

ORBITER

Programming Interface

©2001-2002 Martin Schweiger

martins@medphys.ucl.ac.uk

www.medphys.ucl.ac.uk/~martins/orbit/orbit.html

14 January 2002

1.	Introduction	2
2.	Requirements.....	2
3.	Preparation	2
4.	SDK files	2
5.	Concept.....	3
6.	Some useful hints	3
7.	Sample modules	3
8.	Data types.....	4
9.	Constants.....	6
10.	Class VESSEL	6
10.1.	CONSTRUCTION/CREATION.....	6
10.2.	VESSEL PARAMETERS AND CAPABILITIES	7
10.3.	CURRENT VESSEL STATUS	14
10.4.	STATE VECTORS	16
10.5.	ORBITAL ELEMENTS	17
10.6.	TRANSFORMATIONS	19
10.7.	ATMOSPHERIC PARAMETERS	20
10.8.	VISUAL MANIPULATION.....	20
11.	Plugin callback function reference	23
12.	Vessel callback function reference	25
13.	Planet callback function reference.....	29
14.	API function reference	31
14.1.	OBTAINING OBJECT HANDLES.....	31
14.2.	OBJECT MASS FUNCTIONS.....	34
14.3.	OBJECT STATE VECTORS.....	36
14.4.	SURFACE-RELATIVE PARAMETERS.....	38
14.5.	ENGINE STATUS.....	43
14.6.	SIMULATION TIME	45
14.7.	KEYBOARD INPUT.....	47
14.8.	MFD MANAGEMENT	47
14.9.	FILE MANAGEMENT	47
15.	Standard ORBITER modules.....	49
15.1.	VsOP87	49
15.2.	LUNA.....	49
16.	Index	49

1. Introduction

This reference document contains the specification for the Orbiter Programming Interface. It is not required for running Orbiter.

The programming interface allows the development of third party modules to enhance the functionality of the Orbiter core. Examples for modules are:

- Additional instruments, simulation monitoring devices, and spacecraft controls
- Custom flight models
- Custom instrument panels (planned)
- Multiplayer modules
- Custom calculation of planetary positions

The API is in an early stage of development. Future versions will probably change significantly, mainly by expanding the list of supported functions.

2. Requirements

The following components are required to build an addon module:

- The latest Orbiter package
- The Orbiter SDK libraries and include files (contained in the Orbiter SDK package)
- A C++ compiler running under Windows (the SDK was developed with VC++, but other compilers should also work)

3. Preparation

- Install the Orbiter package, if you haven't already done so.
- Install the Orbiter SDK package. This will generate the *OrbiterSDK* subdirectory containing the header files and libraries required for building plugins.
- Create a project for your plugin DLL (the method depends on the compiler used). Make sure you use thread-safe system libraries ("Multithread DLL"). Add *OrbiterSDK\include* to the include search path, and add *OrbiterSDK\lib\Orbiter.lib* and *OrbiterSDK\lib\OrbiterSDK.lib* to the link stage.
- Write the code for your plugin, compile and link it, and move the resulting DLL to the *Orbiter\Modules\Plugin* folder.
- Run Orbiter, go to the *Modules* tab in the launchpad dialog, and activate your new plugin.

4. SDK files

The following files are contained in the Orbiter development kit:

Orbitersdk\doc*	<i>SDK documentation</i>
Orbitersdk\include Orbitersdk.h	<i>The interface header file</i>
Orbitersdk\lib Orbitersdk.lib Orbiter.lib	<i>The DLL auxiliary library</i> <i>The Orbiter API library</i>
Orbitersdk\tools*	<i>Tools for model and texture generation</i>
Orbitersdk\samples*	<i>Sample source code</i>

5. Concept

Definition of terms used in this document:

Module

A *module* is a dynamic link library (DLL) which extends or replaces functionality of the core Orbiter program. Modules interact with Orbiter via callback functions conforming to the public interface defined below.

Plugin

Plugins are generic modules not linked to any particular object. They may include popup windows for displaying or manipulating general simulation information, multiplayer interfaces, etc. Plugins can be activated or deactivated by the user via the Modules tab in the Orbiter Launchpad dialog.

Planet module

Planet modules are linked to planets or moons and are used specifically for updating planetary position and velocity data. Planet modules are referenced via the planet/moon's configuration file.

Vessel module

Vessel modules are linked to specific spacecraft, to allow customisation of the vessel's behaviour. Vessel modules are referenced via the vessel class configuration file.

In all active modules, Orbiter executes *callback functions* corresponding to certain simulation conditions. For example, whenever the simulation window is opened after the user presses the *Orbiter* button in the launchpad dialog, Orbiter calls the *opcOpenRenderWindowport* callback function in all plugins to allow initialisation routines to be performed. A plugin doesn't need to implement all callback functions defined in the interface. However, the programmer is responsible for implementing callback functions in a consistent way. For example, if the plugin allocates memory for data in *opcOpenRenderWindowport*, then this memory should be deallocated in *opcCloseRenderWindowport*. The SDK allows access to core parts of the Orbiter simulator, and bugs in active plugins may cause the program to crash.

All callback functions use a C stack frame, so they need to be defined as *extern "C"* for compilation with a C++ compiler. For convenience the *DLLCLBK* macro is provided in *Orbitersdk.h* to use as modifier for callback function definitions.

The code for the callback functions may contain calls to the Orbiter API functions, to obtain and set simulation parameters such as object positions and speed, simulation time, etc. API functions use an *oapi* ("orbiter API") prefix. API functions use a C++ stack frame.

6. Some useful hints

- Your plugin should not open popup windows or dialogs outside the render window when running in fullscreen mode. Check the fullscreen flag passed to the *opcOpenRenderWindowport* callback function to see whether it is safe to open a window.
- If you intercept a callback function which is called at each frame (like *opcTimestep*), make it as efficient as possible, or simulation performance will suffer.

7. Sample modules

The *Orbitersdk\samples* folder contains a few projects which can be used as a starting point for creating your own plugins. To compile a sample using VC++:

- Load the project file (*.dsw) into VC++.
- Build the project.
- Copy the DLL from the Debug or Release subdirectory into the *Orbiter\Modules\Plugin* directory (plugins) or into the *Orbiter\Modules* directory (planet and vessel modules).
- To activate new plugins, run Orbiter, activate the plugin under the Modules tab, and launch the simulation.

- New planet or vessel modules are used automatically if they are referenced by the relevant definition files.

Warpcontrol

Opens a dialog which allows fine-tuning of time acceleration. (available only in window mode)

Rcontrol

Opens a dialog which allows to switch and control spacecraft on the fly. (available only in window mode)

Atlantis

The complete code for Orbiter's reference implementation of the Atlantis (Space Shuttle) module, including modules for post-separation SRBs (solid rocket boosters) and main tank.

8. Data types

OBJHANDLE

A handle for a logical object. Objects can be vessels, orbital stations, spaceports, planets, moons or suns.

VISHANDLE

A handle for a visual object. These are representations for logical objects for the purpose of rendering. Visuals exist only if the object is within visual range of the camera, and are created and deleted as needed.

VECTOR3

Double precision vector $\in R^3$

Synopsis:

```
typedef union {
    double data[3];
    struct { double x, y, z; };
} VECTOR3;
```

MATRIX3

Double precision matrix $\in R^{3 \times 3}$

Synopsis:

```
typedef union {
    double data[9];
    struct { double m11, m12, m13,
                  m21, m22, m23,
                  m31, m32, m33; };
} MATRIX3;
```

ELEMENTS

Keplerian orbital elements

Synopsis:

ENGINESTATUS

Defines the thruster status for a spacecraft

Synopsis:

```
struct {
    double main;           main/retro thruster level [-1,+1]
    double hover;         hover thruster level [0,+1]
    int attmode;           attitude thruster mode [0=rot, 1=lin]
} ENGINESTATUS;
```

ENGINE_TYPE

Enumerates thruster types

Synopsis:

```
typedef enum {  
    ENGINE_MAIN,  
    ENGINE_RETRO,  
    ENGINE_HOVER,  
    ENGINE_ATTITUDE  
} ENGINE_TYPE;
```

EXHAUST_TYPE

Enumerates engine groups for exhaust rendering.

Synopsis:

```
typedef enum {  
    EXHAUST_MAIN,  
    EXHAUST_RETRO,  
    EXHAUST_HOVER,  
    EXHAUST_CUSTOM  
} EXHAUST_TYPE;
```

VESSEL_STATUS

Defines vessel status parameters at a given time.

Synopsis:

```
typedef struct {  
    VECTOR3 rpos;  
    VECTOR3 rvel;  
    VECTOR3 vrot;  
    VECTOR3 arot;  
    double fuel;  
    double eng_main;  
    double eng_hovr;  
    OBJHANDLE rbody;  
    OBJHANDLE base;  
    int port;  
    int status;  
} VESSEL_STATUS;
```

rpos	position relative to reference body (ecliptic frame)
rvel	velocity relative to reference body (ecliptic frame)
vrot	rotation velocity about principal axes (ecliptic frame)
arot	vessel orientation against ecliptic frame
fuel	fuel level [0...1]
eng_main	main engine setting [-1...1]
eng_hovr	hover engine setting [0...1]
rbody	handle of reference body
base	handle of docking or landing target
port	designated docking or landing port
status	0=freeflight, 1=landed, 2=taxiing, 3=docked, 99=undefined

Notes:

arot=(α, β, γ) contains angles of rotation [rad] around x,y,z axes in ecliptic frame to produce this rotation matrix \mathbf{R} for mapping from the vessel's local frame of reference to the global frame of reference:

$$\mathbf{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \alpha & \sin \alpha \\ 0 & -\sin \alpha & \cos \alpha \end{bmatrix} \begin{bmatrix} \cos \beta & 0 & -\sin \beta \\ 0 & 1 & 0 \\ \sin \beta & 0 & \cos \beta \end{bmatrix} \begin{bmatrix} \cos \gamma & \sin \gamma & 0 \\ -\sin \gamma & \cos \gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

such that

$$\mathbf{r}_{global} = \mathbf{R} \mathbf{r}_{local} + \mathbf{p}$$

where \mathbf{p} is the vessel's global position.

9. Constants

MFD mode constants

MFD_NONE
MFD_ORBIT
MFD_SURFACE
MFD_MAP
MFD_LANDING
MFD_DOCKING
MFD_OPLANEALIGN
MFD_OSYNC
MFD_TRANSFER
MFD_USERTYPE

MFD position constants

MFD_LEFT
MFD_RIGHT
MFD_USERPOS

10. Class VESSEL

This class constitutes the interface with Orbiter's internal vessel implementation, and provides access to the various status parameters and methods of individual spacecraft. Typically, an instance of VESSEL or a derived class will be constructed in each vessel module. An example for a typical application of the VESSEL class can be found in the Atlantis sample code.

