**()

void **setReferenceFrame**(*SpaceCenter.ReferenceFrame value*)
Reference frame for the positions of the object.

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the object is visible.

void **remove**()
Remove the object.

String **getContent**()

void **setContent**(String *value*)
The text string

String **getFont**()

void **setFont**(String *value*)
Name of the font

static java.util.List<String> **availableFonts**(*Connection connection*)
A list of all available fonts.

int **getSize**()

void **setSize**(int *value*)
Font size.

float **getCharacterSize**()

void **setCharacterSize**(float *value*)
Character size.

UI.FontStyle **getStyle**()

void **setStyle**(*UI.FontStyle value*)
Font style.

org.javatuples.Triplet<Double, Double, Double> **getColor**()

void **setColor**(org.javatuples.Triplet<Double, Double, Double> *value*)
Set the color

String **getMaterial**()

void **setMaterial**(String *value*)
Material used to render the object. Creates the material from a shader with the given name.

UI.TextAlignment **getAlignment**()

void **setAlignment**(*UI.TextAlignment value*)
Alignment.

float **getLineSpacing**()

void **setLineSpacing**(float *value*)
Line spacing.

UI.TextAnchor **getAnchor**()

void **setAnchor**(*UI.TextAnchor value*)
Anchor.

6.5 InfernalRobotics API

Provides RPCs to interact with the InfernalRobotics mod. Provides the following classes:

6.5.1 InfernalRobotics

public class **InfernalRobotics**
This service provides functionality to interact with Infernal Robotics.

boolean **getAvailable**()
Whether Infernal Robotics is installed.

java.util.List<*ServoGroup*> **servoGroups**(*SpaceCenter.Vessel vessel*)
A list of all the servo groups in the given *vessel*.

Parameters

- **vessel** (SpaceCenter.Vessel) –

ServoGroup **servoGroupWithName**(*SpaceCenter.Vessel vessel*, String *name*)
Returns the servo group in the given *vessel* with the given *name*, or `null` if none exists. If multiple servo groups have the same name, only one of them is returned.

Parameters

- **vessel** (SpaceCenter.Vessel) – Vessel to check.
- **name** (*String*) – Name of servo group to find.

Servo **servoWithName**(*SpaceCenter.Vessel vessel*, String *name*)
Returns the servo in the given *vessel* with the given *name* or `null` if none exists. If multiple servos have the same name, only one of them is returned.

Parameters

- **vessel** (SpaceCenter.Vessel) – Vessel to check.
- **name** (*String*) – Name of the servo to find.

6.5.2 ServoGroup

public class **ServoGroup**
A group of servos, obtained by calling *servoGroups(SpaceCenter.Vessel)* or *servoGroupWithName(SpaceCenter.Vessel, String)*. Represents the "Servo Groups" in the InfernalRobotics UI.

String **getName**()

void **setName**(String *value*)
The name of the group.

String **getForwardKey**()

void **setForwardKey**(String *value*)
The key assigned to be the "forward" key for the group.

String **getReverseKey**()

void **setReverseKey**(String *value*)
The key assigned to be the "reverse" key for the group.

float **getSpeed**()

void **setSpeed**(float *value*)
The speed multiplier for the group.

boolean **getExpanded**()

void **setExpanded**(boolean *value*)
Whether the group is expanded in the InfernalRobotics UI.

java.util.List<*Servo*> **getServos**()
The servos that are in the group.

Servo **servoWithName**(String *name*)
Returns the servo with the given *name* from this group, or `null` if none exists.

Parameters

- **name** (*String*) – Name of servo to find.

java.util.List<*SpaceCenter.Part*> **getParts**()
The parts containing the servos in the group.

void **moveRight**()
Moves all of the servos in the group to the right.

void **moveLeft**()
Moves all of the servos in the group to the left.

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

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

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

void **stop**()
Stops the servos in the group.

6.5.3 Servo

public class **Servo**
Represents a servo. Obtained using *ServoGroup.*

getServos(), *ServoGroup.servoWithName(String)* or *servoWithName(SpaceCenter.Vessel, String)*.

String **getName**()

void **setName**(String *value*)
The name of the servo.

SpaceCenter.Part **getPart**()
The part containing the servo.

void **setHighlight**(boolean *value*)
Whether the servo should be highlighted in-game.

float **getPosition**()
The position of the servo.

float **getMinConfigPosition**()
The minimum position of the servo, specified by the part configuration.

float **getMaxConfigPosition**()
The maximum position of the servo, specified by the part configuration.

float **getMinPosition**()

void **setMinPosition**(float *value*)
The minimum position of the servo, specified by the in-game tweak menu.

float **getMaxPosition**()

void **setMaxPosition**(float *value*)
The maximum position of the servo, specified by the in-game tweak menu.

float **getConfigSpeed**()
The speed multiplier of the servo, specified by the part configuration.

float **getSpeed**()

void **setSpeed**(float *value*)
The speed multiplier of the servo, specified by the in-game tweak menu.

float **getCurrentSpeed**()

void **setCurrentSpeed**(float *value*)
The current speed at which the servo is moving.

float **getAcceleration**()

void **setAcceleration**(float *value*)
The current speed multiplier set in the UI.

boolean **getIsMoving**()
Whether the servo is moving.

boolean **getIsFreeMoving**()
Whether the servo is freely moving.

boolean **getIsLocked**()

void **setIsLocked**(boolean *value*)
Whether the servo is locked.

boolean **getIsAxisInverted**()

void **setIsAxisInverted**(boolean *value*)
Whether the servos axis is inverted.

void **moveRight**()
Moves the servo to the right.

void **moveLeft**()
Moves the servo to the left.

void **moveCenter**()
Moves the servo to the center.

void **moveNextPreset**()
Moves the servo to the next preset.

void **movePrevPreset**()
Moves the servo to the previous preset.

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

Parameters

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

void **stop**()
Stops the servo.

6.5.4 Example

The following example gets the control group named "MyGroup",
prints out the names and positions of all of the servos in the group,
then moves all of the servos to the right for 1 second.

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.InfernalRobotics;
import␣
↪krpc.client.services.InfernalRobotics.Servo;
import krpc.
↪client.services.InfernalRobotics.ServoGroup;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Vessel;
```

```
import java.io.IOException;

public class InfernalRoboticsExample {
    public static␣
↪void main(String[] args) throws IOException,
↪ RPCException, InterruptedException {
        Connection connection = Connection.
↪newInstance("InfernalRobotics Example");
        Vessel vessel = SpaceCenter.
↪newInstance(connection).getActiveVessel();
        InfernalRobotics␣
↪ir = InfernalRobotics.newInstance(connection);

        ServoGroup group␣
↪= ir.servoGroupWithName(vessel, "MyGroup");
        if (group == null) {
            System.out.println("Group not found");
            return;
        }

        for (Servo servo : group.getServos()) {
            System.out.println(servo.
↪getName() + " " + servo.getPosition());
        }