Public member functions

10.1. Construction/creation

VESSEL

Constructor. Creates a vessel from a vessel handle.

Synopsis:

```
VESSEL (OBJHANDLE hVessel, int flightmodel)
```

Parameters:

hVessel vessel handle
flightmodel level of realism requested. (0=simple, 1=realistic)

Create

Creates a new vessel.

Synopsis:

```
static OBJHANDLE Create (  
    const char *name,  
    const char *classname)
```

Parameters:

name vessel name
classname vessel class name

Return value

handle of the newly created vessel

Notes:

- This is a static method, so it can be called as `VESSEL::Create(...)` without an existing instance of class `VESSEL`.
- A configuration file for the specified vessel class must exist in the Config subdirectory.
- Note the difference between this method and the `VESSEL` constructor: `Create` creates a new *object* within Orbiter, *not* a `VESSEL` interface to an existing object. If the new vessel's class references a module, this module will likely create its own `VESSEL` interface.

GetHandle

Returns the Vessel handle.

Synopsis:

```
const OBJHANDLE GetHandle (void) const
```

Return value:

vessel handle, as passed to the constructor.

Notes:

- The handle is useful for various API function calls.

10.2. Vessel parameters and capabilities

GetName

Returns the vessel's name.

Synopsis:

```
char *GetName (void) const
```

Return value:

Pointer to vessel's name.

GetClassName

Returns the vessel's class name.

Synopsis:

```
char *GetClassName (void) const
```

Return value:

Pointer to vessel's class name.

GetFlightModel

Returns the requested realism level for the flight model.

Synopsis:

```
int GetFlightModel (void) const
```

Return value:

Realism level. These values are currently supported:
0 = simple
1 = realistic

GetSize

Returns the vessel's mean radius.

Synopsis:

```
double GetSize (void) const
```

Return value:

Vessel mean radius [m].

GetEmptyMass

Returns vessel's empty mass excluding fuel and payload. Equivalent to the oapiGetEmptyMass API function.

Synopsis:

```
double GetEmptyMass (void) const
```

Return value:

Vessel empty mass [kg].

GetMaxFuelMass

Returns vessel's maximum fuel capacity. Equivalent to the oapiGetMaxFuelMass API function.

Synopsis:

```
double GetMaxFuelMass (void) const
```

Return value:

Vessel maximum fuel mass [kg].

GetISP

Returns vessel's fuel-specific impulse.

Synopsis:

```
double GetISP (void) const
```

Return value:

Fuel-specific impulse [m/s]. This is the amount of thrust [N] obtained by burning 1kg of fuel per second.

GetMaxThrust

Returns maximum thrust rating [N] for one of the vessel's engine groups, defined by *eng*.

Synopsis:

```
double GetMaxThrust (ENGINE_TYPE eng) const
```

Parameters:

eng engine group identifier

Return value:

Maximum thrust rating [N]

GetMainThrustModPtr

Returns the address of a variable which can be used to define custom main/retro thrust [Newton] outside the control of the Orbiter core. Orbiter includes this value in the acceleration calculation, but the module is responsible for adjusting the corresponding physical parameters, like mass changes due to fuel consumption. Negative values define retro thrust.

Synopsis:

```
double *GetMainThrustModPtr (void) const
```

Return value:

Pointer to a variable which the module can set to alter orbiter's main/retro thrust setting. The value of this variable is added to the result of the standard thrust calculation.

GetCOG_elev

Returns the altitude of the vessel's centre of gravity over ground level when landed [m].

Synopsis:

```
double GetCOG_elev (void) const
```

Return value:

elevation of vessel's centre of mass [m].

GetCrossSections

Returns the vessel's cross sections projected in the direction of the vessel's principal axes [m²]

Synopsis:

```
void GetCrossSections (VECTOR3 &cs) const
```

Parameters:

cs vector receiving the cross sections of the vessel's projection into the y-z, x-z, and x-y planes, respectively [m²]

GetCW

Returns the vessel's wind resistance coefficients in the principal directions [dimensionless].

Synopsis:

```
void GetCW (
    double &cw_z_pos,
    double &cw_z_neg,
    double &cw_x,
    double &cw_y) const
```

Parameters:

cw_z_pos resistance in positive z direction (forward)
cw_z_neg resistance in negative z direction (back)
cw_x resistance in lateral direction
cw_y resistance in vertical direction

Notes:

- The first value (cw_z_pos) is the coefficient used if the vessel's airspeed z-component is positive (vessel moving forward). The second value is used if the z-component is negative (vessel moving backward).
- Lateral and vertical components are assumed symmetric.

GetWingAspect

Returns the vessel's wing aspect ratio (wingspan² / wing area). Vessels without wing-type airfoils return 0.

Synopsis:

```
double GetWingAspect (void) const
```

Return value:

Wing aspect ratio (wingspan² / wing area)

GetWingEffectiveness

Returns wing form factor: ~3.1 for elliptic wings, ~2.8 for tapered wings, ~2.5 for rectangular wings.

Synopsis:

```
double GetWingEffectiveness (void) const
```

Return value:

Wing form factor.

Notes:

- This form factor describes the wing's effectiveness in producing lift in an atmosphere as a function of its shape.

GetRotDrag

Returns the vessel's resistance $r_{x,y,z}$ against rotation around axes in atmosphere.

Synopsis:

```
void GetRotDrag (VECTOR3 &rd) const
```

Parameters:

rd rotational drag coefficient in the three coordinate axes of the vessel's frame of reference.

Notes:

- rd contains the components $r_{x,y,z}$ against rotation around axes in atmosphere, where angular deceleration due to atmospheric friction is $a^{(w)}_{x,y,z} = -v^{(w)}_{x,y,z} \rho$ $r_{x,y,z}$ with angular velocity $v^{(w)}$ and atmospheric density ρ .

GetPMI

Returns principal moments of inertia, mass-normalised [m^2]

Synopsis:

```
void GetPMI (VECTOR3 &pmi) const
```

Parameters:

pmi Diagonal elements of the inertia tensor

Notes:

For the meaning of the pmi vector, see SetPMI.

GetCameraOffset

Returns the camera position for internal (cockpit) view.

Synopsis:

```
void GetCameraOffset (VECTOR3 &ofs) const
```

Parameters:

ofs camera offset in the vessel's local frame of reference [m,m,m]

SetSize

Sets the vessel's mean radius [m].

Synopsis:

```
void SetSize (double size) const
```

Parameters:

size vessel mean radius [m]

Notes:

- This value is used for visibility calculations, but normally has no influence on the actual visual representation of the object (which is defined by the mesh) unless the module performs mesh scaling operations.

SetEmptyMass

Sets the vessel's empty mass excluding fuel and payload. Equivalent to the `oapiSetEmptyMass` API function.

Synopsis:

```
void SetEmptyMass (double m) const
```

Parameters:

<code>m</code>	vessel empty mass [kg]
----------------	------------------------

SetMaxFuelMass

Sets the vessel's maximum fuel capacity.

Synopsis:

```
void SetMaxFuelMass (double m) const
```

Parameters:

<code>m</code>	Maximum fuel mass [kg].
----------------	-------------------------

SetISP

Sets the vessel's fuel-specific impulse.

Synopsis:

```
void SetISP (double isp) const
```

Parameters:

<code>isp</code>	fuel-specific impulse [m/s].
------------------	------------------------------

Notes:

- The ISP defines the amount of thrust [N] obtained by burning 1 kg of fuel per second.

SetMaxThrust

Sets the maximum thrust rating for engine group *eng* to *th* [N].

Synopsis:

```
void SetMaxThrust (ENGINE_TYPE eng, double th) const
```

Parameters:

<code>eng</code>	engine group identifier
<code>th</code>	maximum thrust rating [N]

SetEngineLevel

Sets the thrust level for an engine group.

Synopsis:

```
void SetEngineLevel (ENGINE_TYPE eng, double level) const
```

Parameters:

<code>eng</code>	engine group identifier
<code>level</code>	thrust level (0..1)

Notes:

- Main engine level $-x$ is equivalent to retro engine level $+x$ and vice versa.

SetCOG_elev

Sets the altitude of the vessel's centre of gravity over ground level when landed [m].

Synopsis:

```
void SetCOG_elev (double h) const
```

Parameters:

h	elevation of the vessel's centre of gravity above the surface plane when landed [m].
---	--

Notes:

- This function is obsolete and has been replaced by SetTouchdownPoints.

SetTouchdownPoints

This defines 3 surface contact points for ground contact calculations (e.g. the points where the landing gear touches the ground).

Synopsis:

```
void SetTouchdownPoints (  
    const VECTOR3 &pt1,  
    const VECTOR3 &pt2,  
    const VECTOR3 &pt4) const
```

Parameters:

pt1	touchdown point of nose wheel (or equivalent)
pt2	touchdown point of left wheel (or equivalent)
pt3	touchdown point of right wheel (or equivalent)

Notes:

- The points are the positions at which the vessel's undercarriage (or equivalent) touches the surface, specified in local vessel coordinates.
- The points should be specified such that the cross product $\text{pt3}-\text{pt1} \times \text{pt2}-\text{pt1}$ defines the horizon UP direction for the landed vessel (given a left-handed coordinate system).

SetCrossSections

Sets the vessel's cross sections projected in the direction of the vessel's principal axes [m²].

Synopsis:

```
void SetCrossSections (const VECTOR3 &cs) const
```

Parameters:

cs	vector of cross sections of the vessel's projection into the y-z, x-z, and x-y planes, respectively [m ²]
----	---

SetCW

Sets the vessel's wind resistance coefficients along the local reference axes [dimensionless].

Synopsis:

```
void SetCW (  
    double cw_z_pos,  
    double cw_z_neg,  
    double cw_x,  
    double cw_y) const
```

Parameters:

cw_z_pos	resistance in positive z direction (forward)
cw_z_neg	resistance in negative z direction (back)
cw_x	resistance in lateral direction
cw_y	resistance in vertical direction

Notes:

- The first value (cw_z_pos) is the coefficient used if the vessel's airspeed z-component is positive (vessel moving forward). The second value is used if the z-component is negative (vessel moving backward).
- Lateral and vertical components are assumed symmetric.

void SetWingAspect (double) const

Sets the wing aspect ratio (wingspan² / wing area). Used for atmospheric drag calculation. Only vessels with wing-type airfoils should call this function.

void SetWingEffectiveness (double) const

Sets the wing form factor: ~3.1 for elliptic wings, ~2.8 for tapered wings, ~2.5 for rectangular wings. Used for lift and drag calculation.

void SetRotDrag (const VECTOR3&) const

Sets the vessel's resistance against rotation around axes in atmosphere.

void SetPitchMomentScale (double scale) const

Sets the magnitude of the moment acting on the vessel's pitch angle which rotates the vessel's longitudinal direction towards the airspeed vector.

void SetBankMomentScale (double scale) const

Sets the magnitude of the moment acting on the vessel's bank angle which rotates the vessel's longitudinal direction towards the airspeed vector.

SetPMI

Sets principal moments of inertia, mass-normalised [m²].

Synopsis:

```
void SetPMI (const VECTOR3 &pmi) const
```

Parameters:

pmi Principal moments of inertia

Notes:

- The principal moments are the diagonal elements of the inertia tensor in a frame of reference where the off-diagonal elements are zero.
- The elements of pmi should be calculated as follows:

$$pmi_1 = \frac{1}{M} \int \rho(r)(r_y^2 + r_z^2) dr$$

$$pmi_2 = \frac{1}{M} \int \rho(r)(r_x^2 + r_z^2) dr$$

$$pmi_3 = \frac{1}{M} \int \rho(r)(r_x^2 + r_y^2) dr$$

where M is the total vessel mass, ρ is the density, and the integration is performed over the vessel volume. The reference frame is chosen so that the off-diagonal elements of the tensor vanish.

- The `shippedit` utility allows to calculate the inertia tensor from a mesh, assuming a homogeneous mass distribution.

void SetTrimScale (double) const

Sets the max. magnitude of the pitch trim control.

Synopsis:

```
void SetTrimScale (double scale) const
```

Parameters:

scale pitch trim scaling factor

Notes:

- If scale is set to zero (default) the vessel does not have a pitch trim control.

SetCameraOffset

Sets the camera position for internal (cockpit) view.

Synopsis:

```
void SetCameraOffset (const VECTOR3 &ofs) const
```

Parameters:

ofs camera offset in the vessel's local frame of reference [m,m,m]

Notes:

- Currently the camera direction in cockpit view is always the vessel's local +z axis (forward).

SetLiftCoeffFunc

Installs callback function for calculation of lift coefficient as a function of angle of attack.

Synopsis:

```
void SetLiftCoeffFunc (LiftCoeffFunc lcf) const
```

Parameters:

lcf callback function pointer with the following interface:
double LiftCoeff (double aoa)

Notes:

- The callback function must be able to deal with aoa values in the range $-\pi \dots \pi$.
- If the function is not installed, the vessel is assumed not to produce any lift.

ParseScenarioLine

Processes an input line from a scenario file.

Synopsis:

```
void ParseScenarioLine (
    char *line,
    VESSELSTATUS *status) const
```

Parameters:

line line to be interpreted
status status parameter set

Notes:

- Normally, this function will be called from within the body of ovLoadState to allow Orbiter to process any generic status parameters which are not processed by the module.

10.3. Current vessel status

GetStatus

Returns vessel's current status parameters.

Synopsis:

```
void GetStatus (VESSELSTATUS &status) const
```

Parameters:

status struct receiving current vessel status

Notes:

- For a definition of VESSELSTATUS see Section 8.

DefSetState

Calls the default Orbiter vessel state initialisation with the specified status.

Synopsis:

```
void DefSetState (const VESSELSTATUS *status) const
```

Parameters:

status vessel status parameters.

Notes:

- This function is most commonly used in ovcSetState to enable default state initialisation.

SaveDefaultState

Causes Orbiter to write default vessel parameters to a scenario file.

Synopsis:

```
void SaveDefaultState (FILEHANDLE scn) const
```

Parameters:

scn scenario file handle

Notes:

- This method should normally only be invoked from within ovcSaveState, to allow Orbiter to save its default vessel status parameters.
- If ovcSaveState is implemented but does not call SaveDefaultState, no default parameters are written to the scenario.

GetMass

Returns current (total) vessel mass. Equivalent to the oapiGetMass API function.

Synopsis:

```
double GetMass (void) const
```

Return value:

Current vessel mass [kg].

GetFuelMass

Returns the vessel's current fuel mass.

Synopsis:

```
double GetFuelMass (void) const
```

Return value:

Current fuel mass [kg]

GetEngineLevel

Returns the thrust level for an engine group.

Synopsis:

```
double GetEngineLevel (ENGINEATYPE eng) const
```

Parameters:

eng engine group identifier

Return value:

thrust level (0..1)

Notes:

- For main engines, this does not include externally defined, module-controlled thrusters
- This function does not work for attitude thrusters.

SetFuelMass

Sets the vessel's current fuel mass [kg].

Synopsis:

```
void SetFuelMass (double m) const
```

Parameters:

m Current fuel mass [kg].

Notes:

- m must be between 0 and MaxFuelMass.
- This should NOT be used to implement normal fuel consumption by thrusters registered with Orbiter, since the Orbiter core takes care of this. Instead, this should be used for component modifications (e.g. stage separation), external thrust calculations, refuelling, etc.

10.4. State vectors

GetGlobalPos

Returns vessel's current position in the global reference frame.

Synopsis:

```
void GetGlobalPos (VECTOR3 &pos) const
```

Parameters:

pos: vector receiving position

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.0.
- Units are meters.
- Equivalent to `oapiGetGlobalPos(GetHandle(), &pos)`

GetGlobalVel

Returns vessel's current velocity in the global reference frame.

Synopsis:

```
void GetGlobalVel (VECTOR3 &vel) const
```

Parameters:

vel vector receiving velocity

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.0.
- Units are meters/second.
- Equivalent to `oapiGetGlobalVel (GetHandle(), &vel)`

GetRelativePos

Returns vessel's current position with respect to another object.

Synopsis:


```
void GetRelativePos (OBJHANDLE hRef, VECTOR3 &pos) const
```

Parameters:

hRef	reference object handle
pos	vector receiving position

Notes:

- Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.
- Equivalent to `oapiGetRelativePos (GetHandle(), hRef, &pos)`

GetRelativeVel

Returns vessel's current velocity relative to another object.

Synopsis:

```
void GetRelativeVel (OBJHANDLE hRef, VECTOR3 &pos) const
```

Parameters:

hRef	reference object handle
vel	vector receiving relative velocity

Notes:

- Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.
- Equivalent to `oapiGetRelativeVel (GetHandle(), hRef, &vel)`

GetEquPos

Returns vessel's current equatorial position (longitude, latitude and radius) with respect to the closest planet or moon.

Synopsis:

```
OBJHANDLE GetEquPos (
    double &longitude,
    double &latitude,
    double &radius) const
```

Parameters:

longitude	variable receiving longitude value [rad]
latitude	variable receiving latitude value [rad]
radius	variable receiving radius value [m]

Return value:

Handle to reference body to which the parameters refer. NULL indicates failure (no reference body available).

10.5. Orbital elements

Note: Calculating elements from state vectors is expensive. If possible, avoid calling the functions in this group at each frame (e.g. inside `ovcTimestep`). On the other hand, once any function in this group has been called, calling other functions during the *same* time step is not expensive.

GetElements

Returns vessel's primary orbital elements w.r.t. dominant gravitational source.

Synopsis:

```
OBJHANDLE GetElements (ELEMENTS &el, double &mjd_ref) const
```

Parameters:

el	primary orbital elements (semi-major axis a , eccentricity e , inclination i , longitude of ascending node θ , longitude of periapsis ϖ , mean longitude at epoch L)
mjd_ref	reference epoch in MJD (Modified Julian Date) format

Return value:

Handle of reference object. NULL indicates failure (no elements available).

Notes:

- There are various ways to specify orbital elements. Note that here we use the *longitude* of the ascending node (not *anomaly* of the ascending node), and *longitude* of periapsis, and that the mean anomaly L refers to epoch (mjd_ref), not to date (so it should not change over time unless the orbit itself changes).

GetArgPer

Returns argument of periapsis.

Synopsis:

```
OBJHANDLE GetArgPer (double &arg) const
```

Parameters:

arg argument of periapsis for current orbit [rad]

Return value:

Handle of reference object. NULL indicates failure (no elements available)

GetSMi

Returns semi-minor axis.

Synopsis:

```
OBJHANDLE GetSMi (double &smi) const
```

Parameters:

smi semi-minor axis for current orbit [m]

Return value:

Handle of reference object. NULL indicates failure (no elements available)

GetApDist

Returns apoapsis distance.

Synopsis:

```
OBJHANDLE GetApDist (double &apdist) const
```

Parameters:

apdist apoapsis distance for current orbit [m]

Return value:

Handle of reference object. NULL indicates failure (no elements available)

GetPeDist

Returns periapsis distance.

Synopsis:

```
OBJHANDLE GetPeDist (double &pedist) const
```

Parameters:

pedist periapsis distance for current orbit [m]

Return value:

Handle of reference object. NULL indicates failure (no elements available)

10.6. Transformations

ShiftCentreOfMass

Register a shift in the centre of mass after a structural change (e.g. stage separation)

Synopsis:

```
void ShiftCentreOfMass (const VECTOR3 &shift)
```

Parameters:

shift CoM displacement vector.

Notes:

- This function should be called after a vessel has undergone a structural change which shifted the centre of mass, and which resulted in a change of the mesh component offsets of *-shift*. It will do two things:
 1. Translate the vessel's world reference point by +shift to compensate for the mesh offset shift.
 2. Drag the camera so that it centers at the new CoM (if in external mode tracking the concerned vessel).

GetRotationMatrix

Returns the vessel's current rotation matrix for transformations from the vessel's local frame of reference to the global (world) frame of reference.

Synopsis:

```
void GetRotationMatrix (MATRIX3 &R) const
```

Parameters:

R rotation matrix

Notes:

- To transform a point $\mathbf{r}_{\text{local}}$ from local vessel coordinates to a global point $\mathbf{r}_{\text{global}}$, the following formula is used:
$$\mathbf{r}_{\text{global}} = \mathbf{R} \mathbf{r}_{\text{local}} + \mathbf{p},$$
where \mathbf{p} is the vessel's global position.
- This transformation can be directly performed by a call to `Local2Global`.

GlobalRot

Performs a rotation of a direction from the local vessel frame to the global frame.

Synopsis:

```
void GlobalRot (  
    const VECTOR3 &rloc,  
    VECTOR3 &rrot) const
```

Parameters:

rloc point in local vessel coordinates (input)
rrot rotated point (output)

Notes:

- This function is equivalent to multiplying `rloc` with the rotation matrix returned by `GetRotationMatrix`.
- Should be used to transform *directions*. To transform *points*, use `Local2Global`, which additionally adds the vessel's global position to the rotated point.