        group.moveRight();
        Thread.sleep(1000);
        group.stop();
        connection.close();
    }
}
```

6.6 Kerbal Alarm Clock API

Provides RPCs to interact with the Kerbal Alarm Clock mod. Provides the following classes:

6.6.1 KerbalAlarmClock

public class **KerbalAlarmClock**

This service provides functionality to interact with Kerbal Alarm Clock.

boolean **getAvailable**()

Whether Kerbal Alarm Clock is available.

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.

6.6.2 Alarm

public class **Alarm**
Represents an alarm. Obtained by calling *getAlarms()*, *alarmWithName(String)* or *alarmsWithType(AlarmType)*.

AlarmAction **getAction**()

void **setAction**(*AlarmAction value*)
The action that the alarm triggers.

double **getMargin**()

void **setMargin**(double *value*)
The number of seconds before the event that the alarm will fire.

double **getTime**()

void **setTime**(double *value*)
The time at which the alarm will fire.

AlarmType **getType**()
The type of the alarm.

String **getID**()
The unique identifier for the alarm.

String **getName**()

void **setName**(String *value*)
The short name of the alarm.

String **getNotes**()

void **setNotes**(String *value*)
The long description of the alarm.

double **getRemaining**()
The number of seconds until the alarm will fire.

boolean **getRepeat**()

void **setRepeat**(boolean *value*)
Whether the alarm will be repeated after it has fired.

double **getRepeatPeriod**()

void **setRepeatPeriod**(double *value*)
The time delay to automatically create an alarm after it has fired.

SpaceCenter.Vessel **getVessel**()

void **setVessel**(*SpaceCenter.Vessel value*)
The vessel that the alarm is attached to.

SpaceCenter.CelestialBody **getXferOriginBody**()

void **setXferOriginBody**(*SpaceCenter.CelestialBody value*)
The celestial body the vessel is departing from.

SpaceCenter.CelestialBody **getXferTargetBody**()

void **setXferTargetBody**(*SpaceCenter.CelestialBody value*)
The celestial body the vessel is arriving at.

void **remove**()
Removes the alarm.

6.6.3 AlarmType

public enum **AlarmType**
The type of an alarm.

public *AlarmType* **RAW**
An alarm for a specific date/time or a specific period in the future.

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

public *AlarmType* **MANEUVER_AUTO**
See *AlarmType.MANEUVER*.

public *AlarmType* **APOAPSIS**
An alarm for furthest part of the orbit from the planet.

public *AlarmType* **PERIAPSIS**
An alarm for nearest part of the orbit from the planet.

public *AlarmType* **ASCENDING_NODE**
Ascending node for the targeted object, or equatorial ascending node.

public *AlarmType* **DESCENDING_NODE**
Descending node for the targeted object, or equatorial descending node.

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

public *AlarmType* **CONTRACT**
An alarm based on the expiry or deadline of contracts in career modes.

public *AlarmType* **CONTRACT_AUTO**
See *AlarmType.CONTRACT*.

public *AlarmType* **CREW**
An alarm that is attached to a crew member.

public *AlarmType* **DISTANCE**
An alarm that is triggered when a selected target comes within a chosen distance.

public *AlarmType* **EARTH_TIME**
An alarm based on the time in the "Earth" alternative Universe (aka the Real World).

public *AlarmType* **LAUNCH_RENDEVOUS**
An alarm that fires as your landed craft passes under the orbit of your target.

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

public *AlarmType* **SOI_CHANGE_AUTO**
See *AlarmType.SOI_CHANGE*.

public *AlarmType* **TRANSFER**
An alarm based on Interplanetary Transfer Phase Angles, i.e. when should I launch to planet X? Based on Kosmo Not's post and used in Olex's Calculator.

public *AlarmType* **TRANSFER_MODELLED**
See *AlarmType.TRANSFER*.

6.6.4 AlarmAction

public enum **AlarmAction**
The action performed by an alarm when it fires.

public *AlarmAction* **DO_NOTHING**
Don't do anything at all...

public *AlarmAction* **DO_NOTHING_DELETE_WHEN_PASSED**
Don't do anything, and delete the alarm.

public *AlarmAction* **KILL_WARP**
Drop out of time warp.

public *AlarmAction* **KILL_WARP_ONLY**
Drop out of time warp.

public *AlarmAction* **MESSAGE_ONLY**
Display a message.

public *AlarmAction* **PAUSE_GAME**
Pause the game.

6.6.5 Example

The following example creates a new alarm for the active vessel. The alarm is set to trigger after 10 seconds have passed, and display a message.

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.KerbalAlarmClock;
import␣
↪krpc.client.services.KerbalAlarmClock.Alarm;
import krpc.
↪client.services.KerbalAlarmClock.AlarmAction;
import␣
↪krpc.client.services.KerbalAlarmClock.AlarmType;
import krpc.client.services.SpaceCenter;

import java.io.IOException;

public class KerbalAlarmClockExample {
    public static void main(String[]␣
↪args) throws IOException, RPCException {
        Connection connection = Connection.
↪newInstance("Kerbal Alarm Clock Example");
        KerbalAlarmClock␣
↪kac = KerbalAlarmClock.newInstance(connection);
        Alarm alarm = kac.createAlarm(AlarmType.
↪RAW, "My New Alarm", SpaceCenter.
↪newInstance(connection).getUT() + 10);
        alarm.setNotes("10 seconds␣
↪have now passed since the alarm was created.");
        alarm.setAction(AlarmAction.MESSAGE_ONLY);
        connection.close();
    }
}
```

6.7 RemoteTech API

Provides RPCs to interact with the RemoteTech mod. Provides the following classes:

6.7.1 RemoteTech

public class **RemoteTech**
This service provides functionality to interact with RemoteTech.

boolean **getAvailable**()
Whether RemoteTech is installed.

java.util.List<String> **getGroundStations**()
The names of the ground stations.

Antenna **antenna**(*SpaceCenter.Part part*)
Get the antenna object for a particular part.