Local2Global

Performs a transformation from local vessel to global coordinates.

Synopsis:

```
void Local2Global (
    const VECTOR3 &local,
    VECTOR3 &global) const
```

Parameters:

local	point in local vessel coordinates (input)
global	transformed point in global coordinates (output)

10.7. Atmospheric parameters

GetAtmPressure

Returns atmospheric pressure [Pascal] at current vessel position.

Synopsis:

```
double GetAtmPressure (void) const
```

Return value:

atmospheric pressure [Pa] at current vessel position.

Note:

- This function returns 0 if the vessel is outside all planetary atmospheric hulls, as defined by the planets' AtmAltLimit parameters.

GetAtmDensity

Returns atmospheric density [kg/m³] at current vessel position.

Synopsis:

```
double GetAtmDensity (void) const
```

Return value:

atmospheric density [kg/m³] at current vessel position.

Note:

- This function returns 0 if the vessel is outside all planetary atmospheric hulls, as defined by the planets' AtmAltLimit parameters.

10.8. Visual manipulation

ClearMeshes

Removes all previously declared meshes for the vessel's visual representation.

Synopsis:

```
void ClearMeshes () const
```

AddMesh

Adds a new mesh to the vessel's visual representation.

Synopsis:

```
void AddMesh (
    const char *meshname,
    const VECTOR3 *ofs=0) const
```

Parameters:

meshname mesh file name (without path and file extension) which must exist in the *Meshes* subdirectory.
ofs optional pointer to a displacement vector which describes the offset (in meter) of the mesh origin against the vessel origin.

AddExhaustRef

Adds a definition for rendering the exhaust from an engine.

Synopsis:

```
UINT AddExhaustRef (  
    EXHAUSTTYPE exh,  
    VECTOR3 &pos,  
    double lscale = -1.0,  
    double wscale = -1.0,  
    VECTOR3 *dir = 0) const
```

Parameters:

exh engine group identifier (main, retro, hover, custom)
pos exhaust reference position (in local vessel coordinates)
lscale longitudinal scaling factor
wscale transversal scaling factor
dir exhaust direction

Return value:

Exhaust identifier

Notes:

- The direction vector should be normalised to 1.
- For the standard engine types (main, retro, hover), default values are used for lscale, wscale and dir, if not supplied in the function call. Custom engines should always provide these values.
- For the standard engine types the scaling factors define the scaling for maximum thrust setting. For thrusts < 100% the scaling factors are adjusted accordingly. For custom engines, the scaling factors are used directly. The module is responsible for adjusting the scaling factors (via SetExhaustScales) to reflect changes in thrust.

DelExhaustRef

Deletes an exhaust render definition.

Synopsis:

```
void DelExhaustRef (EXHAUSTTYPE exh, WORD id) const
```

Parameters:

exh engine group identifier (main, retro, hover, custom)
id engine identifier, as returned by AddExhaustRef

ClearExhaustRefs

Deletes all exhaust render definitions.

Synopsis:

```
void ClearExhaustRefs (void)
```

SetExhaustScales

Sets the longitudinal and transversal scaling factors for exhaust rendering

Synopsis:

```
void SetExhaustScales (  
    EXHAUSTTYPE exh,
```

```
WORD id,
double lscale,
double wscale) const
```

Parameters:

exh	engine group identifier (main, retro, hover, custom)
id	engine identifier, as returned by AddExhaustRef
lscale	longitudinal scaling factor
wscale	transversal scaling factor

Notes:

- This function must be called for custom engines to reflect changes in thrust level. For standard engine types, this is done automatically.

AddAttExhaustRef

Adds an exhaust render definition for an attitude thruster.

Synopsis:

```
UINT AddAttExhaustRef (
    const VECTOR3 &pos,
    const VECTOR3 &dir,
    double wscale = 1.0,
    double lscale = 1.0) const
```

Parameters:

pos	exhaust reference position (in local vessel coordinates)
dir	exhaust direction (normalised)
wscale	exhaust render width scaling factor
lscale	exhaust render length scaling factor

Return value:

Attitude exhaust id.

Notes:

- This function only affects the exhaust rendering, not the physical parameters of the attitude engines.
- After creating an attitude thruster with AddAttExhaustRef, it must be assigned to one or more attitude modes with AddAttExhaustMode.

AddAttExhaustMode

Assign an attitude thruster to an attitude mode.

Synopsis:

```
void AddAttExhaustMode (
    UINT idx,
    ATTITUDEMODE mode,
    int axis,
    int dir) const
```

Parameters:

idx	attitude exhaust id, as returned by AddAttExhaustRef.
mode	ATTMODE_ROT or ATTMODE_LIN
axis	rotation/translation axis (0=x, 1=y, 2=z)
dir	rotation/translation direction (0 or 1)

Notes:

- An attitude thruster can be assigned to more than one mode (e.g. a rotational and a linear mode)
- Multiple attitude thrusters can be assigned to a single mode.
- The following attitude modes are available:

mode	axis	dir	used for
ATTMODE_ROT	0	0	pitch up
ATTMODE_ROT	0	1	pitch down
ATTMODE_ROT	1	0	yaw left
ATTMODE_ROT	1	1	yaw right
ATTMODE_ROT	2	0	roll right
ATTMODE_ROT	2	1	roll left
ATTMODE_LIN	0	0	move right
ATTMODE_LIN	0	1	move left
ATTMODE_LIN	1	0	move up
ATTMODE_LIN	1	1	move down
ATTMODE_LIN	2	0	move forward
ATTMODE_LIN	2	1	move back

ClearAttExhaustRefs

Deletes all attitude thruster exhaust render definitions.

Synopsis:

```
void ClearAttExhaustRefs (void) const
```

RegisterAnimation

Logs a request for calls to ovcAnimate, while the vessel's visual exists.

Synopsis:

```
void RegisterAnimation (void) const
```

Notes:

- This function allows to implement animation sequences in combination with the ovcAnimate callback function. After a call to RegisterAnimation, ovcAnimate is called at each time step, if the vessel's visual exists.
- Use UnregisterAnimation to stop further calls to ovcAnimate.
- Orbiter uses a reference counter to log animation requests. It calls ovcAnimate as long as counter > 0,
- If ovcAnimate is not implemented by the module, RegisterAnimation has no effect.

UnregisterAnimation

Unlogs an animation request.

Synopsis:

```
void UnregisterAnimation (void) const
```

Notes:

- This stops a request for animation callback calls from a previous RegisterAnimation.
- The call to UnregisterAnimation should not be placed in the body of ovcAnimate, since it may be lost if the vessel's visual doesn't exist.

11. Plugin callback function reference

This is a list of callback functions which Orbiter will call for all activated *plugin modules*. (i.e. DLLs in the Modules\Plugin subdirectory which were activated by the user via the Launchpad dialog). Plugin callback functions use an *opc* ("orbiter plugin callback") prefix.

opcDLLInit

Called after the DLL is loaded by Orbiter, before the simulation window is opened. DLLs are loaded either during the program start, or when the user activates a DLL in the *Modules* tab of the launchpad dialog.

Synopsis:

```
DLLCLBK void opcDLLInit (HINSTANCE hDLL)
```

Parameters:

hDLL DLL module handle

opcDLLExit

Called before the DLL is unloaded by Orbiter, after the simulation window has closed. DLLs are unloaded either when Orbiter exits, or when the user deactivates a DLL in the *Modules* tab of the launchpad dialog.

Synopsis:

```
DLLCLBK void opcDLLExit (HINSTANCE hDLL)
```

Parameters:

hDLL DLL module handle

opcOpenRenderWindow

Called after the simulation window has been opened. The DLL should use this function for initialisations which depend on the size of the render window. The size remains valid until the `opcCloseRenderWindow` method is called. Note that for windowed modes the width and height parameters may be smaller than the user-defined window size, to accomodate window borders and title line.

Synopsis:

```
DLLCLBK void opcOpenRenderWindow (
    HWND renderWnd,
    DWORD width,
    DWORD height,
    BOOL fullscreen)
```

Parameters:

renderWnd	render window handle
width	width of the render viewport (pixel)
height	height of the render viewport (pixel)
fullscreen	TRUE if a fullscreen video mode is used, FALSE for a windowed mode

opcCloseRenderWindow

Called after the simulation window has been closed.

Synopsis:

```
DLLCLBK void opcCloseRenderWindow (void)
```

opcTimestep

Called at each time step of the simulation. Note that this is not exactly the same as an animation frame, because rendering may continue during a simulation pause if the camera is movable during pause.

Synopsis:

```
DLLCKBK void opcTimestep (
    double SimT,
    double SimDT,
    double mjd)
```

Parameters:

SimT	elapsed simulation time since simulation start (seconds)
SimDT	time interval since last time step (seconds)
mjd	simulation universal time in MJD (modified Julian date) format.

opcFocusChanged

Called when input focus (keyboard and joystick control) is switched to a new vessel (for example as a result of a call to `oapiSetFocus`).

Synopsis:

```
DLLCLBK opcfocusChanged (  
    OBJHANDLE new_focus,  
    OBJHANDLE old_focus)
```

Parameters:

`new_focus` handle of vessel receiving the input focus
`old_focus` handle of vessel losing focus

Notes:

Currently only objects of type "vessel" can receive the input focus. This may change in future versions.

opcTimeAccChanged

Called when the simulation time acceleration factor changes.

Synopsis:

```
DLLCLBK void opcTimeAccChanged (  
    double nWarp,  
    double oWarp)
```

Parameters:

`nWarp` new time acceleration factor
`oWarp` old time acceleration factor

12. Vessel callback function reference

This is a list of callback functions for *vessel modules* (i.e. modules referenced by the Module entry in vessel class configuration files). Vessel callback functions use an *ovc* ("orbiter vessel callback") prefix.

ovcInit

Called during vessel creation. A vessel module *must* define this function in order to create an instance of the VESSEL interface or a derived class.

Synopsis:

```
DLLCLBK VESSEL *ovcInit (  
    OBJHANDLE hVessel,  
    int flightmodel)
```

Parameters:

`hVessel` handle identifying the newly created vessel.
`flightmodel` level of flight model realism (0=simple, 1=realistic)

Return value:

Module-generated instance of VESSEL or a derived class.

Notes:

A typical implementation will look like this:

```
class MyVessel: public VESSEL  
{  
    ...  
}  
  
DLLCLBK VESSEL *ovcInit (OBJHANDLE hVessel, int flightmodel)  
{  
    return new MyVessel(hVessel, flightmodel);  
}
```

ovcExit

Called before killing the vessel. Should be used for cleanup operations (memory deallocation etc.) and for deallocating the VESSEL interface.

Synopsis:

```
DLLCLBK void ovcExit (VESSEL *vessel)
```

Parameters:

vessel vessel interface

ovcSetClassCaps

Called during vessel initialisation. This allows the module to define vessel class capabilities, such as mass, size, aerodynamic specs, thruster ratings, etc.

Synopsis:

```
DLLCLBK void ovcSetClassCaps (  
    VESSEL *vessel,  
    FILEHANDLE cfg)
```

Parameters:

vessel vessel interface
cfg handle for the vessel class configuration file.

Notes:

- This function should only set general parameters (like maximum fuel mass), not the current state parameters for a specific ship (like current fuel mass).
- Generic parameters directly defined in the vessel class cfg file (e.g. *MaxFuel*) override values set in ovcSetClassCaps. This allows to manipulate values without need to recompile the module.
- The cfg file handle allows to read nonstandard parameters from the class file.

ovcSetState

Called at vessel creation to allow initialisation of the initial state.

Synopsis:

```
DLLCLBK void ovcSetState (  
    VESSEL *vessel,  
    const VESSELSTATUS *status)
```

Parameters:

vessel vessel interface
status vessel state parameters

Notes:

- This function is called after ovcSetClassCaps.
- If this function is not defined, Orbiter will perform default state initialisations.
- To perform Orbiter's default initialisation from within ovcSetState, call vessel->DefSetState (status)

ovcLoadState

Called when the vessel must read its initial status from a scenario file.

Synopsis:

```
DLLCLBK void ovcLoadState (  
    VESSEL *vessel,  
    FILEHANDLE scn,  
    VESSELSTATUS *def_vs)
```

Parameters:

vessel	vessel interface
scn	scenario file handle
def_vs	set of generic vessel parameters

Notes:

- This callback function is provided to allow the module to read nonstandard parameters from the scenario file.
- The function should define a loop which parses lines from the scenario file via `oapiReadScenario_nextline`.
- Any lines which the module parser does not recognise should be forwarded to Orbiter's default scenario parser via `VESSEL::ParseScenarioLine`, to allow the processing of generic options.
- Alternatively, the module parser may intercept generic parameters and directly write values into the generic set `def_vs` (dangerous!)
- A typical parser may look like this:

```

DLLCLBK void ovcLoadState (VESSEL *vessel, FILEHANDLE scn,
    VESSELSTATUS *def_vs)
{
    char *line;
    int my_value;

    while (oapiReadScenario_nextline (scn, line)) {
        if (!strnicmp (line, "my_option", 9)) {
            sscanf (line+9, "%d", &my_value);
        } else if (...) { // more items
            ...
        } else { // anything not recognised is passed to Orbiter
            vessel->ParseScenarioLine (line, def_vs);
        }
    }
}

```

ovcSaveState

Called when a vessel needs to save its current status to a scenario file.

Synopsis:

```

DLLCLBK void ovcSaveState (
    VESSEL *vessel,
    FILEHANDLE scn)

```

Parameters:

vessel	vessel interface
scn	scenario file handle

Notes:

- This function only needs to be implemented if the vessel must save nonstandard parameters. Otherwise Orbiter invokes a default parameter save.
- To allow Orbiter to save its default vessel parameters, use `VESSEL::SaveDefaultState`.
- To write custom parameters to the scenario file, use the `oapiWriteLine` method.

ovcVisualCreated

Called after a the visual representation of a vessel has been created.

Synopsis:

```

DLLCLBK void ovcVisualCreated (
    VESSEL *vessel,
    VISHANDLE vis,
    int refcount)

```

Parameters:

vessel	vessel interface
vis	handle for the newly created visual
refcount	visual reference count

Notes:

- The logical interface to a vessel exists as long as the vessel is present in the simulation. However, the visual interface exists only when the vessel is within visual range of the camera. Orbiter creates and destroys visuals as required. This enhances simulation performance in the presence of a large number of objects in the simulation.
- Whenever Orbiter creates a vessel's visual it reverts to its initial configuration (e.g. as defined in the mesh file). The module can use this function to update the visual to the current state, wherever dynamic changes are required.
- More than one visual representation of an object may exist. The refcount parameter defines how many visual interfaces to the object exist.

ovcVisualDestroyed

Called before the visual representation of a vessel is destroyed.

Synopsis:

```
DLLCLBK void ovcVisualDestroyed (
    VESSEL *vessel,
    VISHANDLE vis,
    int refcount)
```

Parameters:

vessel	vessel interface
vis	handle for the visual to be destroyed
refcount	visual reference count

Notes:

- Orbiter calls this function before it destroys the vessel's visual representation, e.g. when it moves out of the visual range of the current camera.
- The (logical) vessel may still exist, but it is no longer rendered.

ovcTimestep

Called at each simulation time step *before* the vessel updates its position and velocity.

Synopsis:

```
DLLCLBK void ovcTimestep (VESSEL *vessel, double simt)
```

Parameters:

vessel	vessel interface
simt	simulation up time (seconds since simulation start)

Notes:

- This function, if implemented, is called at each frame for each instance of this vessel class, and is therefore time-critical. Avoid any unnecessary calculations here which may degrade performance.

ovcAnimate

Called at each simulation time step if the module has registered an animation request and if the vessel's visual exists.

Synopsis:

```
DLLCLBK void ovcAnimate (VESSEL *vessel, double simt)
```

Parameters:

vessel	vesse interface
--------	-----------------

simt simulation up time (seconds since simulation start)

Notes:

- This callback allows the module to animate the vessel's visual representation (moving undercarriage, cargo bay doors, etc.)
- It is only called as long as the vessel has registered an animation (between matching VESSEL::RegisterAnimation and VESSEL::UnregisterAnimation calls) and if the vessel's visual exists.
- The UnregisterAnimation call should not be placed within the body of ovcAnimate, since it would be lost if the vessel's visual doesn't exist. This should rather be placed in ovcTimestep.

ovcConsumeKey

Keyboard handler. Called at each simulation time step. This callback function allows the installation of a custom keyboard interface for the vessel.

Synopsis:

```
DLLCLBK int ovcConsumeKey (  
    VESSEL *vessel,  
    const char *keystate)
```

Parameters:

vessel	vessel interface
keystate	keyboard state

Return value:

The function should return a nonzero value if it processes a key, and zero otherwise.

Notes:

- The keystate contains the current keyboard state. Use the KEYDOWN macro in combination with the key identifiers as defined in orbitersdk.h (OAPI_KEY_XXX) to check for particular keys being pressed. Example:

```
if (KEYDOWN (keystate, OAPI_KEY_F10))  
    if (oapiAcceptDelayedKey (OAPI_KEY_F10, 1.0))  
    { ... }
```
- For keys to be used as switches the oapiAcceptDelayedKey function is useful, which returns false if the same key was consumed within a given time interval and no other key was consumed since then.

13. Planet callback function reference

This is a list of callback functions Orbiter will call for all *planet modules* (i.e. modules referenced by the *Module* entry in the configuration files of planets or moons). See also the Vsop87 entry in the “Standard Orbiter modules” section below.

In the following <Planet> is a placeholder for the planet's or moon's name as defined in its configuration file (case-sensitive!)

<Planet>_SetPrecision

Define the relative error for the calculations for <Planet>.

Synopsis:

```
DLLCLBK int <Planet>_SetPrecision (double prec)
```

Parameters:

prec	module-specific
------	-----------------

Return value:

not used by Orbiter

Notes:

- Orbiter calls this function at the start of each simulation with the value of the *ErrorLimit* entry of the planet's configuration file. The module can use this to set its calculation precision.
- If the *ErrorLimit* entry is not defined in the cfg file, then `<Planet>_SetPrecision` will not be called, so the module should initialise some default precision.
- It is up to the module how to interpret the passed precision value, but by convention *prec* should specify the relative error for position and velocity calculations.
- This function is optional. If the module doesn't define it, Orbiter will ignore the *ErrorLimit* entry in the cfg file.

<Planet>_EclSphData

Calculate ecliptic positions and velocities in spherical coordinates. Reference frame is ecliptic and equinox of J2000. For planets (i.e. objects defined as "Planet" in the solar system cfg file) heliocentric coordinates should be calculated. For moons (i.e. objects defined as "Moon" in the solar system cfg file) coordinates w.r.t. the moon's reference planet should be calculated, e.g. geocentric for Earth's moon.

Synopsis:

```
DLLCLBK int <Planet>_EclSphData (double mjd, double *ret)
```

Parameters:

mjd	date in MJD format (MJD = JD-2400000.5)
ret	array of results which the function should calculate as follows: ret[0] = longitude [rad] ret[1] = latitude [rad] ret[2] = radius [AU] ret[3] = velocity in longitude [rad/s] ret[4] = velocity in latitude [rad/s] ret[5] = radial velocity [AU/s]

Return value:

Error code (not used)

Notes:

- The function should calculate the values for *ret* in the J2000 ecliptic frame, but Orbiter's precision requirements are not very high, so the ecliptic of a different epoch (or the ecliptic of date) is probably ok.
- Orbiter only calls this function directly to calculate positions at times other than the current simulation time (e.g. for trajectory predictions). Otherwise it calls `<Planet>_CurrentData` (see below).

<Planet>_CurrentData

This function is called by Orbiter at each frame to update planet positions and velocities. Therefore the implementation can make use of interpolation methods to increase the efficiency of the calculation.

Synopsis:

```
DLLCLBK int <Planet>_CurrentData (  
    double simt,  
    double *ret)
```

Parameters:

simt	Time (in seconds) since simulation start
ret	results (as in <code><Planet>_EclSphData</code>)

Return value:

not used

Notes:

- Orbiter passes simt (simulation time in seconds) rather than mjd to this function to allow more precise calculation of the interpolation point.
- The simplest way to implement this function is as

```
return <Planet>_EclSphData (oapiTime2MJD (simt), ret);
```

However this is not recommended. Instead the function should sample the planet data in appropriate intervals and use an interpolation scheme to calculate the data for a given time. This is more efficient and helps smoothing rounding errors in the full updates.
- This function is called at every frame by Orbiter and is therefore extremely time-critical. As a performance target, the execution of this function for *all* planets should take < 10 milliseconds on a low-end machine.
- The sampling times for full position calculations should be staggered for different planets, so that not all full updates occur at the same frame.

14. API function reference

This is the reference list for the Orbiter API functions which can be used by modules to obtain and set simulation parameters from the Orbiter kernel. See index for alphabetical listing.