Parameters

- **part** (SpaceCenter.Part) –

Comms **comms**(*SpaceCenter.Vessel vessel*)
Get a communications object, representing the communication capability of a particular vessel.

Parameters

- **vessel** (SpaceCenter.Vessel) –

6.7.2 Comms

public class **Comms**
Communications for a vessel.

SpaceCenter.Vessel **getVessel**()
Get the vessel.

boolean **getHasLocalControl**()
Whether the vessel can be controlled locally.

boolean **getHasFlightComputer**()
Whether the vessel has a flight computer on board.

boolean **getHasConnection**()
Whether the vessel has any connection.

boolean **getHasConnectionToGroundStation**()
Whether the vessel has a connection to a ground station.

double **getSignalDelay**()
The shortest signal delay to the vessel, in seconds.

double **getSignalDelayToGroundStation**()
The signal delay between the vessel and the closest ground station, in seconds.

double **signalDelayToVessel**(*SpaceCenter.Vessel other*)
The signal delay between the this vessel and another vessel, in seconds.

Parameters

- **other** (SpaceCenter.Vessel) –

java.util.List<*Antenna*> **getAntennas**()
The antennas for this vessel.

6.7.3 Antenna

public class **Antenna**
A RemoteTech antenna. Obtained by calling *Comms.getAntennas()* or *antenna(SpaceCenter.Part)*.

SpaceCenter.Part **getPart**()
Get the part containing this antenna.

boolean **getHasConnection**()
Whether the antenna has a connection.

Target **getTarget**()

void **setTarget**(*Target value*)
The object that the antenna is targetting. This property can be used to set the target to *Target.NONE* or *Target.ACTIVE_VESSEL*. To set the target to a celestial body, ground station or vessel see *Antenna.getTargetBody()*, *Antenna.getTargetGroundStation()* and *Antenna.getTargetVessel()*.

SpaceCenter.CelestialBody **getTargetBody**()

void **setTargetBody**(*SpaceCenter.CelestialBody value*)
The celestial body the antenna is targetting.

String **getTargetGroundStation**()

void **setTargetGroundStation**(String *value*)
The ground station the antenna is targetting.

SpaceCenter.Vessel **getTargetVessel**()

void **setTargetVessel**(*SpaceCenter.Vessel value*)
The vessel the antenna is targetting.

public enum **Target**
The type of object an antenna is targetting. See *Antenna.getTarget()*.

public *Target* **ACTIVE_VESSEL**
The active vessel.

public *Target* **CELESTIAL_BODY**
A celestial body.

public *Target* **GROUND_STATION**
A ground station.

public *Target* **VESSEL**
A specific vessel.

public *Target* **NONE**
No target.

6.7.4 Example

The following example sets the target of a dish on the active vessel then prints out the signal delay to the active vessel.

```
import krpc.client.Connection;
import krpc.client.RPCException;
import krpc.client.services.RemoteTech;
import krpc.client.services.RemoteTech.Antenna;
import krpc.client.services.RemoteTech.Comms;
import krpc.client.services.SpaceCenter;
import krpc.client.services.SpaceCenter.Part;
import krpc.client.services.SpaceCenter.Vessel;

import java.io.IOException;

public class RemoteTechExample {
    public static void main(String[]␣
↪args) throws IOException, RPCException {
        Connection connection␣
↪= Connection.newInstance("RemoteTech Example");
        SpaceCenter␣
↪sc = SpaceCenter.newInstance(connection);
        RemoteTech␣
↪rt = RemoteTech.newInstance(connection);
        Vessel vessel = sc.getActiveVessel();

        // Set a dish target
        Part part = vessel.
↪getParts().withTitle("Reflectron KR-7").get(0);
        Antenna antenna = rt.antenna(part);
        antenna.
↪setTargetBody(sc.getBodies().get("Jool"));

    ␣
↪    // Get info about the vessels communications
        Comms comms = rt.comms(vessel);
        System.out.printf("Signal delay␣
↪= %.1f seconds\n", comms.getSignalDelay());
        connection.close();
    }
}
```

6.8 User Interface API

6.8.1 UI

public class **UI**

Provides functionality for drawing and interacting with in-game user interface elements.

Canvas **getStockCanvas**()

The stock UI canvas.

Canvas **addCanvas**()

Add a new canvas.

Note: If you want to add UI elements to KSPs stock UI canvas, use *getStockCanvas()*.

void **message**(String *content*, float *duration*, *MessagePosition position*)
Display a message on the screen.

Parameters

- **content** (*String*) – Message content.
- **duration** (*float*) – Duration before the message disappears, in seconds.
- **position** (MessagePosition) – Position to display the message.

Note: The message appears just like a stock message, for example quicksave or quickload messages.

void **clear**(boolean *clientOnly*)
Remove all user interface elements.

Parameters

- **clientOnly** (*boolean*) – If true, only remove objects created by the calling client.

public enum **MessagePosition**
Message position.

public *MessagePosition* **TOP_LEFT**
Top left.

public *MessagePosition* **TOP_CENTER**
Top center.

public *MessagePosition* **TOP_RIGHT**
Top right.

public *MessagePosition* **BOTTOM_CENTER**
Bottom center.

6.8.2 Canvas

public class **Canvas**
A canvas for user interface elements. See *getStockCanvas()* and *addCanvas()*.

RectTransform **getRectTransform**()
The rect transform for the canvas.

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the UI object is visible.

Panel **addPanel**(boolean *visible*)
Create a new container for user interface elements.

Parameters

- **visible** (*boolean*) – Whether the panel is visible.

Text **addText** (String *content*, boolean *visible*)
Add text to the canvas.

Parameters

- **content** (*String*) – The text.
- **visible** (*boolean*) – Whether the text is visible.

InputField **addInputField** (boolean *visible*)
Add an input field to the canvas.

Parameters

- **visible** (*boolean*) – Whether the input field is visible.

Button **addButton** (String *content*, boolean *visible*)
Add a button to the canvas.

Parameters

- **content** (*String*) – The label for the button.
- **visible** (*boolean*) – Whether the button is visible.

void **remove** ()
Remove the UI object.

6.8.3 Panel

public class **Panel**
A container for user interface elements. See *Canvas.addPanel(boolean)*.

RectTransform **getRectTransform** ()
The rect transform for the panel.

boolean **getVisible** ()

void **setVisible** (boolean *value*)
Whether the UI object is visible.

Panel **addPanel** (boolean *visible*)
Create a panel within this panel.

Parameters

- **visible** (*boolean*) – Whether the new panel is visible.

Text **addText** (String *content*, boolean *visible*)
Add text to the panel.

Parameters

- **content** (*String*) – The text.
- **visible** (*boolean*) – Whether the text is visible.

InputField **addInputField** (boolean *visible*)
Add an input field to the panel.

Parameters

- **visible** (*boolean*) – Whether the input field is visible.

Button **addButton** (String *content*, boolean *visible*)
Add a button to the panel.

Parameters

- **content** (*String*) – The label for the button.
- **visible** (*boolean*) – Whether the button is visible.

void **remove** ()
Remove the UI object.

6.8.4 Text

public class **Text**
A text label. See *Panel.addText(String, boolean)*.

RectTransform **getRectTransform** ()
The rect transform for the text.

boolean **getVisible** ()

void **setVisible** (boolean *value*)
Whether the UI object is visible.

String **getContent** ()

void **setContent** (String *value*)
The text string

String **getFont** ()

void **setFont** (String *value*)
Name of the font

java.util.List<String> **getAvailableFonts** ()
A list of all available fonts.

int **getSize** ()

void **setSize** (int *value*)
Font size.

FontStyle **getStyle** ()

void **setStyle** (*FontStyle value*)
Font style.

org.javatuples.Triplet<Double, Double, Double> **getColor** ()

void **setColor** (org.javatuples.Triplet<Double, Double, Double> *value*)
Set the color

TextAnchor **getAlignment** ()

void **setAlignment**(*TextAnchor value*)
Alignment.

float **getLineSpacing**()

void **setLineSpacing**(float *value*)
Line spacing.

void **remove**()
Remove the UI object.

public enum **FontStyle**
Font style.

public *FontStyle* **NORMAL**
Normal.

public *FontStyle* **BOLD**
Bold.

public *FontStyle* **ITALIC**
Italic.

public *FontStyle* **BOLD_AND_ITALIC**
Bold and italic.

public enum **TextAlignment**
Text alignment.

public *TextAlignment* **LEFT**
Left aligned.

public *TextAlignment* **RIGHT**
Right aligned.

public *TextAlignment* **CENTER**
Center aligned.