14.1. Obtaining object handles

oapiGetObjectByName

Retrieve the handle for an object from its name. Objects may be vessels, orbital stations, planets, moons or suns. The handle remains valid until the object is deleted or the simulation terminates.

Synopsis:

```
OBJHANDLE oapiGetObjectByName (char *name)
```

Parameters:

name object name (not case-sensitive)

Return value:

object handle. (NULL indicates that the object does not exist)

oapiGetObjectByIndex

Retrieve the handle for an object from its index. This is useful to construct loops over a series of objects. The handle remains valid until the object is deleted or the simulation terminates.

Synopsis:

```
OBJHANDLE oapiGetObjectByIndex (int index)
```

Parameters:

index object index (≥ 0)

Return value:

object handle. (NULL indicates that the object does not exist)

Notes:

$0 \leq \text{index} < \text{oapiGetObjectCount}()$ is required. The function does not perform a range check!

oapiGetObjectCount

Returns the number of objects currently present in the simulation.

Synopsis:

```
DWORD oapiGetObjectCount (void)
```

Return value:
object count

oapiGetVesselByName

Retrieve the handle for a vessel from its name. The handle remains valid until the object is deleted or the simulation terminates.

Synopsis:
`OBJHANDLE oapiGetVesselByName (char *name)`

Parameters:
name object name (not case-sensitive)

Return value:
object handle. (NULL indicates that the vessel does not exist)

oapiGetVesselByIndex

Retrieve the handle for a vessel from its index. This is useful to construct loops over a series of vessels. The handle remains valid until the object is deleted or the simulation terminates.

Synopsis:
`OBJHANDLE oapiGetVesselByIndex (int index)`

Parameters:
index object index (≥ 0)

Return value:
vessel handle. (NULL indicates that the vessel does not exist)

Notes:
 $0 \leq \text{index} < \text{oapiGetVesselCount}()$ is required. The function does not perform a range check!

oapiGetVesselCount

Returns the number of vessels currently present in the simulation.

Synopsis:
`DWORD oapiGetVesselCount (void)`

Return value:
vessel count

oapiGetStationByName

Retrieves the handle of an orbital station from its name.

Synopsis:
`OBJHANDLE oapiGetStationByName (char *name)`

Parameters:
name station name (not case-sensitive)

Return value:
object handle. (NULL indicates that the station does not exist)

oapiGetStationByIndex

Retrieves the handle of an orbital station from its list index.

Synopsis:


```
OBJHANDLE oapiGetStationByIndex (int index)
```

Parameters:

index object index (≥ 0)

Return value:

object handle. (NULL indicates that the station does not exist)

oapiGetStationCount

Returns the number of stations currently in the simulation.

Synopsis:

```
DWORD oapiGetStationCount (void)
```

Return value:

station count

oapiGetGbodyByName

Retrieves the handle of a “massive” object (a gravitational field source: sun, planet or moon) from its name.

Synopsis:

```
OBJHANDLE oapiGetGbodyByName (char *name)
```

Parameters:

name object name (not case-sensitive)

Return value:

object handle. (NULL indicates that the object does not exist)

oapiGetGbodyByIndex

Retrieves the handle of a massive object from its list index.

Synopsis:

```
OBJHANDLE oapiGetGbodyByIndex (int index)
```

Parameters:

index object index (≥ 0)

Return value:

object handle. (NULL indicates that the object does not exist)

oapiGetGbodyCount

Returns the number of massive objects (suns, planets and moons) currently in the simulation.

Synopsis:

```
DWORD oapiGetGbodyCount ()
```

Return value:

Number of objects

oapiGetObjectName

Returns the name of an object.

Synopsis:

```
void oapiGetObjectName (  
    OBJHANDLE hObj,  
    char *name,  
    int n)
```

Parameters:

hObj	object handle
name	pointer to character array to receive object name
n	length of string buffer

Notes:

name must be allocated to at least size *n* by the calling function.
If the string buffer is not long enough to hold the object name, the name is truncated.

oapiGetFocusObject

Retrieve the handle for the current focus object. The focus object is the user-controlled vessel which receives keyboard and joystick input.

Synopsis:

```
OBJHANDLE oapiGetFocusObject (void)
```

Return value:

focus object handle. This is guaranteed to exist during the simulation (between *opcOpenRenderViewport* and *opcCloseRenderViewport*)

Notes:

Currently the focus object is guaranteed to be a vessel. This may change in future versions.

oapiSetFocusObject

Switches the input focus to a different vessel object.

Synopsis:

```
OBJHANDLE oapiSetFocus (OBJHANDLE hVessel)
```

Parameters:

hVessel	handle of vessel to receive the focus
---------	---------------------------------------

Return value:

handle of vessel losing focus, or NULL if focus did not change

Notes:

hVessel must refer to a vessel object. Trying to set the focus to a different object type (e.g. orbital station) will fail.

14.2. Object mass functions

oapiGetMass

Returns the mass [kg] of an object. For vessels, this is the total mass, including current fuel and payload mass.

Synopsis:

```
double oapiGetMass (OBJHANDLE hObj)
```

Parameters:

hObj	object handle
------	---------------

Return value:

object mass [kg]

oapiGetEmptyMass

Returns empty mass of a vessel, excluding fuel and payload.

Synopsis:

```
double oapiGetEmptyMass (OBJHANDLE hVessel)
```

Parameters:

hVessel vessel handle

Return value:

empty vessel mass [kg]

Notes:

- hVessel must be a vessel handle. Other object types are invalid.
- Do not rely on a constant empty mass. Structural changes (e.g. discarding a rocket stage) will affect the empty mass.
- For multistage configurations, the fuel mass of all currently inactive stages contributes to the empty mass. Only the fuel mass of active stages is excluded.

oapiGetFuelMass

Returns current fuel mass of a vessel.

Synopsis:

```
double oapiGetFuelMass (OBJHANDLE hVessel)
```

Parameters:

hVessel vessel handle

Return value:

Current fuel mass [kg]

Notes:

- hVessel must be a vessel handle. Other object types are invalid.
- For multistage configurations, this returns the current fuel mass of active stages only.

oapiGetMaxFuelMass

Returns maximum fuel mass of a vessel.

Synopsis:

```
double oapiGetMaxFuelMass (OBJHANDLE hVessel)
```

Parameters:

hVessel vessel handle

Return value:

Maximum fuel mass [kg]

Notes:

- hVessel must be a vessel handle. Other object types are invalid.
- For multistage configurations, this returns the sum of the max fuel mass of active stages only.

oapiSetEmptyMass

Set the empty mass of a vessel (excluding fuel and payload)

Synopsis:

```
void oapiSetEmptyMass (OBJHANDLE hVessel, double mass)
```

Parameters:

hVessel vessel handle

mass empty mass [kg]

Notes:

- Use this function to register structural mass changes, for example as a result of jettisoning a fuel tank, etc.

14.3. Object state vectors

oapiGetGlobalPos

Returns the position of an object in the global reference frame.

Synopsis:

```
void oapiGetGlobalPos (OBJHANDLE hObj, VECTOR3 *pos)
```

Parameters:

hObj	object handle
pos	pointer to vector receiving coordinates

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.
- Units are meters.

oapiGetGlobalVel

Returns the velocity of an object in the global reference frame.

Synopsis:

```
void oapiGetGlobalVel (OBJHANDLE hObj, VECTOR3 *vel)
```

Parameters:

hObj	object handle
vel	pointer to vector receiving velocity data

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.
- Units are meters/second.

oapiGetFocusGlobalPos

Returns the position of the current focus object in the global reference frame.

Synopsis:

```
void oapiGetFocusGlobalPos (VECTOR3 *pos)
```

Parameters:

pos	pointer to vector receiving coordinates
-----	---

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.0.
- Units are meters.

oapiGetFocusGlobalVel

Returns the velocity of the current focus object in the global reference frame.

Synopsis:

```
void oapiGetFocusGlobalVel (VECTOR3 *vel)
```

Parameters:

vel pointer to vector receiving velocity data

Notes:

- The global reference frame is the heliocentric ecliptic system at ecliptic and equinox of J2000.
- Units are meters/second.

oapiGetRelativePos

Returns the distance vector from hRef to hObj in the ecliptic reference frame.

Synopsis:

```
void oapiGetRelativePos (
    OBJHANDLE hObj,
    OBJHANDLE hRef,
    VECTOR3 *pos)
```

Parameters:

hObj	object handle
hRef	reference object handle
pos	pointer to vector receiving distance data

Notes:

Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.

oapiGetRelativeVel

Returns the velocity difference vector of hObj relative to hRef in the ecliptic reference frame.

Synopsis:

```
void oapiGetRelativeVel (
    OBJHANDLE hObj,
    OBJHANDLE hRef,
    VECTOR3 *vel)
```

Parameters:

hObj	object handle
hRef	reference object handle
vel	pointer to vector receiving velocity difference data

Notes:

Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.

oapiGetFocusRelativePos

Returns the distance vector from hRef to the current focus object.

Synopsis:

```
void oapiGetFocusRelativePos (OBJHANDLE hRef, VECTOR3 *pos)
```

Parameters:

hRef	reference object handle
pos	pointer to vector receiving distance data

Notes:

Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.

oapiGetFocusRelativeVel

Returns the velocity difference vector of the current focus object relative to hRef.

Synopsis:

```
void oapiGetFocusRelativeVel (OBJHANDLE hRef, VECTOR3 *vel)
```

Parameters:

hRef	reference object handle
vel	pointer to vector receiving velocity difference data

Notes:

Results are w.r.t. ecliptic frame at equinox and ecliptic of J2000.0.

14.4. Surface-relative parameters

oapiGetAltitude

Returns the altitude of a vessel over a planetary surface.

Synopsis:

```
BOOL oapiGetAltitude (OBJHANDLE hVessel, double *alt)
```

Parameters:

hVessel	vessel handle
alt	pointer to variable receiving altitude value

Return value:

Error flag (FALSE on failure)

Notes:

- Unit is meter [m]
- Returns altitude above *closest* planet.
- Altitude is measured above *mean* planet radius (as defined by SIZE parameter in planet's cfg file)
- The handle passed to the function must refer to a *vessel*.

oapiGetFocusAltitude

Returns the altitude of the current focus vessel over a planetary surface.

Synopsis:

```
BOOL oapiGetFocusAltitude (double *alt)
```

Parameters:

alt	pointer to variable receiving altitude value [m]
-----	--

Return value:

Error flag (FALSE on failure)

oapiGetPitch

Returns a vessel's pitch angle w.r.t. the local horizon.