public enum **TextAnchor**
Text alignment.

public *TextAnchor* **LOWER_CENTER**
Lower center.

public *TextAnchor* **LOWER_LEFT**
Lower left.

public *TextAnchor* **LOWER_RIGHT**
Lower right.

public *TextAnchor* **MIDDLE_CENTER**
Middle center.

public *TextAnchor* **MIDDLE_LEFT**
Middle left.

public *TextAnchor* **MIDDLE_RIGHT**
Middle right.

public *TextAnchor* **UPPER_CENTER**
Upper center.

public *TextAnchor* **UPPER_LEFT**
Upper left.

public *TextAnchor* **UPPER_RIGHT**
Upper right.

6.8.5 Button

public class **Button**
A text label. See *Panel.addButton(String, boolean)*.

RectTransform **getRectTransform**()
The rect transform for the text.

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the UI object is visible.

Text **getText**()
The text for the button.

boolean **getClicked**()

void **setClicked**(boolean *value*)
Whether the button has been clicked.

Note: This property is set to true when the user clicks the button. A client script should reset the property to false in order to detect subsequent button presses.

---

void **remove**()
Remove the UI object.

6.8.6 InputField

public class **InputField**
An input field. See *Panel.addInputField(boolean)*.

RectTransform **getRectTransform**()
The rect transform for the input field.

boolean **getVisible**()

void **setVisible**(boolean *value*)
Whether the UI object is visible.

String **getValue**()

void **setValue**(String *value*)
The value of the input field.

Text **getText**()
The text component of the input field.

Note: Use *InputField.getValue()* to get and set the value in the field. This object can be used to alter the style of the input field's text.

---

boolean **getChanged**()

void **setChanged**(boolean *value*)
Whether the input field has been changed.

Note: This property is set to true when the user modifies the value of the input field. A client script should reset the property to false in order to detect subsequent changes.

---

void **remove**()
Remove the UI object.

6.8.7 Rect Transform

public class **RectTransform**
A Unity engine Rect Transform for a UI object. See the Unity manual for more details.

org.javatuples.Pair<Double, Double> **getPosition**()

void **setPosition**(org.javatuples.Pair<Double, Double> *value*)
Position of the rectangles pivot point relative to the anchors.

org.javatuples.Triplet<Double, Double, Double> **getLocalPosition**()

void **setLocalPosition**(org.javatuples.Triplet<Double, Double, Double> *value*)
Position of the rectangles pivot point relative to the anchors.

org.javatuples.Pair<Double, Double> **getSize**()

void **setSize**(org.javatuples.Pair<Double, Double> *value*)
Width and height of the rectangle.

org.javatuples.Pair<Double, Double> **getUpperRight**()

void **setUpperRight**(org.javatuples.Pair<Double, Double> *value*)
Position of the rectangles upper right corner relative to the anchors.

org.javatuples.Pair<Double, Double> **getLowerLeft**()

void **setLowerLeft**(org.javatuples.Pair<Double, Double> *value*)
Position of the rectangles lower left corner relative to the anchors.

void **setAnchor**(org.javatuples.Pair<Double, Double> *value*)
Set the minimum and maximum anchor points as a fraction of the size of the parent rectangle.

org.javatuples.Pair<Double, Double> **getAnchorMax**()

void **setAnchorMax**(org.javatuples.Pair<Double, Double> *value*)
The anchor point for the lower left corner of the rectangle defined as a fraction of the size of the parent rectangle.

org.javatuples.Pair<Double, Double> **getAnchorMin**()

void **setAnchorMin**(org.javatuples.Pair<Double, Double> *value*)
The anchor point for the upper right corner of the rectangle defined as a fraction of the size of the parent rectangle.

org.javatuples.Pair<Double, Double> **getPivot**()

void **setPivot**(org.javatuples.Pair<Double, Double> *value*)
Location of the pivot point around which the rectangle rotates, defined as a fraction of the size of the rectangle itself.

org.javatuples.Quartet<Double, Double, Double, Double> **getRotation**()

void **setRotation**(org.javatuples.Quartet<Double, Double, Double, Double> *value*)
Rotation, as a quaternion, of the object around its pivot point.

org.javatuples.Triplet<Double, Double, Double> **getScale**()

void **setScale**(org.javatuples.Triplet<Double, Double, Double> *value*)
Scale factor applied to the object in the x, y and z dimensions.