Synopsis:

```
BOOL oapiGetPitch (OBJHANDLE hVessel, double *pitch)
```

Parameters:

hVessel	vessel handle
pitch	pointer to variable receiving pitch value

Return value:

Error flag (FALSE on failure)

Notes:

- Unit is radian [rad]
- Returns pitch angle w.r.t. closest planet

- The local horizon is the plane whose normal is defined by the distance vector from the planet centre to the vessel.
- The handle passed to the function must refer to a *vessel*.

oapiGetFocusPitch

Returns the pitch angle of the current focus vessel w.r.t. the local horizon.

Synopsis:

```
BOOL oapiGetFocusPitch (double *pitch)
```

Parameters:

pitch pointer to variable receiving pitch value

Return value:

Error flag (FALSE on failure)

oapiGetBank

Returns a vessel's bank angle w.r.t. the local horizon.

Synopsis:

```
BOOL oapiGetBank (OBJHANDLE hVessel, double *bank)
```

Parameters:

hVessel vessel handle
bank pointer to variable receiving bank value

Return value:

Error flag (FALSE on failure)

Notes:

- Unit is radian [rad]
- Returns bank angle w.r.t. closest planet
- The local horizon is the plane whose normal is defined by the distance vector from the planet centre to the vessel.
- The handle passed to the function must refer to a *vessel*.

oapiGetFocusBank

Returns the bank angle of the current focus vessel w.r.t. the local horizon.

Synopsis:

```
BOOL oapiGetFocusBank (double *bank)
```

Parameters:

bank pointer to variable receiving bank angle [rad]

Return value:

Error flag (FALSE on failure)

oapiGetHeading

Returns a vessel's heading (against geometric north) calculated for the local horizon plane.

Synopsis:

```
BOOL oapiGetHeading (OBJHANDLE hVessel, double *heading)
```

Parameters:

hVessel vessel handle
heading pointer to variable receiving heading value [rad]

Return value:

Error flag (FALSE on failure)

Notes:

- Unit is radian [rad] 0=north, $\pi/2$ =east, etc.
- The handle passed to the function must refer to a vessel.

oapiGetFocusHeading

Returns the heading (against geometric north) of the current focus vessel calculated for the local horizon plane.

Synopsis:

```
BOOL oapiGetFocusHeading (double *heading)
```

Parameters:

heading pointer to variable receiving heading value [rad]

Return value:

Error flag (FALSE on failure)

oapiGetEquPos

Returns a vessel's spherical equatorial coordinates (longitude, latitude and radius) with respect to the closest planet or moon.

Synopsis:

```
BOOL oapiGetEquPos (  
    OBJHANDLE hVessel,  
    double *longitude,  
    double *latitude,  
    double *radius)
```

Parameters:

hVessel vessel handle
longitude pointer to variable receiving longitude value [rad]
latitude pointer to variable receiving latitude value [rad]
radius pointer to variable receiving radius value [m]

Return value:

Error flag (FALSE on failure)

Notes:

- The handle passed to the function must refer to a vessel; stations are not supported at present.

oapiGetFocusEquPos

Returns the current focus vessel's spherical equatorial coordinates (longitude, latitude and radius) with respect to the closest planet or moon.

Synopsis:

```
BOOL oapiGetFocusEquPos (  
    double *longitude,  
    double *latitude,  
    double *radius)
```

Parameters:

longitude pointer to variable receiving longitude value [rad]
latitude pointer to variable receiving latitude value [rad]
radius pointer to variable receiving radius value [m]

Return value:

Error flag (FALSE on failure)

oapiGetAirspeed

Returns a vessel's airspeed w.r.t. the closest planet or moon.

Synopsis:

```
BOOL oapiGetAirspeed (OBJHANDLE hVessel, double *airspeed)
```

Parameters:

hVessel vessel handle
airspeed pointer to variable receiving airspeed value [m/s]

Return value

Error flag (FALSE on failure)

Notes:

- This function works even for planets or moons without atmosphere. It returns an "airspeed-equivalent" value.

oapiGetFocusAirspeed

Returns the current focus vessel's airspeed w.r.t. the closest planet or moon.

Synopsis:

```
BOOL oapiGetFocusAirspeed (double *airspeed)
```

Parameters:

airspeed pointer to variable receiving airspeed value [m/s]

Return value:

Error flag (FALSE on failure)

oapiGetAirspeedVector

Returns a vessel's airspeed vector w.r.t. the closest planet or moon in the local horizon's frame of reference.

Synopsis:

```
BOOL oapiGetAirspeedVector (  
    OBJHANDLE hVessel,  
    VECTOR3 *speedvec)
```

Parameters:

hVessel vessel handle
speedvec pointer to variable receiving airspeed vector [m/s in x,y,z]

Return value:

Error flag (FALSE on failure)

Notes:

- This function returns the airspeed vector with respect to the local horizon reference frame. To get the vector with respect to the local vessel coordinates, use oapiGetShipAirspeedVector.

oapiGetFocusAirspeedVector

Returns the current focus vessel's airspeed vector w.r.t. the closest planet or moon in the local horizon's frame of reference.

Synopsis:

```
BOOL oapiGetFocusAirspeedVector (VECTOR3 *speedvec)
```

Parameters:

speedvec pointer to variable receiving airspeed vector [m/s in x,y,z]

Return value:

Error flag (FALSE on failure)

oapiGetShipAirspeedVector

Returns a vessel's airspeed vector w.r.t. the closest planet or moon in the vessel's local frame of reference.

Synopsis:

```
BOOL oapiGetShipAirspeedVector (  
    OBJHANDLE hVessel,  
    VECTOR3 *speedvec)
```

Parameters:

hVessel vessel handle
speedvec pointer to variable receiving airspeed vector [m/s in x,y,z]

Return value:

Error flag (FALSE on failure)

Notes:

- This function returns the airspeed vector with respect to the vessel's frame of reference. To get the vector with respect to the local horizon's frame of reference, use oapiGetAirspeedVector.

oapiGetFocusShipAirspeedVector

Returns the current focus vessel's airspeed vector w.r.t. closest planet or moon in the vessel's local frame of reference.

Synopsis:

```
BOOL oapiGetFocusShipAirspeedVector (VECTOR3 *speedvec)
```

Parameters:

speedvec pointer to variable receiving airspeed vector [m/s in x,y,z]

Return value:

Error flag (FALSE on failure)

oapiGetAtmPressureDensity

Returns the atmospheric pressure and density caused by a planetary atmosphere at the current vessel position.

Synopsis:

```
void oapiGetAtmPressureDensity (  
    OBJHANDLE hVessel,  
    double *pressure,  
    double *density)
```

Parameters:

hVessel vessel handle
pressure pointer to variable receiving pressure value [Pa]
density pointer to variable receiving density value [kg/m³]

Notes:

- Pressure and density are calculated using an exponential barometric equation, without accounting for local variations.

oapiGetFocusAtmPressureDensity

Returns the atmospheric pressure and density caused by a planetary atmosphere at the current focus vessel's position.

Synopsis:

```
void oapiGetFocusAtmPressureDensity (  
    double *pressure,  
    double *density)
```

Parameters:

pressure	pointer to variable receiving pressure value [Pa]
density	pointer to variable receiving density value [kg/m ³]

14.5. Engine status

oapiGetEngineStatus

Retrieve the status of main, retro and hover thrusters for a vessel.

Synopsis:

```
void oapiGetEngineStatus (  
    OBJHANDLE hVessel,  
    ENGINESTATUS *es)
```

Parameters:

hVessel	vessel handle
es	pointer to an ENGINESTATUS structure which will receive the engine level parameters

Notes:

The main/retro engine level has a range of [-1,+1]. A positive value indicates engaged main/disengaged retro thrusters, a negative value indicates engaged retro/disengaged main thrusters. Main and retro thrusters cannot be engaged simultaneously. For vessels without retro thrusters the valid range is [0,+1]. The valid range for hover thrusters is [0,+1].

oapiGetFocusEngineStatus

Retrieve the engine status for the focus vessel.

Synopsis:

```
void oapiGetFocusEngineStatus (ENGINESTATUS *es)
```

Parameters:

es	pointer to an ENGINESTATUS structure which will receive the engine level parameters
----	---

Notes:

See oapiGetEngineStatus

oapiSetEngineLevel

Engage the specified engines.

Synopsis:

```
void oapiSetEngineLevel (  
    OBJHANDLE hVessel,  
    ENGINETYPE engine,  
    double level)
```

Parameters:

hVessel	vessel handle
engine	identifies the engine to be set
level	engine thrust level [0,1]

Notes:

- Not all vessels support all types of engines.
- Setting main thrusters >0 implies setting retro thrusters to 0 and vice versa.
- Setting main thrusters to -level is equivalent to setting retro thrusters to +level and vice versa.

oapiGetAttitudeMode

Returns a vessel's attitude thruster mode (rotational or linear)

Synopsis:

```
int oapiGetAttitudeMode (OBJHANDLE hVessel)
```

Parameters:

hVessel vessel handle

Return value:

Current attitude mode: 0=rotational, 1=linear, -1=error

Notes:

- The handle must refer to a vessel. This function does not support stations or other object types.
- Return value < 0 indicates an error, e.g. if the vessel does not have attitude thrusters.

oapiToggleAttitudeMode

Flip a vessel's attitude thruster mode between rotational and linear.

Synopsis:

```
int oapiToggleAttitudeMode (OBJHANDLE hVessel)
```

Parameters:

hVessel vessel handle

Return value:

The new attitude mode: 0=rotational, 1=linear, -1=error

Notes:

- The handle must refer to a vessel. This function does not support stations or other object types.
- Return value < 0 indicates an error, e.g. if the vessel does not have attitude thrusters, or if the mode can't be changed.

oapiSetAttitudeMode

Set a vessel's attitude thruster mode.

Synopsis:

```
BOOL oapiSetAttitudeMode (OBJHANDLE hVessel, int mode)
```

Parameters:

hVessel vessel handle
mode attitude mode: 0=rotational, 1=linear

Return value:

Error flag; FALSE indicates failure

Notes:

- The handle must refer to a vessel. This function does not support stations or other object types.

oapiGetFocusAttitudeMode

Returns the current focus vessel's attitude thruster mode (rotational or linear)

Synopsis:

```
int oapiGetFocusAttitudeMode ()
```

Return value:

Current attitude mode: 0=rotational, 1=linear, -1=error

Notes:

- Return value < 0 indicates an error, e.g. if the vessel does not have attitude thrusters.

oapiToggleFocusAttitudeMode

Flip the current focus vessel's attitude thruster mode between rotational and linear.

Synopsis:

```
int oapiToggleFocusAttitudeMode ()
```

Return value:

The new attitude mode: 0=rotational, 1=linear, -1=error

Notes:

- Return value < 0 indicates an error, e.g. if the vessel does not have attitude thrusters, or if the mode can't be changed.

oapiSetFocusAttitudeMode

Set the current focus vessel's attitude thruster mode.

Synopsis:

```
BOOL oapiSetFocusAttitudeMode (int mode)
```

Parameters:

mode attitude mode: 0=rotational, 1=linear

Return value:

Error flag; FALSE indicates failure

14.6. Simulation time

oapiGetSimTime

Retrieve time (in seconds) since simulation start.

Synopsis:

```
double oapiGetSimTime ()
```

Return value:

Simulation up time (seconds)

Notes:

Since the simulation up time depends on the simulation start time, this parameter is useful mainly for time differences. To get an absolute time parameter, use `oapiGetSimMJD`.

oapiGetSimStep

Retrieve length of last time step (from previous to current frame) in seconds.

Synopsis:

```
double oapiGetSimStep ()
```

Return value:

Step length (seconds)

Notes:

This parameter is useful for numerical (finite difference) calculation of time derivatives.

oapiGetSimMJD

Retrieve absolute time measure (Modified Julian Date) for current simulation state.

Synopsis:

```
double oapiGetSimMJD ( )
```

Return value:

Current Modified Julian Date (days)

Notes:

Orbiter defines the *Modified Julian Date* (MJD) as $JD - 240\,0000.5$, where JD is the *Julian Date*. JD is the interval of time in mean solar days elapsed since 4713 BC January 1 at Greenwich mean noon.

oapiTime2MJD

Convert a simulation up time value into a Modified Julian Date.

Synopsis:

```
double oapiTime2MJD (double simt)
```

Parameters:

simt simulation time (seconds)

Return value:

Modified Julian Date (MJD) corresponding to simt.

oapiGetTimeAcceleration

Returns simulation time acceleration factor.

Synopsis:

```
double oapiGetTimeAcceleration (void)
```

Return value:

time acceleration factor

Notes:

This function will not return 0 when the simulation is paused. Instead it will return the acceleration factor at which the simulation will resume when unpaused.

oapiSetTimeAcceleration

Set the simulation time acceleration factor

Synopsis:

```
void oapiSetTimeAcceleration (double warp)
```

Parameters:

warp new time acceleration factor

Notes:

Warp factors will be clamped to the valid range [1,1000]. If the new warp factor is different from the previous one, all DLLs (including the one that called oapiSetTimeAcceleration) will be sent a opcTimeAccChanged message.

14.7. Keyboard input

oapiAcceptDelayedKey

A helper function which allows to implement key delays.

Synopsis:

```
bool oapiAcceptDelayedKey (char key, double interval)
```

Parameters:

key	key identifier (OAPI_KEY_xxx), see orbitersdk.h for complete list.
interval	key delay (seconds)

Return value:

This function returns false if the last call to the function for 'key' occurred within 'interval' seconds, and if it was not called for another key in the mean time.

Notes:

- This function is useful for implementing a keyboard control which switches between discrete states, where a time delay should occur between state changes. By default, keyboard parsing is not delayed, so the system would rapidly loop through all states if the user held down the key too long.
- See `ovcConsumeKey` for an example.

14.8. MFD management

oapiOpenMFD

Set an MFD (multifunctional display) to a specific mode.

Synopsis:

```
void oapiOpenMFD (int mode, int pos)
```

Parameters:

mode	MFD mode id (see Section 9)
pos	MFD position (see Section 9)

Notes:

mode MFD_NONE will turn off the MFD.
User-defined modes and positions are not currently supported.

oapiGetMFDMode

Get the current mode of the specified MFD.

Synopsis:

```
int oapiGetMFDMode (int pos)
```

Parameters:

pos	MFD position (see Section 9)
-----	------------------------------

Return value:

MFD mode id (see Section 9)

14.9. File management

oapiWriteLine

Writes a line to a file.

Synopsis:

```
void oapiWriteLine (FILEHANDLE file, char *line)
```

Parameters:

file	file handle
line	line to be written (zero-terminated)

oapiWriteScenario_string

Writes a string-valued item to a scenario file.

Synopsis:

```
void oapiWriteScenario_string (  
    FILEHANDLE scn,  
    char *item,  
    char *string)
```

Parameters:

file	file handle
item	item id
string	string to be written (zero-terminated)

oapiWriteScenario_int

Writes an integer-valued item to a scenario file.

Synopsis:

```
void oapiWriteScenario_int (  
    FILEHANDLE scn,  
    char *item,  
    int i)
```

Parameters:

file	file handle
item	item id
i	integer value to be written

oapiWriteScenario_float

Writes a floating point-valued item to a scenario file.

Synopsis:

```
void oapiWriteScenario_float (  
    FILEHANDLE scn,  
    char *item,  
    double d)
```

Parameters:

file	file handle
item	item id
d	floating point value to be written

oapiWriteScenario_vec

Writes a vector-valued item to a scenario file.

Synopsis:

```
void oapiWriteScenario_vec (  
    FILEHANDLE scn,  
    char *item,  
    const VECTOR3 &vec)
```

Parameters:

file	file handle
item	item id
vec	vector to be written

oapiReadScenario_nextline

Reads an item from a scenarion file.

Synopsis:

```
bool oapiReadScenario_nextline (
    FILEHANDLE scn,
    char *&line)
```

Parameters:

scn	file handle
line	pointer to the scanned line

Notes:

- The function returns true as long as an item for the current block could be read. It returns false at EOF, or when an “END” token is read.
- Leading and trailing whitespace, and trailing comments (from “;” to EOL) are automatically removed.
- “line” points to an internal static character buffer.

15. Standard ORBITER modules

15.1. Vsop87

Vsop87.dll is a full implementation of the VSOP87 planetary solutions for Mercury to Neptune.¹ Orbiter uses the VSOP87 “B” series which computes the heliocentric positions for the ecliptic and equinox of J2000. Positions and velocities are calculated by a perturbation method which uses a series of trigonometric perturbation terms. The number of included terms defines the precision of the result. Therefore the computation time will depend on the selected precision. Vsop87.dll supports precision settings between 1e-3 and 1e-8.

Vsop87.dll supports the following planets: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus and Neptune.

According to the VSOP documentation, at full precision (1e-8), the relative error is within 1” for

- Mercury, Venus, Earth and Mars over 4000 years before and after J2000
- Jupiter and Saturn over 2000 years before and after J2000.
- Uranus and Neptune over 6000 years before and after J2000.

If you want to replace Vsop87 with your own code:

- Check section 12 for the callback interface.
- The code for different planets doesn’t need to be implemented in a single DLL. You can replace the calculations for a single planet by writing a module for it, and referencing this module from the planet’s cfg file, while keeping the standard Vsop87 module for the other planets.

15.2. Luna

Luna.dll calculates lunar positions and velocities using a perturbation method. The implementation is derived from Elwood Downey’s xephem,² with time derivative terms added by myself. Coordinates are for equinox and ecliptic of date.

16. Index

<	A
<Planet>_CurrentData.....	30
<Planet>_EclSphData	30
<Planet>_SetPrecision.....	29
	E
	ENGINESTATUS
	4

ENGINETYPE	5
EXHAUSTTYPE	5

L

Luna.....	49
-----------	----

M

MATRIX3	4
---------------	---

O

oapiAcceptDelayedKey	47
oapiGetAirspeed	41
oapiGetAirspeedVector	41
oapiGetAltitude	38
oapiGetAtmPressureDensity	42
oapiGetAttitudeMode	44
oapiGetBank	39
oapiGetEmptyMass	34
oapiGetEngineStatus	43
oapiGetEquPos	40
oapiGetFocusAirspeed	41
oapiGetFocusAirspeedVector	41
oapiGetFocusAltitude	38
oapiGetFocusAtmPressureDensity	42
oapiGetFocusAttitudeMode	44
oapiGetFocusBank	39
oapiGetFocusEngineStatus	43
oapiGetFocusEquPos	40
oapiGetFocusGlobalPos	36
oapiGetFocusGlobalVel	36
oapiGetFocusHeading	40
oapiGetFocusObject	34
oapiGetFocusPitch	39
oapiGetFocusRelativePos	37
oapiGetFocusRelativeVel	37
oapiGetFocusShipAirspeedVector	42
oapiGetFuelMass	35
oapiGetGbodyByIndex	33
oapiGetGbodyByName	33
oapiGetGbodyCount	33
oapiGetGlobalPos	36
oapiGetGlobalVel	36
oapiGetHeading	39
oapiGetMass	34
oapiGetMaxFuelMass	35
oapiGetMFDMode	47
oapiGetObjectByIndex	31
oapiGetObjectByName	31
oapiGetObjectCount	31
oapiGetObjectName	33
oapiGetPitch	38
oapiGetRelativePos	37
oapiGetRelativeVel	37
oapiGetShipAirspeedVector	42
oapiGetSimMJD	46
oapiGetSimStep	45
oapiGetSimTime	45
oapiGetStationByIndex	32
oapiGetStationByName	32
oapiGetStationCount	33

oapiGetTimeAcceleration	46
oapiGetVesselByIndex	32
oapiGetVesselByName	32
oapiGetVesselCount	32
oapiOpenMFD	47
oapiReadScenario_nextline	48
oapiSetAttitudeMode	44
oapiSetEmptyMass	35
oapiSetEngineLevel	43
oapiSetFocusAttitudeMode	45
oapiSetFocusObject	34
oapiSetTimeAcceleration	46
oapiTime2MJD	46
oapiToggleAttitudeMode	44
oapiToggleFocusAttitudeMode	45
oapiWriteLine	47
oapiWriteScenario_float	48
oapiWriteScenario_int	48
oapiWriteScenario_string	48
oapiWriteScenario_vec	48
OBJHANDLE	4
opcCloseRenderWindow	24
opcDLLExit	24
opcDLLInit	23
opcFocusChanged	24
opcOpenRenderWindow	24
opcTimeAccChanged	25
opcTimestep	24
ovcAnimate	28
ovcConsumeKey	29
ovcExit	26
ovcInit	25
ovcLoadState	26
ovcSaveState	27
ovcSetClassCaps	26
ovcSetState	26
ovcTimestep	28
ovcVisualCreated	27
ovcVisualDestroyed	28

R

Rcontrol	4
----------------	---

V

VECTOR3	4
VESSEL	6
AddAttExhaustMode	22
AddAttExhaustRef	22
AddExhaustRef	21
AddMesh	20
ClearAttExhaustRefs	23
ClearExhaustRefs	21
ClearMeshes	20
Constructor	6
Create	6
DefSetState	15
DelExhaustRef	21
GetApDist	18
GetArgPer	18
GetAtmDensity	20

